Flatland Draw Engine Domain

Author

Leon Starr

Model document organization

The Flatland models belong to a single Draw Engine Domain representing a coherent and self contained subject matter. For document management purposes, this domain is spread out over several interconnected Subsystems. All Subsystems are at the same level of abstraction and freely reference one another via interconnecting class model relationships and imported classes (dashed classes on the class diagrams).

The first section of this document describes the types (data types) used throughout the domain. Subsequent sections describe the classes, attributes and relationships of each subsystem.

License / Copyright

Types

Count

A positive integer > 0 representing an discrete unitless quantity.

Position

(Descriptions to be added later. For now see attribute descriptions.)

Distance

A positive integer of typesetting points (72/inch)

Description

Connection Geometry

Stroke Style

Plus Minus

Root Vine

Face Placement

Name

Adjacent Layer Last

Rect Size

Degrees

Boolean

Hollow Solid Open

Ordinal

Padding

Alignment

Orientation

Text Style

Count

Rounding

Text

Diagram Coordinates

Binary Connector Subsystem

This subsystem describes the anatomy of a Binary Connector which connects from a position on one Node face to a different position on the same or another Node face. This includes the use of Tertiary Stems which branch off from a Binary Connector to connect a third Node face position forming a T-shaped line. Binary Connectors may form a straight line between Node faces or be bent at one or more corners.

Relationship numbering range: R100-R149

Class Descriptions

Anchored Binary Stem

This is an Anchored Stem that is one of the opposing (non-Tertiary) Stems in a Binary Connector.

Attributes

ID

Same as**Binary Stem.ID**

Connector

Same as**Binary Stem.Connector**

Identifiers

1. **ID** + **Connector**

Bend

A Bend is the line drawn between two Corners in a Bending Binary Connector. One is drawn for each user specified Path.

Attributes

T location

Type: Same as **Corner.Location**

P location

Type: Same as **Corner.Location**

Path

Type: Same as **Path.Sequence**

Connector

Same as **Path.Connector** and **Corner.Connector**

Identifiers

1. **T location** No two Bends can share the same Corner
2. **P location** Same reasoning as #1
3. **Path** + **Connector**

Bending Binary Connector

This is a Binary Connector that must turn one or more corners to connect its opposing Binary Stems. In such a case the two Binary Stems will be counterparts and we can arbitrarily start drawing a line from one of the Counterpart Binary Stems to the other. In fact, we could start from both ends and work toward the middle or start from the middle and work our way out. So the terms “start” and “end” could just as easily have been labeled “A” and “B”.

Attributes

ID

Same as **Binary Connector.ID** and, for each of the two Counter Part Binary Stems, **Counterpart Binary Stem.Connector**

Start stem

Same as **Counterpart Binary Stem.ID**

End stem

Same as **Counterpart Binary Stem.ID** and not the same as the**Start stem** value

Identifiers

1. **ID**

Binary Connector

The defining property of a Binary Connector is that it connects two points, each on some Node face. Common examples are a transition from one state to another on a state diagram or an association between two classes on a class diagram.

While each Binary Stem must be in a unique position (Stems never overlap) both Binary Stems may be on the same Node or even on the same Node face in a Binary Connector. For example, a state may transition to itself or a class may be associated with itself via a reflexive association.

A Binary Connector may also include a Tertiary Stem which attaches to a third Node face position and then extends in a straight line to some point on the line connecting the two Binary Stems. Since the Tertiary Stem is a straight line, it cannot attach to the same Node as either of the Binary Stems in the Binary Connector. So the Tertiary Stem will be attached to the face of a Node that has neither of the Binary Stems attached.

At present, the only known example of a Tertiary Stem's usage is to represent an association class relationship on a class diagram.

Attributes

ID

Same as **Connector.ID**

Identifiers

ID

Binary Stem

This is a Stem that is one of the opposing (non-Tertiary) Stems in a Binary Connector. It's position may be specified by the user as an anchor point, or computed in the case of a Floating Binary Stem.

Attributes

ID

Same as**Binary Stem.ID**. Also the union of the ID values in each subclass.

Connector

Same as**Binary Stem.Connector** Also the union of the Connector values in each subclass.

Identifiers

1. **ID** + **Connector**

Corner

This is a point on the Canvas where two lines of a Bending Binary Connector meet at a right angle. Corners are not specified by the user, they are computed from user specified anchor positions and Paths.

Attributes

Location

The computed Canvas x and y coordinate of the Corner

Type: Position

Connector

Same as **Bending Binary Connector.ID**

Identifiers

1. **Location** (No two Corners may overlap)
2. **Location** + **Connector** (super identifier to support R111)

Counterpart Binary Stem

When a Binary Connector bends at least once the user must specify an anchor position for each Binary Stem. Since this means that we must have a pair of Anchored Binary Stems, we can think of these as required counterparts within such a Binary Connector.

Attributes

ID

Same as**Anchored Binary Stem.ID**

Connector

Same as**Anchored Binary Stem.Connector**

Identifiers

1. **ID** + **Connector**

Floating Binary Stem

The point where this Stem meets a Node Face is determined by drawing a straight line across from a Projecting Binary Stem in a Binary Connector. It “floats” because because the face position is computed rather than being specified by the user.

Attributes

ID

Same as**Binary Stem.ID**

Connector

Same as**Binary Stem.Connector**

Identifiers

1. **ID** + **Connector**

Lane

The corridor formed by either a Row or Column in the Grid. For the purpose of drawing a line as part of a Connector, Rows and Columns are regarded similarly.

Attributes

Number

Type: Same as either**Column.Number or Row.Number**

Orientation

Type: Row_Column `:: [ row | column ]`

Identifiers

Number + **Direction**

Since you can have both a Row and Column with the same number in the Grid, we need the direction to distinguish them.

Path

If a Bending Binary Connector requires more than one Corner, it will be necessary for the user to specify where to place each Corner to Corner stretch. Depending on the orientation, either a Row or Column is chosen along with an alignment preference.

Attributes

Connector

Same as **Bending Binary Connector.ID**

Sequence

Paths are sequenced from the Node in the T position toward the Node in the P position.

Type: Ordinal

Lane

Type: Same as **Lane.Number**

Direction

Type: Same as **Lane.Direction**

Rut

We can imagine the Path guided a long a rut somewhere in the Lane. In a horizontal lane this could be the center, top or bottom. Finer gradations in a Lane are possible. For now only three rut positions will be available, but finer gradations should eventually be supported.

Type: Lane Placement

Identifiers

Sequence + **Connector**

Paths are numbered uniquely within each Connector

Projecting Binary Stem

This is an Anchored Binary Stem participating on one of the opposing (non-Tertiary) sides of a Binary Connector. It could be one component of a pair of opposing Anchored Binary Stems in the case where the Binary Connector bends around at least one corner or a Projecting Binary Stem that establishes the position of it opposing Floating Binary Stem

Attributes

ID

Same as**Anchored Binary Stem.ID**

Connector

Same as**Anchored Binary Stem.Connector**

Identifiers

1. **ID** + **Connector**

Straight Binary Connector

This is a Binary Connector drawn as a single straight vertical or horizontal line. Since the line is straight, only one of its Binary Stems has an anchor position specified by the user. The opposite Binary Stem will be placed where the straight line projecting from the Anchored Binary Stem intersects the target Node face. At that point a Floating Binary Stem is drawn which won't necessarily line up with any specific Face Placement position on the Node face.

The anchored Stem is called a Projecting Binary Stem with the non-anchored Stem referred to as a Floating Binary Stem.

Attributes

ID

Same as **Binary Connector.ID**, **Floating Binary Stem.Connector** and **Projecting Binary Stem.Connector**

Floating stem

Same as **Floating Binary Stem.ID**

Projecting stem

Same as **Projecting Binary Stem.ID**

Identifiers

ID

Tertiary Stem

Drawn from a Node face to the middle of a Binary Connector where the root end is on the Node and the vine end touches the Binary Connector between its two Opposing Stems.

Attributes

ID

Same as **Anchored Stem.ID**

Connector

Same as **Anchored Stem.Connector** and **Binary Connector.ID**

Identifiers

1. **ID** + **Connector**

Relationship Descriptions

R100 / 1:1

Projecting Binary Stem, establishes x or y coordinate of *one* **Floating Binary Stem**

Floating Binary Stem gets x or y coordinate from *one* **Projecting Binary Stem**

If a Binary Connector is unbent (straight) we want to specify an anchor position on one Node face and then just draw the Connector line straight across to stop on the opposite Node face. What we don't want to do is try to connect anchor to anchor since that will lead to a diagonal line if the anchors on each side aren't on the same x or y axis.

So we pair an Anchored Binary Stem which we will call the Projecting Binary Stem with a non-Anchored Binary Stem which we call the Floating Binary Stem. Thus the x or y value of the Floating Binary Stem is strictly determined by that of its paired Projecting Binary Stem anchor position. This pairing then forms a Straight Binary Connector.

Formalization

Straight Binary Connector association class

R101 / Generalization

Anchored Binary Stem is a **Projecting Binary Stem**or **Counterpart Binary Stem**

These are the two roles played by a Binary Stem that has a user specified anchor position. In the projecting case, the Stem serves to establish the x or y coordinate of a line shared by a corresponding Floating Binary Stem. In the counterpart case, two Anchored Binary Stems are the terminating points of a line bent at least once.

Formalization

The identifier in each of the subclasses referring to the superclass identifier.

R102 / 1:M

Bending Binary Connector turns at right angle on *one or many* **Corner**

Corner is a right angle turn of *one* **Bending Binary Connector**

By definition, a there is at least one right angle turn in a Binary Bending Connector and hence, one Corner.

Formalization

Referential attribute in the Corner class

R103 / Generalization

Binary Connector is a **Straight Binary Connector**or **Bending Binary Connector**

A Straight Binary Connector is a single horizontal or vertical line that connects both of its non-tertiary Stems. Only one of the two non-tertiary Stems is an Anchored Stem since the opposing Stem is positioned based on the intersection of the connector line and the opposing Node face. Thus the face position of only one Stem, the Anchored Stem, need be specified by the user.

A Bending Binary Connector has at least one corner and requires all of its Stems to be Anchored Stems (fixed on user specified node face positions).

Formalization

The identifier in each of the subclasses referring to the superclass identifier

R104 / 1:1

Counterpart Binary Stem, starts line toward *one* **Counterpart Binary Stem**

Counterpart Binary Stem ends line from *one* **Counterpart Binary Stem**

In a Bending Binary Connector, a line is drawn between two Counterpart Binary Stems. The terms “start” and “end” are used to establish an arbitrary ordering of the Bends so that we can refer to a next or previous Bend while copumting and drawing the lines.

Formalization

Bending Binary Connector association class

R105 / 1:M

Bending Binary Connector turns at right angle on *one or many* **Bend**

Bend is a right angle turn of *one* **Bending Binary Connector**

The line forming a Bending Binary Connector must, by definition, bend at least once. At each Bend the line turns 90 degrees in either direction and proceeds to the end Stem or to the next Bend.

We would like most of the connector lines in our model diagrams to be straight as much as possible. This can be achieved more easily for non-Binary Connectors. For now at least bending is only supported for Binary Connectors and, therefore, a Bend can only exist as part of a Binary Connector.

Formalization

Referential attributes in the Bend class

R107 / 1:Mc

Bending Binary Connector takes *zero, one or many* **Path**

Path is taken by *one* **Bending Binary Connector**

In the simplest and most common case, a Bending Binary Connector turns at only one point forming a single Corner. In this case there is no need for the user to specify a Path as the anchor positions on each Stem establish the single Corner location. You just find the intersection of the lines projecting from each Stem.

When more than one Corner is desired, the user must choose where to place each pair of Corners. Consider two Nodes in the same Row where the Connector will be drawn between the top face of the T node to the top face of the P node. Each Corner will lie somewhere above each Node on the same y coordinate, since we want a straight line. But where is the y coordinate? One Row above? Two Rows above? It's up the the user to decide.

A Path represents both the choice of a Row or Column (Lane) and the alignment within that Lane.

The number of Paths that must be specified is equal to one less than the number of desired Corners in the Bending Binary Connector. So, zero in the case of a single Corner as previously discussed and then incrementing from there.

A Path is defined specific to its Bending Binary Connector. A variety of constraints will prevent two Paths from different Connectors from overlapping, such as the enforcing the uniqueness of Corner coordinates.

Formalization

Referential attributes in the Path class

R109 / Generalization

Binary Stem is a **Floating Binary Stem**or **Anchored Binary Stem**

A Stem used in a Binary Connector may or may not be anchored (user specified). In the case of a Straight Binary Connector, one will be anchored with the other left floating (derived from the anchored Stem). With a Bending Binary Connector each Stem must be anchored.

Formalization

The identifier in each of the subclasses referring to the superclass identifier. Also the superclass identifier is defined as the union of the corresponding identifiers in each of the subclasses.

R110 / 1:1c

Tertiary Stem connects to the middle of *one* **Binary Connector**

Binary Connector connects with *zero or one* **Tertiary Stem**

A third Node face position may be connected into a Binary Connector, effectively making it tertiary. Rather than define a new kind of Connector we just say that a Tertiary Stem may or may not latch onto the middle of any given Binary Connector. This is because the properties of a Binary Connector, bent or straight, are not affected by the existence of any optional third Stem.

When we say “middle of” we mean anywhere between the vine ends of the Binary Connector’s two Binary Stems.

Formalization

Referential attribute in the Tertiary Stem class

R111 / 1c:1c

Corner is toward the P/T anchor of *zero or one* **Corner**

When there are more than two Corners in a Bending Binary Connector the Corners are connected in sequence proceeding from the T node to the P node. This establishes an arbitrary but consistent sequence for the purpose of determining how all of the Bends are interconnected. So if we know which Node is designated as T, we can proceed from one Path to the next filling in Bends to get to the P node.

For a Bending Binary Connector with only one Corner there are no Bends.

Formalization

Referential attributes in the Bend association class

R112 / 1:1

Bend is drawn along *one* **Path**

Path establishes line of *one* **Bend**

Whereas a Path is a user specified request for the placement of a line, a Bend is the actual line computed between two Corners.

For each Path specified by the user it is necessary to compute the corresponding Bend so that it can be drawn.

Formalization

Referential attribute in the Bend class

Tree Connector Subsystem

This subsystem describes the anatomy of a Tree Connector which connects from a position on one trunk Node face to a different position on one or more other branch Node faces. It can be used to express a generalization relationship on a class diagram. But there are surely other uses for this type of Connector on other Diagram Types.

