Disposition: Resolved

OMG Issue No: 16371

Title: Merge DI Diagrams

Source:

International Business Machines (Mr. Maged Elaasar, melaasar@ca.ibm.com)

Summary:

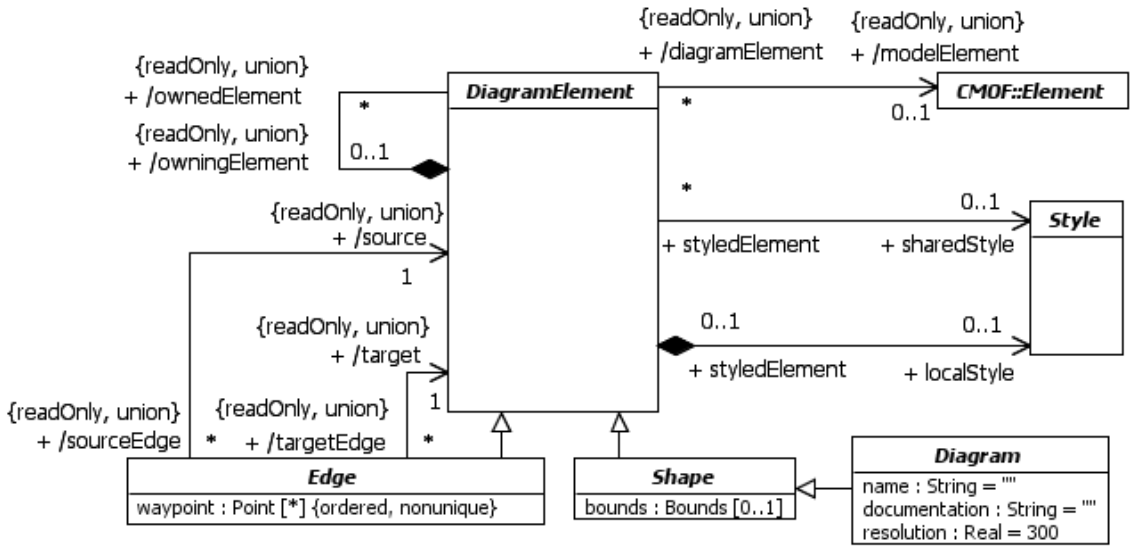
After the first FTF ballot, the DI diagrams are small enough to be merged into one.

Resolution:

The DI metamodel is indeed small enough after Ballot 1 that all concepts can fit in one diagram.

Revised Text:

Replace Figures 9.2 through 9.6 (all figures in Clause 9.2 after Figure 1) with the following:



In clauses 9.3.1 through 9.3.9 (all subclauses of Clause 9.3), replace the bullets under the Diagram heading with

- Figure 9.2 (DI Package)

Disposition: Resolved

Disposition: Resolved

OMG Issue No: 16372

Title: Update Annex A

Source:

International Business Machines (Mr. Maged Elaasar, melaasar@ca.ibm.com)

Summary:

Update Annex A (UML Diagram Definition Example) for the changes in FTF ballot 1 and complete the example.

Resolution:

Add a small example defining a subset of the UML Class diagram with DD..

Revised Text:

Replace Annex A's title and text with the following:

Annex A - UML Class Diagram Definition Example

(Informative)

This annex gives an example of using the DD specification to define a small subset of the UML class diagram. Subsection A.1 gives the UML DI metamodel as an extension of the DI metamodel. Subsection A.2 gives the mapping from this UML DI metamodel to the DG metamodel.

The UML class diagram is chosen due to its widespread use and familiarity. However, to control the scope, the example is limited to representative subset of class diagram elements consisting of three classifiers (Class, Interface and DataType) and three relations (Association, Generalization and InterfaceRealization). This subset exemplifies the notation of shapes (with labels, compartments and alternative notation) and edges (with labels, markers and line styles) of the class diagram.

Note: This example is adapted from the following paper:

Elaasar, M., and Labiche, Y., "Diagram Definition: a Case Study with the UML Class Diagram", ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS), October 2011.

