8 Diagram Common

The Diagram Common (DC) package contains abstractions shared by the Diagram Interchange and the Diagram Graphics packages.

8.1 Overview

The Diagram Common (DC) package contains a number of common primitive types as well as structured data types that are used by the other DD packages, namely the Diagram Graphics(DI) package (Clause 9) and the Diagram Interchange(DG) package (Clause 10). The DC package itself does not depend on other packages.

The following sub clauses discuss common assumptions that are made by DC and the other DD packages.

8.1.1 Measurement Unit

All coordinates and lengths defined by the DD packages are assumed to be in user units. A user unit is a value in the user coordinate system, which initially (before any transformation is applied) aligns with the device's coordinate system (for example, a pixel grid of a display). A user unit, therefore, represents a logical rather than physical measurement unit. Since some applications might specify a physical dimension for a diagram as well (mainly for printing purposes), a mapping from a user unit to a physical unit can be specified as a diagram's resolution. (Inch is chosen in this specification to avoid variability but tools can easily convert from/to other preferred physical units.) Resolution specifies how many user units fit within one physical unit (for example, a resolution of 300 specifies that 300 user units fit within 1 inch on the device).

8.1.2 Coordinate System

This specification assumes a two-dimensional x-y coordinate system that has its origin at coordinate x=0, y=0. The x-axis is horizontal and its coordinate values increases to the right with negative coordinates allowed. Similarly, the y-axis is vertical and its coordinate values increases to the bottom with negative coordinates allowed.

8.1.3 Z-Order

Diagram (or graphical) elements may overlap in some situations (their renderings may intersect), in which case it is important to determine which ones appear below or more hidden (have lower z-order) and which ones appear above or more visible (have higher z-order). The general rules for determining the relative z-order are:

- Owned elements are higher in z-order than their owning elements.
- Elements that appear higher in the same "ordered" composition collection have higher z-order than those that appear lower in the same collection.
- The relative z-order between different composition collections of elements in the same owning element cannot be specified directly in the metamodel but needs to be specified in the description of the owning element (for example, labels and compartments are separate collection of children of a UML Class shape, but labels are always higher in z-order than compartments).

8.1.4 Rotation

Rotations specified throughout this specification are made in degrees and can be positive (clock-wise) or negative (counter-clock-wise).

8.2 Abstract Syntax

Issue: 17453 – Use MOF Primitive Types

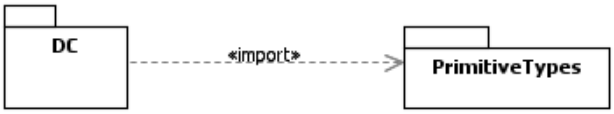

Figure 8.1 Dependencies of the DC package

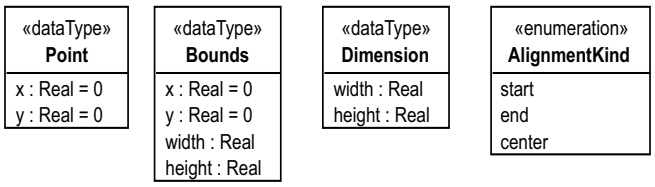

Figure 8.2 - The layout data types

Figure 8.3 - The color data type

8.3 Classifier Descriptions

8.3.1 AlignmentKind [Enumeration]

AlignmentKind enumerates the possible options for alignment for layout purposes.

Description

AlignmentKind enumerates the possible kinds for alignment for layout purposes (e.g., for text alignment within a bounding box).

Diagrams

- Figure 8.2 (Layout Types)

Literals

- start - an alignment to the start of a given length.
- end - an alignment to the end of a given length
- center - an alignment to the center of a given length

8.3.2 Bounds [DataType]

Bounds specifies a rectangular area in some x-y coordinate system that is defined by a location (x and y) and a size (width and height).

