Introduction to patterns in DDI-Views

*Flavio Rizzolo*

*V1.0 – Sep 22, 2016*

Introduction

DDI-Views introduces a new feature for those developing the model. There are “pattern” packages that consist entirely of abstract classes that can be “realized” by other classes. The pattern classes do not appear in the bindings (e.g. XML or RDF) that users of DDI will see. The realizing classes must mirror all of the properties and relationships of the pattern class although the properties of relationships may be renamed to make the semantics clearer. This document describes several of the pattern classes and their use.

Using the Collections pattern

DDI-Views introduces a generic Collections pattern that can be used to model different types of groupings, from simple unordered sets to all sorts of hierarchies, nesting and ordered sets/bags.

A collection is a container, which could be either a set (i.e. unique elements) or a bag (i.e. repeated elements), of Members. Collections can also be extended with richer semantics (e.g. generic, partitive, and instance, among others) to support a variety of DDI 3.x and GSIM structures, such as Node Sets, Schemes, Groups, sequences of Process Steps, etc. Collections, together with their related classes in the pattern, provide an abstraction to capture commonalities among a variety of seemingly disparate structures.

A Collection consists of zero, one or more Members. A Member could potentially belong to multiple Collections. A Collection is also a Member, which allows for nesting of Collections in complex structures. Members have to belong to some Collection, except in the case of nested Collections where the top level Collection is a Member that doesn’t belong to any Collection.

This pattern can be used via a special type of association called *realizes*. DDI-Views uses *realizes* to say that a class “behaves” like a Collection. For instance, consider a Set that consists of Elements, they implement the Collection pattern as follows: Set *realizes* Collection and Element *realizes* Member.

To realize this pattern all classes involved must be associated in a way that is compatible with the pattern. As a rule of thumb, a more restrictive type of association than the one that appears in the pattern is compatible, a looser one is not. For instance, since the collection pattern has an aggregation association (denoted by the empty diamond), classes realizing the Collection pattern need to be related by either an aggregation or a composition, nothing else. In addition, source and target, when applicable, must also match, e.g. the diamond of the aggregation/composition needs to be on the class realizing Collection, not Member. Similar compatibility rules apply to cardinality. Furthermore, all associations must be realized, with the exception of *IsA* associations, which are usually part of the pattern definition and do not apply to individual realizations in the same way. Renaming associations does not affect compatibility as long as the documentation clearly explains how they map to the association in the pattern.

For instance, consider the model diagram on the right. In this example, a Set class is defined as being composed of at least one Element, i.e. no empty Sets are allowed. In addition, an Element always belong to one and only one Set, which means that deleting the Set will also delete its Elements. Such an association is compatible with the *contains* association in the Collection pattern and thus Set and Element can realize Collection and Member, respectively. In contrast, Schema and XML Instance cannot realize the pattern: the association is neither an aggregation nor a composition, Schema is not a grouping of XML Instances, and the association points from XML Instance to Schema. None of this is compatible with the Collection pattern, in particular with the semantics of the *contains* association between Collection and Member.

Collections can be structured with Binary Relations, which are sets of pairs of Members in a Collection. Binary Relations can have different properties, e.g. *totality*, *reflexivity*, *symmetry*, and *transitivity*, all of which can be useful for reasoning.

A Binary Relation is said to be symmetric if for any pair of Members *a*, *b* in the associated Collection, whenever *a* is related to *b* then also *b* is related to *a*. Based on this property we define two specializations of Binary Relation: Symmetric Binary Relation, when the property is *true*, and Asymmetric Binary Relation, when the property is *false*. Symmetric Binary Relations can be viewed as collections of Unordered Pairs themselves, whereas Asymmetric Binary Relations can be viewed as collections of Ordered Pairs. However, for simplicity, we do not model Relations themselves with the Collection pattern.

We can further classify Binary Relations based on additional properties. We say that a Binary Relation is *total* if all Members of the associated Collection are related to each other. We call it *reflexive* if all Members of the associated Collection are related to themselves. Finally, we say it is *transitive* if for any Members *a*, *b*, *c* in the associated Collection, whenever *a* is related to *b* and *b* is related to *c* then *a* is also related to *c*. [Refer to Dan’s document on Relations for more details.]

These properties can be combined to define subtypes of Binary Relations, e.g. Equivalence Relation, Order Relation, Strict Order Relation, Immediate Precedence Relation, and Acyclic Precedence Relation, among others. Equivalence Relations are useful to define partitions and equivalence classes (e.g. Levels in a Classification). Order Relations can be used to represent lattices (e.g. class hierarchies, partitive relationships), Immediate Precedence Relations can define sequences and trees (e.g. linear orderings, parent-child structures) and Acyclic Precedence Relation can represent directed acyclic graphs (e.g. molecular interactions, geospatial relationships between regions).

