Why Flatland?

You are done pixel pushing. Nicely laid out diagrams from model markup.

Introducing a model markup language that lets you provide essential layout cues to get the diagrams you want without all the pixel pushing. And by putting all your model data in a human readable text file, you can diff and use normal configuration management. Problem solved. Best of all, it's all open source.

It is expressly designed for Executable UML, but can be extended to other diagram types.

How does it work?

Nodes are laid out using a flexible grid system like a spreadsheet. Each cell of the grid contains at most one node, so you can never overlap nodes. A node can be contained entirely within a cell or it can span multiple cells. For each node, you specify:

1. The cell to assign (R3, C4) for example
2. Or cell range (R3-R5, C4) (to span above or below multiple nodes)
3. Default alignment is center, but you can specify top, bottom, left or right

The diagram type you specify determines the notation used for the nodes (round corners, number of UML style text compartments, thickness of borders, fonts and font sizes. If you specify xuml class diagram, for example, you get square corners, three compartments per node, and so forth.

The diagram type also specifies the connector styles.

To place a connector, specify the node or nodes to connect, the face on each node, the relative position on each face (default is center 0) or an integer such as 1, 2, -2, to indicate distance from the center up down or left right. You may also need to specify the decorator to place on each end of the connector, depending on connector type. For example, a generalization might specify four nodes with the bottom of one as the generalization and the top of the other three nodes.

So here's the problem. You have some sort of model drawing that has more than a handful of elements, say 10+nodes and connections. Next question is, do you care about how they are laid out? For example, is something like this:

<tangled mess>

less preferable to something like this:

<nicely laid out>

When you only have a handful of elements, maybe you don't care. But if you do industrial scale modeling, you want control of the layout. If you can't control the layout of your complex diagrams, you lose the benefit of diagramming which is to make relationships clear. But it's not just about simplifying the diagram or making it pretty. It's about expressing your design strategy.

If you are using a monolithic modeling environment such as Enterprise Architect, No Magic, BridgePoint or Simulink, you are out of luck. You just have to work with the draw editor they give you if you want to see your models transformed into something useful. Happy pixel pushing!

For those of us that use a flexible tool chain (where any draw editor can be plugged in) as described in Models to Code, we have more options. Unfortunately, we are left with the choice of pixel pushing so we have control over our layout OR we can use a layout engine like PlantUML which controls they layout for you, typically with dismal results (if you have more than a handful of elements).

For quite some time I have been using a combination of the two approaches. PlantUML for sequence diagrams (it does a nice job) and you have reasonably good control over the layout. That's because you are systematically moving horizontally or vertically in steps, so it's not too difficult to get it right.

But

The Flatland Draw Engine

So let's say that you want to build a detailed model and lay it out nicely. You have a few choices.

First, you can use a model development environment that suits you such as Magic Draw or Simulink or Enterprise Architect or BridgePoint (xtUML). We lump these into the Monolithic Tool category. Here the idea is that you have a complete model development environment that takes care of your every need.

There are a number of downsides to this approach. You might like the code generation, simulation, diagnostic and other properties of the tool, but not the model editor.

Alternatively, you can use a more freeform tool such as Umlet, LucidChart or, my favorite, Omnigraffle (only on the mac). Advantage is that you can draw pretty much anything you want and have total control over the layout. Downside, is that you will need to do some conversion to feed your model into the other tools.

Our approach in Models to Code is that you break away entirely from the Monolithic tool and get yourself a, preferably, open source tool chain. You get flexibility on modifying the building blocks, but you need to know what you are doing.

How it works

We'll show the process of encoding and extracting layout information in an xUML model file. If you want to use this with non-xUML models and your own model file format, you can easily define your own commands and extractor. You can use the xUML example as a template.

The xUML model file format looks like this:

(simple model example, with non-model stuff left out)

In this file we know what all the model elements are, what models they belong to and what's connected to what.

What we don't know is where the nodes and connectors go, so we will insert that information, and that information only. A model populator will ignore the layout information and populate a metamodel or otherwise process the non-graphical information. Once that's done, the layout extractor will match the layout info with the corresponding elements in the metamodel. Then a layout program can be generated as input to the layout engine.

The pipeline looks like this:

model file | metamodel pop

model file, metamodel pop | layout extractor | layout plan

layout plan | draw engine

You model must be encoded as a text file. For a given model file text format, you need to define a couple of commands that will encode your layout information. When the model file is processed, the layout information is extracted and combined with the model elements you want to layout.

So let's say you have a file format like this:

The layout commands you need to add are these:

```
Node (row span, col span) [valign alignment] [ halign align]
Connector (side, align) (side, align)
```

Now you will need to define a text format for expressing each of these commands that won't be confused with your model information.

And you will need to modify your model file processor so that it binds the layout information with each node and connector element. Your file processor must output commands recognized by your model layout engine. Whatever diagram, node and connector types you specified in your layout engine must be matched with the layout commands.

