Gaphor Documentation

Arjan J. Molenaar

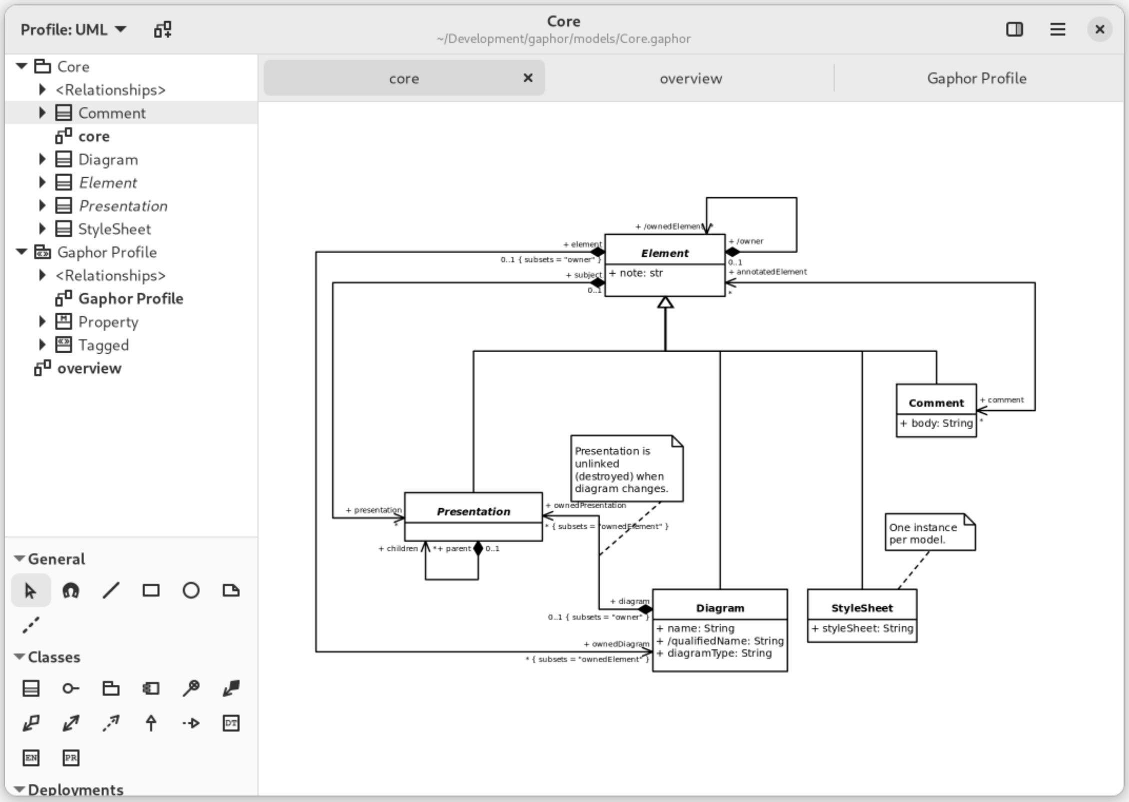
May 01, 2023

GETTING STARTED

Note: The documentation is up to date for Gaphor 2.18.1

Gaphor is a UML and SysML modeling application written in Python. It is designed to be easy to use, while still being powerful. Gaphor implements a fully-compliant UML 2 data model, so it is much more than a picture drawing tool.

You can use Gaphor to quickly visualize different aspects of a system as well as create complete, highly complex models.



Gaphor is 100% Open source. The code and issue tracker can be found on GitHub.

What are you waiting for? *Let's get started*!

For download instructions, and the blog, please visit the Gaphor Website.

Gaphor has excellent integration with *Sphinx* and *Jupyter notebooks*.