A.1 UML DI

Some design principles used in this example are:

- Avoid interchanging notational information that can be derived from the UML model to minimize redundancy between the DI and UML models.
- Interchange simple layout constraints (bounds for all shapes/labels and waypoints for all edges) and avoid constraints of more complex layout algorithms to make it easier for tools to map to/from their native layouts.
- Interchange the overlapping order of sibling diagram elements (which can happen when a diagram is crowded) by making all nested element collections ordered (a higher index implies a higher overlap order).
- Avoid interchanging purely stylistic properties (e.g., colors/fonts) that tools may give users control over since they may vary dramatically between tools. However, we made an exception to some font properties (e.g., name and size) that we suspected could affect layout.
- Keep the DI class hierarchy small, thus easier to maintain and evolve, by avoiding extensive sub-classing (resembling the UML class hierarchy). Instead, we allow DI classes to have a mixed bag of optional properties that apply in specific UML contexts only.

The UML DI metamodel (Figure A.1) extends the DI metamodel, where appropriate, using metamodel extension semantics (subclassing and property subsetting and redefinition). Specifically, the class *UMLDiagram* composes a collection of elements of type *UMLDiagramElement*. The latter optionally references an element from a UML model and can be styled with instances of class *UMLStyle*, which has two properties: *fontName* and *fontSize*.

Note — the reason property UMLDiagramElement::modelElement can redefine DiagramElement::modelElement even though UML::Element is not explicitly a subclass of CMOF::Element is due to the semantics of clause 9.1 of MOF, which says that class Element is an implicit superclass of all MOF-based model elements.

Classes are defined for interchanging the chosen shapes and edges of the class diagram, based on three notational patterns in the UML specification (Figure A.2):

- pattern (a): a shape that has a label and an optional list of compartments, each of which has an optional list of other labels (e.g., the classifier box notation).
- pattern (b): a shape that has a label only (e.g., the interface ball notation)
- pattern (c): an edge that has an optional list of labels (e.g., the association notation)

However, pattern (b) is really a special case of (a) when there is no compartment. Based on that, three shape classes (*UMLShape*, *UMLLabel* and *UMLCompartment*) and one edge class (*UMLEdge*) are defined and related with the multiplicities of patterns (a) and (c). These classes (except *UMLCompartment*) are subclasses of *UMLDiagramElement* to enable them to be styled independently, to reference their own UML elements, and to be connectable (an edge is made to only connect elements of type *UMLDiagramElement*). Properties on the classes disambiguate their notation. For example, a *kind* can be set on a label to indicate what aspects of the UML element to show textually. A flag *showClassifierShape* can be set on a classifier's shape to indicate whether to use the box notation (this covers only a subset of the possible notational options for brevity).