These subtypes can also have various semantics, e.g. Part-Of and Subtype-Of for Order Relations, to support a variety of use cases and structures, such as Node Sets, Schemes, Groups, sequences of Process Steps, etc. Note that some of them include temporal semantics, e.g. Strict Order Relation and Acyclic Precedence Relation.



A modeller can use the different semantics types as a guide when trying to decide what type of Binary Relation to realize. For instance, if the new class to be added to the model is a Node Set containing Nodes that will be organized in a parent-child hierarchy, the modeller can define a Node Hierarchy class with PARENT\_OF semantics to structure the Node Set. The type of Binary Relation to realize then is Immediate Precedence Relation because it is the one that has the required semantics in its Semantics Type.

Alternatively, a modeller familiar with the definitions of the Binary Relation properties, i.e. symmetry, reflexivity and transitivity, could make the choice based on what combination represents the type they are looking for. For instance, a parent-child hierarchy requires the Binary Relation to be ANTI\_SYMMETRIC (if a Node is the parent of another, the latter is not the parent of the former), ANTI\_REFLEXIVE (a Node cannot be a parent of itself) and ANTI\_TRANSITIVE (a Node is not the parent of its children’s children). It is easy to see that the only one that satisfies that criteria is the Immediate Precedence Relation.

The next diagram shows an example of the realization of the pattern. We can model Node Hierarchy and Node Hierarchy Pair classes as realizations of Immediate Precedence Relation and Ordered Pair, respectively.



Let us illustrate how this model works with a simple instance. Consider a Geography Statistical Classification with Classification Items representing Canada, its provinces and cities.

Since Statistical Classifications are Node Sets and Classification Items are Nodes, we can view Classification Items such as Canada, Ontario, Quebec, Toronto, etc. as Members in a Collection structured by a Node Hierarchy Relation. Node Hierarchy Pairs represent the parent-child relationships in the Node Hierarchy Relation. For instance, (Canada, Ontario) is a Node Hierarchy Pair in which Canada is the parent and Ontario is the child. Other Node Hierarchy Pairs are (Canada, Quebec) and (Canada, Toronto).

Note that by maintaining the hierarchy in a separate structure, i.e. the Node Hierarchy, Items can be reused in multiple Classifications. For instance, in another geography Statistical Classification provinces grouped into regions, Ontario can be made the child of the Central Region instead of Canada without changing the definition of the Classification Items involved, i.e. Canada, Ontario and Central Region in this case.



Interestingly, Binary Relations might not be enough for some purposes. First of all, some structures cannot be reduced to binary representations, e.g. hypergraphs, without cumbersome supporting structures. In addition, a Binary Relation could be too verbose in some cases since the same Member in a Collection could appear multiple times in different pairs, e.g. one-to-many relationships like parent-child and ancestor-descendent. An n-ary Relation provides a more compact representation for such cases. Like Binary Relations they come in two flavours: Symmetric Relation and Asymmetric Relation.

Asymmetric Relations provide an equivalent, yet more compact, n-ary representation for multiple Ordered Pairs that share the same source and/or target Members. In addition, they can be realized by Correspondence Tables to map Node Sets and their Nodes based on some criterion (e.g. similarity, provenance, etc.). Correspondence Table and Map realize Asymmetric Relation and Ordered Tuple, respectively.

Consider again the geography classification tree of the Canadian example above. All Node Hierarchy Pairs that have the same parent Member could be represented with a single Node Hierarchy Tuple that realizes the Ordered Tuple in the model. The realization will also rename source as parent and target as child.

Although only two cities are shown in the example, Ontario has hundreds of them. Using a Node Hierarchy realizing an Asymmetric Relation, all pairs that have Ontario as parent, e.g. (Ontario, Toronto), (Ontario, Ottawa), (Ontario, Kingston), etc., could be joined into a single n-ary Node Hierarchy Tuple with Ontario as parent and Toronto, Ottawa, Kingston, etc. as children. With this representation we replaced multiple pairs with a single tuple and avoid the repetition of Ontario hundreds of times for each individual pair.

Classifications sometimes need to be mapped to each other. In our geography example we could have two variants with similar structure as shown in the next figure.



Some of the Classification Items in both Classifications are the exactly the same, e.g. Toronto and T.O., they just have different names. Others, e.g. Ottawa and National Capital Region, are only approximate, e.g. Ottawa is part of the National Capital Region (NCR), but the latter is larger and contains other municipalities. For such a case, we use a realization of Ordered Member Correspondence to create a containment mapping between Ottawa and NCR whereas the other mappings are exact.