Here's an example of the output you might produce:

You need to find a way to specify this information such as:

```
@N (R1, C2-C4) valign left, halign +2
```

And you need to ensure that when the file is scanned it will be possible to figure out which command is associated with which model element (such as placing on the next line after the model element is declared)

Model Markup

Nodes

```
[type], rows, columns, [alignment]
rels
R1:Face

state, R1, C7, from
```

Connectors

```
type, faces,
```

Flatland3 Domain

Binary Connector Subsystem Description

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-R150

Flatland3 Domain Data Types

Author

Leon Starr

Document

mint.flatland3.td.1b / Version 0.8.0 / February 2, 2020

Data Type Descriptions

All data types used in the Flatland3 Domain class models are described here.

Document organization

This document is organized into three sections. The first section lists all supplied (base) types that are used in this domain. The second section defines general purpose data types which are common across many domains and are used in this domain. General purpose data types inherit from the base types. The third section describes data types crafted specifically for this domain. These data types inherit from either general purpose data types or base types directly.

limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Supplied Data Types

A supplied data type is defined for the xUML modeling language and is implemented by the Model Execution environment. Here we list only those supplied data types referred to directly in the local domain by the class model and inside of any activities.

Boolean

This is a base type.

Operations

All boolean operations

set / unset

```
set()::Boolean
```

Sets the value to true and returns it

```
unset()::Boolean
```

Sets the value to false and returns it

Initialization

```
init()::Boolean
```

Initializes the value to false and returns it

Nominal

A natural number representing a name for the purpose of establishing uniqueness. Since this type is just for naming, no relative comparison other than equality or math operations are relevant.

Base

Integer[1..+]

Units

None

Operations

Inherit assignment and equality comparison only

Initialization

```
init(c: Class Name)::Nominal
```

Assigns the next available unique number across the **c** population given by `c( i1:v1, i2:v2, ...)` where i<n> is the set if identifier attributes representing the scope of uniqueness excluding the typed attribute and v<n> are the values restricting that scope. (It would be nice to have a more formal description of this idea). Informally, we take the identifier of the class **c** minus the attribute being typed Nominal and project on the typed attribute. We then assign a value not in that set.

Ordinal

A number assigned for the purpose of imposing sequential order. It is often used instead of a reflexive association for this purpose.

Base

Integer[1..+]

Units

None

Operations

Inherit assignment, subtraction, addition and compare only

first

```
first( c : Class Name )::Ordinal
```

Assigns the lowest ordered value and returns it. The values of all other instances are incremented by one.

last

```
last( c : Class Name )::Ordinal
```

Adds one to the highest value for all instances, assigns and returns it. In other words, assigns the next number in sequence across the **c** population given by `c( i1:v1, i2:v2, ...)` where i<n> is the set if identifier attributes representing the scope of uniqueness excluding the typed attribute and v<n> are the values restricting that scope. (It would be nice to have a more formal description of this idea). Informally, we take the identifier of the class **c** minus the attribute being typed Ordinal and project on the typed attribute. We then assign the next number in sequence.

insert

```
insert( c : Class Name, o : Ordinal )::Ordinal
```

Set the value to `o` and increment all higher numbered instances by one. (The first and last operations can be performed by insert, but are provided for convenience).

remove

```
remove( o : Ordinal )
```

Renumbers all higher numbered instances within the identifier limited population (see description under init operation). This operation should be called when the associated instance is deleted.

Initialization

The last() operation is called by default on the local class.

Deletion

The remove() operation is called.

distance

```
distance( o : Ordinal)::Positive Integer
```

Returns `abs( value – i )`, the absolute difference between two ordinal values. (Is there a more appropriate mathematical term such as reach, span, degree, separation? Otherwise, could be confused with physical distance.)

least

```
least()::Ordinal
```

Returns the lowest ordered value that can possibly be assigned to any Ordinal typed element. If we start counting from 1, as is currently defined for Ordinals, the value 1 will always be returned.

General Purpose Data Types

A general purpose data type is built upon a base data type or some other general purpose data type. By "general purpose" we mean that the data type has potential use across multiple modeled domains. It may, therefore, make sense to build the definition of a general purpose data type directly into the model execution environment. Here, we assume not, and require its definition here.

Each non-base data type designates some `init()::<type>`

Name

A series of 1 to 80 alphanumeric and characters, space delimited, no other whitespace characters. No leading spaces, no more than one space in sequence, no ending space.

Base

String[1..80] / / I forget the regex for this!

Operations

Inherit assignment and equality comparison only

Initialization

```
init()::Name
```

Sets the data type's designated default value and returns it

Count

A natural number representing a counted quantity such as the number of times that some event has occurred. A Count may be incremented, decremented or reset.

Base

Integer[0..+]

Units

None

Operations

Inherit Integer assignment and comparison only

incr()::Count, ++ prefix operator

Increments by one returning the updated value

decr()::Count, – prefix operator

Decrements by one if value is > 0. returning the updated value

reset()::Count

Sets value to zero returning zero

Initialization

Calls the reset operation.

Domain Specific Data Types

A domain specific data type is intended to be of use only within the local domain.