Relationship numbering range: R151-R199

Class Descriptions

Anchored Tree Stem

Any Stem within a Tree Connector attached to a user specified anchor position is an Anchored Tree Stem.

Attributes

ID

Same as**Anchored Stem.ID** and **Tree Stem.ID**

Connector

Same as **Anchored Stem.Connector**. **Branch.Connector** and **Tree Stem.Connector**

Branch

Same as **Branch.ID**

Identifiers

1. **ID** + **Connector**

Branch Path

If the placement of a Branch can not be unambiguously computed by the specified grafts or Node face placement on the Diagram, the user must specify a Path aligned in some Lane. This user supplied information is a Branch Path.

Attributes

ID

Same as**Path.ID**

Connector

Same as **Path.Connector**

Identifiers

1. **ID** + **Connector**

Leaf Stem

Each Node participating in a leaf role within a Tree Connector attaches to the Connector via a Leaf Stem. This is generally an Anchored Stem, unless the Leaf Stem does not attach at a right angle to its Branch.

Attributes

ID

Same as**Anchored Branch Stem.ID** or **Floating Leaf Stem.ID**

Connector

Same as **Tree Connector.ID** and also same as**Anchored Branch Stem.Connector** or **Floating Leaf Stem.-Connector**

Identifiers

1. **ID** + **Connector**

Tree Connector

A Tree Connector connects a Node in a trunk role to one or more Nodes each in a leaf role in a hierarchical structure. It can be used to draw a generalization relationship on a class diagram, for example.

Attributes

ID

Same as**Connector.ID**

Identifiers

1. **ID**

Trunk Stem

Every tree connector pattern connects a single Node playing the role of a trunk with one or more other Nodes playing a leaf role. The Trunk Stem is an Anchored Tree Stem attached to the trunk Node.

Attributes

ID

Same as**Anchored Tree Stem.ID**

Connector

Same as **Tree Connector.ID** and **Anchored Tree Stem.Connector**

Identifiers

1. **ID** + **Connector**

Relationship Descriptions

OR161 / Ordinal

Branch bends corner at

In a Tree Connector with multiple Branches, each Branch is sequenced to establish adjacency. It must be possible, given a single Branch to move in either direction and find the adjacent Branch which must be oriented at a right angle. Starting from a Branch that originates at some Anchored Tree Stem, attached collinear or at a right angle, the Branch either terminates the Tree Connector at some other Tree Stem (anchored or floating) or it terminates at a corner which bends to form an adjacent Branch. This sequence continues until the final Branch terminates.

This ordering is important because it defines where the corner is located between two adjacent Branches.

Formalization

ID is an ordinal sequenced within a **Connector**

R151 / 1:1

Tree Connector is rooted in *one* **Trunk Stem**

Trunk Stem is root of *one* **Tree Connector**

We can visualize a hierarchical or tree connector pattern as originating in a trunk that extends out to one or more Branches which sprout one or more Leaf Stems. Here we establish that the Tree Connector must originate in a single Trunk Stem.

In the case of a class diagram generalization relationship, for example, the Trunk Stem would extend from the relationship's superclass.

Formalization

Referential attribute in the Trunk Stem class

R152 / 1:M

Tree Connector radiates out to *one or many* **Leaf Stem**

Leaf Stem sprouts from *one* **Tree Connector**

While there is only one Trunk Stem in a Tree Connector, there may be one or more Leaf Stems to form a hierarchical pattern. By policy, the pattern does not support a leaf-less tree.

In the example of a class diagram generalization relationship, each subclass would extend a Leaf Stem to attach to a Branch in the Tree Connector.

Formalization

Referential attribute in the Leaf Stem class

R153 / 1:M

Rut Branch follows *one* **Branch Path**

Branch Path guides *one* **Rut Branch**

The user can specify a Branch Path which establishes a Lane and a Rut where a Rut Branch is drawn. Only one Rut Branch may occupy the same Rut to avoid coincident or overlapping connector lines.

Formalization

Referential attribute in the Rut Branch class

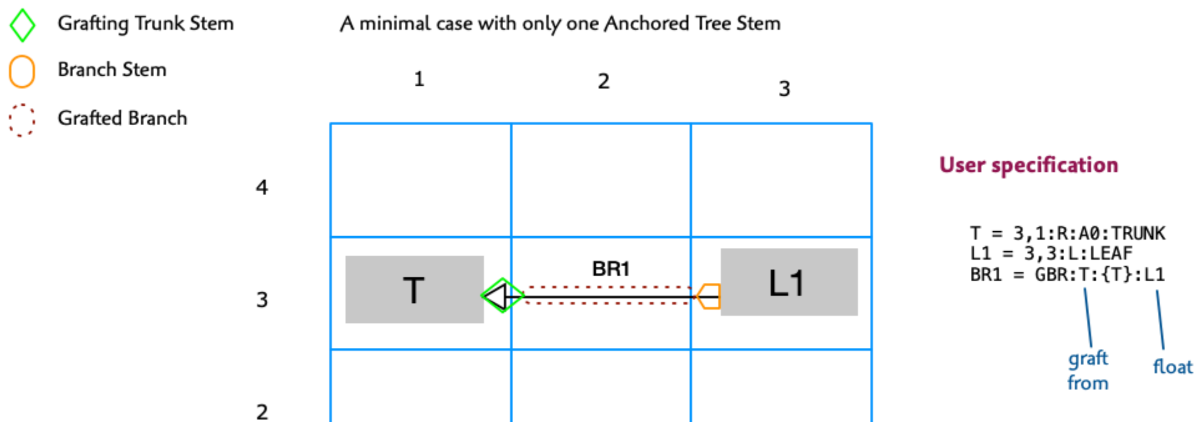
R154 / 1:M

Anchored Tree Stem hangs from *one* **Branch**

Branch hangs *one or many* **Anchored Tree Stem**

Every Branch has to connect at least one Anchored Tree Stem. In the minimal case (shown) this could be a Trunk Stem that grafts a Grafted Branch leading to a Floating Leaf Stem on the opposite side.

Pattern 7



Every Anchored Tree Stem must attach to a Branch at some point. This is either at a right angle to the Branch in which case the stem is hanging or it is in line with the Branch in which case the Branch is grafted from the Anchored Tree Stem.

Formalization

Referential attribute in the Anchored Tree Stem class

R155 / Generalization

Path is a **Branch** or **Binary Path**

At present there are two kinds of line segments that can be guided down a Row or Column by a Rut. In each case, a part of a bending Connector is proceeding in a straight line as guided by a Path specified by the User.

Since the rules for bending are specific to each Connector geometry, it is necessary to distinguish each type of Path.

Formalization

Referential attributes in the subclasses

R156 / 1:1c

Floating Leaf Stem is positioned by *one* **Grafted Branch**

Grafted Branch positions *zero or one* **Floating Leaf Stem**

Once a Grated Branch is established by an Anchored Tree Stem, it proceeds in a straight line and ends in one of three cases. It can end at a final Anchored Tree Stem hanging at a right angle to the Grafted Branch. If there is an adjacent Branch, it proceeds to where it meets that Branch at a right angle. In this case the Tree Connector is bending around a corner. In the third case, the Grafted Branch line meets the **Vine end** of a Floating Leaf Stem. As is the case with all Floating Stems, the user does not specify an anchor point on the Stem's Node face. The point on the Node face is determined by projecting a line from the Grafted Branch to the face.

Formalization

Referential attribute in the Floating Leaf Stem class

R157 / 1:1c

Anchored Tree Stem establishes axis of *zero or one* **Grafted Branch**

Grafted Branch is a collinear extension of *one* **Anchored Tree Stem**

The line segment of a Branch can be defined by starting at the **Vine end** of an Anchored Tree Stem and extending on the same axis toward one or more other attached Tree Stems. If the Branch is defined in this manner it is a Grafted Branch.

By definition, a Grafted Branch extends out from some Anchored Tree Stem.

Most Anchored Tree Stems will not define a Grafted Branch and instead simply hang at a right angle from some Branch which may or may not be a Grafted Branch.

Formalization

Referential attribute in the Grafted Branch class

R158 / Generalization

Tree Stem is an **Anchored Tree Stem**or **Floating Leaf Stem**

Every Stem within a Tree Connector is either anchored or floating. The utility of this abstraction is not immediately clear. It is nonetheless true.

Formalization

The union of the subclass identifiers in the superclass as well as the referential attributes in each subclass.

R162 / Generalization

Branch is a **Grafted**, **Interpolated** or **Rut Branch**

There are three ways to determine the placement of a Branch. In the case of a Rut Branch the user specified a Path which establishes a Lane and a Rut. An Interpolated Branch is placed at the halfway point in between opposing Node faces. This is determined by taking all of the faces hanging in the Rut Branch, finding the two closest opposing faces and then identifying the halfway point between them. Finally, a Grafted Branch is collinear with a user specified Anchor Tree Stem.

Formalization

Referential attributes in the subclasses

R163 / Generalization

Anchored Tree Stem is a **Trunk Stem** or **Anchored Leaf Stem**

Every Anchored Stem in a Tree Connector attaches a Node in the trunk role (via its Trunk Stem) or in the leaf role (via its Anchored Leaf Stem).

Formalization

Referential attributes in the subclasses

R164 / Generalization

Leaf Stem is a **Floating Leaf Stem** or **Anchored Leaf Stem**

If a Leaf Stem is anchored to its Node it is either hanging at a right angle to its Branch or defining a grafting point from which its Grafted Branch is extended. It is also possible that a Leaf Stem is not anchored, but instead floats to be collinear with its Branch.

Formalization

Referential attributes in the subclasses

Node Subsystem

This subsystem considers the Canvas and Diagram as a whole, the grid system for Node placement, the Notation applied to the Diagram and the various types of Nodes that may be placed on the Diagram. Connectors are modeled in a different subsystem.

Relationship numbering range: R1-R49

Class Descriptions

Diagram Layout Specification

Defines a set of values that determine how a Diagram and Grid is positioned on a Canvas and how Nodes are positioned relative to the Diagram and Grid.

Attributes

Name

In this version there is assumed to be only a single specification instance, so the name is here merely expresses unique model identity.

Type: Name

Default margin

The distance from each canvas edge that may not be occupied by the Diagram.

Type: Padding

Default diagram origin

The lower left corner of the Diagram in Canvas coordinates.

Type: Position

Default cell padding

The distance from each Cell edge inward that may not be occupied by any Node. This prevents two Nodes in adjacent Cells from being too close together.

Type: Padding

Default cell alignment

The horizontal and vertical alignment of a Node in its Cell or Cells

Type: Padding

Identifiers

Name

Diagram Notation

A Notation supported by the Flatland draw engine to render Diagrams of a given Diagram Type. See R32 for more details.

Attributes

Diagram type

Same as **Diagram Type.Name**

Notation

Same as **Notation.Name**

Identifiers

Diagram type + **Notation**

Consequence of a many-many association

Diagram Type

A standard diagram such as ‘class diagram’, ‘state machine diagram’ or ‘collaboration diagram’. Each of these types draws certain kinds of Nodes and Connectors supported by one or more standard Notations.

Attributes

Name

A commonly recognized name such as ‘class diagram’, ‘state machine diagram’, etc.

Type: Name

Identifiers

Name

Node

On Diagrams, model entity semantics such as states, classes, subsystems and so forth can be symbolically represented as polygonal or rounded shapes. These shapes can then be connected together with lines representing model relationship semantics. A Node represents the placement of a shape symbol at a specific location (Cell) on a Diagram.

Every Node, regardless of its specific shape as determined by its Node Type, is considered to be roughly or completely rectangular. This means that every Node has four faces, top, bottom, left and right where one or more Connectors may attach.

Attributes

ID

Each Node is numbered uniquely on its Diagram.

Type: Nominal

Node type

Type: Same as **Node Type.Name**

Diagram type

Type: Same as **Node Type.Diagram type**

Size (derived)

The height and width of the Node. This height is derived from the combined heights of its visible Compartments. The width is determined as a result of computing the required width of all of the visible Compartments.

Type: Rect Size

Location

The lower left corner of the Node relative to the Diagram.

Type: Diagram Coordinates

Identifiers

ID

We only handle one Diagram at a time, so the Node.ID is always unique.

Node Type

Specifies characteristics common to all Nodes of a given type. A class node, for example, has three compartments, sharp corners a certain border style, etc. For now, to support a different visual style for a class node, let's say, you would need to define a new node/diagram type combination (UML class on a UML class diagram type vs. Shlaer-Mellor class on a Shlaer-Mellor class diagram type), for example). Since, most diagrams we are considering have notational variation in the Connector Types and not the Node Types, we're baking in the visual characteristics of a Node Type for now and making it flexible for Connector Types.

Attributes

Name

A name like "class", "state", "imported class", "domain", etc.

Type: Name

Diagram type

Type: Same as **Diagram Type.Name**

Rounded

Whether or not all four node corners are rounded

Type: Boolean

Compartments

The number of UML style text compartments visible.

Type: Count1 `:: integer > 0`

Border

Type: Border style

Default size

Initial assumption about a Node size.

Type: Rect Size

Max Size

Node may not be drawn larger than this size.

Type: Rect Size

Corner margin

The minimum distance permitted between a Stem Root end and the nearest Node corner. The intention it to prevent lines attaching on or very close to a Node's corner which looks glitchy.

Type: Distance

Identifiers

Name + **Diagram type**

Notation

A standard (supported by a large or small community) set of symbols used for drawing a Diagram Type.

Attributes

Name

A name such as ‘xUML’, ‘UML’, ‘Starr’, ‘Shlaer-Mellor’, etc.

Type: Name

Identifiers

Name

Relationship Descriptions

R30 / 1:1c

Diagram is rendered using *one* **Diagram Notation**

Diagram Notation renders *zero or one* **Diagram**

When a Diagram is created, there may be a choice of multiple Notations that it can be displayed in. A class diagram, for example, could be displayed as Starr, xUML or Shlaer-Mellor notation. Each potential Diagram would mean the same thing, but the drawn notation would be different in each case.

A Diagram can use only a Notation that is defined for its Diagram Type. Since a Diagram Type must be supported by at least one Notation, there will always be at least one possible choice.

Only one Diagram is rendered at a time. This means that while, in theory, the same Diagram Notation could render multiple Diagrams and certainly does over time, during the runtime of the draw engine, a given Diagram Notation either is or isn't the one that determines the look of a Diagram, thus the 1c multiplicity in this association.