Symmetric Relations are similarly structured as Asymmetric Relations. They provide an equivalent, yet more compact, n-ary representation for multiple Unordered Pairs that have some Members in common. In addition, they can be used to model (unordered) Correspondences between Collections and Members.

Using the Process pattern

Another pattern introduced in DDI-Views is Process. It consists of a basic set of classes to describe process steps and information flows between them. Some of its classes are extensions of classes in the Collections Pattern. The diagram on the left shows the Process pattern.

A Process Step performs one or more business functions at any granularity. Each Process Step can be performed by a Service. Process Steps and Services can have Interfaces with input, outputs and other interface definitions. Process Steps can be nested and thus describe processes at multiple levels of detail. The Process Control Step handles the execution flow of the steps in its scope. The Information Flow specifies how information objects move between Process Steps by mapping inputs and outputs.

The Process Pattern can be realized in multiple ways. DDI-Views include a well-known realization called Workflow, which can be mapped to process execution languages like BPEL or BPMN.

A Workflow is Sequence of Workflow Steps that performs one or more business functions. Each Workflow Step can be performed by a Workflow Service. There are two types of Workflow Steps: Acts and Control Constructs. Acts represent actions and are atomic Process Steps, i.e. they cannot be composed of other Process Steps. An Act is similar to a terminal in the production rules of a formal grammar and an instruction in a programming language. A Control Construct describes logical execution flows between Process Steps. Control Constructs can be nested via its sub-classes and thus describe workflows at multiple levels of detail. The nesting of Workflow Steps always terminates in an Act. All nesting of Workflow Steps occurs via Workflow Sequences and Conditional Control Constructs, both specializations of Control Constructs. The former models linear execution of Workflow Steps whereas the latter includes three types of iterative constructs: repeatWhile, repeatUntil and Loop. The Workflow Sequence at the end of the *contains* association represents the body of the Conditional Control Construct, which is executed depending on the result of the *condition* evaluation. The specialized sub-classes determine whether the Sequence is executed in each iteration before the condition is evaluated (RepeatUntil), or after (RepeatWhile, Loop). The Loop also provides a counter with *initialValue* and *stepValue* that can be used to specify in the condition how many times the Sequence in the body is executed.



In addition to the iterative constructs, Conditional Control Constructs includes IfThenElse, which provides a means to specify branching control flows. It contains the *condition* inherited from the parent class and two associations: *contains* (also inherited from the parent class), to the Sequence of Process Steps that is executed when the condition is true, and *containsElse*, to an optional Sequence to be executed if the condition is evaluated to false. Optionally, IfThenElse can also have an associated ElseIf construct to model switch statements.

A Workflow Sequence can be viewed as a Collection whose Members are Workflow Steps that can be ordered in two different ways: with one or more Temporal Interval Relations, i.e. a design-time temporal constraint, or with a Rule (constructor) to determine ordering at run-time. Let’s begin discussing temporal constraints.

Temporal Interval Relations provide a mechanism for capturing Allen’s interval relations, one of the best established formalisms for temporal reasoning. Temporal Interval Relations can be used to define *temporal constraints* between pairs of Workflow Steps, e.g. whether the execution of two Workflow Steps can overlap in time or not, or one has to finish before the other one starts, etc. Note that this also supports parallel processing, which is also implicit in the model: the definition of Conditional Control Constructs can contain multiple Workflow Sequences that could be executed in parallel.

There are thirteen Temporal Interval Relations: twelve asymmetric ones, i.e. *precedes*, *meets*, *overlaps*, *finishes*, *contains*, *starts* and their converses, plus *equals*, which is the only one that has no converse or, rather, it is the same as its converse. Together these relations are distinct (any pair of definite intervals are described by one and only one of the relations), exhaustive (any pair of definite intervals are described by one of the relations), and qualitative (no numeric time spans are considered).

Following Allen’s, Temporal Interval Relations are defined as follows.



In DDI-Views, each of the asymmetric Allen’s interval relations is a Temporal Interval Relation that realizes different Binary Relations with specific temporal semantics. All asymmetric Temporal Interval Relations contain Ordered Interval Pairs whereas the only symmetric one, i.e. Equals, contains Unordered Interval Pairs. For instance, the Precedes Interval Relation realizes the pattern as shown in the diagram below.



Precedes Interval Relation and Ordered Interval Pair realize Strict Order Relation and Ordered Pair, respectively. If we look back to the Binary Relations Specialization diagram in the previous section we notice that Strict Order Relation has the TEMPORAL\_PRECEDES semantics, among others, which means that the Process Step at the end of the *source* association in the Ordered Interval Pair has to finish before the one at the end of *target* starts.

The Equals Interval Relation is a slightly different case because it is an equivalence relation rather than an asymmetric one and therefore it contains Unordered Interval Pairs.