Direction

Enumeration [**up** | **down**]

Distance

A physical distance

Base

Rational

Units

Meters

Range

[0..800]

Operations

Inherit addition and multiplication

Multiply (override)

Cannot multiply by a negative number

separation

```
separation( d : Distance )::Distance
```

Subtracts one distance from another and returns the absolute difference representing the separation between two locations

Level Name

A short name such as **Lobby**, **L**, **3**, **48**, **P1**, **P2**, etc as you might see on the label of a button inside of a Cabin.

Base

String[1..14]

Operations

Inherit assignment and equality comparison only

Speed

The vertical speed of a Cabin

Base

Rational

Units

Meters per sec

Range

[0..100]

Operations

Inherit all except subtract

Subtract (override)

If result is negative return absolute value

Multiply (override)

Cannot multiply by a negative number

Class Descriptions

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**

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**

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**

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**

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**

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**

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.

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**

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)

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

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 attache 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

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

Relationship Descriptions

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

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

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

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

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

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

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

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.

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 Bi-

nary 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.

Flatland3 Domain

Node Subsystem Description

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-R50

Flatland3 Domain Data Types

Author

Leon Starr

Document

mint.flatland3.td.1b / Version 0.8.0 / February 2, 2020

Data Type Descriptions

All data types used in the Flatland3 Domain class models are described here.

Document organization

This document is organized into three sections. The first section lists all supplied (base) types that are used in this domain. The second section defines general purpose data types which are common across many domains and are used in this domain. General purpose data types inherit from the base types. The third section describes data types crafted specifically for this domain. These data types inherit from either general purpose data types or base types directly.

limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Supplied Data Types

A supplied data type is defined for the xUML modeling language and is implemented by the Model Execution environment. Here we list only those supplied data types referred to directly in the local domain by the class model and inside of any activities.

Boolean

This is a base type.

Operations

All boolean operations

set / unset

```
set()::Boolean
```

Sets the value to true and returns it

```
unset()::Boolean
```

Sets the value to false and returns it

Initialization

```
init()::Boolean
```

Initializes the value to false and returns it

Nominal

A natural number representing a name for the purpose of establishing uniqueness. Since this type is just for naming, no relative comparison other than equality or math operations are relevant.

Base

Integer[1..+]

Units

None

Operations

Inherit assignment and equality comparison only

Initialization

```
init(c: Class Name)::Nominal
```

Assigns the next available unique number across the **c** population given by `c( i1:v1, i2:v2, ...)` where i<n> is the set if identifier attributes representing the scope of uniqueness excluding the typed attribute and v<n> are the values restricting that scope. (It would be nice to have a more formal description of this idea). Informally, we take the identifier of the class **c** minus the attribute being typed Nominal and project on the typed attribute. We then assign a value not in that set.

Ordinal

A number assigned for the purpose of imposing sequential order. It is often used instead of a reflexive association for this purpose.

Base

Integer[1..+]

Units

None

Operations

Inherit assignment, subtraction, addition and compare only

first

```
first( c : Class Name )::Ordinal
```

Assigns the lowest ordered value and returns it. The values of all other instances are incremented by one.

last

```
last( c : Class Name )::Ordinal
```

Adds one to the highest value for all instances, assigns and returns it. In other words, assigns the next number in sequence across the **c** population given by `c( i1:v1, i2:v2, ...)` where i<n> is the set if identifier attributes representing the scope of uniqueness excluding the typed attribute and v<n> are the values restricting that scope. (It would be nice to have a more formal description of this idea). Informally, we take the identifier of the class **c** minus the attribute being typed Ordinal and project on the typed attribute. We then assign the next number in sequence.

insert

```
insert( c : Class Name, o : Ordinal )::Ordinal
```

Set the value to `o` and increment all higher numbered instances by one. (The first and last operations can be performed by insert, but are provided for convenience).

remove

```
remove( o : Ordinal )
```

Renumbers all higher numbered instances within the identifier limited population (see description under init operation). This operation should be called when the associated instance is deleted.

Initialization

The last() operation is called by default on the local class.

Deletion

The remove() operation is called.

distance

```
distance( o : Ordinal)::Positive Integer
```

Returns `abs( value – i )`, the absolute difference between two ordinal values. (Is there a more appropriate mathematical term such as reach, span, degree, separation? Otherwise, could be confused with physical distance.)

least

```
least()::Ordinal
```

Returns the lowest ordered value that can possibly be assigned to any Ordinal typed element. If we start counting from 1, as is currently defined for Ordinals, the value 1 will always be returned.

General Purpose Data Types

A general purpose data type is built upon a base data type or some other general purpose data type. By "general purpose" we mean that the data type has potential use across multiple modeled domains. It may, therefore, make sense to build the definition of a general purpose data type directly into the model execution environment. Here, we assume not, and require its definition here.

Each non-base data type designates some `init()::<type>`

Name

A series of 1 to 80 alphanumeric and characters, space delimited, no other whitespace characters. No leading spaces, no more than one space in sequence, no ending space.

Base

String[1..80] / / I forget the regex for this!

Operations

Inherit assignment and equality comparison only

Initialization

```
init()::Name
```

Sets the data type's designated default value and returns it

Count

A natural number representing a counted quantity such as the number of times that some event has occurred. A Count may be incremented, decremented or reset.