CHAPTER

ONE

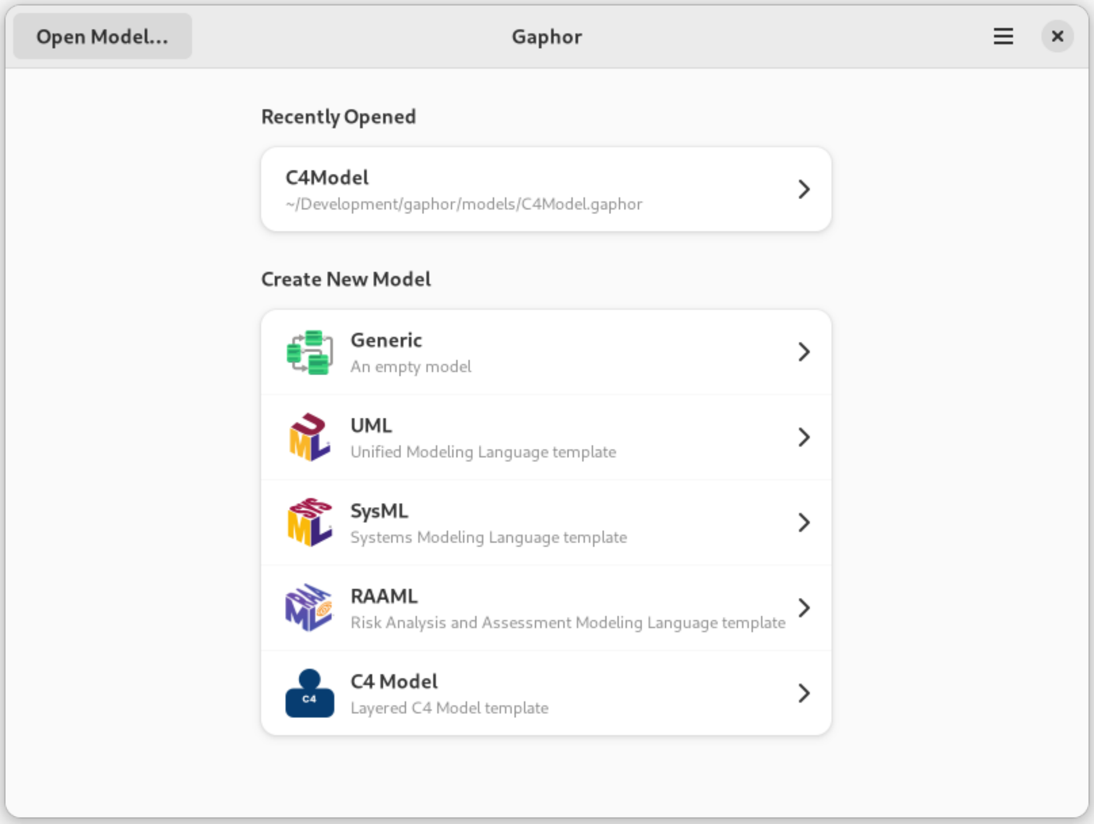
GET STARTED WITH GAPHOR

Gaphor is more than a diagram editor: it's a modeling environment. Where simple diagram editors such as Microsoft Visio and draw.io allow you to create pictures, Gaphor actually keeps track of the elements you add to the model. In Gaphor you can create diagrams to track and visualize different aspects of the system you're developing.

Enough talk, let's get started.

You can find installers for Gaphor on the Gaphor Website. Gaphor can be installed on Linux (Flatpak, AppImage), Windows, and macOS.

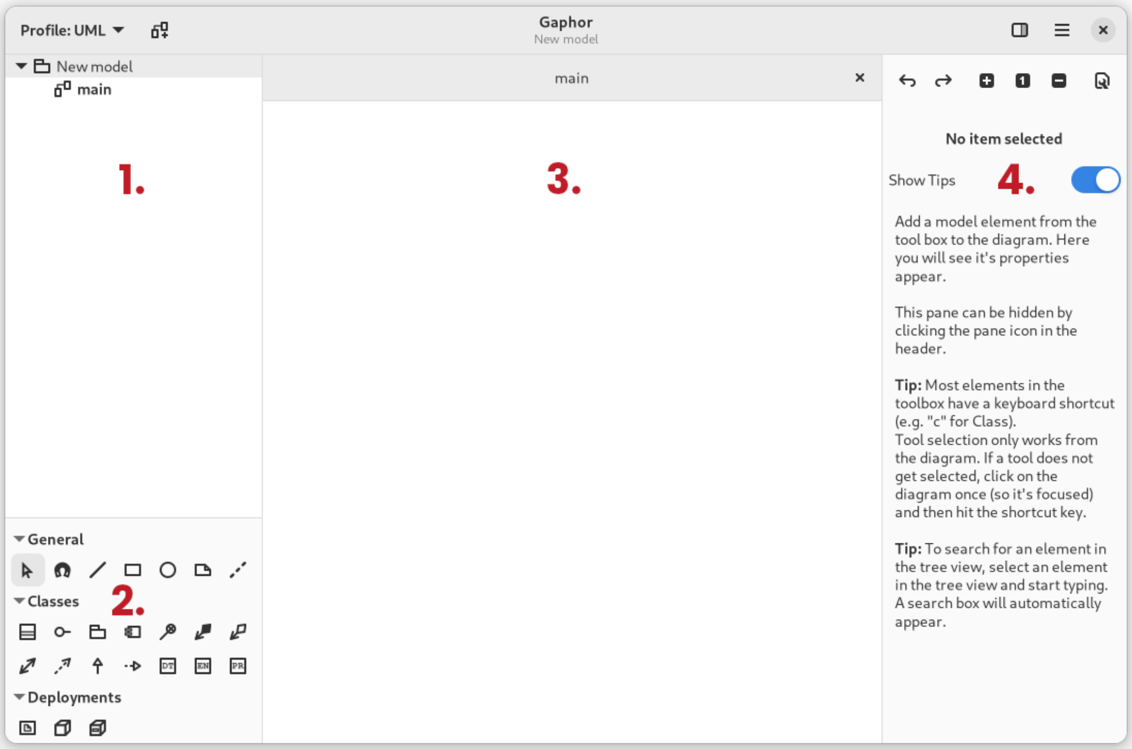
Once Gaphor is launched, it provides you a welcome screen. It shows you previously opened models and model templates.


Open Model...
Gaphor
Recently Opened
C4Model
~/Development/gaphor/models/C4Model.gaphor
Create New Model
Generic
An empty model
UML
Unified Modeling Language template
SysML
Systems Modeling Language template
RAAML
Risk Analysis and Assessment Modeling Language template
C4 Model
Layered C4 Model template

You can select a template to get started.

- **Generic:** a blank model to start with
- **UML:** A template for the *Unified Modeling Language* for modeling a software system
- **SysML:** A template for the *Systems Modeling Language* for modeling a wide range of systems and systems-of-systems
- **RAAML:** A template for the *Risk Analysis and Assessment Modeling language* for safety and reliability analysis
- **C4 Model:** A template for *Context, Containers, Components, and Code* which for lean modeling of software architecture

Once the model interface is loaded you'll see the modeling interface.



Add a model element from the tool box to the diagram. Here you will see it's properties appear.

This pane can be hidden by clicking the pane icon in the header.

Tip: Most elements in the toolbox have a keyboard shortcut (e.g. "c" for Class).
Tool selection only works from the diagram. If a tool does not get selected, click on the diagram once (so it's focused) and then hit the shortcut key.

Tip: To search for an element in the tree view, select an element in the tree view and start typing. A search box will automatically appear.

The layout of the Gaphor interface is divided into four sections, namely:

1. Model Browser
2. Diagram Element Toolbox
3. Diagrams
4. Property Editor

Each section has its own specific function.

1.1 Model Browser

The Model Browser section of the interface displays a hierarchical view of your model. Every model element you create will be inserted into the Model Browser. This view acts as a tree where you can expand and collapse different elements of your model. This provides an easy way to view the elements of your model from an elided perspective. That is, you can collapse those model elements that are irrelevant to the task at hand.