Formalization

Diagram.Notation -> Diagram.Notation.Notation and Diagram.Type -> Diagram Notation.Notation and Diagram Type.Name

The shared Diagram.Type value enforces the constraint that a Diagram's notation must be supported by its specified Diagram Type on R11.

R7 / 1:Mc

Compartment is filled by *zero one or many* **Text Line**

Text Line fills *one* **Compartment**

.

Formalization

Reference in Text Line class.

R8 / 1:Mc

Compartment is a **Title** or **Data Compartment**

.

Formalization

Subclass references to superclass.

R1 / Ordinal

Compartment Type is stacked above

Vertical stacking of corresponding Compartment Types of a Node Type. For example a title compartment is drawn above an attribute compartment which is drawn above a method compartment in a class diagram.

Formalization

Compartment Type identifier I2 with **Stack order**, numbered within **Node type** and **Diagram type**

R32 / M:Mc-1

Diagram Type is supported by *one or many* **Notation**

Notation supports *zero, one or many* **Diagram Type**

The term 'supports' should not be confused with 'compatible'.

Compatibility means that a Notation has been defined, in the real world, to be used with a certain kind of diagram. Support means that the Flatland draw engine currently has the ability to draw a particular Diagram Type in a specified Notation.

Here we assume that compatibility is understood when this relationship is populated and that a given Notation is associated only with those Diagram Types where it makes sense to use it.

For example, the xUML notation is relevant to a wide variety of diagram types, but for now it may only be supported for class diagrams and state machine diagrams. On the other hand, the Starr notation applies only to class diagrams.

So this relationship represents which Notations have been selected to support certain Diagram Types supported by the Flatland drawing tool.

So that they can be drawn, it is essential to ensure that at least one compatible Notation is supported for each Diagram Type defined in the Flatland draw engine.

Formalization

Diagram Notation association class

R11 / 1:1c

Diagram Type specifies *zero or one* **Diagram**

Diagram is specified by *exactly one* **Diagram Type**

A Diagram Type embodies a diagramming standard and so constrains a Diagram to be drawn a certain way, with certain types of Nodes and Connectors. The associated Notation further constrains the drawn look of these elements.

A Connector Type, say a binary association which has meaning in a class diagram won't be available in a state machine diagram, for example.

Therefore, a Diagram is always specified by a single Diagram Type. A given Diagram Type may, or may not be the Diagram Type employed to constrain the currently managed Diagram.

Formalization

Diagram.Type

Sheet Subsystem

This subsystem considers the placement of graphics, title blocks and text fields on the sheet underlying a model diagram. This makes it possible to display metadata such as title, copyright, date, version etc. All of this is in the Sheet Subsystem since the placement and sizing of these elements typically varies by sheet size.

Relationship numbering range: R300-R350

Class Descriptions

Box

A bounded rectangle forming part of a Title Block Pattern.

Attributes

ID

Arbitrary identifying value, local to a Title Block Pattern.

Type: Based on Nominal

Identifiers

1. **ID** + **Pattern**

ID value is unique within each Pattern

Box Placement

When a Title Block Pattern is scaled and then positioned within a Frame it becomes possible to compute the placement of each of its Boxes.

Attributes

Placement

The location of the lower left corner of the Box in Canvas coordinates.

Type: Position

Box size

The height and width of the Box

Type: Rect Size

Identifiers

1. **Frame** + **Sheet** + **Box** + **Pattern**

Many to many identifier

Boxed Field

A Boxed Field is placed inside a Data Box within a Title Block Pattern.

Attributes

Order

When there is more than one unit of Metadata present in the same Data Box, the items are stacked vertically. If there is only one Boxed Field in a Data Box, that field has an order value of 1. If there is more than one item, the bottommost item is 1 with ascending values going vertically.

Identifiers

1. **Metadata** + **Frame** + **Sheet**

   Super class reference

Boxed Text Line

Each unit of Metadata is displayed as a single line of text in a Data Box. These lines are ordered vertically if there is more than one Metadata item sharing the same Data Box.

So a Boxed Text Line is the entry of Metadata content in some line number of a Data Box.

Identifiers

1. **Metadata** + **Box** + **Title block pattern**

A Metadata item can appear at most once in a Data Box

2. **Box** + **Title block pattern** + **Order**

The numbering of text lines is local to each Data Box

Attributes

Order

When there is more than one unit of Metadata present in the same Data Box, the items are stacked vertically. If there is only one Boxed Field in a Data Box, that field has an order value of 1. If there is more than one item, the bottommost item is 1 with ascending values going vertically.

Compartment Box

A Compartment Box is split in two with a Partition yielding two Partitioned Boxes. So a Compartment Box is a rectangular border wrapping two internal Boxes.

Attributes

(No non-referential attributes)

Identifiers

1. **ID** + **Pattern**

Forms the union of the subclass identifier values

Data Box

A Box that is not further partitioned, but is instead intended to wrap presented Meta Data is a Data Box.

Attributes

Alignment

The vertical and horizontal alignment of text content displayed in this Data Box.

Type: VH Align `:: ([ T | C | B ], [L | C | R])`

Identifiers

1. **ID** + **Pattern**

Refers to each superclass identifier

Envelope Box

A Title Block Pattern is wrapped by a single Envelope Box. So it constitutes the outer boundary of the entire Title Block Pattern.

Attributes

(No non-referential attributes)

Identifiers

1. **ID** + **Pattern**

Refers to each superclass identifier

Field

A location within a Frame where an item of Metadata is placed.

Attributes

Type

The location in Canvas coordinates of the lower left corner where the Meta Data will be rendered.

Type: Open Box `:: [ Open | Block ]`

Max area

The maximum rectangular dimension to be used for rendering the Metadata.

Type: Rect Size

Identifiers

1. **Metadata** + **Frame** + **Sheet** + **Type**

Since the same unit of Metadata could be displayed both in a Boxed and an Open Field, we need the **Type** attribute to distinguish the two cases.

Field Content

This is data supplied with a Canvas that is entered into a Field.

Attributes

Value

This is the data supplied with the Canvas that goes into the Field.

Type: Text

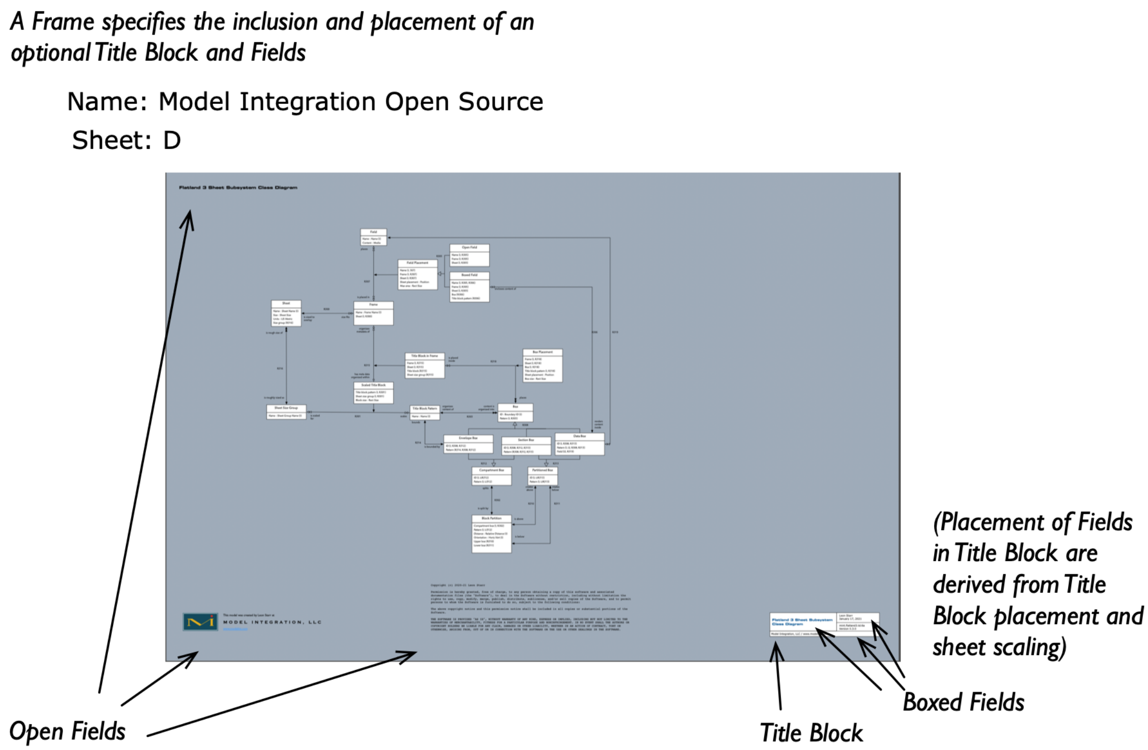
Identifiers

1. **Metadata** + **Location** + **Frame** + **Sheet**

Identifier is taken from the many side of the association

Frame

On any serious project it is not adequate to generate model diagrams absent any meta data such as authors, dates, revision numbers, copyright notices, organization logos and so forth.



A Frame represents a pattern of Fields and/or a Title Block Pattern on the surface area defined by a Sheet. The lower left corner placements of each Frame element (Field or Scaled Title Block) are customized to fit the dimensions of a given Sheet.

Attributes

Name

Describes the usage or organization affiliation, `Model Integration Open Source` or `Model Integration Proprietary`, as examples.

Type: Frame Name based on system type Name

Identifiers

1. **Name** + **Sheet**

Since a Frame is custom fit to a Sheet we will refer to a Frame using the Sheet Name, `D-Model Integration Open Source`, for example.

Frame Metadata

Metadata that is displayed in either an Open Field or a Block Field or both.

Identifier

1. **Metadata** + **Frame** + **Sheet**

Attributes

(No non-referential attributes)

Metacontent

The user supplies a value for each relevant Metadata item as Metacontent. If you think of Metadata as a variable, then Metacontent is the value assigned to that variable.

Identifiers

1. **Metadata**

Attributes

No non-referential attributes

Metadata

Information that can be displayed in the Frame of a model diagram other than the model itself.

Attributes

Name

A name that reflects the content to be inserted in a Frame such as `Modification date`, `Author`, `Organization Logo`.

Type: Field Name based on Name system type.

Media

They kind of data to be inserted in the Field. Since the underlying graphics library will handle various image formats, we need not specify them here. We really just need to know what rendering facilities to use.

Type: `[ text | image ]`

Identifiers

1. **Name**

Open Field

An Open Field is not part of a Title Block and can be placed anywhere in a Frame.

Identifiers

1. **Metadata** + **Frame** + **Sheet**

Refers to Superclass identifier

Attributes

Placement

The location, in Canvas Coordinates, within the Frame of the lower left corner of the Open Field.

Type: Position

Partition

A Compartment Box is divided horizontally or vertically to yield two Partitioned Boxes.

Attributes

Distance

A fraction of the distance moving upward or rightward. For example, a Compartment Box split at `0.33` yields two Partitioned Boxes with the lower being one third the area of the upper Box. So the distance is always measured away from the lowest boundary coordinate value either vertically or horizontally.

Orientation

This is the direction of the partition axis.

Type: `[ horizontal | vertical ]`

Identifiers

1. **ID** + **Pattern**

Forms the union of the subclass identifier values

Partitioned Box

A Partitioned Box results from partitioning a Compartment Box as either the upper (rightmost or uppermost) box or lower (leftmost or lowermost) Box.

Attributes

(No non-referential attributes)

Identifiers

1. **ID** + **Pattern**

Forms the union of the subclass identifier values

Resource

The name of an image that can be located somewhere.

Identifiers

1. **Metadata**

Attributes

Name

A descriptive name to be supplied with the diagram semantics that can be mapped to a file somewhere.

Type: Text

Location

A resource locator such as a file path or URL. (Currently only file name's are supported)

Type: Resource Locator (file path, for now)

Scaled Title Block

The dimensions of a Title Block and all of the Boxes it contains can be determined for a given Sheet Size Group. A Title Block Pattern is defined as a set of relative Box sizes. The actual Box sizes are determined by ratios relative to the Envelope Box. The size of the Envelope Box is determined by choosing a size that works for a given Sheet Size Group and saving it as a Scaled Title Block.

Attributes

Size

The height and width of the Envelope Box for the given Sheet Size Group.

Type: Rect Size

Margin

The distance from the horizontal and vertical box boundary to any text content that must remain clear for all boxes in the Title Block Pattern (for the current scaling). This spacing is necessary for aesthetics and readability of the text content.

Type: HV Spacing `:: (H::distance, V::distance)`

Identifiers

1. **Title block pattern** + **Sheet size group**

Section Box

A rectangle inside the Envelope that is further partitioned is a Section Box.

Attributes

(No non-referential attributes)

Identifiers

1. **ID** + **Pattern**

Refers to each superclass identifier

Sheet

A pre-defined size representing the total rectangular surface area available for drawing is considered a Sheet. In the engineering world standard sheet sizes are named such as A, B, C D and E in the US or A0, A1, etc internationally. Each size constitutes a possible instance of Sheet. Many drawing programs provide the feature of an arbitrary sized sheet that expands as you draw. Flatland, however, encourages, but does not enforce, adherence to a standard sheet size so that you have the option of creating a properly scaled drawing or plot on actual paper if you have the printing equipment. Flatland also encourages you to divide complex domains into appropriately sized subsystems rather than just creating one monolithic tangle of model elements. That said, any size of Sheet can be pre-defined and made available for drawing in Flatland.

Attributes

Name

A US name like letter, tabloid, B, C, D or international name like A4, A3, etc. Or even a user defined name like 'extra wide'

Type: Sheet Name based on the Name system type

Size

Type: Sheet Size, a height and width value as unit-less positive rational numbers

Units

The units are specified so that the Size values can be converted into the correct internal drawing units (probably points).

Type: `[ US | Metric ]`

Identifiers

1. **Name**

Sheet Size Group

It is not really necessary to customize a title block for each sheet size. A title block sized to fit in a Tabloid (11inx17in) Sheet is equally adequate to an A3 Sheet. So it is convenient to group these two sizes for the purpose of determining which Scaled Title Block Pattern to use. It might be possible to work out a scaling formula that would perfectly size a Title Block Pattern given Sheet dimensions, but this formula is not obvious. It certainly is not a linear scale. (If you take the Title Block Pattern to Sheet size ration for Letter, and scale that up for an A1 Sheet, it doesn't look right). Rather than try to devise a formula, we can just scale each Title Block Pattern to fit a given Sheet Size Group which will be less work than doing the same for every Sheet size, US and International.