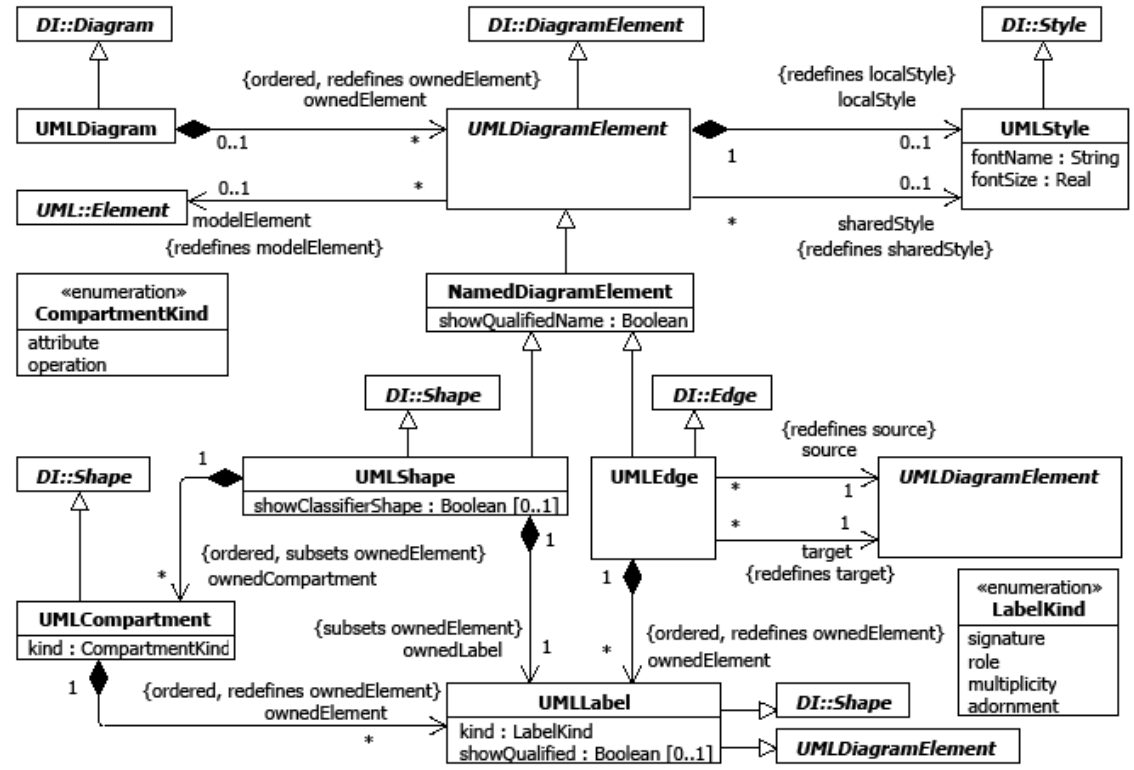


Figure A.1 - UML DI Metamodel

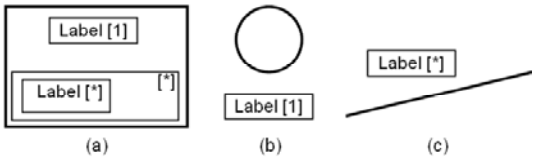


Figure A.2 - UML DI Notational Patterns

A.2 Mapping UML DI to DG using QVT Operational

Recall from Section 7 that a modeling language can specify its concrete graphical syntax as a mapping from its DI metamodel, which references its abstract syntax metamodel, to the DG metamodel. The mapping can be expressed using any suitable mapping language. This examples expresses the mapping using the QVT Operational (QVTO) transformation language (for brevity, only selected parts of the transformation are shown). QVTO is a standard language and has an available implementation (on Eclipse). This assumes reader's familiarity with QVTO, OCL and the UML 2.x metamodel.

The UML class diagram's concrete syntax is defined with a QVTO transformation (Figure A.3) from the UML DI metamodel (Section A.1) to the DG metamodel. The transformation starts by looking for all instances of UMLDiagram and initiating the mapping for them (lines 2-4). Mappings (like operations) are defined on UML DI classes and have DG classes as return types. For example, a mapping named toGraphics is defined on the class *UMLDI::UMLDiagram* and has *DG::Canvas* as a return type (line 5). This maps an instance of *UMLDiagram* to an instance of *Canvas*, and initializes the properties of the latter according to the body of the mapping. In this case, the body iterates on all the owned elements of the diagram, mapping each one in turn to graphics, and adding the resulting graphical elements as members of the canvas (line 6).

Furthermore, a *UMLShape* maps to a *Group* (lines 12-17) consisting of the following: a graphic for the model element (line 14), a graphic for the owned label (line 15) and a graphic for each owned compartment (line 16). These graphics are produced by other nested mappings (defined later). A similar mapping is defined for *UMLEdge* (lines 18-22). However, the mapping for *UMLCompartment* (lines 23-26) is different as the first member graphic is fixed as a *Rectangle* whose bounds are defined by the compartment. The mapping for *UMLLabel* (lines 27-40) is also different as it maps to a *Text* whose bounds are defined by the label and whose data value is defined based on the label's kind. For example, if the kind is *signature*, the value is defined by a query *getSignature* defined on *UML::NamedElement* (line 33-34). Also notice how the mapping inherits (line 28) another mapping (lines 8-11) that copies over the local and shared styles. Another local style is added (line 39) based on the label's model element (e.g., the *fontItalic* property is set to true for the *signature* label in the case of an abstract classifier).

Some of the queries used for the label mapping are shown in Figure A.4. The *getSignature* query (lines 1-3) returns the (simple or qualified) name of an element based on a flag. The query is overridden for different UML types to specify their unique signatures. For example, *UML::Interface* (lines 4-6) overrides it to prefix the name with the «Interface» keyword. *UML::Property* (lines 12-17) overrides it to return the full signature of a property in an attribute compartment (with type, multiplicity, etc.).

Figure A.5 shows mappings between UML classifiers and their corresponding graphical elements (e.g., box or ball notation). The first mapping (lines 1-3), defined on *UML::Element*, delegates to other mappings depending on the type of the element. Notice that both *UML::Class* (lines 4-6) and *UML::DataType* (lines 7-9) have one mapping each creating a rectangle, while *UML::Interface* has two mappings, one creating a *DG::Rectangle* (lines 10-13) and the other creating a *DG::Circle* (lines 14-20), based on the flag *showClassifierShape* (lines 11, 15) on *UMLShape*.