In the figure above, you will see that there are two elements in the Model Browser. The root element, *New Model* is a package. Notice the small arrow beside *New Model* that is pointing downward. This indicates that the element is expanded. You will also notice the two sub-elements are slightly indented in relation to *New Model*. The *main* element is a diagram.

In the Model Browser view, you can also right-click the model elements to get a context menu. This context menu allows you to find out in which diagram model elements are shown, add new diagrams and packages, and delete an element.

Double-clicking on a diagram element will show it in the Diagram section. Elements such as classes and packages can be dragged from the tree view on the diagrams.

1.2 Toolbox

The toolbox is used to add new items to a diagram. Select the element you want to add by clicking on it. When you click on the diagram, the selected element is created. The arrow is selected again, so the element can be manipulated.

Tools can be selected by simply left-clicking on them. By default, the pointer tool is selected after every item placement. This can be changed by disabling the "Reset tool" option in the Preferences window. Tools can also be selected by keyboard shortcuts. The keyboard shortcut can be displayed as a tooltip by hovering over the tool button in the toolbox. Finally, it is also possible to drag elements on the Diagram from the toolbox.

1.3 Diagrams

The diagram section contains diagrams of the model and takes up the most space in the UI because it is where most of the modeling is done. Diagrams consist of items placed on the diagram. There are two main types of items:

1. Elements
2. Relationships

Multiple diagrams can be opened at once: they are shown in tabs. Tabs can be closed by pressing Ctrl+w or left-clicking on the x in the diagram tab.

1.3.1 Elements

Elements are the shapes that you add to a diagram, and together with Relations, allow you to build up a model.

To resize an element on the diagram, left-click on the element to select it and then drag the resize handles that appear at each corner.

To move an element on the diagram, drag the element where you want to place it by pressing and holding the left mouse button, and moving the mouse before releasing the button.

1.3.2 Relations

Relations are line-like elements that form relationships between elements in the diagram. Each end of a relation is in one of two states:

1. Connected to an element and the handle turns red
2. Disconnected from an element and the handle turns green

If both ends of a relation are disconnected, the relation can be moved by left-clicking and dragging it.

A new segment in a relation can be added by left-clicking on the relation to select it and then by hovering your mouse over it. A green handle will appear in the middle of the line segments that exist. Drag the handle to add another segment. For example, when you first create a new relation, it will have only one segment. If you drag the segment handle, then it will now have two segments with the knee of the two segments where the handle was.

1.3.3 Undo and Redo

Undo a change press Ctrl+z or left-click on the back arrow at the top of the Property Editor. To re-do a change, hit Ctrl+Shift+z or press the forward arrow at the top of the Property Editor.

1.4 Property Editor

The Property Editor is present on the right side of the diagrams. When no item is selected in the diagram, it shows you some tips and tricks. When an item is selected on the diagram, it contains the item details like name, attributes and stereotypes. It can be opened with F9 and the icon in the header bar.

The properties that are shown depend on the item that is selected.

1.5 Model Preferences

The Property Editor also contains model preferences: Click the button. Here you can set a few model related settings and edit the *style sheet*.

CHAPTER

TWO

YOUR FIRST MODEL

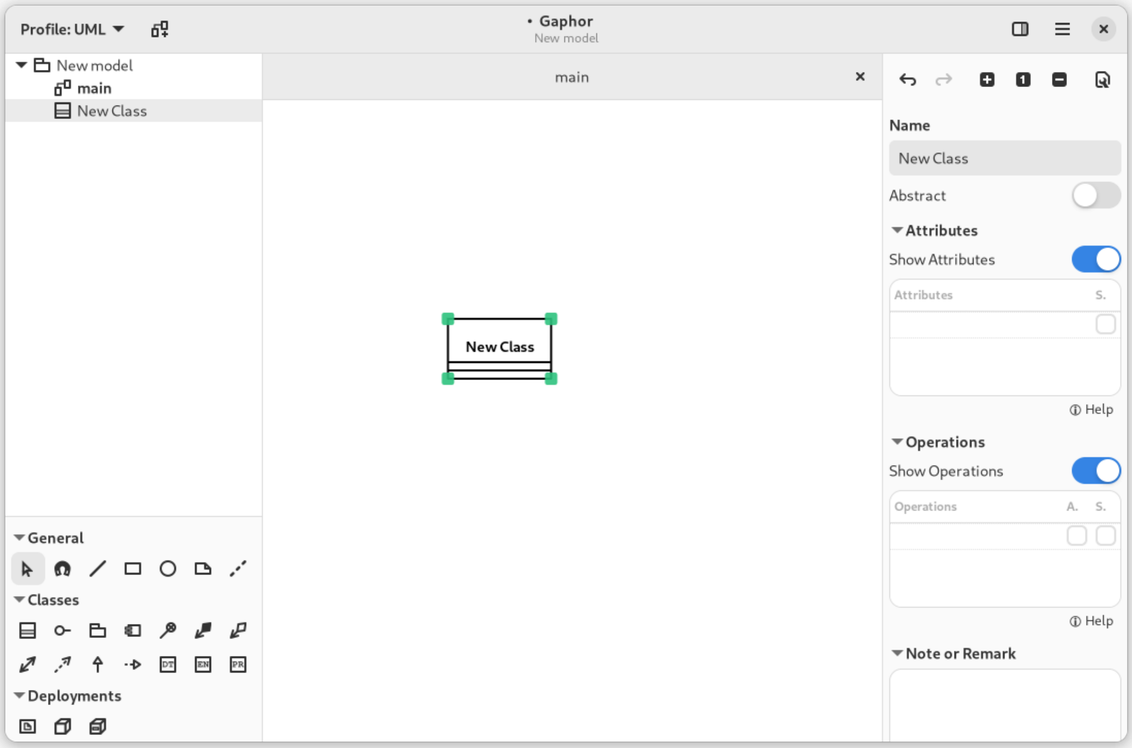
Note: In this tutorial we refer to the different parts of the gaphor interface: *Model Browser*, *Toolbox*, *Property Editor*.