Attributes

Name

A descriptive name like 'Dish' or 'LetterA4'. Or maybe use a 'Tshirt size' system like 'small', 'medium', etc.

Type: Sheet Group Name based on the system Name data type

Identifiers

1. **Name**

Text content

Content such as an author name, date, document title and so forth is defined as text and rendered into any Field that designates the associated Metadata item.

Identifiers

1. **Metadata**

Attributes

Value

The actual text to be displayed on the diagram wherever the Metadata is referenced.

Type: Text

Title Block Pattern

This is typically a rectangle partitioned into multiple internal rectangles where fields like “Title”, “Approved by:”, “Version” and so forth appear as in a standard engineering diagram.

Attributes

Name

A descriptive name.

Type: Name

Identifiers

1. **Name**

Title Block Placement

Whereas the sizes of each Box can be worked out knowing the Sheet Size Group, the actual coordinates of each Box lower left corner is unique to each Frame. That's because the Frame is custom fit to its Sheet.

Attributes

(No non-referential attributes)

Identifiers

1. **Frame** + **Sheet**

Identifier formed from the association class many-side (Frame)

Relationship Descriptions

OR304 / Ordinal

stacking order

Metadata items are stacked vertically within the Data Box with the lowest ordinal value in the highest vertical position.

Formalization

Boxed Field.Order

R300 / 1:Mc

Frame is sized to overlay *one* **Sheet**

Sheet size fits *zero or many* **Frame**

The dimensions and Fields of a Frame are adjusted to perfectly fit a particular Sheet (sheet size). A `D-Model Architect` frame, for example will only fit on a `D` Sheet.

A variety of Frames can be designed for any given Sheet. If no Frames are defined for a given Sheet, no title blocks or Fields will be drawn when a diagram is generated when that Sheet is selected by the user.

Formalization

Frame.Sheet -> Sheet.Name

R301 / Mc:Mc

Title Block Pattern is scaled to look good in *zero, one or many* **Sheet Size Group**

Sheet Size Group scales for appearance *zero, one or many* **Title Block Pattern**

A given Title Block Pattern can be scaled to look nicely (not take up too much space and still display contents legibly) in the Sheet sizes defined by a Sheet Size Group. For each Sheet Size Group where the Title Block Pattern might be used, it would have a different scale and all of its internal Boxes would size differently.

It is possible to define a Title Block Pattern without necessarily scaling it, though this would be unusual as it would get used. But there may be some benefit in defining a Title Block Pattern in advance of employing it.

Similarly a Sheet Size Group may exist that doesn't support any Title Block Pattern. This could be the case in a project were title blocks are not used at all.

Formalization

Scaled Title Block.Title block pattern -> Title Block Pattern.Name, Scaled Title Block.Sheet size group -> Sheet Size Group.Name

R302 / 1:1

Partition splits *one* **Compartment Box**

Compartment Box is split by *one* **Partition**

By definition, a Compartment Box is split by a single Partition to yield an upper and a lower Partitioned Box. By 'upper' and 'lower' we refer to ascending values along the cartesian axes. Upper refers to 'further right' if the Partition is vertical or 'further up' if the Partition is horizontal. In other words, 'up' means a higher coordinate value along the x or y axis.

Formalization

Partition.Compartment box -> Compartment Box.ID

Partition.Pattern -> Compartment Box.Pattern

R303 / 1:M

Title Block Pattern defines a nested rectangular hierarchy of*one or many* **Box**

Box is rectangle in nested hierarchy defined by *one* **Title Block Pattern**

A Title Block Pattern is divides an Envelope Box into multiple sub-rectangles which may be further partitioned to form a number of Data Boxes where Meta Data can be presented. At a minimum, a Title Block consists of an Envelope Box split in two to form two Data Boxes. Anything less would constitute a simple Open Field.

Formalization

Box.Pattern -> Title Block Pattern.Name

R305 / Generalization

Field is an **Open**or **Boxed Field**

Each Field is either embedded within a Title Block Pattern or scattered by itself somewhere out on the Canvas.

Formalization

<subclass>.Metadata -> Field.Metadata

<subclass>.Frame -> Field.Frame

<subclass>.Sheet -> Field.Sheet

R306 / 1:Mc

Boxed Field is enclosed as text block within *one* **Data Box**

Data Box encloses in text block *zero or many* **Boxed Field**

Each Box Field within a Data Box corresponds to distinct line in a multiline block of text. The lower left corner of this text box is determined by the Scaled Title Block spacing applied to the Data Box's Box Placement.

Since a Data Box is just a position within a Title Block Pattern, it can be referenced each time its Title Block Pattern is applied to a Frame.

Formalization

Boxed Field.Box -> Data Box.ID

Boxed Field.Title block pattern -> Data Box.Title box pattern

R307 / Mc:Mc-1

Metadata is displayed in *zero, one or many* **Frame**

Frame displays *zero, one or many* **Metadata**

When a Frame is specified Fields may be defined at various locations in the Frame, in Canvas Coordinates, where the associated Meta Data will be rendered.

In some cases the same piece of Metadata, `Title`, let's say, might be presented in more than one location for easy reference. Consequently, we cannot use the Meta Data name exclusively as the name of the field.

Fields are not required since a Frame may be empty or specify just a single Scaled Title Block.

Metadata may be defined which is never used, though this should be a rare occurrence.

Formalization

Frame Metadata.Name -> Metadata.Name

Frame Metadata.Frame -> Frame.Name

Frame Metadata.Sheet -> Frame.Sheet

R308 / Generalization

Box is an **Envelope**, **Section** or **Data Box**

A Title Block Pattern is divided up into a number of sub-rectangles. The outermost rectangle is the Envelope. A partition of this Box results in two internal Boxes. If neither is partitioned further, each becomes a Data Box where Meta Data can be presented. But if one of the boxes is further subdivided via a Partition, that enclosing Box is a Section Box which simply encloses two other Boxes either or both of which may be a Section Box or a Data Box. Sooner or later we are left with nothing but Data Boxes and the Title Block Pattern is complete. Envisioned as a binary tree, we can think of the Envelope Box as the root, intermediate nodes as Section Boxes and the leaves as Data Boxes.

Formalization

<subclass>.ID -> Box.ID

<subclass>.Pattern -> Box.Pattern

R309 / 1:1c

Metacontent is provided for *one* **Metadata**

Metadata has provided *zero or one* **Metacontent**

The user supplies content to be filled in for any reference to a given Metadata item.

Only one value may be supplied for each item of Metadata. If no value is supplied, any associated Metadata Field will be rendered blank.

Formalization

Metacontent.Metadata -> Metadata.Name

R310 / 1:1

Partition creates above *one* **Partitioned Box**

Partitioned Box is above *one* **Partition**

A Partition will always yield a rightmost or uppermost Partitioned Box.

Formalization

Partition.Upper box -> Partitioned Box.ID

Partition.Pattern -> Partitioned Box.Pattern

R311 / 1:1

Partition creates below *one* **Partitioned Box**

Partitioned Box is below *one* **Partition**

A Partition will always yield a leftmost or lowermost Partitioned Box.

Formalization

Partition.Lower box -> Partitioned Box.ID

Partition.Pattern -> Partitioned Box.Pattern

R312 / Generalization

Compartment Box is an **Envelope** or **Section Box**

Both an Envelope and a Section Box are, by definition, partitioned.

Formalization

<subclass>.ID -> Compartment Box.ID

<subclass>.Pattern -> Compartment Box.Pattern

The union of <subclass> ID + <subclass> Pattern values forms the domain of the Compartment Box.ID + Compartment Box.Pattern values.

R313 / Generalization

Partitioned Box is a **Section** or **Data Box**

A Box resulting from a Partition is either another Compartment Box, which is a Section Box or it is not further partitioned in which case it is a Data Box.

Formalization

<subclass>.ID -> Partitioned Box.ID

<subclass>.Pattern -> Partitioned Box.Pattern

The union of <subclass> ID + <subclass> Pattern values forms the domain of the Partitioned Box.ID + Partitioned Box.Pattern values.

R314 / 1:1

Title Block Pattern is bounded by *one* **Envelope Box**

Envelope Box bounds *one* **Title Block Pattern**

The rectangle that completely surrounds a Title Block is the Envelope Box.

Formalization

Envelope Box.Pattern -> Title Block Pattern.Name

R315 / 1c:Mc

Frame places *zero or one* **Scaled Title Block**

Scaled Title Block is placed in *zero, one or many* **Frame**

A Scaled Title Block is positioned within a Frame by creating a Title Block Placement where a lower left corner is specified.

A Frame can position at most one title block, typically in the lower right corner of the Canvas.

A Title Block Pattern, if it is used, will likely have a different position in each Frame.

Formalization

Title Block Placement.Frame -> Frame.Name

Title Block Placement.Sheet -> Frame.Sheet

(constrained to be a Sheet belonging to the Title Block Placement.Sheet size group)

Title Block Placement.Title block pattern -> Scaled Title Block.Title block pattern

Title Block Placement.Sheet size group -> Scaled Title Block.Sheet size group

R316 / 1:M

Sheet is roughly sized as *one* **Sheet Size Group**

Sheet Size Group roughly sizes *many* **Sheet**

Sheet Size Groups are used to determine the scaling for each available Title Block Pattern.

Roughly similar sizes such as Letter, A4 and Legal may be grouped together in the same Sheet Size Group since the same Title Block scale will work for all three sizes.

Since any Sheet must specify a scale to be used for any Title Block Patterns, each Sheet must be categorized in a Sheet Size Group.

Formalization

Sheet.Size group -> Sheet Size Group.Name

R317 / 1:1c

Frame Metadata appears in *zero or one* **Boxed Field**

Boxed Field shows *one* **Frame Metadata**

A unit of Metadata that is designated for display within a Frame might be rendered in a Box Field. (If not, it must be rendered in an Open Field).

A Boxed Field must be assigned a Metadata item, but that item may not have any content supplied, in which case the field will appear blank when rendered.

Formalization

Open Field.Metadata -> Frame Metadata.Name

Open Field.Frame -> Frame Metadata.Frame

Open Field.Sheet -> Frame Metadata.Sheet

R318 / M:Mc

Title Block Placement determines placement of *one or many* **Box**

Box placement is determined by *zero, one or many* **Title Block Placement**

Once a Title Block Pattern is scaled and positioned in a Frame, it is possible to derive the placement and size of each Box in the Title Block Pattern with respect to the drawing Canvas.

Formalization

Box Placement.Frame -> Title Block Placement.Frame

Box Placement.Sheet -> Title Block Placement.Sheet

Box Placement.Box -> Box.ID

Box Placement.Title block pattern -> Box.Pattern

R319 / 1:1c

Frame Metadata appears in *zero or one* **Open Field**

Open Field shows *one* **Frame Metadata**

A unit of Metadata that is designated for display within a Frame might be rendered in an Open Field. (If not, it must be rendered in a Boxed Field).

An Open Field must be assigned a Metadata item, but that item may not have any content supplied, in which case the field will appear blank when rendered.

Formalization

Open Field.Metadata -> Frame Metadata.Name

Open Field.Frame -> Frame Metadata.Frame

Open Field.Sheet -> Frame Metadata.Sheet

R320 / 1:M

Frame Metadata must appears in *one or many* **Field**

Field must show *one* **Frame Metadata**

If an item of Metadata is included in a Frame, it must appear in at least one Field. At most, it can appear in two, one of each type, Boxed and Open. The reasoning is that there shouldn't be any need to duplicate frame data in multiple locations inside or outside of a title block. But it can be helpful to take some piece of data that appears in a title block and present it outside for emphasis. The specific case motivating this rule is the diagram `Title` Metadata item. Even though the title information could be included in the title block, it is often nice to show it in the upper left corner of the sheet also for quick reference.

Formalization

Field.Metadata -> Frame Metadata.Name

Field.Frame -> Frame Metadata.Frame

Field.Sheet -> Frame Metadata.Sheet

R321 / 1:Mc

Boxed Field is placed on *one* **Box Text Line**

Box Text Line places *zero, one or many* **Boxed Field**

.

Formalization

Boxed Field.Metadata -> Boxed Text Line.Metadata

Boxed Field.Box -> Boxed Text Line.Box

(constrained such that Boxed Field.Metadata == /R321/Box Text Line.Metadata)

Boxed Field.Title block pattern -> Boxed Text Line.Title block pattern

R322 / Generalization

Metacontent is **Text Content** or **Resource**

The data to be filled in for an item of Metadata can be plain text, such as an author name or a document title, or it can be a resource such as an image in an png file. (In fact, that's the only kind of resource currently supported).

Formalization

<subclass>.Metadata -> Metacontent.Metadata

Connector Subsystem

This subsystem describes the overall geometry for all connector types as well as the placement of symbols and labels on connector stems.

Relationship numbering range: R50-R99

Class Descriptions

Anchored Stem

Not to be confused with the beer made in San Francisco, California. This is a Stem whose root end is determined by a user specified Face Placement position on the Node Face.

Attributes

ID

Same as**Stem.ID**

Connector

Same as**Stem.Connector**

Node

Same as **Stem.Node**

Face

Same as **Stem.Face**

Anchor position

Relative distance from the center of the Node face.

Type: Face Placement `-5..+5` where zero represents the center with + to the right or top and - to the left or bottom, both away from the center

Identifiers

1. **ID** + **Connector**
2. **Node** + **Face** + **Anchor position**

To prevent any drawing overlap, two Stems may not anchor at the same Node face placement location.

Annotation

The application of a Label to a Decorated Stem is an Annotation. Whereas a Decoration is drawn on a Stem on one end or the other (root or vine), a Label is offset from the Stem so that it doesn't overlap the Stem line and relative to the Node face where the Stem is attached.

Attributes

Stem type

Same as **Decorated Stem.Stem type**

Semantic

Same as **Decorated Stem.Semantic**

Diagram type

Same as **Decorated Stem.Diagram type**

Notation

Same as **Decorated Stem.Notation**

Label

Same as **Label.Name**

Default stem side

By default the Rendered Label will appear on this side of the Stem axis in its vicinity. The user can override this default by specifying a Label flip. If the stem is drawn vertically near the Label, it will appear to the right or left and if the stem is drawn horizontally the label will be above or below the stem. If a + value is specified, it means to the right or above since the x or y axis increases in that direction.