Description

Bounds is used to specify a rectangular area in some x-y coordinate system. The area is specified with a (x, y) location, representing the distance of the area’s top-left corner from the origin, and a size (width and height) along the x-y axes.

Diagrams

- Figure 8.2 (Layout Types)

Attributes

- x : Real [1] = 0 - a real number (>=0 or <=0) that represents the x-coordinate of the bounds
- y : Real [1] = 0 - a real number (>=0 or <=0) that represents the y-coordinate of the bounds
- width : Real [1] - a real number (>=0) that represents the width of the bounds
- height : Real [1] - a real number (>=0) that represents the height of the bounds

Constraints

- non_negative_size: the width and height of bounds cannot be negative
  [OCL] width >= 0 and height >=0

8.3.3 Color [DataType]

Color is a data type that represents a color value in the RGB format.

Description

Color is used as a type for attributes that represent color. The color value is encoded using the RGB format as three separate integers in the range (0...255) representing the red, green, and blue components of the color. For example the color yellow is (red=255, green=255, blue=0).

Diagrams

- Figure 8.3 (Color Type)

Attributes

- red : Integer [1] - the red component of the color in the range (0..255)
- green : Integer [1] - the red component of the color in the range (0..255)
- blue : Integer [1] - the red component of the color in the range (0..255)

Constraints

- valid_rgb: the red, green, and blue components of the color must be in the range (0...255).
  [OCL] red >= 0 and red <=255 and green >= 0 and green <=255 and blue >= 0 and blue <=255

8.3.4 Dimension [DataType]

Dimension specifies two lengths (width and height) along the x and y axes in some x-y coordinate system.

Description

Dimension is used to specify two lengths, a width along the x-axis and a height along the y-axis, in a x-y coordinate system.

Diagrams

- Figure 8.2 (Layout Types)

Attributes

- width : Real [1] - a real number (>=0) that represents a length along the x-axis.
- height : Real [1] - a real number (>=0) that represents a length along the y-axis.

Constraints

- non_negative_dimension: the width and height of a dimension cannot be negative
  [OCL] width >= 0 and height >=0

8.3.5 KnownColor [Enumeration]

KnownColor is an enumeration of 17 known colors.

Description

KnownColor enumerates 17 known colors, defined by the CSS specification, which are: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white, and yellow.

Diagrams

- Figure 8.3 (Color Type)

Literals

- maroon - a color with a value of #800000
- red - a color with a value of #FF0000
- orange - a color with a value of #FFA500
- yellow - a color with a value of #FFFF00
- olive - a color with a value of #808000
- purple - a color with a value of #800080
- fuchsia - a color with a value of #FF00FF
- white - a color with a value of #FFFFFF
- lime - a color with a value of #00FF00
- green - a color with a value of #008000
- navy - a color with a value of #000080
- blue - a color with a value of #0000FF
- aqua - a color with a value of #00FFFF
- teal - a color with a value of #008080
- black - a color with a value of #000000
- silver - a color with a value of #C0C0C0
- gray - a color with a value of #808080

8.3.6 Point [DataType]

A Point specifies a location in some x-y coordinate system.

Description

Point is used to specify a coordinate that is at a given distance (along the x and y axes) from the origin of some x-y coordinate system. The point (0, 0) is considered to be at the origin of that coordinate system. Coordinates increase towards the right of the x-axes and towards the bottom of the y-axis.

Diagrams

- Figure 8.2 (Layout Types)

Attributes

- x : Real [1] = 0 - a real number (<= 0 or >= 0) that represents the x-coordinate of the point.
- y : Real [1] = 0 - a real number (<= 0 or >= 0) that represents the y-coordinate of the point.

Issue: 17453 – Use MOF Primitive Types

9 Diagram Interchange

The Diagram Interchange (DI) package enables interchange of graphical information that language users have control over, such as position of nodes and line routing points. Language specifications specialize elements of DI to define diagram interchange elements for a language.