Although the names should speak for themselves, you can check out the *Getting Started* page for more information about those sections.

Once Gaphor is started, and you can start a new model with the *Generic* template. The initial diagram is already open in the Diagram section.

Select an element you want to place, in this case a Class () by clicking on the icon in the Toolbox and click on the diagram. This will place a new Class item instance on the diagram and add a new Class to the model – it shows up in the Model Browser. The selected tool will reset itself to the Pointer tool after the element is placed on the diagram.

The Property Editor on the right side will show you details about the newly added class, such as its name (*New Class*), attributes and operations (methods).



It's simple to add elements to a diagram.

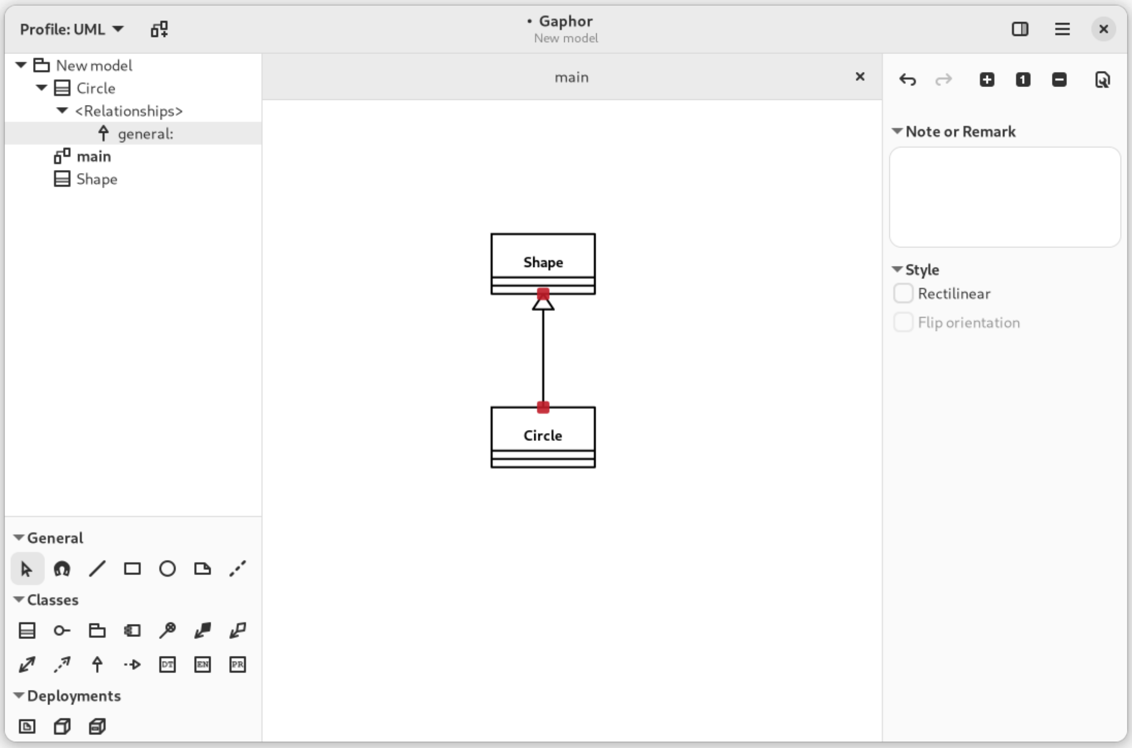
Gaphor does not make any assumptions about which elements should be placed on a diagram. A diagram is a diagram. UML defines all different kinds of diagrams, such as Class diagrams, Component diagrams, Action diagrams, Sequence diagrams. But Gaphor does not place any restrictions.

2.1 Adding Relations

Add another Class. Change the names to `Shape` and `Circle`. Let's define that `Circle` is a sub-type of `Shape`. You can do this by selecting one and changing the name in the Property Editor, or by double-clicking the element.

Select Generalization ().

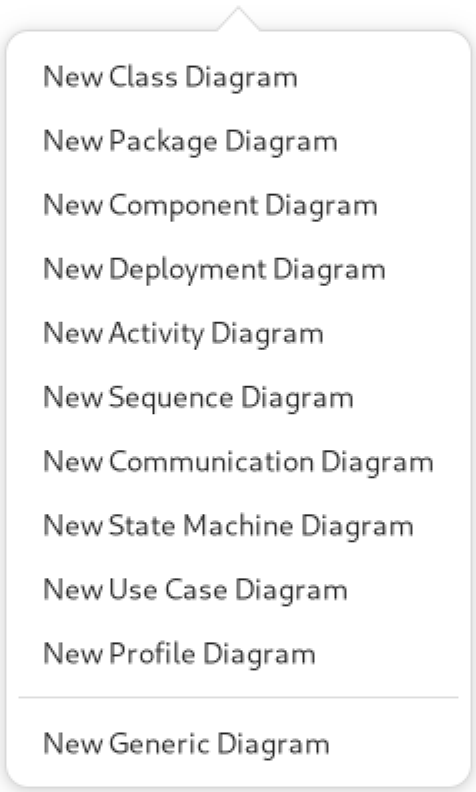
Move the mouse cursor over `Shape`. Click, hold and drag the line end over `Circle`. Release the mouse button, and you should have your relationship between `Shape` and `Circle`. You can see both ends of the relation are red, indicating they are connected to their class.



Optionally you can run the auto-layout (→ Tools → Auto Layout) to align the elements on the diagram.

2.2 Creating New Diagrams

To create a new diagram, use the Model Browser. Select the element that should contain the new diagram. For now, select *New Model*. Click the New Diagram menu () in the header bar.



Select *New Generic Diagram* and a new diagram is created.