Type: [+ | −]

Vertical stem offset

If the Stem is drawn horizontally, this is the vertical space between the Label content rectangle and the Stem.

Type: Distance

Horizontal stem offset

If the Stem is drawn vertically, this is the horizontal space between the Label content rectangle and the Stem.

Type: Distance

Node face offset

The minimum (and default) distance between the Label content rectangle face parallel and closest to the Node face at the root end of the Stem.

Type: Distance

Identifiers

Stem type + **Semantic** + **Diagram type** + **Notation**

Consequence of a one-many association with id formed from reference to the many side.

Connector

A Connector is a set of Stems connected by one or more lines to form a contiguous branch bringing one or more Nodes into a drawn model level relationship. On a class diagram, for example, a Connector is drawn for each binary association, generalization and association class relationship.

The Connector Type and its Stem Types determine how the Connector should be drawn.

Attributes

ID

Each Connector is numbered uniquely on its Diagram.

Type: Nominal

Diagram

Same as **Diagram.ID**

Connector type

Same as **Connector type.Name**

Diagram type

Same as **Connector type.Diagram type**

Identifiers

ID

Since only one Diagram is drawn at a time, there is only ever one instance of Diagram and so the Connector.ID suffices as a unique identifier.

Connector Layout Specification

Defines a set of values that determine how a Connector is drawn.

Attributes

Name

In this version there is assumed to be only a single specification instance, so the name is here merely expresses unique model identity.

Type: Name

Default stem positions

The number of equally spaced positions relative to a center position (included in the count) on a Node face where a Stem can be attached. A value of one corresponds to a single connection point in the center of a Node face. A value of three is a central connection point with one on either side, and so on. In practice, five is usually the right number, especially for a class or state diagram. But this could vary by diagram and node type in the future.

Type: Odd Quantity **::** Odd Integer > 0

Default rut positions

The number of ruts where Path can be defined in a Lane. These work like stem/anchor positions on a Lane as opposed to a Node face. For a value of 3 we get positions -1, 0 and +1 with 0 representing the Lane Center and +1 high/right and -1 low/left.

Type: Odd Quantity **::** Odd Integer > 0

Default new path row height

When a new empty row must be added to to accommodate a Path in a Connector use this initial height.

Type: Distance

Runaround lane width

When a new empty row or column must be added to accommodate a Path that bends outside the grid, this is the initial height or width to use in creating that Lane.

Type: Distance

Identifiers

1. **Name** // This is a singleton, so the name is certainly unique

Connector Name

The user may supply a name for any or all Connectors in a Diagram. On a class diagram, for example, the user would specify names like R2, R35, etc. for each relationship Connector.

Attributes

Connector

Same as **Connector.ID**

Name

The user supplied name to be drawn on or near the Connector axis.

Type: Text

Bend

If the Connector is bent, we proceed clockwise from the first attached Node starting from 1 for each bend. The term "bent" can be applied liberally. In the case of a Binary Connector, we really mean Bend. With a Tree Connector, the quantity can represent each Branch.

The Name will then be placed at the center of the line segment of this Bend.

In the case of a non-bent Connector, this value is ignored

Type: Count

Side

For a horizontal Connector, this will be above or below and for a vertical Connector it will be left or right. Since both right and above are at increasing coordinate values along one coordinate axis, we can just use a positive or negative sign to indicate the Side. Positive (1) means above or right while negative (-1) is the other side.

Type: `[ 1 | -1 ]` as an integer value

Location

The coordinates of the lower left text bounding box.

Type: Position

Size

The dimensions of the text bounding box.

Type: Rect Size

Identifiers

1. **Connector type** + **Diagram type** + **Notation**

Connector Name Specification

A Diagram Notation may specify that a given Connector Type be named along with the default placement information for that name. For diagram generation purposes, we leave it to the user to supply a name with a format appropriate to the Diagram Type and Notation. But we can retain layout information so that the user need not specify precise placement of each name. For example, we can say that a certain name be placed in the center of each connector overlaying it, or at a certain distance above a horizontal connector and to the right of a vertical connector.

The name of a Connector is not associated with any particular end of the Connector. In that case you would use a Stem name instead.

Attributes

Connector type

Same as **Connector Type.Name**

Diagram type

Same as both **Diagram Notation.Diagram type** and **Connector Type.Diagram type**

Notation

Same as **Diagram Notation.Notation**

Vertical axis buffer

The buffer ensures that there is consistent whitespace between the name and the connector axis. For example, all names for a given Connector Type can be drawn with 7 points of empty space above or below the Connector line segment.

This buffer is the vertical gap above or below a horizontal connector bend. The distance is measured from the edge of the text bounding box closest to the adjacent connector axis.

If the value is zero, the name is drawn centered on top of the Connector with a solid fill around the text so that the connector line is never drawn through the text.

Type: Distance

Horizontal axis buffer

Same concept as the **Vertical axis buffer** except that this is the horizontal gap, right or left, of a vertical connector bend.

Type: Distance

Default name

A text value to be used in case the user does not supply a name.

Type: Text

Optional

Whether or not the name is required or optional. If required and no name is supplied a warning can be raised and the default name applied.

Type: Boolean

Identifiers

1. **Connector type** + **Diagram type** + **Notation**

Connector Style

Connectors are ordinarily drawn as un-patterned lines. If the lines in a Connector will be drawn with some other pattern, such as dashed, a Connector Style is defined. For example, the xUML dependency connectors in a domain diagram (package dependency) are dashed.

Attributes

Connector type

Same as Connector Type.Name

Diagram type

Same as both **Diagram Notation.Diagram type** and **Connector Type.Diagram type** This enforces the constraint that a line style can be defined only for a Notation defined on the Connector Type.

Notation

Same as **Diagram Notation.Notation**

Stroke

The stroke style to use when drawing the Connector lines.

Type: Stroke Style

Identifiers

1. **Connector type** + **Diagram type** + **Notation**

From association multiplicity

Connector Type

One or more Nodes may be interrelated by some model level relationship such as a state transition, generalization, association, dependency and so forth. Each such relationship is drawn with one or more connecting lines and terminating symbols. A Connector Type defines the symbols, line connection geometry and appearance of Connectors corresponding to some model level relationship.

Attributes

Name

The name of the model level relationship such as “Transition” or “Generalization”.

Type: Name

Diagram type

Same as **Diagram Type.Name**

Geometry

This describes the way that a Connector is drawn, pulling together all of its Stems. Many geometries are possible, but only a handful are supported which should cover a wide range of diagramming possibilities.

Unary – Relationship is rooted in some Node on one end and not connected on the other end. An initial transition on a state machine diagram is one example where the target state is connected and the other end of the transition just has a dark circle drawn at the other end (not a Node). It consists of a single Stem.

Binary – Relationship is drawn from one Node face position to another on the same or a different Node. This could be a state transition with a from and to state or a binary association from one class to another or a reflexive relationship starting and ending on the same class or state. It consists of two Stems, one attached to each Node face position connected together with a line. A Tertiary geometry where a third Stem connects a Node face to the binary connection is also possible in this geometry. It is considered an optional extension that can be defined on any Binary Connector.

Tree – Here one Node is a root connecting to two or more other Nodes. A Stem emanates from the root Node and another type of Stem emanates from each of the subsidiary Nodes and one or more lines are drawn to connect all the Stems. A class diagram generalization relationship is a typical case.

Type: Connection Geometry`:: [ unary | binary | tree ]`

Identifiers

Name + **Diagram type**

The Name is unique for each Diagram Type by policy. It seems likely that a name like “Transition, for example, could be useful and defined differently across Diagram Types.

Decorated Stem

A Stem Signification that is decorated somehow when it appears on a Diagram is considered a Decorated Stem. Not all Stem Significations are decorated. The stem attaching a class diagram subclass is not notated in many class diagram notations.

See R55 description for more details.

Attributes

Stem type

Stem Signification.Stem type

Semantic

Stem Signification.Semantic

Diagram type

Type: Same as**Stem Signification.Diagram type** and **Diagram Notation.Diagram type**

Notation

Diagram Notation.Notation

Stroke

This is the style used to draw the Stem where it isn't occluded by any Symbols. In most cases it is probably just the default connector style. But in at least the case of an `xUML- associative mult` Decorated Stem, a dashed line is typically drawn.

Type: Stroke Style

Identifiers

Stem type + **Semantic** + **Diagram type** + **Notation**

Consequence of a many-many association with a shared Diagram Type.

Floating Stem

The user specifies the Node face, but not the attachment position of a Floating Stem. The point on the Node face where a Floating Stem attaches is determined by the position of an opposing Anchored Stem so that a straight line between them is ensured.

Attributes

ID

Same as**Stem.ID**

Connector

Same as**Stem.Connector**

Identifiers

1. **ID** + **Connector**

Free Stem

This type of Stem is used to create a Unary Connector. In fact, a Free Stem comprises the entire Unary Connector.

Attributes

ID

Same as**Stem.ID**

Connector

Same as**Stem.Connector**

Identifiers

1. **ID** + **Connector**

Rendered Label

The application of a Label to a Decorated Stem is an Annotation. Whereas a Decoration is drawn on a Stem on one end or the other (root or vine), a Label is offset from the Stem so that it doesn't overlap the Stem line and relative to the Node face where the Stem is attached.

Attributes

Stem

Same as **Stem.ID**

Connector

Same as **Stem.Connector**

Location

The location of the lower left corner in Diagram coordinates

Type: Position

Stem type

Same as both **Annotation.Stem type** and **Stem.Stem type**

Semantic

Same as both **Annotation.Semantic** and **Stem.Semantic**

Diagram type

Same as both **Annotation.Diagram type** and **Stem.Diagram type**

Notation

Same as both **Annotation.Notation** and **Stem Type.Notation**

Identifiers

1. **Stem** + **Connector** // Reference to many side
2. **Location**// Otherwise there could be an illegal overlap on the Diagram

Rendered Symbol

This is the Symbol as drawn on one end of a Stem on the Diagram.

Attributes

Stem

Same as **Stem.ID**

Connector

Same as **Stem.Connector**

Stem type

Same as both **Stem End Decoration.Stem type** and **Stem.Stem type**

Semantic

Same as both **Stem End Decoration.Semantic** and **Stem.Semantic**

Diagram type

Same as both **Stem End Decoration.Diagram type** and **Stem.Diagram type**

Notation

Same as both **Stem End Decoration.Notation** and **Stem Type.Notation**

End

Same as **Stem End Decoration.End**

Growth

The distance from the Stem End (vine or root) to the edge of the Symbol on the Stem.

Type: Distance

Identifiers

1. **Stem type** + **Semantic** + **Diagram type** + **Notation** + **Stem** + **Connector** + **End**// From multiplicity

Stem

This is a line drawn from a face on a Node outward. The terminator on the Node face is the root and the terminator on the other side of the line is the vine. Both terminators are generally referred to as the Stem ends.

A Stem may be decorated on either, both or neither end. A decoration consists of a graphic symbol such as an arrow or a circle or a fixed text Label such as the UML `0..1` multiplicity text. A graphic symbol may be combined with a text Decoration such as the Shlaer-Mellor open arrow head and `c` conditionality Label combination.

Attributes

ID

Distinguishes one Stem from another within the same Connector.

Type: Nominal

Connector

Same as**Connector.ID**

Stem type

Same as **Stem Type.Name** and **Stem Signification.Stem Type**

Diagram type

Same as **Stem Type.Diagram type** and **Stem Signification.Diagram type**

Node

Same as **Node.ID**

Face

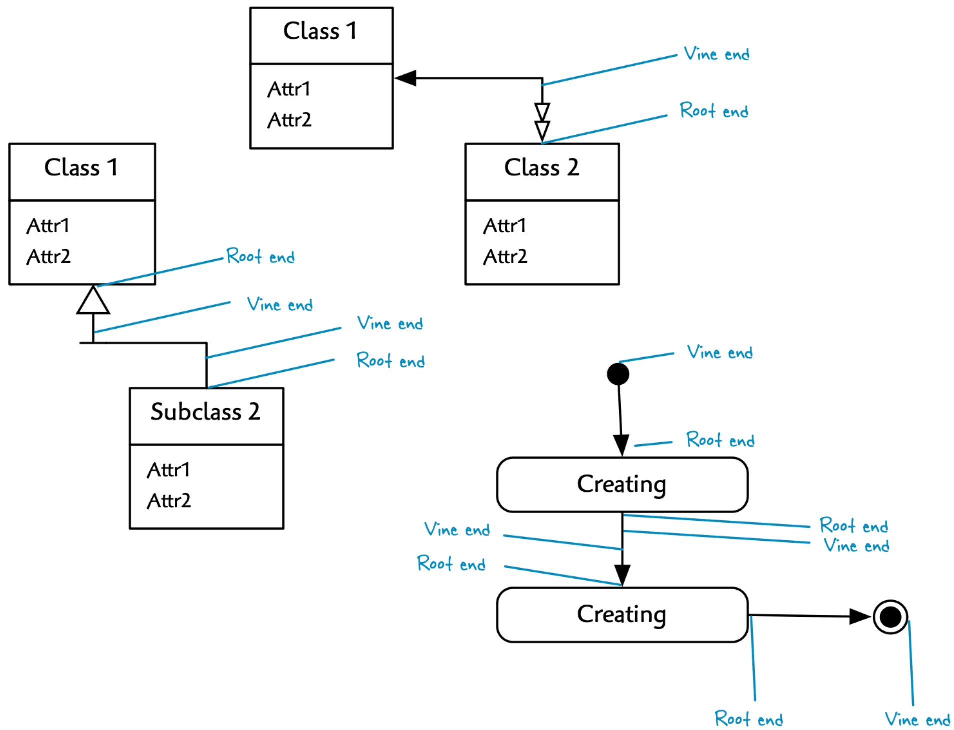
The side of the Node where the Stem is anchored.

Type: Node Face `:: [ Top | Bottom | Right | Left ]`

Root end

The point on the attached Node face where the Stem root is anchored.

Stem End positions



Type: Position

Vine end

The point where the Stem vine ends away from the attached Node. See figure in **Root end** description.

Type: Position

Identifiers

1. **ID** + **Connector**

Each Stem is uniquely numbered local to its Connector. The **ID** attribute is added since this is a -M association class which means that multiple instances of Stem may correspond to the same Connector–Stem Type pair.

2. **ID** + **Connector** + **Node** + **Face**

Superkey is provided so that Anchored Stem subclass can enforce a constraint on Stem placement to avoid coincident Stems (see Anchored Stem).