9.1 Overview

Issue: 16628 – DiagramElements cannot represent multiple model Elements

The Diagram Interchange (DI) package contains a number of types used in the definition of diagram interchange models. The package imports the Diagram Common package (Clause 8) and MOF, as shown in Figure 9.1, that contains various relevant data types. The DI package contains many abstract types for extension and refinement by concrete types in domain-specific DI packages. DI is a framework meant for extension rather than a component ready to be used out of the box. It provides typically needed diagram interchange information for customized DI models in specific graphical domains.

9.2 Abstract Syntax

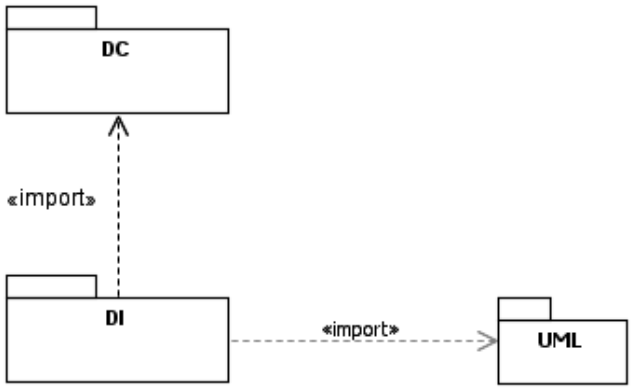

Figure 9.1 - Dependencies of the DI package

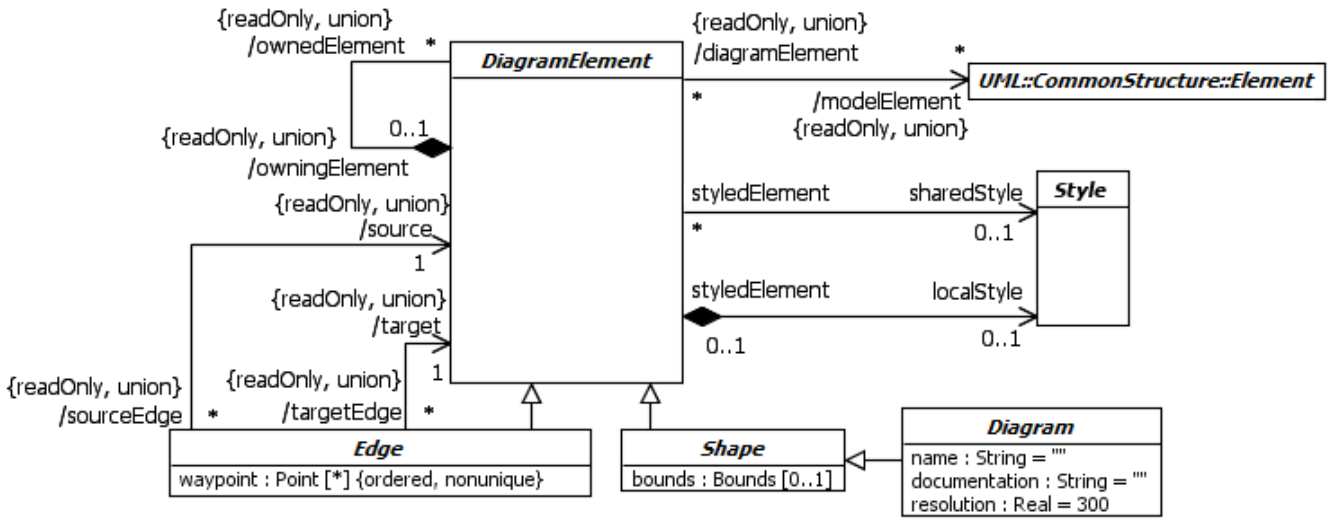

Figure 9.2 - Diagram Element

9.3 Classifier Descriptions

9.3.1 Diagram [Class]

Diagram is an abstract container of a graph of diagram elements. Diagrams are diagram elements with an origin point in the x-y coordinate system. Their elements are laid out relative to their origin point.