Equals Interval Relation and Unordered Interval Pair realize Equivalence Relation and Unordered Pair, respectively. Equivalence Relation is simply a Symmetric Binary Relation that is REFLEXIVE and TRANSITIVE with some additional semantics, among which we find TEMPORAL\_EQUALS, the one required by the Equals Interval Relation. This means that the execution of the two Process Steps at the end of the *maps* association in Unordered Interval Pair begin and end at the same time.

Let’s see how Temporal Interval Relations work with an example.

Consider a questionnaire with set of questions and a control flow logic defined between them. The flow between questions can be modeled with a Workflow Sequence where each question is a realization of a Workflow Step. Now, let’s assume a couple of conditions: (i) question Q3 requires the answer of both Q1 and Q2, and (ii) question Q4 is triggered by answering question Q2. Note the difference between requiring an answer and being triggered by an answer. The former means that there is no necessary immediate execution, i.e. Q3 can be executed long after both Q1 and Q2 have been answered. In other words, the exact moment for executing Q3 will depend on other parts of the control flow logic of the questionnaire, i.e. other constraints defined in the questionnaire flow. However, when a question triggers another it means that the latter is executed right after the former is answered.

With that in mind, the precedence constraint between Q1, Q2 and Q3 can be modeled with a Precedes Interval Relation whereas the one between Q2 and Q4 can be represented by a Meets Interval Relation. Remember that the fact that Q1 precedes Q3 means that Q1 finishes before Q3 starts whereas Q2 meets Q4 means that Q4 starts exactly when Q2 finishes, which is exactly what the two Temporal Interval Relations mentioned above can express.

The following is the resulting model.



So far we described the control flow logic that is specified by Control Constructs. How does information flow between the steps in the control flow logic? We mentioned that the pattern has a class for that purpose, i.e. Information Flow. Binding is the realization of Information Flow for Workflows. A Binding is a design-time class that maps Input and Output Parameters of different steps in a Workflow Sequence or between a step and its sub-steps.



Let’s illustrate the use of Bindings with an example.

Consider a Workflow that implements a fragment of the GSBPM Process phase. This is a specific implementation of GSBPM within the context of an organization in which some sub-steps are not implemented.



The step at the top, Process Data, has four Parameters, three Input and one Output, each with its type between brackets, i.e. Classification, Instance Variable and Edit Rules for Input and Instance Variable for Output. This means that each parameter can hold at runtime only objects of the specified type.

There are also two sub-steps organized in a Workflow Sequence. The first sub-step, Code, has an Instance Variable and a Classification as Inputs and an Instance Variable as Output. The second sub-step, Edit and Impute, has the coded Instance Variable and Edit Rules as Input and an Instance Variable as Output.

How do we put all the pieces together? The top step gets a collected Instance Variable and produces a coded, edited and imputed Instance Variable based on a Classification and some Edit Rules. This is achieved by invoking the two sub-steps just described in sequence.

In order for this to work, the Input and Output Parameters of each step have to map so that the Instance Variable the top step receives moves thru the two sub-steps and is returned back, transformed, to the top step. This is made possible by the Bindings.

The first Binding on the left maps the Classification Input Parameter of Process Variable to that of the Code sub-step. This is an example of input-to-input Binding between two levels, i.e. a container step and its sub-steps. The bottom Binding maps the Instance Variable Output Parameter of Code to the Instance Variable Input Parameter of Edit and Impute. This is an example of an output-to-input Binding between steps at the same level, i.e. within the same Control Construct container. The other Bindings make sure that all Parameters are mapped.

Using the Signification pattern

Based on “Identifiers, Labels, Names, and Designations”, by Frank France and Dan Gillman.

A Sign links a Signified with a Signifier that denotes it. A Signifier is a concept whose extension consists of tokens (perceivable objects).

Signifier, Sign and Signified become part of the Signification Pattern.



A Designation is simply a Sign where the Signified is a Concept. Therefore Designation and Sign realize Sign and Signified, respectively. Signifier becomes the Data Type of the representation property of Sign.

The reason for making Signifier, Sign and Signified into a pattern to be realized as opposed to classes to be extended is that Concepts are not always Signifieds, which is what a specialization would imply. In fact, a Concept is a Signified only if there is a Designation that denotes it. The realization means that the Concept is going to behave like a Signified only in the context described.

Codes enter into the picture as Designations. A Code then is a type of Designation that has Non-Linguistic Signifiers and where the Signified is a Category (Concept).

A Node takes meaning from a *single* Category and has an optional set of Designations. In the case of the Code Item subtype, at least *one* Designation is required, i.e. a Code. It’s important to note that a Code Item can have only *one* Category, thus if there are multiple Codes associated with a Code Item, all of them will have to denote the same Category (see constraint in diagram).