Figure A.6 shows mappings between UML relations and poly lines. The first mapping (lines 10-13), defined on *UML::Element*, delegates to other mappings depending on the type of the element. The mapping of relation *UML::InterfaceRealization* (lines 14-19) copies the edge's waypoints to the poly line'a points (line 15). As the notation of this relation depends on whether the interface shape was shown as a box or a ball, this is checked first (line 16). If it is shown as a box, a shared style with a dash pattern (lines 1-2) and a closed arrow marker (lines 3-9) are used (lines 17-18).

```
01    transformation UMLDIToDG(in umldi : UMLDI, out DG);
02    main() {
03       umldi.objectsOfType(UMLDiagram)->map toGraphics();
04    }
05    mapping UMLDiagram::toGraphics() : Canvas {
06       member += self.ownedElement->map toGraphics();
07    }
08    mapping UMLDiagramElement::toGraphics() : Group {
09       localStyle := copyStyle(self.localStyle);
10       sharedStyle := copyStyle(self.sharedStyle);
11    }
12    mapping UMLShape::toGraphics() : Group
13       inherits UMLDiagramElement::toGraphics {
14       member += self.modelElement.map toGraphics(self);
15       member += self.ownedLabel.map toGraphics ();
16       member += self.ownedCompartment->map toGraphics();
17    }
18    mapping UMLEdge::toGraphics() : Group
19       inherits UMLDiagramElement::toGraphics {
20       member += self.modelElement.map toGraphics(self);
21       member += self.ownedElement->map toGraphics ();
22    }
23    mapping UMLCompartment::toGraphics() : Group {
24       member += object Rectangle {bounds := self.bounds};
25       member += self.ownedElement->map toGraphics ();
26    }
27    mapping UMLLabel::toGraphics () : Text
28       inherits UMLDiagramElement::toGraphics {
29       var e := self.modelElement;
30       var q := self.showQualified;
31       bounds := self.bounds;
32       data := switch {
33          case (self.kind = LabelKind::signature)
34             e.oclAsType(NamedElement).getSignature(q);
35          case (self.kind = LabelKind::role)
36             e.oclAsType(Property).getRole();
37          ...
38       };
39       localStyle += e.map toStyle(self);
40    }
```

Figure A.3 - QVTO Mapping from UML D to DG

```
query NamedElement::getSignature(q : Boolean) : String {
   return self.getName(q);
}
query Interface::getSignature(q : Boolean) : String {
   return "«Interface»\n" + self.getName(q);
}
query Property::getSignature(q : Boolean) : String {
   var t := if self.type->notEmpty() then ":" +
            self.type.getSignature(q) else "" endif;
   return self.getRole()+ t + self.getAdornment();
}
query Property::getRole() : String {
   var d := if self.isDerived then "/" else "" endif;
   var v := if self.visibility = VisibilityKind::public
      then "+" else ... endif;
   return d + v + self.getName(false);
}
query NamedElement::getName(q : Boolean) : String {
   return if q then self.qualifiedName
          else self.name endif;
}
query Property::getAdornment() : String {
   return "{" + ... + "}";
}
```

Figure A.4 - Queries used by the UML Label Mapping

```
mapping Element::toGraphics(s:UMLShape):GraphicalElement
   disjuncts Interface::toRectangle, Interface::toCircle,
             Class::toRectangle, DataType:toRectangle {}
mapping Class::toRectangle (s:UMLShape) : Rectangle {
   bounds := s.bounds;
}
mapping DataType::toRectangle (s:UMLShape) : Rectangle {
   bounds := s.bounds;
}
mapping Interface::toRectangle (s:UMLShape) : Rectangle
   when { s.showClassifierShape=true } {
   bounds := s.bounds;
}
mapping Interface::toCircle (s:UMLShape) : Circle
   when { s.showClassifierShape=false } {
   var b := s.bounds;
   center := object Point{b.x+b.width/2;b.y+b.height/2};
   radius := if b.width<b.height then b.width/2
             else b.height/2 endif;
}
```