3. **Node** + **Face** + **Root end**

Now two Stems may share the same Root end position on a Node Face. Same coincident Stem constraint as supported by identifier #2 above, but enforced at the point when the coordinates are resolved.

Stem End Decoration

Either the root or vine end of a Decorated Stem that features a Symbol when drawn.

See R58 description for more details.

Attributes

Stem type

Same as **Decorated Stem.Stem type**

Semantic

Same as **Decorated Stem.Semantic**

Diagram type

Same as **Decorated Stem.Diagram type**

Notation

Same as **Decorated Stem.Notation**

Symbol

Same as **Symbol.Name**

End

A Stem has two ends, root and vine. Either, both or neither end may be decorated.

Type: `[ root | vine ]`

Identifiers

Stem type + **Semantic** + **Diagram type** + **Notation** + **Symbol**+ **End**

Consequence of a many-many association with the addition of an extra attribute **End placement** to distinguish the -M associative multiplicity.

Stem Name

The user may supply a name for any or all Connectors in a Diagram. On a class diagram, for example, the user would specify names like R2, R35, etc. for each relationship Connector.

Attributes

Stem

Same as **Stem.ID**

End

The end of the Stem where the name is placed.

Type: `[ root | vine ]`

Name

The user supplied name to be drawn on or near the Stem. The text will be right or left aligned depending on the location relative to the Stem.

Type: Text

Side

For a horizontal Stem, this will be above or below and for a vertical Stem it will be left or right. Since both right and above are at increasing coordinate values along one coordinate axis, we can just use a positive or negative sign to indicate the Side. Positive (1) means above or right while negative (-1) is the other side.

Type: `[ 1 | -1 ]` as an integer value

Location

The coordinates of the lower left text bounding box.

Type: Position

Size

The dimensions of the text bounding box.

Type: Rect Size

Identifiers

1. **Connector type** + **Diagram type** + **Notation**

Stem Name Specification

For a given Notation, certain Stem Types are named. For each case we can establish the uniform placement of such names relative to the associated Stems.

Attributes

Stem type

Same as **Stem Type.Name**

Diagram type

Same as both **Diagram Notation.Diagram type** and **Connector Type.Diagram type**

Notation

Same as **Diagram Notation.Notation**

End

The end of the Stem where the name is placed.

Type: `[ vine | root ]`

Vertical axis buffer

The buffer ensures that there is consistent whitespace between the name and the connector axis. For a Stem, this is the distance away from the Stem which should be greater than half the width of any Stem Decoration to avoid overlap.

In the vertical case, this is the vertical distance from a horizontally aligned Stem.

Type: Distance

Horizontal axis buffer

Same concept as for **Vertical axis buffer**.

In the horizontal case, this is the horizontal distance from a vertically aligned Stem.

Type: Distance

Vertical end buffer

The buffer ensures that there is consistent whitespace between the name and the root or vine end of the Stem. In the case of a root end, this is the gap between the name bounding box and a Node Face. In the case of a vine end, it depends on the connector type. For a tertiary connector, the gap is between the name bounding box and a binary connector line segment.

In general the distance should never be zero since there is a risk that the name would be drawn on top of a stem decoration. But if there is no decoration it may make sense to specify zero so that the name is drawn over the top of the stem with a solid background.

A vertical end buffer is associated with a vertical connector line segment where the name will be to the right or left of the line

Type: Distance

Horizontal end buffer

Same concept as the **Vertical end buffer** except that this is the horizontal gap, right or left, of a vertical connector line segment.

Type: Distance

Default name

A text value to be used in case the user does not supply a name.

Type: Text

Optional

Whether or not the name is required or optional. If required and no name is supplied a warning can be raised and the default name applied.

Type: Boolean

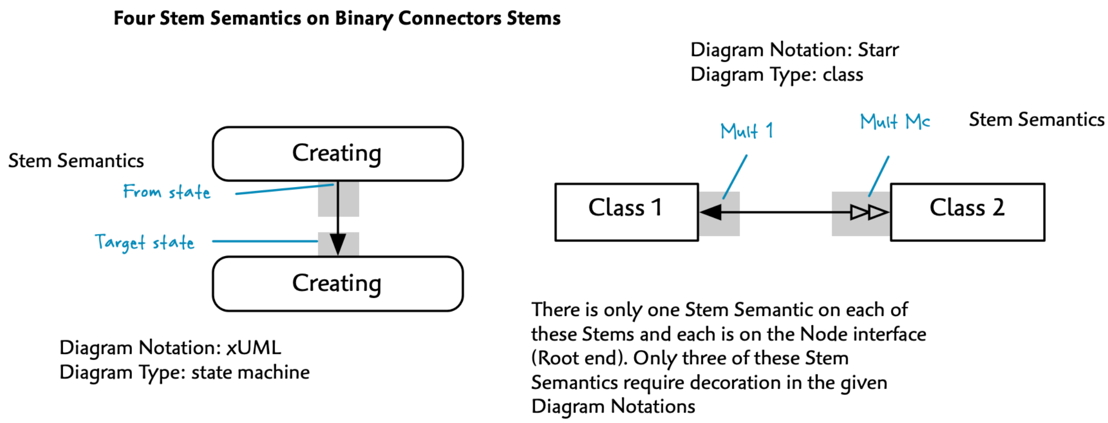
Identifiers

1. **Stem type** + **Diagram type** + **Notation** + **End**

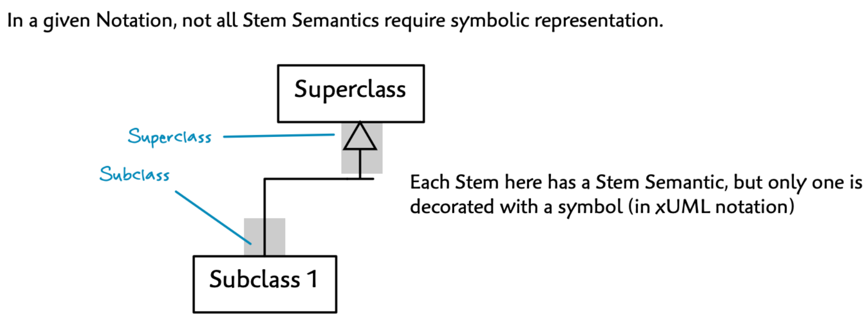
Stem Semantic

A Stem Semantic is some notation independent meaning that can be attributed to either end (root/vine) of a Stem. When combined with a Diagram Notation, it may or may not be represented by some visual representation such as an arrow or text.

A Stem always has meaning where it attaches to its Node since the connected Node is playing some sort of role (target state, class multiplicity, subclass, etc).

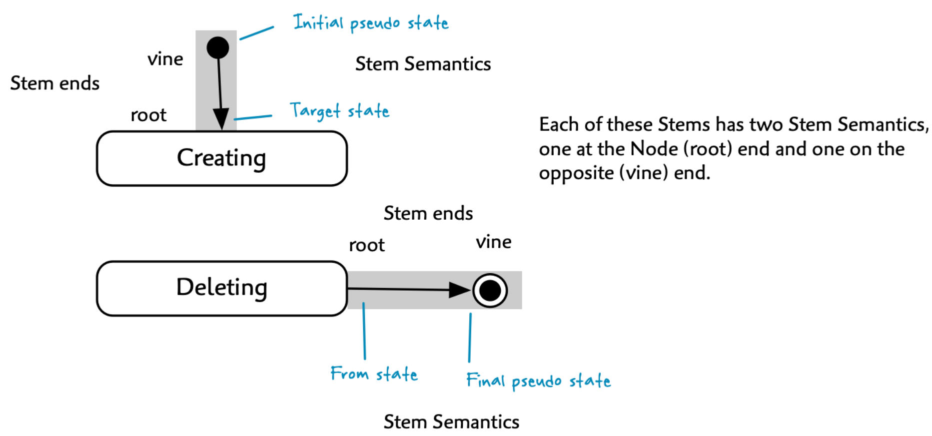


In a given Diagram Notation, a Stem Semantic may or may not require any Symbols or Labels. The from state semantic, for example, is just an undecorated line in xUML. Subclasses in xUML, Starr and Shlaer-Mellor class diagrams are similarly undecorated.



In some cases, the Stem end away from the Node face (vine end) will also have significance. Usually this is only the case when the Stem is not connected to any other Stem as it is in a Unary Connector.

Two Stems on Unary Connectors



On a state machine diagram, for example, the line that touches a state Node (root end) is terminated with an arrow to indicate a target state. The opposing end of the Stem (vine end) is undecorated unless the state Node is an initial state. In this case there is a decoration on each end of the Stem.

In the case of a deletion transition on a state machine diagram, the root end of the Stem attached to the Node is undecorated while its opposite vine end features a dot filled circle.

Attributes

Name

A name that reflects the meaning (semantic) of the Stem termination such as “target state” (goes to this state) or “Mc mult” (many conditional multiplicity) or “final psuedo-state”. Care is taken to describe meaning and not notation.

Type: Name

Identifiers

Name

Unique by policy

Stem Signification

This is a meaning that is relevant to a particular Stem Type. See the description of R62 for more details.

Attributes

Stem type

Type: Same as **Stem Type.Name**

Semantic

Type: Same as **Stem Semantic.Name**

Diagram type

Type: Same as both **Stem Semantic.Diagram type** and **Stem Type.Diagram type**. It establishes the constraint that a Stem Type may signify only a Stem Semantic that is defined on the same Diagram Type.

Identifiers

1. **Stem type** + **Semantic** + **Diagram type**

Determined by the association multiplicity

Stem Type

Defines the characteristics of the portion of a Connector attached to a Node called a 'Stem'.

In a binary association connector of a class model, for example, there are two `class mult` Stem Types and one `associative mult` Stem Type defined. A transition Connector Type in a state machine diagram defines two Stem Types, `to state` and `from state`.

Characteristics of primary interest are the semantics and notation and any other visual aspects of a Stem.

Attributes

Name

Describes the type of Node to which a Stem will be attached such as to state or association class.

Type: Name

Diagram type

Type: Same as **Diagram Type.Name**

Connector type

Type: Same as **Connector Type.Name**

About

A description of the purpose and usage of this Stem Type

Type: Description

Minimum length

A Stem of this type can never be shorter than this length. This keeps a bend or the Diagram edge from getting too close to the Node face. You wouldn't want to bend at 90 degrees less than a point away from a Node face, for example.

This value also serves to provide a default distance between the Root and Vine Ends, thus readily establishing the coordinate of the Vine End (assuming the Stem's Vine end isn't based on some other factor. In the case of a Tertiary Stem in a Binary Connector, for example, the Vine End will extend out to the nearest normal connector line, thus exceeding the Minimum Length usually.

Type: Distance

Identifiers

1. **Name** + **Diagram type**

Stem Type Names are unique to each Diagram Type by policy

Unary Connector

This type of Connector is rooted on some Node face with a vine end that does not attach to anything. It is therefore placed at some fixed distance away from the root end. The initial and final psuedo-transitions on a UML state machine diagram are both examples of Unary Connectors.

Attributes

ID

Same as **Connector.ID**

Identifiers

ID

Relationship Descriptions

R50 / 1:Mc

Connector Type can be drawn in *exactly one* **Diagram Type**

Diagram Type can draw *zero, one or many* **Connector Type**

These are the types of Connectors that can be drawn on a given Diagram Type. On an xUML state machine diagram you can draw initial, final and normal transitions, for example, whereas on an xUML class diagram you can draw generalizations, binary associations and association class relationships. More to the point, you cannot draw a state transition on a class diagram. So this relationship constrains what can be drawn on a given Diagram Type. (Though nothing prevents you from defining a new Diagram Type where this would be possible!)

Most Diagram Types will have at least one kind of Connector Type, otherwise the associated diagrams will just be a layout of unconnected Nodes. That said, there is no reason to require connections on any given Diagram Type.

A Connector Type is defined exclusively to a Diagram Type. Thus, transition on a state machine diagram may be defined differently than transition on some other kind of diagram.

Formalization

Reference in the Connector Type class

R51 / 1:Mc

Connector Type specifies *zero, one or many* **Connector**

Connector is specified by *exactly one* **Connector Type**

This is a standard specification relationship where the Connector Type defines various characteristics of a Connector. Whereas a Connector Type defines properties of all Connectors, a Connector is a manifestation of a Connector Type actually drawn on a Diagram.

When a Connector is created, it will need to grow a Stem for each connected Node and then draw a line that ties the Stems all together.

Formalization

Reference in the Connector class

R52 / 1:Mc

Node is source of *zero, one or many* **Stem**

Stem is rooted in *exactly one* **Node**

The root end of Stem is always attached to a single Node. In fact, a Stem never attaches more than one Node, though a Connector certainly can via multiple Stems. There is no such thing as a free floating Stem unattached to any Node.

A Node, on the other hand, may or may not be part of a connection. A free floating unconnected Node will not be attached to any Stem.

Formalization

Referential attribute in Stem class

R53 / M:Mc-M

Connector sprouts as *one or many* **Stem Type**

Stem Type sprouts in *zero, one or many* **Connector**

A Connector is drawn by creating all necessary Stems and then connecting them together with one or more lines. The **Connector Type.Geometry** attribute determines how these Stems and connecting lines will be drawn.

The same Stem Type may be used multiple times in a Connector. For example, an xUML class diagram binary association will need two class multiplicity Stems, one for each side of the Connector. A class diagram generalization will need one subclass stem for each subclass Node. Each connection to a Node will result in a new Stem.

If no Connectors have been drawn that use a particular Stem Type, that Stem Type will just be a definition that hasn't been used yet. In this case the Stem Type won't refer to any Connectors.

Formalization

Stem association class

R54 / 1c:Mc-1

Decorated Stem is annotated by *zero or one* **Label**

Label annotates *zero, one or many* **Decorated Stem**

A Decorated Stem may or may not have an associated text Label. In the Starr class diagram notation a generalization arrow has no associated text. In xUML, however, the arrow is accompanied by the UML tag `{ disjoint, complete }`. There seems to be no reason to support multiple fixed text Labels as none of the supported notations require them.

A given Label may be used with more than one Decorated Stem. The Shlaer-Mellor `c` label is associated with any class multiplicity where zero is a possibility, for example.

A Label may be defined that is not used with any notation, though this is unlikely. It can be done in anticipation of supporting a future notation, however.

