Flatland3 Domain

Node Subsystem Description

Description…

Flatland3 Domain Data Types

Author

Leon Starr

Document

mint.flatland3.td.21 / Version 0.4.0 / December 31, 2019

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.

License / Copyright

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

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.21 / Version 0.4.0 / December 31, 2019

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.

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

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

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 Semantic

Meaning is often, but not always, attributed to the point where a Stem line anchors on a Node face or on the opposite end of the Stem line (where it hangs free in the case of a unary Connector Type or it meets the branch that ties all the Stems together). On a state machine diagram, for example, the line that touches a state Node is terminated with an arrow to indicate a transition path. Each possible termination meaning, or semantic, can be described independent of any particular notation. The state transition arrow, for example, can be abstracted as a target state semantic. Whether or not it is in fact represented by an arrow symbol is determined by whatever notational system is applied on a given diagram, xUML state machine Diagram Notation, for example.

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

Connector Symbol

Defines a visual symbol to represent the associated Connector Semantic for the specified Diagram Type. See R60 description for more details.

Attributes

Semantic

Same as **Connector Semantic.Name**

Diagram type

Same as **Diagram Notation.Diagram type**

Notation

Same as **Diagram Notation.Notation**

Identifiers

Connector semantic + **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 (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.

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

Connector semantic

Connector Symbol.Connector semantic

Diagram type

Connector Symbol.Diagram type and **Stem Type.Diagram type**

Notation

Connector Symbol.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 Connector Symbol. 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 ]`

Identifiers

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

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

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.

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

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

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**

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 flower.

Attributes

ID

Distinguishes one Stem from another within the same Connector.

Type: Nominal

Connector

Same as**Connector.ID**

Stem type

Stem Type.Name

Node

Same as **Node.ID**

Face

The side of the Node where the Stem is anchored.

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

Lateral 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**

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. **Connector** + **Stem Type** + **Node** + **Face** + **Lateral position**

To prevent any drawing overlap, two stems may not anchor at the same Node face position.

Stem Notation

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**

Connector semantic

Same as **Decorated Stem End.Connector 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 + **Connector semantic** + **Diagram type** + **Notation** + **Symbol**

Consequence of a many-many association

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 assoc class.

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.

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.

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

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

Connector Type.Diagram type

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

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 / 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**

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

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

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

R60 / 1:Mc-M

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

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

A Diagram Notation may supply some kind of notation for any Connector 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 Connector 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

Connector Symbol 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

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**

R63 / 1:Mc

Connector Type connects stems using *zero, one or many* **Branch Pattern**

Branch Pattern connects stems for *one* **Connector Type**

A non-unary Connector Type will require at least one Branch Pattern to pull together all of its Stems into a complete Connector.

For example, an xUML class diagram Ternary Association requires two Branch Patterns. One connects the class multiplicity Stems and the other connects the

Formalization

Branch Pattern.Connector type and **Branch Pattern.Diagram type**

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.

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

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