Description

A diagram does not need to be nested. It can be persisted in the same resource as the abstract syntax model or in a different resource. It can also be owned by elements of the abstract syntax model, or by no element at all (like being the root of the resource).

A diagram represents a two dimensional x-y coordinate system that is used to layout nested and inter-connected diagram elements. A diagram has an origin point (0, 0) on the x and y. The coordinate system of a diagram increases along the x-axis from left to right and along the y-axis from top to bottom. All the nested diagram elements are laid out relative to their nesting diagram's origin.

Issue: 16628 – DiagramElements cannot represent multiple model Elements

As a kind of diagram element, a diagram may reference model elements from an abstract syntax model, in which case the whole diagram is considered a depiction of those elements (e.g., an activity diagram is a depiction of a UML activity). Alternatively, a diagram without such a reference is simply a layout container for its diagram elements (e.g., a class diagram is a container for UML class shapes and edges).

A diagram can have a name and a documentation. This information is not shown as part of the rendering of the diagram itself but can be used in an application to label a diagram (e.g., "DI Package Diagram") in a browser and show its intent (e.g., "A diagram that shows the classes of the DI package").

A diagram also specifies a resolution expressed in units per inch. The resolution specifies the conversion ratio between the logical units used by the diagram and a unit of physical measurement (an inch in this case). For example, a resolution value of 300 specifies that every 300 logical unit of length map to an inch. The resolution value is mainly used when printing diagrams or when rendering diagrams on display in their physical size.

Styles contain combinations of style property values used by different elements across the diagram. This allows a large number of elements in a diagram to reference a small number of styles, which would dramatically reduce a diagram's footprint.

Diagrams

- Figure 9.2 (DI Package)

Generalizations

- Shape [Abstract Class]

Attributes

- name : String [1] = "" - the name of the diagram.
- documentation : String [1] = "" - the documentation of the diagram.
- resolution : Real [1] = 300 - the resolution of the diagram expressed in user units per inch.

9.3.2 DiagramElement [Abstract Class]

DiagramElement is the abstract super type of all elements in diagrams, including diagrams themselves. When contained in a diagram, diagram elements are laid out relative to the diagram's origin.

Description

Issue: 16628 – DiagramElements cannot represent multiple model Elements

A diagram element can be useful on its own (i.e., purely notational) or more commonly used as a depiction of other MOF-based elements from an abstract syntax model (like a UML model). In the latter case, the diagram element references the depicted model elements and defines notational properties for those elements. An example of a depicting diagram element is a Class shape on a UML diagram that specifies the bounds of the class, its colors, its compartments...etc. An example of a purely notational diagram element is a Note shape on a UML diagram that provides a textual description of part of the diagram. The diagram element's reference to model elements is defined abstractly as derived union to allow language-specific extensions of DI to refine it further to suit their purposes (like specializing its type).

A diagram element can own other diagram elements in a graph-like hierarchy. The collection of owned elements is defined abstractly as a derived union to allow language-specific extensions of DI to define the allowed topologies for their diagram elements (e.g., a UML class shape can own UML compartments). This collection is also specialized in subclasses of diagram element in the DI package.

More specialized diagram element types define properties that characterize their nature. However, a subset of those properties is stylistic in nature and tends to have similar values across many diagram elements. Examples of such properties are fill properties, stroke properties, and font properties. To minimize the footprint of diagram interchange models, those stylistic properties are not defined on diagram elements directly but are rather defined on Style elements

that can be owned and/or shared by diagram elements. Shared style elements are owned by other elements, which might be packaging elements in the language incorporating diagram interchange. Style property values are calculated based on a well-defined algorithm given in “Style [Abstract Class]” on page 19.