Now drag the elements from the Model Browser onto the new diagram. First the classes `Shape` and `Circle`. Add the generalization last. Drop it somewhere between the two classes. The relation will be created to the diagram.

Now change the name of class `Circle` to `Ellipse`. Check the other diagram. The name has been changed there as well.

Important: Elements in a diagram are only a *representation* of the elements in the underlaying model. The model is what you see in the Model Browser.

Elements in the model are automatically removed when there are no more representations in any of the diagrams.

CHAPTER

THREE

STYLE SHEETS

Since Gaphor 2.0, diagrams can have a different look by means of style sheets. Style sheets use the Cascading Style Sheets (CSS) syntax. CSS is used to describe the presentation of a document written in a markup language, and is most commonly used with HTML for web pages.

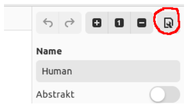
On the W3C CSS home page, CSS is described as:

> Cascading Style Sheets (CSS) is a simple mechanism for adding style (e.g., fonts, colors, spacing) to Web documents.

Its application goes well beyond web documents, though. Gaphor uses CSS to provide style elements to items in diagrams. CSS allows us, users of Gaphor, to change the visual appearance of our diagrams. Color and line styles can be changed to make it easier to read the diagrams.

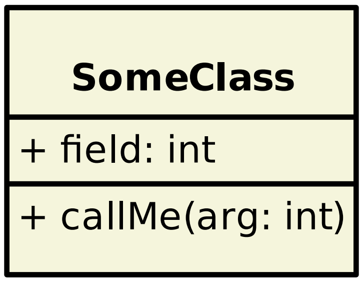
Since we're dealing with a diagram, and not a HTML document, some CSS features have been left out.

The style is part of the model, so everyone working on a model will have the same style. To edit the style press the tools page button at the top right corner in gaphor:



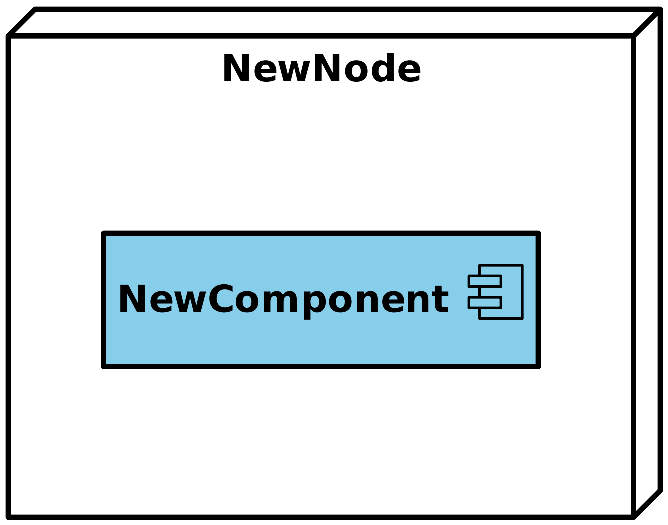
Here is a simple example of how to change the background color of a class:

```
class {
  background-color: beige;
}
```



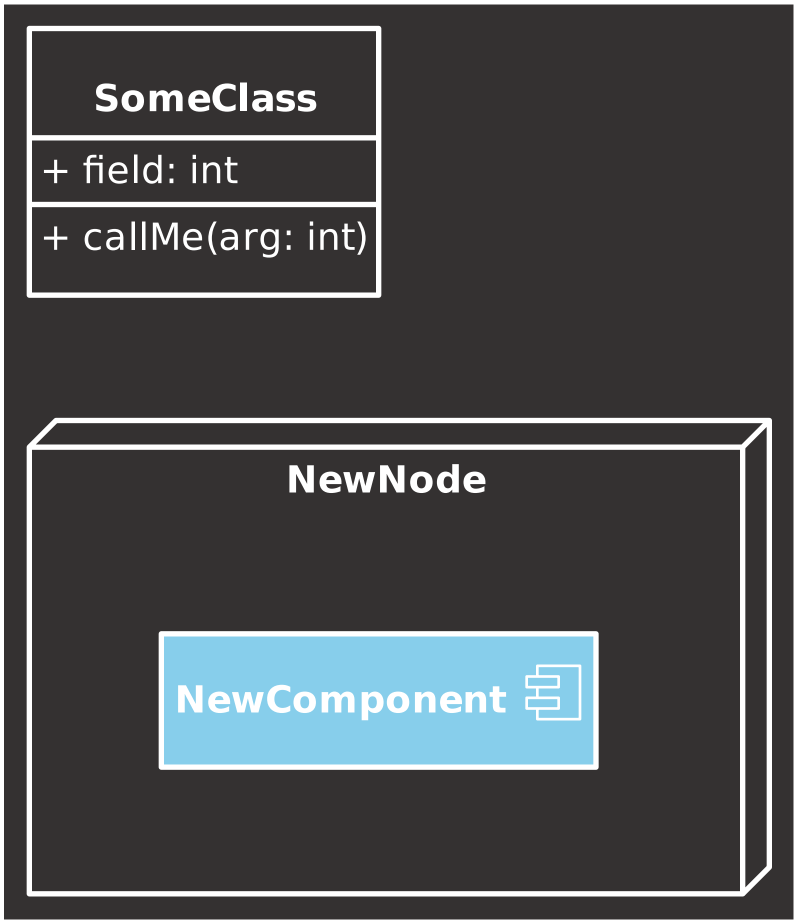
Or change the color of a component, only when it's nested in a node:

```
node component {
  background-color: skyblue;
}
```



The diagram itself is also expressed as a CSS node. It's pretty easy to define a "dark" style:

```
diagram {
  background-color: #343131;
}

* {
  color: white;
  text-color: white;
}
```



Here you already see the first custom attribute: `text-color`. This property allows you to control the color of the text drawn in an item. `color` is used for the lines (strokes) that make the layout of a diagram item.