Base

Integer[0..+]

Units

None

Operations

Inherit Integer assignment and comparison only

incr()::Count, ++ prefix operator

Increments by one returning the updated value

decr()::Count, – prefix operator

Decrements by one if value is > 0. returning the updated value

reset()::Count

Sets value to zero returning zero

Initialization

Calls the reset operation.

Domain Specific Data Types

A domain specific data type is intended to be of use only within the local domain.

Direction

Enumeration [**up** | **down**]

Distance

A physical distance

Base

Rational

Units

Meters

Range

[0..800]

Operations

Inherit addition and multiplication

Multiply (override)

Cannot multiply by a negative number

separation

```
separation( d : Distance )::Distance
```

Subtracts one distance from another and returns the absolute difference representing the separation between two locations

Level Name

A short name such as **Lobby**, **L**, **3**, **48**, **P1**, **P2**, etc as you might see on the label of a button inside of a Cabin.

Base

String[1..14]

Operations

Inherit assignment and equality comparison only

Speed

The vertical speed of a Cabin

Base

Rational

Units

Meters per sec

Range

[0..100]

Operations

Inherit all except subtract

Subtract (override)

If result is negative return absolute value

Multiply (override)

Cannot multiply by a negative number

Class Descriptions

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

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

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

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**

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.

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.

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

Flatland3 Domain

Node Subsystem Description

Description…

Flatland3 Domain Data Types

Author

Leon Starr

Document

mint.flatland3.td.2a / Version 0.4.0 / February 2, 2020

Data Type Descriptions

All data types used in the Flatland3 Domain class models are described here.

Document organization

This document is organized into three sections. The first section lists all supplied (base) types that are used in this domain. The second section defines general purpose data types which are common across many domains and are used in this domain. General purpose data types inherit from the base types. The third section describes data types crafted specifically for this domain. These data types inherit from either general purpose data types or base types directly.

limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Domain Specific Data Types

A domain specific data type is intended to be of use only within the local domain.

Stroke Style

Composition of: `Line width::integer > 0, Pattern::Pattern anme`

Pattern Name

`[ dashed | solid ]`

Class Descriptions

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.

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

Diagram type

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

Notation

Same as **Stem Notation.Notation** constrained to mach **/R53/Connector/R70/Diagram.Notation**

Node

Same as **Node.ID**

Face

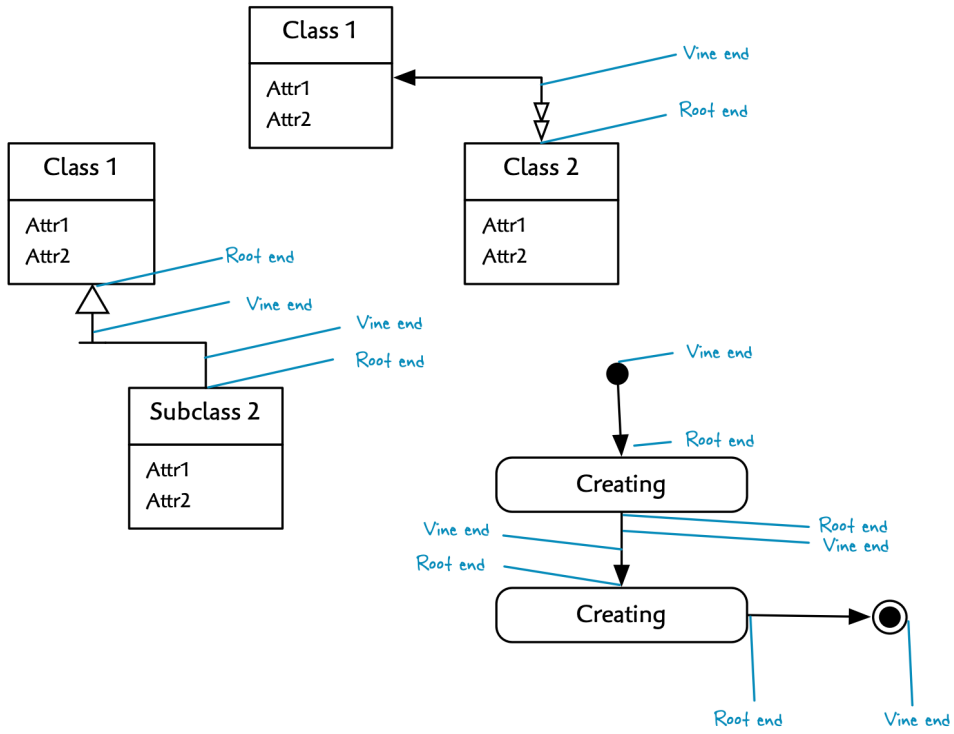
The side of the Node where the Stem is anchored.

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.

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.

Decorated Stem End

Either end of a Stem (root or vine) may be decorated with one or more Symbols. A state transition on a state machine diagram, for example, connects two state nodes. Each node interface results in a Stem. In this example, only the Stem on the target state is decorated. Here a solid arrow symbol is used to decorate the root end (where it attaches to the node).