Formalization

Referential attributes in the Annotation class

R55 / Mc:Mc-1

Diagram Notation decorates *zero, one or many* **Stem Signification**

Stem Signification is decorated with *zero, one or many* **Diagram Notation**

Each Diagram Notation may specify a different decoration for a Stem Signification. The Starr class diagram notation, for example assigns a double hollow arrow at the root end of a `class mult – Mc mult` Stem Signification. xUML, on the other hand specifies only a text label of `0..*`for that same Stem Signification.

In fact, a Stem Signification may not be decorated at all in a given Diagram Notation. The `from state – source state` Stem Signification on a state machine diagram, for example, is not decorated in xUML while the `to state – target state` is.

A given Diagram Notation only specifies decoration for those Stem Significations relevant to the associated Diagram Type. Thus the, `Starr – class` Diagram Notation does not specify decoration on any Stem Significations on a state machine diagram.

Formalization

Stem Decoration association class

R56 / 1:Mc

Stem indicates *one* **Stem Signification**

Stem Signification is indicated on *zero, one or many* **Stem**

When a Stem is drawn it binds to one of the Stem Significations that its Stem Type may signify. A Stem whose type is `class mult` (class multiplicity) must indicate one of the available multiplicity significations, namely: `1`, `M`, `1c` or `Mc`. The selection will be user specified. For many Stem Types there will be only one Stem Signification to choose from so the indication is automatic.

Formalization

Referential attributes in the Stem class

R57 / 1:Mc

Diagram Type is context for *zero, one or many* **Stem Semantic**

Stem Semantic has meaning on *exactly one* **Diagram**

Consider a Stem Semantic such as `class mult` (class multiplicity) or maybe another `target state`. Each Stem Semantic defines the meaning associated with the point where a Connector attaches to some Node. The `class mult` Stem Semantic only makes sense on a class diagram while the `target state` Stem Semantic is intended for state machine diagrams.

In fact, each Stem Semantic is specific to the context defined by a type of Diagram. In other words, each Diagram Type establishes a set of relevant Stem Semantics that make sense only on that Diagram Type.

True, you may create a Diagram Type with semantics similar or almost identical to another Diagram Type. Say you define a `petri net` Diagram Type which also specifies `target state`. We still want to keep the semantics custom specified for each Diagram Type so that we don't elide subtle distinctions among them. No problem since the name of a Stem Semantic is local to its own Diagram Type. Thus a `petri net-target state` is distinct from a `state machine-target state`. The semantics may be equivalent or slightly different, but they are two distinct semantics as far as Flatland is concerned.

If a Diagram Type does not specify any Stem Semantics, this means that the Diagram Type does not support Connectors of any type. Perfectly legal, but of questionable value. Flatland will draw them at any rate!

Formalization

Reference in Stem Semantic class

R58 / Mc:Mc-M

Decorated Stem is terminated by *zero, one or many* **Symbol**

Symbol terminates *zero, one or many* **Decorated Stem End**

Each end of a Decorated Stem may or may not be adorned by a single Symbol. Keep in mind that a Symbol can be compound and built up from many graphical elements. So each terminal can be as ornate as necessary. This effectively means that at most two Symbols can be associated with a given Decorated Stem. See note in formalization section below to see how the two-ness constraint is addressed.

It is also possible for the same Symbol to be used at both ends of a Decorated Stem. Consequently this relationship is many-associative. (A given pairing of Decorated Stem and Symbol can result in two association class instances, differentiated by the **End** component of the class identifier).

If neither end of a Decorated Stem features a Symbol, there may be a Label associated with the Stem. If there is no Label either, perhaps the Stem is notated by changing its line stroke pattern. For example, in xUML an associative 1 multiplicity on a class diagram is shown by drawing the stem as a dashed pattern with no other label or symbol.

A Decorated Stem that does not have a special line pattern, Symbol or Label is not decorated and should not be declared as such. No harm can come from falsely declaring a Decorated Stem with no Decoration, it will just be rendered as a linear Stem, but it is bad practice.

A given Symbol can be used in as many Decorated Stems as you like. A `solid arrow` for example might be used both in a state transition and in a domain diagram dependency. If a Symbol is not used at all, there is no harm as it may become useful in a Diagram Notation defined later.

Formalization

Referential attributes in the Stem End Decoration association class along with enforcement of the two-ness constraint. This is accomplished by integrating the **End** attribute into the Decoration identifier. See the class description for more details.

R59 / 1:M

Connector Type connects nodes with *one or many* **Stem Type**

Stem Type defines node connections for *one or many* **Connector Type**

We define the structure of a Connector by describing it as a set of Stems of various types that are lashed together with connecting lines. Each Stem Type establishes the meaning of the interface between a Connector line and the Node where it attaches. For each type of Connector, certain types of Stems are relevant.

For example, a generalization Connector Type defined on a class diagram requires only two types of Stems, a superclass and a subclass Stem Type to designate the meaning of each connection point. Furthermore, the subclass Stem Type has relevance only to a class diagram generalization Connector Type.

A Connector Type without any Stem Types makes no sense because it couldn't connect to any Nodes. And a Stem Type only has utility as part of some Connector Type.

Formalization

Stem Type Usage association class with shared **Diagram type**

R60 / Mc:Mc-1

Connector Type lines are styled in *zero, one or many* **Diagram Notation**

Diagram Notation styles lines of *zero, one or many* **Connector Type**

A Connector Style is defined only for those Connector Types that are not simple solid black lines. So most Connector Types do not need any special style in a given Diagram Notation.

A given Diagram Notation may or may not set styles for Connector Types.

Formalization

References in the Connector Style association class

R61 / Mc:Mc-1

Stem End Decoration is rendered near *zero, one or many* **Stem**

Stem renders *zero, one or many* **Stem End Decoration**

When a Stem is drawn, any corresponding Symbols are positioned on one or both Stem end axes and rendered as specified by the Stem End Decoration. (There are at most two Symbols placed on a given Stem, one at each end).

Formalization

Referential attributes in the Rendered Symbol class

R62 / M:M-1

Stem Type may signify *one or many* **Stem Semantic**

Stem Semantic may be signified by *one or many* **Stem Type**

A Stem Semantic refines the general meaning specified by a Stem Type. A `class mult` Stem Type, for example, indicates the dual concepts of multiplicity and conditionality. A variety of Stem Semantics are available that each establish a precise pairing of multiplicity and conditionality `1 mult` (unconditional 1), `Mc mult` (conditional many), and so forth. When a Stem is created, it must bind to one of the Stem Semantics available to the Stem's Stem Type.

A given Stem Semantic may be relevant to more than one Stem Type. The unconditional multiplicity `1 mult` and `M mult` Stem Semantics, for example, also apply to the `associative mult` Stem Type that defines the Stem on a class diagram's association class.

A Stem Semantic is not useful if it has no relevance to any Stem Type, so it must be relevant to at least one.

Many Stem Types have meanings that cannot be further modified and therefore may signify only one available Stem Semantic. A `to state` Stem Type can only mean `target state`, for example. But every Stem Type does not have a specific meaning unless it can signify at least one Stem Semantic.

Formalization

References in Stem Semantic Option association class

R63 / 1:Mc

Diagram shows *zero, one or many* **Connector**

Connector appears on *one* **Diagram**

A Connector is rendered on the one and only Diagram. And it is certainly possible to create a Diagram with Nodes and no Connectors.

Formalization

Reference in Connector class

R65 / Generalization

Anchored Stem is an **Anchored Binary Stem**, **Tertiary Stem**, **Anchored Tree Stem** or **Free Stem**

Each of these subclasses of Anchored Stem are Stems that attached at a user specified anchor position on a Node face.

Each Connector subclass determines the quantity and combination of various types of Stems. A Tree Connector, for example, consists of one Trunk Stem and one or more Leaf Stems. A Unary Connector consists of a single Free Stem.

See each relevant connector subsystem to see how each subclass of Anchored Stems are applied.

Formalization

ID + Connector referenced from each subclass

R66 / Generalization

Floating Stem is a **Floating Binary Stem** or **Floating Leaf Stem**

Floating Stems have utility in both the Binary and Tree Connector subsystems. Though they play different roles in each, a Floating Stem always derives its axis from a coincident Anchored Stem guide.

Formalization

References in the subclasses

R67 / Generalization

Stem is a **Floating Stem** or **Anchored Stem**

An Anchored Stem is positioned by the user with an **Anchor position**. This position is later resolved to diagram coordinates. Anchored Stems are used in all Connector Types.

A Floating Stem is lined up with an opposing Anchored Stem so that a straight line is formed. The pairing of Anchored and Floating Stems is useful in both Binary and Tree Connectors.

With a Straight Binary Connector, there is no need for two user specified anchor positions. Since the Connector is a straight line, only one anchor position is necessary. In fact, there should only be one to ensure that we end up with a non-diagonal line when the coordinates are resolved.

The non-anchored Stem in a Straight Binary Connector is understood to float so that it is level with the opposing Anchored Stem. The position of a Floating Binary Stem is computed for a horizontal line by sharing the x coordinate of the opposing Anchored Stem. This is the y coordinate if the line is vertical.

The same situation can occur in a Tree Connector where one Leaf Stem is anchored while another is lined up with it straight across.

Formalization

ID + Connector in either subclass or **ID + Connector + Stem type + Node + Face + Anchor position** in the Anchored Stem subclass. Two different ID's are referenced since the Anchored Stem is enforcing a constraint preventing two Anchored Stems from being placed in the same location on the same Node face.

R68 / 1c:Mc-1

Annotation is rendered near *zero, one or many* **Stem**

Stem renders *zero or one* **Annotatio**

When a Stem is drawn, any corresponding Label is positioned on the Diagram and rendered as specified by the Annotation.

Formalization

Referential attributes in the Rendered Label class

R69 / Generalization

Connector is a **Hierarchy**, **Unary** or **Binary Connector**

Different rules and constraints may apply to each geometry so they are subclassed. Primarily an unbent Binary Connector has a special relationship to a Floating Stem.

The type is determined by the **Connector Type.Geometry** attribute where both binary and tertiary geometries are folded into the Binary Connector and distinguished by the **Binary Connector.Tertiary** stem boolean attribute.

Formalization

The identifier in each of the subclasses referring to the superclass identifier

Decorator Subsystem

This subsystem describes the various adornments that can be placed on or in the vicinity of a connector stem. These include text labels and graphic symbol geometry.

Relationship numbering range: R100-R149

(Conflict with Binary Connector! Renumber as R200-R249 in next version)

Class Descriptions

Arrow Symbol

Describes a triangular geometry that can be used to define an arrow head.

Attributes

Name

Same as **Shape Element.Name**

Base

The width of the triangle base

Type: Distance

Height

The height of the triangle

Type: Distance

Stroke

The width and pattern of the border around the triangle

Type: Stroke Style

Fill

Defines the overall look of the Arrowhead as either a hollow arrow (border as a closed triangle), solid arrow (solid fill triangle) or open (v-shape with no base line drawn)

Type: Hollow_Solid_Open `:: [ hollow | solid | open ]`

Identifiers

Name

Unique across all Shape ELements

Circle Symbol

Describes a circular geometry. These appear on the initial and final transitions on state diagrams, for example.

Attributes

Name

Same as **Simple Symbol.Name**

Radius

The radius of the circle

Type: Distance

Solid

Whether or not the circle is filled

Type: Boolean

Identifiers

1. **Name**

Compound Symbol

As the name suggests, a Compound Symbol is built up from multiple Simple Symbols stacked together in some arrangement.

Attributes

Name

Same as **Symbol.Name**

Stroke

The border line width, pattern and color

Type: Stroke style

Identifiers

1. **Name**

Cross Symbol

Describes a line drawn at an angle to the Stem. These appear on Shlaer-Mellor superclass stems, for example.

Attributes

Name

Same as **Simple Symbol.Name**

Width

The length of the crossing line segment

Type: Distance

Angle

Angle relative to the Stem axis. 90 degrees yields a cross normal to the Stem.

Type: Degrees

Identifiers

1. **Name**

Decoration

Any notational element, graphical or textual that adorns the vicinity of a Stem is a Decoration.

Attributes

Name

In the case of Annotation the name is the same text that is drawn for the notation. So the name could be: `0..1` or `{ disjoint, complete }`

For a Symbol, the name is purely descriptive such as double solid arrow or small solid circle.

It is important not to use a model semantic name such as initial psuedo state since the symbol might be used with other notations with other meanings. So it is safest to stick to a description of appearance.

Type: Name

Size

When drawn, this is the total rectangular area consumed. This may be useful for detecting and avoiding overlapping drawn elements.

Type: Rect Size

Identifiers

1. **Name**

Decorations are named uniquely by policy.

Label

A fixed text annotation drawn next to a Stem. On a class diagram, these could be standard UML labels such as `1..*` or a tag like`{ disjoint, complete }`. These are not to be confused with variable text such as the name of an event on a state diagram or a relationship such as `R33` on a class diagram.

Attributes

Name

Same as **Decoration.Name**. Since Labels are text, it is convenient to simply make the label content the name of the Label. For example, `0..1` serves as both the name and rendered content of a UML class diagram multiplicity label.

Identifiers

1. **Name**

Simple Symbol

A Simple Symbol is a graphical element that may form all or part of an entire Symbol. It is “simple” in the sense that it is an atomic geometric element.

Attributes

Name

Same as **Symbol.Name**

Stroke

The border line width, pattern and color

Type: Stroke style

Terminal offset

Distance from either end (root/vine) of a Stem. An arrow symbol can be drawn so that it touches the Stem end (0 distance) from a Node face in the case of a root Stem end. Or a cross used in a Shlaer-Mellor superclass might be drawn at some distance from the vine end.

This value is distinct from the **Stack Placement.Offset** which defines an offset between Simple Symbols that are stacked.

Identifiers

1. **Name**

Symbol

A geometric shape such as an arrow drawn at either end of a Stem is a Symbol.

Attributes

Name

Same as **Decoration.Name**. The name can refer to the visual appearance of the Symbol (wide hollow arrow) or to its general usage (gen arrow). Care should be taken to avoid model semantic names such as ‘1 multiplicity’ since the same symbol might be useful for a variety of meanings in different contexts or Diagram Types. A solid arrow, for example, could be used to indicate a method invocation, a state transition or a unit of multiplicity.

Identifiers

1. **Name**

Symbol Stack Placement

This class represents both the inclusion and arrangement of a Simple Symbol in a Compound Symbol.

In the case of a double headed arrow, for example, one Arrow Symbol is drawn at the head of a Stem end and the other behind it. Ordering progresses either along the stem (adjacency), or upward toward the viewer on the z axis (layering).