3.1 Supported selectors

Since we are dealing with diagrams and models, we do not need all the features of CSS. Below you'll find a summary of all CSS features supported by Gaphor.

`*`	All items on the diagram, including the diagram itself.
`node component`	Any component item which is a descendant of a node.
`node > component`	A component item which is a child of a node.
`generaliation[subject]`	A generalization item with a subject present.
`class[name=Foo]`	A class with name “Foo”.
`diagram[name^=draft]`	A diagram with a name starting with “draft”.
`diagram[name$=draft]`	A diagram with a name ends with “draft”.
`diagram[name*=draft]`	A diagram with a name containing the text “draft”.
`diagram[name~=draft item]`	A diagram with a name of “draft” or “item”.
`diagram[name|=draft]`	A diagram with a name is “draft” or starts with “draft-“.
`*:focus`	The focused item. Other pseudo classes are: • `:active` selected items • `:hover` for the item under the mouse • `:drop` if an item is dragged and can be dropped on this item • `:disabled` if an element is grayed out during handle movement
`node:empty`	A node containing no child nodes in the diagram.
`:root`	An item is at the top level of the diagram. This is only applicable for the diagram
`:has()`	The item contains any of the provided selectors. E.g. `node:has(component)`: a node containing a component item.
`:is()`	Match any of the provided selectors. E.g. `:is(node, subsystem) > component`: a node or subsystem.
`:not()`	Negate the selector. E.g. `:not([subject])`: Any item that has no “subject”.

- The official specification of CSS3 attribute selectors.
- Gaphor provides the `|=` attribute selector for the sake of completeness. It’s probably not very useful in this context, though.
- Please note that Gaphor CSS does not support IDs for diagram items, so the CSS syntax for IDs (`#some-id`) is not used. Also, class syntax (`.some-class`) is not supported currently.

3.2 Style properties

Gaphor supports a subset of CSS properties and some Gaphor specific properties. The style sheet interpreter is relatively straight forward. All widths, heights, and sizes are measured in pixels. You can’t use complex style declarations, like the `font` property in HTML/CSS which can contain font family, size, weight.

3.2.1 Colors

`background-color`	Examples:
	`background-color: azure;`
	`background-color: rgb(255, 255, 255);`
	`background-color: hsl(130, 95%, 10%);`

`color`	Color used for lines.
`text-color`	Color for text.
`opacity`	Color opacity factor (`0.0` - `1.0`), applied to all colors.

- A color can be any CSS3 color code, as described in the CSS documentation. Gaphor supports all color notations: `rgb()`, `rgba()`, `hsl()`, `hsla()`, Hex code (`#ffffff`) and color names.

3.2.2 Text and fonts

`font-family`	A single font name (e.g. `sans`, `serif`, `courier`).
`font-size`	An absolute size (e.g. `14`) or a size value (e.g. `small`).
`font-style`	Either `normal` or `italic`.
`font-weight`	Either `normal` or `bold`.
`text-align`	Either `left`, `center`, `right`.
`text-decoration`	Either `none` or `underline`.
`vertical-align`	Vertical alignment for text. Either `top`, `middle` or `bottom`.
`vertical-spacing`	Set vertical spacing for icon-like items (actors, start state). Example: `vertical-spacing: 4`.

- `font-family` can be only one font name, not a list of (fallback) names, as is used for HTML.
- `font-size` can be a number or CSS absolute-size values. Only the values `x-small`, `small`, `medium`, `large` and `x-large` are supported.

3.2.3 Drawing and spacing

`border-radius`	Radius for rectangles: `border-radius: 4`.
`dash-style`	Style for dashed lines: `dash-style: 7 5`.
`justify-content`	Content alignment for boxes. Either `start`, `end`, `center` or `stretch`.
`line-style`	Either `normal` or `sloppy [factor]`.
`line-width`	Set the width for lines: `line-width: 2`.
`min-height`	Set minimal height for an item: `min-height: 50`.
`min-width`	Set minimal width for an item: `min-width: 100`.
`padding`	CSS style padding (top, right, bottom, left). Example: `padding: 3 4`.

- `padding` is defined by integers in the range of 1 to 4. No unit (px, pt, em) needs to be used. All values are in pixel distance.
- `dash-style` is a list of numbers (line, gap, line, gap, …)
- `line-style` only has an effect when defined on a `diagram`. A sloppiness factor can be provided in the range of -2 to 2.

3.2.4 Diagram styles

Only a few properties can be defined on a diagram, namely `background-color` and `line-style`. You define the diagram style separately from the diagram item styles. That way it's possible to set the background color for diagrams specifically. The line style can be the normal straight lines, or a more playful "sloppy" style. For the sloppy style an optional wobliness factor can be provided to set the level of line wobbliness. 0.5 is default, 0.0 is a straight line. The value should be between -2.0 and 2.0. Values between 0.0 and 0.5 make for a subtle effect.

Gaphor supports dark and light mode since 2.16.0. Dark and light color schemes are exclusively used for on-screen editing. When exporting images, only the default color scheme is applied. Color schemes can be defined with `@media` queries. The official `prefers-color-scheme = dark` query is supported, as well as a more convenient `dark-mode`.

```css
/* The background you see in exported diagrams: */
diagram {
  background-color: transparent;
}

/* Use a slightly grey background in the editor: */
@media light-mode {
  diagram {
    background-color: #e1e1e1;
  }
}

/* And anthracite a slightly grey background in the editor: */
@media dark-mode {
  diagram {
    background-color: #393D47;
  }
}
```

3.2.5 Variables

Since Gaphor 2.16.0 you can use CSS variables in your style sheets.

This allows you to define often used values in a more generic way. Think of things like line dash style and colors.

The `var()` function has some limitations:

- Values can't have a default value.
- Variables can't have a variable as their value.

Example:

```css
diagram {
  --bg-color: whitesmoke;
  background-color: var(--bg-color);
}

diagram[diagramType=sd] {
  --bg-color: rgb(200, 200, 255);
}
```

All diagrams have a white background. Sequence diagrams get a blue-ish background.