Only those stem ends that are decorated are included here. In our example, the stem end attached to the from state is just a line with no decoration on either the root or vine ends and, therefore, has no decorated stem ends.

See R55 description for more details.

Attributes

Stem type

Stem Type.Name

Semantic

Stem Notation.Semantic

Diagram type

Stem Notation.Diagram type and **Stem Type.Diagram type**

Notation

Stem Notation.Notation

End

A Stem has two ends, root and vine. Either, both or neither end may be classified as a Stem Decoration. The root end is always attached to a Node face while the vine end sprouts in the opposite direction standing free as in the case of an xUML state machine diagram initial or final pseudo state or hooking onto the Connector at some point. When it stands free, it usually is decorated with some Stem Notation. When it hooks, it may or may not be. In the case of a Starr class diagram association class Stem Type, it hooks onto its Connector branch and carries an arrow symbol. In the case of an xUML class diagram association class Stem Type, there is no symbol on the vine end.

Type: Stem End `:: [ root | vine ]`

Clearance

A Decorated Stem End must be at least a certain length otherwise, the shape decoration will overlap with a bend or connect too close to the opposing Stem on a nearby Node face.

Clearance represents the minimum length of a Stem End used in a Connector Type with a non-unary geometry. For a root end, the distance is from its attached Node face to the other end of the Stem End. It should always be longer than the width of any combined shape symbols. Thus, it is the distance neces-

sary to clear the shapes and provide a minimum connecting line before any bends or joins with an opposing Stem end can occur. For a vine end, it is measured from the end of the Stem away from the attached node face back toward the root end of the same Stem.

For a unary stem such as the xUML initial state transition, the clearance will be zero and disregarded. Instead, a special value in the Connector Layout will be use instead. (Yeah, it's a bit of a model hack, but we'll live with it for now).

It is important to set this value so that it is longer than the combined length of all of the decoration symbols and leaves enough room for any text symbols as well. The value can be computed by a bit of trial and error, but once established, no bend is allowed inside this length.

Type: Distance

Identifiers

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

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

Stem Decoration

This is a Symbol used to illustrate the meaning of a Decorated Stem End.

See R58 description for more details.

Attributes

Stem type

Same as **Decorated Stem End.Stem type**

Semantic

Same as **Decorated Stem End.Semantic**

Diagram type

Same as **Decorated Stem End.Diagram type**

Notation

Same as **Decorated Stem End.Notation**

Symbol

Same as **Symbol.Name**

Identifiers

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

Consequence of a many-many association

Text Symbol

The text drawn next to a Stem End. See R57 for more details.

Attributes

Name

Same as **Symbol.Name**

Identifiers

Name

All Symbols are named uniquely across Shape and Text Symbols

Symbol

This is a predefined text element drawn adjacent to a Decorated Stem End or a graphical element drawn on the stem.

Attributes

Name

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

For a Shape 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 purely geometric description.

Type: Name

Identifiers

Name

All Symbols are named uniquely across all Symbol types.

Shape Symbol

Describes the geometric and style properties of a graphical symbol applied to a Decorated Stem End.

Attributes

Name

Same as **Symbol.Name**

Identifiers

Name

Shape Element

This is a component that defines all or part of a Shape Symbol. For example, a single solid arrowhead is an element of a double solid arrowhead which is used in the Starr notation to represent one-many multiplicity on a class diagram.

Attributes

Name

A name that describes the shape geometry such as small solid arrow or large bounded circle. These names are purely geometric and do not describe notational usage since the same element might be useful in different notations to mean different things.

Type: Name

Identifiers

Name

Stack Method

A Simple Shape is either drawn next to another or on top of it as determined by the value of the Arrange attribute.

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 (the first circle always drawn after the tip). Then a small solid circle is drawn on top of a large unfilled circle. (Circles are always stacked)

Attributes

Compound shape

Same as **Shape Draw Order.Compound shape**

Stack position

Same as **Shape Draw Order.Position**

Next position

Same as **Shape Draw Order.Position** constrained such that the value is equal to the instance's **Stack position** + 1.

This constraint ensures that the order established by **OR62** is used and that a Stack Method only exists between two existing **Shape Draw Order** instances.

Arrange

The order in which Simple Shape Symbols are to be drawn. If the direction is `below`, the next symbol is drawn on top, along the z-axis. If the direction is `next_to`, the next symbol is drawn adjacent to the current shape away from the stem end down the stem's axis, either x or y.

Type: Stack Direction `:: [ below | next_to ]`

Identifiers

1. **Compound shape** + **Stack position**
2. **Compound shape** + **Next position**

Shape Draw Order

This class represents both the inclusion of a Shape Element in a Shape Symbol as well as its draw order.

In the case of a double headed arrow, for example, one Arrowhead Shape is drawn at the head of a Stem end and the other behind it. Ordering progresses either down the stem, from its end or upward (toward the viewer) on the z axis.

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.

In both cases the **Stack Method.Arrange** attribute determines whether symbols are drawn adjacent or on top of one another.

Attributes

Position

The order in which Simple Shape Symbols are to be drawn.