Figure A.5 - UML Classifier Mapping to Graphics

```
01    property interfaceRealStyle = object DG::Style {
02       strokeDashLength := Sequence {2, 2} };
03    property interfaceRealMarker = object Marker {
04       size := object Dimension {width := 10; height := 10};
05       reference := object Point {x := 10; y := 5};
06       member += object Polylgon {
07          point += object Point{ x:=0; y:=0 };
08          point += object Point{ x:=10; y:=5 };
09          point += object Point{ x:=0; y:=10 }; }; };
10    mapping Element::toGraphics(e:UMLEdge):GraphicalElement
11       disjuncts Association::toPolyline,
12        Generalization::toPolyline,
13        InterfaceRealization::toPolyline {}
14    mapping InterfaceRealization::toPolyline(e:UMLEdge):Polyline{
15       point := e.waypoint;
16       var s = e.target.showClassifierShape;
17       sharedStyle := if s then interfaceRealStyle endif;
18       endMarker := if s then interfaceRealMarker endif;
19    }
```

Figure A.6 - UML Relation Mapping to Graphics

Disposition: Resolved

Disposition: Resolved

OMG Issue No: 16386

Title: MOF Compliance

Source:

International Business Machines (Mr. Maged Elaasar, **melaasar@ca.ibm.com**)

Summary:

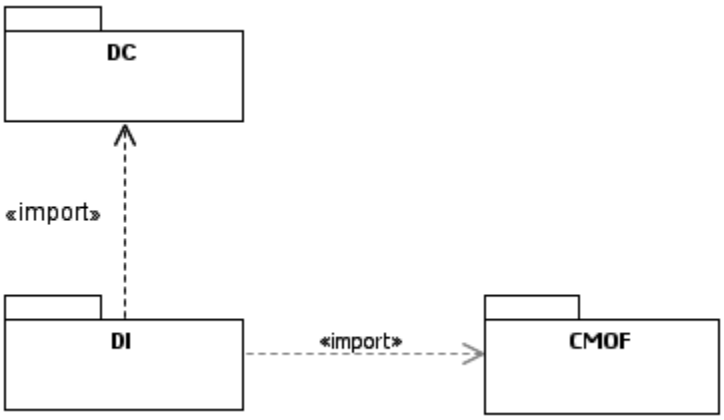
Update for compliance with MOF 2.4 (for example UML:Element instead of MOF:Element), and change text to not refer to MOF, just "metamodel extension".

Resolution:

MOF 2.4 is not formalized yet. It is valid to leave the property typed by CMOF::Element as it is the super class of all metaclasses. We should add a package import from DI to CMOF to make the dependency explicit.

Revised Text:

Replace figure 9.1 by the following to add a package import to CMOF:



In Clause 9.1, first paragraph, after “(Section 8)” insert “and CMOF”.

Disposition: **Resolved**

Disposition: Resolved

OMG Issue No: 16387

Title: Owning styles

Source:

International Business Machines (Mr. Maged Elaasar, **melaasar@ca.ibm.com**)

Summary:

After issue 15902, the spec should not refer to owning styles, for example, see DI, DiagramElement, fourth paragraph, next to last sentence, and DI, Style, second paragraph, first sentence.

Resolution:

Revise as suggested.

Revised Text:

In Clause 9.3.1 (Diagram), Description, last paragraph, remove the first sentence (the one starting “A diagram can own”).

In 9.3.2 (DiagramElement), Description, last paragraph, replace the next to last sentence, as modified by issue 15902, with “Shared style elements are owned by other elements, which might be packaging elements in the language incorporating diagram interchange.”.

In Clause 9.3.5 (Style), Description, second paragraph, first sentence, as modified by 15902, replace “owned by other elements at a higher level like a diagram or other diagram elements with nested elements” with “owned elsewhere (e.g., by packaging elements in the language incorporating diagram interchange)”.

Disposition: Resolved

Disposition: Resolved

OMG Issue No: 16388

Title: Cascading called inheritance

Source:

International Business Machines (Mr. Maged Elaasar, **melaasar@ca.ibm.com**)

Summary:

The spec sometimes refers to style cascading incorrectly as inheritance. Inheritance applies to classes, cascading applies to instances. Search on "inherit" to find them.

Resolution:

Revise as suggested.

Revised Text:

Clause 9.3.5 (Style), Description, third paragraph, replace “style inheritance”, with “cascading styles”, and “inherits its” with “gets its”.

Clause 10.3.11 (GraphicalElement), Description, second paragraph, second sentence, replace “gets inherited” with “gets its value”.

Clause 10.3.12 (Group), Description, second sentence, replace “are inherited by” with “cascade to”.

Clause 10.3.18 (Marker), Description, third paragraph, next to last sentence, replace “Even though referencing marked elements are not in the group chain of a marker, it inherits their styles in the context of every reference” with “The styles of marked elements cascade to their referenced markers in each case”.