Diagrams

- Figure 9.2 (DI Package)

Specializations

- Edge [Abstract Class]
- Shape [Abstract Class]

Association Ends

- /modelElement : Element [*] {readOnly, union} - a reference to a depicted model elements, which can be any MOF-based element.
- /owningElement : DiagramElement [0..1] {readOnly, union} - a reference to the diagram element that directly owns this diagram element.
- ♦ /ownedElement : DiagramElement [*] {readOnly, union} - a collection of diagram elements that are directly owned by this diagram element.
- ♦ localStyle : Style [0..1] - a reference to an optional locally-owned style for this diagram element.
- sharedStyle : Style [0..1] - a reference to an optional shared style element for this diagram element.

9.3.3 Edge [Abstract Class]

Edge is a diagram element that renders as a polyline, connecting a source diagram element to a target diagram element, and is positioned relative to the origin of the diagram.

Description

Edge represents a diagram element defined with a sequence of connected waypoints forming a polyline that connects two diagram elements: a source element and a target element (could be the same as the source as in self connection). The waypoints are positioned relative to the origin of the nesting diagram, specifying a route for the polyline on the diagram.

An edge can be purely notational, i.e., does not reference any model element. An example is the line attaching a comment to a UML element. On the other hand, an edge can be a depiction of a relational element from an abstract syntax model. Examples include UML generalization edge or a BPMN message flow edge. In that case, the edge’s source and target reference diagram elements depicting the relationship’s source and target elements (or its two related elements if the relationship is not directed) respectively. The edge’s source and target references are defined abstractly as derived unions. In an extending language-specific DI metamodel, these references need to be refined. In case the source and target references can be derived unambiguously from the model element, the properties can be redefined with that derivation logic. Otherwise, the properties can be specialized with concrete settable properties.

Diagrams

- Figure 9.2 (DI Package)

Generalizations

- DiagramElement [Abstract Class]

Attributes

- waypoint : Point [*] {ordered, nonunique} - an optional list of points relative to the origin of the nesting diagram that specifies the connected line segments of the edge.

Association Ends

- /source : DiagramElement [1] {readOnly, union} - the edge's source diagram element, i.e., where the edge starts from.
- /target : DiagramElement [1] {readOnly, union} - the edge's target diagram element, i.e., where the edge ends at.

9.3.4 Shape [Abstract Class]

Shape is a diagram element with given bounds that is laid out relative to the origin of the diagram.

Description

Shape is an abstract class that is expected to be further sub classed in a language-specific DI metamodel.

A shape can be purely notational, i.e., does not reference any model element. Examples include a note shape on a UML class diagram with some text describing the diagram and an overlay shape with some semi-transparent fill enclosing a bunch of shapes on the diagram to make them stand out. On the other hand, a shape can be a depiction of a component (non-relational) element from an abstract syntax model. Examples include a UML class shape and a BPMN activity shape.

Diagrams

- Figure 9.2 (DI Package)

Generalizations

- DiagramElement [Abstract Class]

Specializations

- Diagram [Class]

Attributes

- bounds : Bounds [0..1] - the optional bounds of the shape relative to the origin of its nesting plane.

9.3.5 Style [Abstract Class]

Style contains formatting properties that affect the appearance or style of diagram elements, including diagram themselves.

Description

A Style is a set of properties (e.g., fontName, fillColor or strokeWidth) that affect the appearance or style of diagram elements rather than their intrinsic semantics. Style is defined as an abstract class without prescribing any style properties to leave it up to language-specific DI extensions to define concrete style classes with their own properties that are applicable to their diagram element types.

A style element can either be local to (owned by) a diagram element or shared between (referenced by) several diagram elements, in which case it is owned elsewhere (e.g., by packaging elements in the language incorporating diagram interchange). A value set to a local style property in a diagram element overrides one that is set to the same property on a shared style referenced by the same diagram element.