3.3 Working with model elements

Gaphor has many model elements. How can you find out which item should be styled?

Gaphor only styles the elements that are in the model, so you should be explicit on their names. For example: `Component` inherits from `Class` in the UML model, but changing a color for `Class` does not change it for `Component`.

If you hover over a button the toolbox (bottom-left section), a popup will appear with the item's name and a shortcut. As a general rule, you can use the component name, glued together as the name in the stylesheet. A *Component* can be addressed as `component`, *Use Case* as `usecase`. The name matching is case insensitive. CSS names are written in lower case by default.

However, since the CSS element names are derived from names used within Gaphor, there are a few exceptions.

Profile	Group	Element	CSS element
*	*	*element name*	element name without spaces E.g. `class`, `usecase`.
UML	Classes	all Association's	`association`
UML	Components	Device/Node	`node`
UML	Actions	Decision/Merge Node	`decisionnode`
UML	Actions	Fork/Join Node	`forknode`
UML	Actions	Swimlane	`partition`
UML	Interactions	Reflexive message	`message`
UML	States	Initial Pseudostate	`pseudostate`
UML	States	History Pseudostate	`pseudostate`
UML	Profiles	Metaclass	`class`
C4 Model	C4 Model	Person	`c4person`
C4 Model	C4 Model	Software System	`c4container[type="Software System"]`
C4 Model	C4 Model	Component	`c4container[type="Component"]`
C4 Model	C4 Model	Container	`c4container[type="Container"]`
C4 Model	C4 Model	Container: Database	`c4database`
SysML	Blocks	ValueType	`datatype`
SysML	Blocks	Primitive	`datatype`
SysML	Requirements	Derive Requirement	`derivedreq`
RAAML	FTA	any AND/OR/… Gate	`and`, `or`, etc.

3.4 Ideas

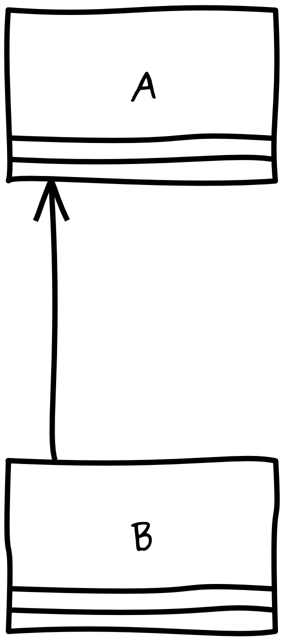
Here are some ideas that go just beyond changing a color or a font. With the following examples we dig in to Gaphor's model structure to reveal more information to the users.

To create your own expression you may want to use the Console (Tools -> Console, in the Hamburger menu). Drop us a line on Gitter and we would be happy to help you.

3.4.1 The drafts package

All diagrams in the package "Drafts" should be drawn using sloppy lines:

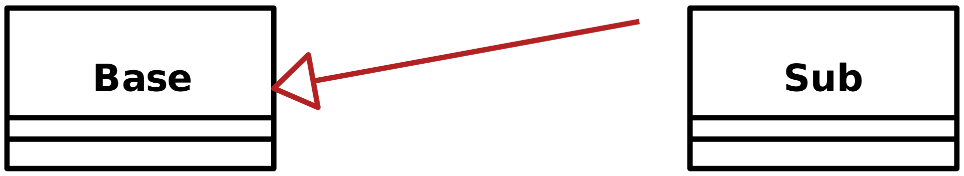
```
diagram[owner.name=drafts] {
  line-style: sloppy 0.3;
}

diagram[owner.name=drafts] * {
  font-family: Purisa; /* Or use some other font that's installed on your system */
}
```



3.4.2 Unconnected relationships

All items on a diagram that are not backed by a model element, should be drawn in a dark red color. This can be used to spot not-so-well connected relationships, such as Generalization, Implementation, and Dependency. These items will only be backed by a model element once you connect both line ends. This rule will exclude simple elements, like lines and boxes, which will never have a backing model element.

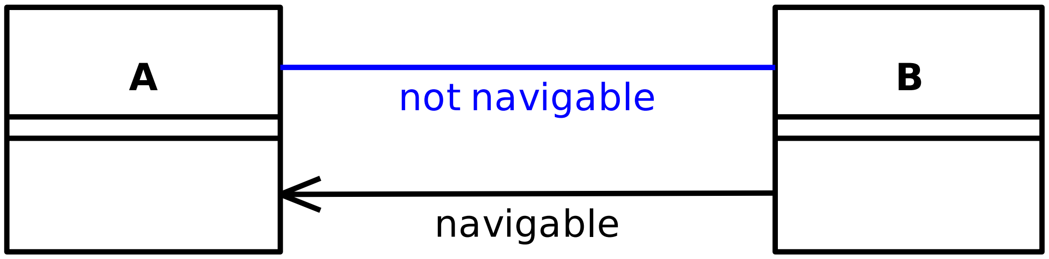
```
:not([subject], :is(line, box, ellipse, commentline)) {
  color: firebrick;
}
```



3.4.3 Navigable associations

An example of how to apply a style to a navigable association is to color an association blue if neither of the ends are navigable. This color could act as a validation rule for a model where at least one end of each association should be navigable. This is actually the case for the model file used to create Gaphor's data model.

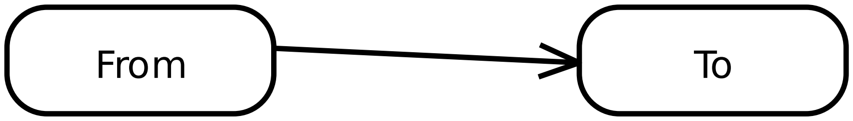
```
association:not([memberEnd.navigability*=true]) {
  color: blue;
}
```



3.4.4 Solid Control Flow lines

In Gaphor, Control Flow lines follow the SysML styling: dashed. If you want, or need to strictly follow the official UML specifications, you can simply make those solid lines.