Clause 10.3.23 (Pattern), Description, next to last sentence, replace “Even though referencing graphical elements filled with patterns are not in the group chain of the pattern's tile, the tile inherits their styles in the context of every reference” with “The styles of those graphical elements cascade to the pattern tiles in each case”.

Clause 10.3.32 (Style), Description, second paragraph, last sentence, replace “style inheritance” with “cascading styles”, and replace “inherits” with “gets”.

Disposition: Resolved

Disposition: Resolved

OMG Issue No: 16389

Title: Optional bounds

Source:

International Business Machines (Mr. Maged Elaasar, **melaasar@ca.ibm.com**)

Summary:

After 15996, bounds should not be required, for example, search on "with a given bounds". Explain that if DI does not specify bounds, then the mapping to DG does.

Resolution:

The only text remaining after 15902 that implies the presence of bounds is examples were bounds is supposed to be specified.

Revise the description of bounds to indicate it is optional.

Revised Text:

In clause 9.3.4 (Shape), under associations, in the text of the bounds bullet, before "bounds of the shape", insert "optional ".

Disposition: **Resolved**

Disposition: Resolved

OMG Issue No: 16390

Title: Annex B

Source:

International Business Machines (Mr. Maged Elaasar, **melaasar@ca.ibm.com**)

Summary:

Update or remove Annex B (DG to SBV Mapping).

Resolution:

Remove the annex until we have content for it.

Revised Text:

Remove Annex B (DG to SVG Mapping).

Disposition: **Resolved**

Disposition: Resolved

OMG Issue No: 16392

Title: Optional waypoints

Source:

International Business Machines (Mr. Maged Elaasar, melaasar@ca.ibm.com)

Summary:

After 15959, multiple waypoints should not be required, for example, see DI Edge, first paragraph, and in the attribute description.

Resolution:

Revise and suggested.

Revised Text:

Clause 9.3.4 (Edge),

Description,

First sentence, remove “two or more”.

Second sentence, replace “as follows: the first waypoint is positioned at the source element, the last waypoint is positioned at the target element and the waypoints in between specify” with “, specifying”.

Under Attributes, replace “a list of two or more” with “an optional list of”.

Disposition: **Resolved**

Disposition: Resolved

OMG Issue No: 16408

Title: Spurious question marks

Source:

NIST (Mr. Conrad Bock, conrad.bock(at)nist.gov)

Summary:

Remove "?"s before attribute names.

Resolution:

The "♦" (composition) symbols got replaced by "?" symbols by mistake during the editing process of the Beta 2 specification. The fix is to restore the "♦" symbols

Revised Text:

In all bulleted lists, replace any "?" found just after the bullet with "♦".

Disposition: **Resolved**

Disposition: Resolved

OMG Issue No: 16409

Title: Visibility markers not needed

Source:

NIST (Mr. Conrad Bock, conrad.bock(at)nist.gov)

Summary:

Would be easier to read the specification with the "+" visibilitiy markers in the figures and attribute/association sections.

Resolution:

We agree with the proposal to remove the visibility symbol as it is not relevant for metamodels.

Revised Text:

In clause 8, 9 and 10, under attribute/associations sections, remove the “+” symbol from the beginning of the bulleted items,

Update figures (8.2, 8.3, 9.2, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7 and 10.8) to remove the “+” symbols.

Disposition: **Resolved**

Disposition: Closed, No Change

OMG Issue No: 16410

Title: Collections => sets

Source:

NIST (Mr. Conrad Bock, conrad.bock(at)nist.gov)

Summary:

Collections should be called sets, except when they are non-unique.

Resolution:

The word collection is valid as it is a generic way of describing sets, lists, sequences...etc. All metamodel properties are already annotated with the specific kinds of collections.

Disposition: **Closed, No Change**

Disposition: Resolved

OMG Issue No: 16485

Title: Diagram Description

Source:

International Business Machines (Mr. Maged Elaasar, **melaasar@ca.ibm.com**)

Summary:

Some of the clause 9.3.1 description is redundant with 8.1.3 and it does not make sense after changes from ballot 1.

Resolution:

Remove the redundant paragraph as suggested.

Revised Text:

In clause 9.3.1 (Diagram), under description, remove the third paragraph starting with “The collection of nested elements in a diagram is ordered and”.

Disposition: **Resolved**