Style properties are typically defined as optional to allow the state of “unset” to be legal. This is needed to implement cascading style, where an unset style property in one diagram element gets its value from the closest diagram element in its owning element chain that has a value set for that property.

The above semantics effectively specify that a value for a style property is based on the following mechanisms (in order of precedence):

- if there is a cascading value set on a local style, use it.
- Otherwise, if there is a cascading value set on a shared style, use it.
- Otherwise, if a cascading value is available from a diagram element in the owning element chain, use it from the closest owning element.
- Otherwise, use the style property’s default value.

Diagrams

- Figure 9.2 on page 16 (DI Package)

Annex A: UML Class Diagram Definition Example

(Informative)

This annex provides an example of using the DD specification to define a small subset of the UML class diagram. Sub clause A.1 gives the UML DI metamodel as an extension of the DI metamodel. Sub clause 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*.

Issue: 16628 – DiagramElements cannot represent multiple model Elements

Note – When DI is used for metamodels other than UML's, the modelElement property can be redefined even though the

metaclasses are not explicit subclasses of UML::Element. This is due to the semantics of sub 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):

1. pattern (a): a shape that has a label and an optional list of compartments, each of which having an optional list of other labels (e.g., the classifier box notation).
2. pattern (b): a shape that has a label only (e.g., the interface ball notation).
3. 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).

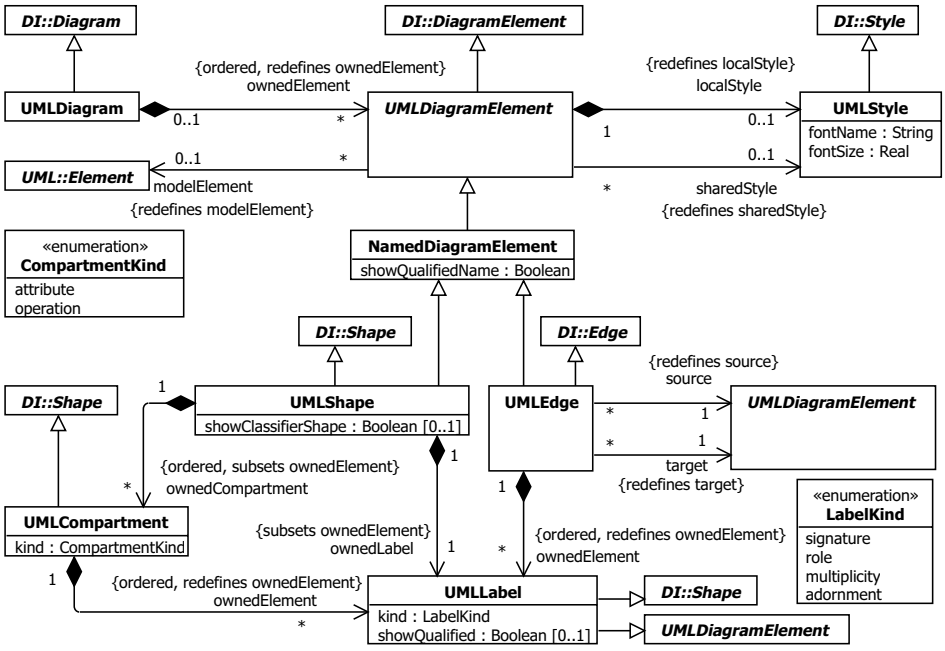

Figure A.1 - UML DI Metamodel

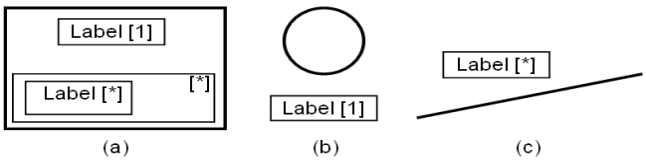

Figure A.2 - UML DI Notational Patterns

A.2 Mapping UML DI to DG using QVT Operational

Recall from Clause 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 example 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 (Sub clause 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