Type: Ordinal

Compound shape

Same as **Compound shape.Name**

Stacked shape element

Same as **Simple Shape Symbol.Name**

Identifiers

Position + **Compound shape**

Since the same **Stacked Shape Element**, say a solid arrowhead, may be used in multiple positions within the same **Compound Shape Symbol**, the **Position**, which is numbered sequentially per **Compound Shape** is used.

Arrowhead Shape

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 Shape

Describes a circular geometry that can be used to define a lollipop style terminator.

Attributes

Name

Same as **Shape Element.Name**

Radius

The radius of the circle

Type: Distance

Solid

Whether or not the circle is filled

Type: Boolean

Stroke

The width and pattern of the circle perimeter

Type: Stroke Style

Identifiers

Name

Unique across all Shape Elements

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 (U) – 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 (B) – 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.

Ternary (T) – This is a binary relationship with an additional Stem connecting to the line between the binary relationship Stems. A typical example is a class diagram association class where a third Stem emanates from the association class Node connecting to the line between the binary relationship Stems.

Hierarchical (H) – 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`:: [ U B T H ]`

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.

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

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

Undecorated stem clearance

For a Stem that has no graphic decoration, such as an xUML class

binary association connection or a xUML subclass connection, this is the minimum distance from the node face to either a bend or the opposing Stem end. It prevents a bend too close to a Node face or a connection too close to another Node.

Type: Distance

Default unary branch length

The length of a unary geometry Stem from its Node face to

the beginning of any graphic decoration on the Vine (free) end of the Stem.

Type: Distance

Identifiers

Name

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.

Hierarchy Connector

This type of Connector is drawn like a tree with one primary Stem and two or more subsidiary Stems. A generalization relationship on a class diagram is one such example.

Attributes

ID

Same as **Connector.ID**

Identifiers

ID

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

Stem Type

A line drawn a short distance outward from a Node. In a unary Connector Type it will not connect to anything else and in any other connection geometry it will connect to one or more other Stems. This specification may define a Stem Symbol to be drawn at either or both ends of a Stem.

For example, an xUML class Stem Type defines a short line emanating from an xUML class diagram class node which can be decorated according to the given notation. It represents only one end of the entire Connector adjacent to the attached Node.

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**

Identifiers

Name

Unique by policy

Stem Type Style

A special line style (other than solid, thin or whatever default is in place) applied to a Stem Type for a given Diagram Notation. See R59 for more details.

Attributes

Stem type

Same as **Stem Type.Name**

Diagram type

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

Notation

Same as **Diagram Notation.Notation**

Stroke

The special line style applied to this Stem Type

Type: Stroke Style

Identifiers

Stem type + **Diagram type**

Consequence of 1c:Mc association where identifier is taken from reference to the many side

Stem Type Usage

This is a Stem Type used by a given Connector Type. See R54 for a complete description.

Attributes

Connector type

Same as **Connector Type.Name**

Diagram type

Same as **Diagram Type.Name**

Stem type

Same as **Stem Type.Name**

Identifiers

Connector type + **Diagram type** + **Stem type**

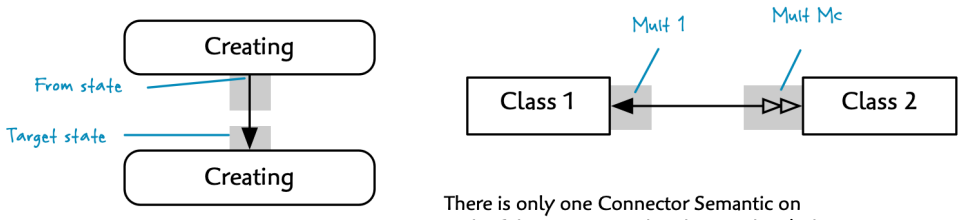
Consequence of many-to-many association where both Connector and Stem type are defined on the same Diagram Type.

Connector Semantic

A Connector Semantic is some notation independent meaning that can be attributed to either end of a Stem. When combined with a 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).

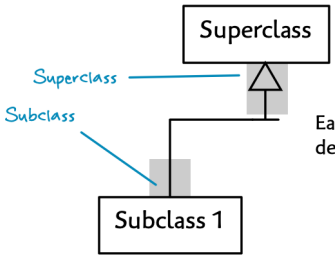
Four Stems on non-unary (binary) Connectors



There is only one Connector Semantic on each of these Stems and each is on the Node interface (Root end)

In a given Notation, a Connector Semantic may or may not require a special symbol. The from state semantic, for example, is just an undecorated line in xUML. Subclasses in xUML, Starr and Shlaer-Mellor class diagrams are similarly un-notated.

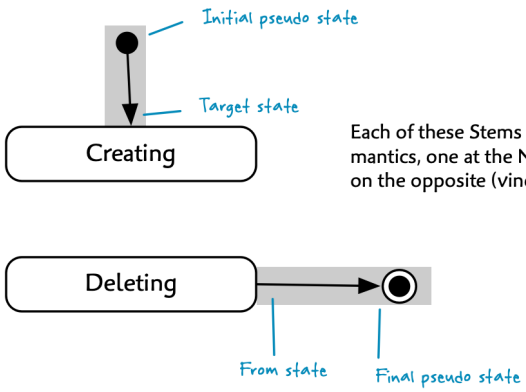
In a given Notation, not all Connector Semantics require symbolic representation.