In the case of an xUML final psuedo state, a solid arrow is drawn at the tip of the Stem end with a large circle after the tip. Then a small solid circle is drawn on top of a large unfilled circle.

Attributes

Position

The order proceeding from the Stem end outward and upward by layer.

Type: Ordinal

Compound symbol

Same as **Compound symbol.Name**

Simple symbol

Same as **Simple Symbol.Name**

Arrange

Whether the next Simple Symbol in the Position sequence will be layered or placed adjacent to this Simple Symbol. If this is the final Simple Symbol in the position sequence, it is simply marked as the last one.

Type: `[ adjacent | layer | last ]`

Offset

This is the distance in the opposite direction of the Stem end relative to the next Simple Symbol, if any, in the position order. For example, one Arrow might be spaced at a certain distance from the adjacent Arrow. Or a circle might be off center when layered on top of another circle. A double hatched cross would have a certain amount of space between the two cross line segments along the Stem axis.

Type: Distance

Identifiers

1. **Position** + **Compound symbol**

Each position within a Compound Symbol is unique. We define positions 1 and 2 within the ‘double solid arrow’ Compound Symbol, for example, where each simple ‘solid arrow’ is rendered.

Relationship Descriptions

R100 / Generalization

Simple Symbol is an **Arrow**, **Circle** or **Cross Symbol**

All of the Stem symbols in supported notations can be created by an arrangement of these Simple Symbol subclasses. Any notation that requires a stem end shape that cannot be created with these elements, will need to extend this generalization (or some subclass) to include the desired element geometry.

Formalization

Name referential attribute in each subclass

R101 / M:Mc-M

Compound Symbol stacks *one or many* **Simple Symbol**

Simple Symbol is stacked in *zero, one or many* **Compound Symbol**

A Simple Symbol is drawn relative to another (or the same) Simple Symbol to form all or part of a Compound Symbol. For example, in a double arrowhead configuration, one arrow is drawn adjacent to the other on a Decorated Stem End. Or, in the case of an xUML final pseudo state, a small solid circle is drawn on top of a larger hollow circle.

The same Simple Symbol can be useful in multiple Compound Symbols. A solid arrow, for example, is useful in both a single and double arrow configuration.

The same Simple Symbol may be used more than once in a Compound Symbol where each usage corresponds to a unique Stack Placement. For example, a solid arrow appears in both the first and second Stack Placement Positions of a double solid arrowhead.

Without any Simple Symbol elements, there would be nothing to draw, so a Compound Symbol requires at least one with at least two Stack Placement positions.

Formalization

Stack Placement association class

R102 / Ordinal

Stack Placement draw order

In a Compound Symbol, each constituent Simple Symbol is drawn in a specific relative location. This order corresponds to a progression along the stem axis or a z-axis toward the viewer.

See the **Stack Placement.Arrange** attribute description for an explanation of which axis applies for a given stacking direction.

Formalization

Position is ordered sequentially using an Ordinal value within each **Compound Shape**

R103 / Generalization

Symbol is a **Compound Symbol** or **Simple Symbol**

A Symbol is either made up of an arrangement of one or more Simple Symbols in various positions or it is just a single Simple Symbol in one position.

Formalization

Name referential attribute in each subclass

R104 / Generalization

Decoration is a **Label** or **Symbol**

In all cases, a Symbol is just a name associated with an icon that can be drawn on the end of a Stem.

There are only two ways to designate the meaning of a Stem end. In some notations, such as Starr class diagramming, hollow and solid arrows indicate multiplicity and conditionality of class model associations. In xUML notation, text labels are used instead. With Shlaer-Mellor notation, arrows are used to indicate multiplicity while a text `c` symbol (conditional) indicates when zero is an option.

Formalization

Name referential attribute in each subclass

Tablet Subsystem

This subsystem constitutes the entire Drawing Domain. It provides a layer of abstraction between the Flatland Model Diagram Layout Application domain and the graphics library.

Relationship numbering range: R1-49

Class Descriptions

Asset

Any given Drawing Type specifies the kinds of text or graphical Elements that might be drawn on it. Each kind of Element is defined as an Asset. A `class name` might be an Asset defined for the drawing type `Executable UML Class Diagram` fore example. A `solid dot` might be an Asset defined for an `Executable UML State Machine Diagram`. The Asset abstraction makes it possible to define the various presentation characteristics such as font size or color fill that can be used to render any associated Elements defined by that Asset. Thus, all `class names` with the same Presentation will be rendered with a common style.

Identifiers

1. **ID**

Singleton value

Attributes

ID

This is a singleton, so any value is fine.

Type: Nominal

Presentation

Same as **Presentation.Name**

Drawing type

Same as **Drawing Type.Name**

Size

The extent of a rectangular drawing area expressed in points.

Type: Rect Size

Asset Presentation

To define the styles that are applied when rendering an Asset, a Presentation is applied. See relationship description for details.

Identifiers

1. **Asset** + **Presentation** + **Drawing type**

Attributes

No non referential attributes

Closed Shape

This is a Shape Element whose line segments form a polygon. As such a fill must be specified in addition to any border style.

Identifiers

1. **ID** + **Layer**

Attributes

No non-referential attributes

Drawing Type

A mix of graphic and text content commonly used together to fashion a particular type of drawing is regarded as a Drawing Type. An example might be a certain kind of model diagram, say an `Executable UML Class Diagram'`. Drawing Types are not limited to model diagrams, however. A frame of headers, footers and a title block for a certain range of sheet sizes is also a Drawing Type. A set of annotations scattered across a sheet with callout lines might also be a Drawing Type.

Multiple Drawing Types might be layered on a Tablet. This makes it possible to create a model diagram with one set of styles (Presentation) combined with a header-footer frame using its own Presentation. And that might even be combined with a layer of annotations using yet another Presentation.

Identifiers

1. Name

Attributes

Name

A name indicating the purpose and size (if there is more than one) of a drawing's style requirements. Examples could be `xUML class diagram` or `xUML class diagram large` or `OS Engineer Frame D` and so on.

Type: Drawing Type Name based on Name system type

Element

A unit of text or graphics placed and rendered on a Layer and styled by an Asset.

Identifiers

1. **ID** + **Layer**

Attributes

ID

A unique identifier

Type: Element ID based on the Nominal system data type

Size

The extent of a rectangular drawing area expressed in points.

Type: Rect Size

Layer

A common feature of many drawing and image editing applications is the ability to stack layers of content along a z axis toward the user's viewpoint. Similarly, the Tablet renders each layer working from the lowest value on the z axis toward the highest. Thus, content on the higher layers may overlap content underneath.

Identifiers

1. Name
2. Z coord

Since there is only one Tablet instance, we can don't need to include the Tablet referential attribute in either identifier.

Attributes

Name

Describes the overall purpose or reason why content should be drawn together at the same level. Examples might be "diagram", "notes", "header footer", etc.

Z coord

This value indicates the z axis ordering with lower numbered values being drawn underneath higher valued numbers. So Level 1 is drawn first working upward. (To be user, rather than programmer friendly, our Tablet z-axis starts at 1 going upward).

Type: Ordinal

Line Segment

This is a Shape Element that connects two points with a straight line.

Identifiers

1. **ID** + **Layer**

Attributes

From

When drawing a Line Segment it is necessary to draw from one position to another on the Tablet. In fact, it makes no difference which direction you draw, so the from/to naming could just as easily be named a/b. But many draw facilities use this terminology, so we adopt it here.

Type: Tablet Coord

To

See **From** attribute description

Type: Tablet Coord

Polygon

A series of line segments that wrap around to form a closed shape.

Identifiers

1. **ID** + **Layer**

Attributes

No non-referential attributes

Presentation

A set of compatible visual styles including fonts, colors, border widths and so forth as appropriate to a given Drawing Type form a selectable Presentation. For example, an `Executable UML State Machine Diagram` might be drawn using certain fonts for state names and possibly different colors for transient and non-transient states. Alternatively, only black and white might be used with purple for a certain kind of connector in a diagnostic Presentation.

Identifiers

1. Name

Attributes

Name

A name describing the overall appearance and purpose of the Presentation such as 'diagnostic connectors' or 'normal' or 'color coded states'.

Type: Presentation Name based on Name system type

Rectangle

A Rectangle is specified with a lower left corner and width/height size. While it is technically a polygon, most drawing facilities provide a separate method for rendering rectangular shapes.

Identifiers

1. **ID** + **Layer**

Attributes

Size

The height and width of the Rectangle on the Tablet.

Type: Rect Size

Lower left

The position of the lower left corner in Tablet coordinates.

Type: Tablet Coord

Shape Element

This is an Element rendered directly using line segments or closed shapes with border widths, fill colors and so forth.

Identifiers

1. **ID** + **Layer**

Attributes

No non referential attributes.

Tablet

A virtual drawing surface serving as a proxy for the drawing area provided by the graphics library.

Identifiers

1. **ID**

Attributes

ID

This is a singleton, so any value is fine.

Type: Singleton

Size

The extent of a rectangular drawing area expressed in points.

Type: Rect Size

Text Element

This is an Element rendered using text facilities and styles rather than directly as line segments.

Identifiers

1. **ID** + **Layer**

Attributes

Content

The text to be rendered

Type: Text

Lower left

The lower left corner of the text area.

Type: Tablet Coord

Vertex

A point in a Polygon where two line segments meet to form a corner.

Identifiers

1. **Polygon** + **Layer** + **Position**

Note that no two vertices may overlap on the same Layer. This ‘safety’ feature prevents undesirable drawing artifact.

Attributes

Position

The location of the Vertex on the Layer

Type: Tablet Coord

Relationship Descriptions

OR20 / Ordinal

Layer is rendered above

Layers are rendered in ascending Z coord order. Since each Layer is at its own dedicated Z coord and all Layer instances are on the same (only) Tablet instance, there is a total ordering on all Layer instances.

Formalization

Layer.Z coord

R1 / 1:M

Drawing Type appears as defined by *many* **Presentation**

Presentation defines appearance of *one* **Drawing Type**

We initially define a single Presentation for each Drawing Type. For an `Executable UML Class Diagram`, for example, we might use only black for fonts and borders, choose a basic font with one size for class names and another for attributes, methods, relationship names and so forth: easy to read, but nothing special. Later, we might decide that it would be nice to use dashed lines and light green for imported class node types. Eventually, there might be a choice of several Presentations that could be selected to display an `Executable UML Class Diagram`.

A Presentation is specific to a given Diagram Type since it applies styles to particular Node and Connector Types relevant to its Diagram Type.

Formalization

Presentation.Drawing type -> Drawing type.Name

R2 / 1:Mc

Presentation styles content of *zero, one or many* **Layer**

Layer content is styled by *one* **Presentation**

When a Layer is defined, it must be associated with a Presentation. Then, when a given Asset is drawn on that Layer it is known what graphic and text styles to apply. A Layer that draws a title block, for example, will apply an appropriate Presentation for that while another Layer for a model diagram would use a different Presentation.

It is also possible to split a diagram to multiple layers so that the content of one part of the diagram might be styled differently than content on another part. For example, you might want separate Presentations (colors perhaps) for adjoining subsystems.

Formalization

Layer.Presentation -> Presentation.Name

Layer.Drawing type -> Presentation.Drawing type

R4 / M:M-1

Presentation stylizes *one or many* **Asset**

Asset is styled by *one or many* **Presentation**

To draw an Element on a Layer, it is necessary to use the set of styles defined for the Element's Asset. So, let's say that you want to draw a `class compartment` Asset on an `xUML class diagram` Drawing Type using the `default` Presentation. You simply specify the Asset and Layer. The Presentation defined for that Layer will be associated with your Asset and then provide a set of styles (line width, line style, fill color) necessary to draw your Asset (a filled rectangle in this case).

By definition, a Presentation defines a set of styles to be applied to all the Assets defined for the Presentation's Drawing Type. Since a Drawing Type must included at least one Asset, there must be at least one in every Presentation.

A given Asset cannot be drawn unless it has the relevant styles defined. Since a Drawing Type must specify at least one Presentation, there must be at least one Presentation for each Asset.

Formalization

Asset Presentation.Asset -> Asset.Name

Asset Presentation.Presentation -> Presentation.Name

Drawing type -> Presentation.Drawing type

Drawing type -> Asset.Drawing type

Constraint: Note that both the Asset and Presentation must be defined for the same Drawing type, formalized by the merged Drawing type referential attribute.

R12 / Generalization

Shape Element is a **Line Segment**or **Closed Shape**

Shape Elements all required line style specifications. Closed Shapes additionally require a fill specification.

Formalization

<subclass>.ID -> Shape Element.ID

<subclass>.Layer -> Shape Element.Layer

R13 / 1:M

Tablet organizes content on z axis with *one or many* **Layer**

Layer holds z axis coordinate content of *one* **Tablet**

A Layer is a horizontal slice of content at some position on the z axis. To draw an Element on a Tablet, the Element must be placed on some Layer. To be useful then, a Tablet must be created with an initial Layer. More can be added later.

Since there is only one Tablet instance, all Layers present content for that Tablet.

Formalization

Layer.Tablet -> Tablet.ID

R15 / Generalization

Element is a **Shape**or **Text Line**

Each Element is drawn somewhere on a single Layer. Effectively, then, an Element has an x,y,z position with z being the Layer and x,y being a position on that layer. The x,y might be a lower left corner or an end point depending on whether the Shape is open or closed.

Since the applicable styles differ for Shapes (open and closed) and lines of text (Text Lines), Elements are classified accordingly.

Formalization

<subclass>.ID -> Element.ID

R19 / 1:Mc

Element is drawn on *one* **Layer**

Layer draws *zero, one or many* **Element**

To organize and layer all graphical content, each Element is placed on a specific Layer when it is created.

A blank Layer isn't much use since it won't render any content. That said, it may be necessary to create one without knowing in advance whether or not any content will be placed on it. Since blank Layers are harmless, there is no reason to preclude them.

Formalization

Element.Layer -> Layer.Name

R22 / Generalization

Closed Shape is a **Rectangle**or **Polygon**

Here we distinguish based on how the rendering of a shape is specified. In the case of a simple Rectangle there is generally a direct facility for drawing one using a corner and size. But in the case of a Polygon, especially one that is irregular, a set of Vertex positions must be specified.

Both types of Closed Shape require a fill specification.

Formalization

<subclass>.ID -> Closed Shape.ID

<subclass>.Layer -> Closed Shape.Layer