Introduction to patterns in DDI-Views

*Flavio Rizzolo*

*Draft 0.7 – April 9, 2016*

Using the Collection pattern

DDI-Views introduces a generic Collection 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 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 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\_SIMMETRIC (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 enough, Binary Relations might not be enough for some purposes. First of all, some structures cannot be reduced to binary representations, e.g. hypergraphs. 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. Let us consider the asymmetric case first.



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 used to model Ordered Correspondences to map Collections and their Members based on some criterion (e.g. similarity, provenance, etc.). Ordered Collection and Member Correspondences realize Asymmetric Relation and Ordered Tuple, respectively.

Consider now a geography classification tree like 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 and 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.

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.



Back to our geography Statistical Classification example, we can have two variants with similar structure as shown in the next figure.



Using the Process pattern

Another pattern introduced in DDI-Views is Process. It consists of classes to describe business functions and workflows that can be mapped to process execution languages like BPEL or BPMN.

A Process is a Sequence of Process Steps that perform one or more business functions. Each Process Step can be performed by a Service. There are two types of Process 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 an instruction in a programming language and a terminal in the production rules of a formal grammar. A Control Construct describes logical flows between Process Steps.

Process Steps can be nested and thus describe processes at multiple levels of detail. The nesting of Process Steps always terminate in an Act. All nesting of Process Steps occur via Sequences, a specialization of Control Constructs.

Control Constructs include Sequence and Conditional Control Construct. The former models linear execution of Process Steps whereas the latter includes three types of iterative constructs: repeatWhile, repeatUntil and Loop. The 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.

It is important to note that the model also covers parallel processing: Conditional Control Constructs can contain multiple Sequences that could be executed in parallel (more on this later).

A Sequence can be viewed as a Collection whose Members are Process Steps that can be ordered in three different ways: with a Sequence Order (traditional design-time total ordering), with one or more Temporal Interval Relations (design-time temporal constraint or “fuzzy” ordering), or with a Rule (constructor) to determine ordering at run-time.

We discuss Sequence Order first. A Sequence Order realizes Collection pattern as follows.



Sequence Order and Sequence Order Pair realize Strict Order Relation and Ordered Pair, respectively. Let us remember from the Binary Relations Specialization diagram in the previous section that Strict Order Relation is an Asymmetric Binary Relation that is ANTI\_SYMMETRIC, ANTI\_REFLEXIVE and TRANSITIVE. In addition, the Sequence Order realization has *totality*=TOTAL and *semantics*=SUCCESSOR\_OF, which means that it can specify a total order of the Process Steps in the Sequence where the order semantics is given by SUCCESOR\_OF.

We discuss next Temporal Interval Relations. They 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 Process Steps, e.g. whether the execution of two Process Steps can overlap in time or not, or one has to finish before the other one starts, etc. This also supports parallel processing.

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 one 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 Relation contains 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 follows.



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.

Temporal Interval Relations and Sequence Orders can be combined in the same Sequence. For instance, consider the following example:



Question Q2 requires the answer of question Q1 so it has to be executed after Q1. That is an example of the traditional sequence ordering given by the Sequence Order Relation. However, note that there is no dependency between Q2 and Q3 since both require only the answer of Q1. Therefore Q2 and Q3 could be executed at the same time, which can be expressed with the Equals Interval Relation.