Each Stem here has a Connector Semantic, but only one is decorated with a symbol (in xUML notation)

In some cases, the Stem end opposite the Node interface 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 Connector with a Unary geometry.

Two Stems on Unary Connectors



On a state machine diagram, for example, the line that touches a state Node is terminated with an arrow to indicate a target state. The opposite end does not; a line is simply drawn to the opposing Stem whose Node end which indicates the from state.

Attributes

Name

A name that reflects the meaning (semantic) of the Stem termination such as “target state” (goes to this state) or “Mc” (many conditional) or “final psuedo-state”.

Type: Name

Identifiers

Name

Unique by policy

Stem Notation

A Stem Semantic such as Mc multiplicity is assigned a specific representation for a combination of Diagram Type and Notation such as Class Diagram/xUML or Class Diagram/Shlaer-Mellor. The notation for Mc multiplicity in each of these two example Diagram Notations constitutes a distinct Stem Notation, `0..*` vs. single arrowhead and a `c` symbol, for example.

See R60 description for more details.

Attributes

Semantic

Same as **Stem Semantic.Name**

Diagram type

Same as **Diagram Notation.Diagram type**

Notation

Same as **Diagram Notation.Notation**

Identifiers

Stem semantic + **Diagram type** + **Notation**

From association multiplicity

Relationship Descriptions

R60 / 1:Mc-M

Diagram Notation defines representation of *zero, one or many* **Stem Semantic**

Stem Semantic is represented in *one* **Diagram Notation**

A Diagram Notation may supply some kind of notation for any Stem Semantic with relevance in the associated Diagram Type.

For example, the xUML class Diagram Notation designates a `1..*` symbol to indicate one-to-many multiplicity. The Starr class Diagram Notation, on the other hand, represents the same semantic with a pair of solid arrows and no text label.

In some cases a Stem Semantic may be relevant to a given Diagram Type, yet have no notation. For example, in the class diagram Diagram Type, the connection to a subclass Node is relevant, but no notation is necessary. Only the superclass Node is notated with all other connected Nodes understood to be subclasses. In such a case, there is no need for a Connector Symbol for the subclass Stems.

Formalization

Stem Notation association class

R61 / Generalization

Shape Element is an **Arrowhead Shape** or **Circle Shape**

All of the Stem symbols in considered notations can be created by a combination of circle and arrowhead shapes. Any notation that requires a stem end shape that cannot be created with these elements, will need to extend this subclass to include the desired element geometry.

Formalization

Name referential attribute in each subclass

R67 / Generalization

Stem is a **Floating Binary** 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.

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.

Formalization

ID + Connector in Floating Binary Stem or **ID + Connector + Stem type + Node + Face + Anchor position** in the Anchored Stem. 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.

R65 / Generalization

Anchored Stem is a **Binary Stem**, **Tertiary Stem**, **Root Stem**, **Branch Stem** or **Free Stem**

The anatomy and constraints of each Connector geometry is determined by the permitted quantity and combination of various Stem positions. A hierarchical geometry, for example, consists of one Root Stem and one or more Branch Stems. A unary geometry consists of a single Free Stem.

Since most Stems are anchored, this is a subclassification of Anchored Stem. All of these types, then, have a user specified anchor position. They differ, however, in how they are combined to form a complete Connector for a given geometry.

Formalization

ID + Connector referenced from each subclass

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

R57 / Generalization

Symbol is a **Text Symbol**, **Compound Shape Symbol** or **Simple Shape 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 are used to 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.

Shape Symbols can be drawn either simple shapes or built up from multiple shapes.

Formalization

Name referential attribute in each subclass

R63 / 1:Mc

Stem Notation is the meaning of *zero, one or many* **Stem**

Stem has meaning defined by *one* **Stem Notation**

Each Stem has some meaning. A Stem might indicate the connection to a Subclass, or the multiplicity of one side of an association. The meaning may or may not be illustrated with any special notation (other than just the connecting line).

A Stem Notation may or may not be employed on a given Diagram any number of times. An Mc multiplicity, for example, could apply to many association ends on a class diagram. (Or it might not be used at all).

Formalization

Referential attributes in Stem class

R64 / 1c:1c

Shape Draw Order is stacked before/after *zero or one* **Shape Draw Order**

When one Simple Shape Symbol is stacked on another, a method must be specified indicating whether the two symbols are to be stacked vertically on top of one another (z-axis) or laid adjacent to each other (x/y axis). Consequently, the relationship between two consecutive instances of Shape Draw Order must specify a stacking method.

Formalization

Compound shape + **Next position** referring to **Compound shape** + **Stack position** with the **Compound shape** value shared.

R62 / Ordinal

Stack Position draw order

In a Compound Shape Symbol, each constituent Simple Shape Symbol is drawn in a specific order. This order corresponds to a progression along the x, y or z axis.

See the **Stack Order.Arrange** attribute description for an explanation of which axis applies for a given draw order.

Formalization

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

R68 / Ordinal

Bend distance from start stem

The first Bend is drawn at the Vine end of the designated start Stem. Each subsequent Bend, if any, connects to the previous Bend until the opposite Stem Vine end is reached.

Formalization

Sequence is ordered sequentially using an Ordinal value for each **Bend Route**

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

Connector Type.Diagram type

R71 / 1:Mc

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

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

A Diagram Type specifies certain Stem Types that can be drawn for its Connector Types.

A class multiplicity or a generalization Stem Type, for example, can only be drawn on an xUML class diagram. The appearance of a Stem Type, however, may vary for each supported notation, xUML, Starr, Shlaer-Mellor and so forth.

If it turns out that a certain Stem Type might be useful on some other Diagram Type, it will be necessary to redefine that Stem Type copying some or all of its Stem Terminators. This should happen so rarely, that it shouldn't present a problem.

Formalization

Stem Type.Diagram type

R70 / 1:Mc

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

Connector appears on *exactly one* **Diagram**

A Connector is drawn on a Diagram which will be the only Diagram instance since only one Diagram is managed at a time.

Since a Diagram can be completely blank with no Nodes, it is certainly possible to have a Diagram with no Connectors on it.

Formalization

Connector.Diagram

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

Connector. Connector type

R52 / 1:Mc

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

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

A Node may or may not participate in any Connections.

If a Connector involves a Node, it will grow a Stem of the appropriate Stem Type from a face on that Node. A Stem must be attached to some Node. We say that it is rooted in the Node to distinguish the two ends of the Stem. The root is in the Node and the flower is away from the Node.

Formalization

Stem.Node

R54 / M:M-1

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

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

Since a Connector draws a line between Stems, when there are multiple, A Connector Type must specify at least one Stem Type. Depending on the geometry of the Connector Type, multiple Stem Types may be required. An xUML class diagram generalization, for example, requires a superclass Stem Type where a large arrow is drawn to highlight the root of the generalization, and a subclass Stem Type with no notation at all (just a short line).

The same Stem Type can be useful in multiple Connector Types on a given Diagram Type when model level relationships are built on one another. For example, in the class diagram Diagram Type, both the binary association and an association class Connector Types make use of a class Stem Type where multiplicity and conditionality is specified. This is because an association class Connector Type is just a binary association with an extra Stem Type for the association class connection.

There is no use for a Stem Type that is not used in a Connector Type.

Formalization

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

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

R55 / Mc:Mc-M

Connector Symbol decorates *zero, one or many* **Stem Type**

Stem Type is decorated with *zero, one or many* **Connector Symbol**

A Stem Type may specify the placement of Connector Symbol at either, both or neither end of a Stem. For example, an xUML class multiplicity Stem Type applies Connector Symbols such as Starr single-double arrows or xUML `n..m` labels at the root end of a Stem with no notation on the opposite end. An xUML state machine diagram initial state transition is constructed from a single Stem with an arrow at the root end (on the state Node face) and a solid circle at the flower end of the Stem.

Formalization

Stem Decoration association class

R56 / M:Mc-M

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

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

A Simple Shape Element is drawn relative to another (or the same) Simple Shape Element to form all or part of a Compound Shape Symbol. For example, in a double arrowhead configuration, one arrow is drawn before 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 Shape Symbol can be useful in multiple Shape Symbols. A solid arrowhead, for example, is useful in both a single and double arrowhead configuration.

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

Without any Shape Elements, there would be nothing to draw, so a Shape Symbol requires at least one.

A Shape Element is not defined unless it is used in at least one Shape Symbol.

Formalization

Stack Position association class

R58 / M:M-1

Decorated Stem End is notated by *one or many* **Symbol**

Symbol notates *one or many* **Decorated Stem End**

The meaning of a Decorated Stem End is indicated with one or more text or graphic symbols. There has to be at least one, or it is an un-decorated Stem end, basically just a line end.

If a Symbol is not used anywhere, it won't be defined. An example of a Symbol used on more than one Decorated Stem End is the Starr or Shlaer-Mellor class diagram multiplicity single and multiple arrowheads which can be used both on a class stem (against a class node face) or on an association class stem (connecting to a binary association branch). Also a generalization arrow can be used both with a Starr or xUML generalization Stem Type.

Formalization

Referential attributes in the Stem Notation class

R59 / 1c:Mc-1

Diagram Notation styles *zero, one or many* **Stem Type**

Stem Type is styled by *zero or one* **Diagram Notation**

Normally all Stem Type lines are drawn in the default style used for all Connector Types (thin, solid typically). In at least one case, the xUML class diagram association class, the Stem Type requires a dashed line.

Rather than define a style for each Stem Type, since most of them are the same, we define a line style only for those Stem Types that vary from the default.

So a given Stem Type may or may not be given a unique line style by a Diagram Notation. A given Diagram Notation may require a special style for any number of Stem Types.

Formalization

Referential attributes in the Stem Type Style class