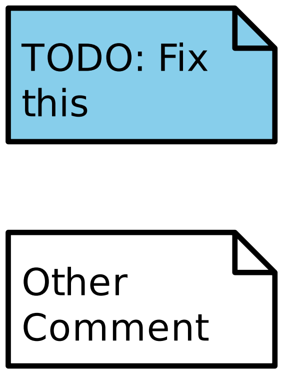
```
controlflow {
  dash-style: 0;
}
```



3.4.5 Todo note highlight

All comments beginning with the phrase "todo" can be highlighted in a different user-specific colour. This can be used to make yourself aware that you have to do some additional work to finalize the diagram.

```
comment[body^="TODO"] {
  background-color: skyblue;
}
```


TODO: Fix this
Other Comment

CHAPTER

FOUR

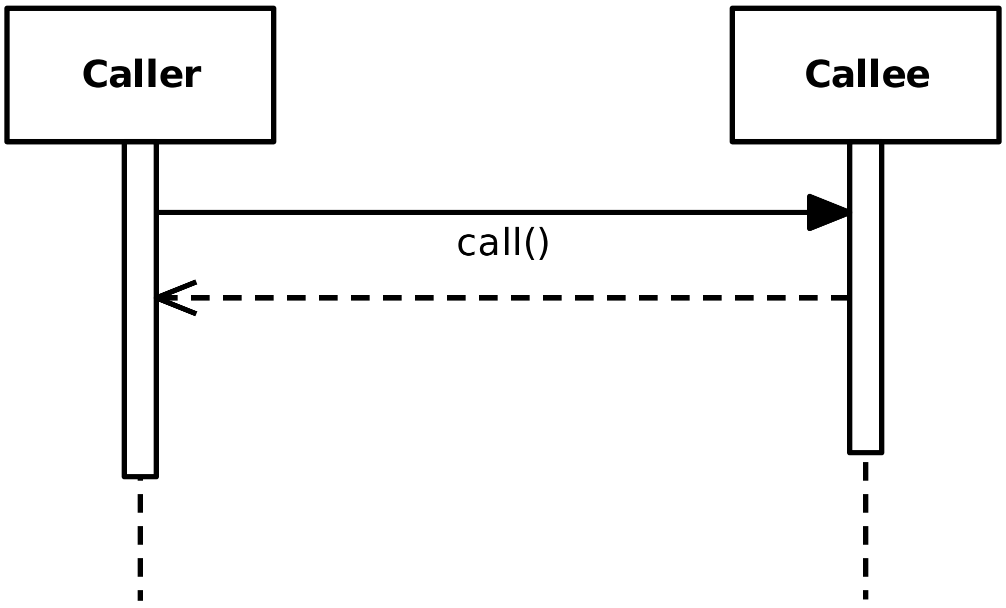
SPHINX EXTENSION

What's more awesome than to use Gaphor diagrams directly in your Sphinx documentation. Whether you write your docs in reStructured Text or Markdown, we've got you covered.

Tip: Here we cover the reStructured Text syntax. If you prefer markdown, we suggest you have a look at the MyST-parser, as it supports Sphinx directives.

It requires *minimal effort to set up*. Adding a diagram is as simple as:

```
.. diagram:: main
```



In case you use multiple Gaphor source files, you need to define a `:model:` attribute and add the model names to the Sphinx configuration file (`conf.py`).

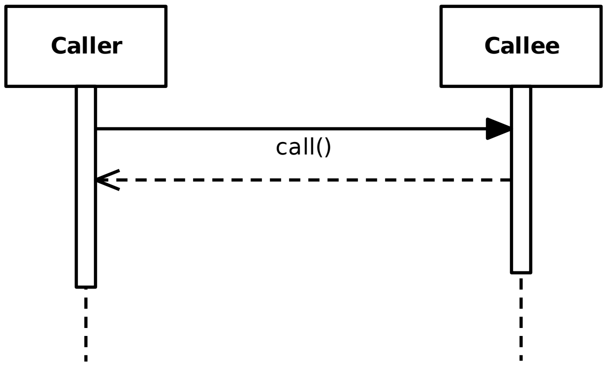
```
.. diagram:: main
   :model: example
```

Diagrams can be referenced by their name, or by their fully qualified name.

```
.. diagram:: New model.main
```

Image properties can also be applied:

```
.. diagram:: main
   :width: 50%
   :align: right
   :alt: A description suitable for an example
```



4.1 Configuration

To add Gaphor diagram support to Sphinx, make sure Gaphor is listed as a dependency.

Important: Gaphor requires at least Python 3.9.

Secondly, add the following to your `conf.py` file:

Step 1: Add gaphor as extension.

```
extensions = [
    "gaphor.extensions.sphinx",
]
```

Step 2: Add references to models

```
# A single model
gaphor_models = "../examples/sequence-diagram.gaphor"

# Or multiple models
gaphor_models = {
    "connect": "connect.gaphor",
    "example": "../examples/sequence-diagram.gaphor"
}
```

Now include `diagram` directives in your documents.

4.1.1 Read the Docs

The diagram directive plays nice with Read the docs. To make diagrams render, it's best to use a .readthedocs.yaml file in your project. Make sure to include the extra `apt_packages` as shown below.

This is the `.readthedocs.yaml` file we use for Gaphor:

```
version: 2
formats: all
build:
  os: ubuntu-22.04
  tools:
    python: "3.11"
  apt_packages:
  - libgirepository1.0-dev
  - gir1.2-pango-1.0
  - graphviz
sphinx:
  configuration: docs/conf.py
  fail_on_warning: true
python:
  install:
  - method: pip
    path: .
    extra_requirements:
    - docs
```

- `libgirepository1.0-dev` is required to build PyGObject.
- `gir1.2-pango-1.0` is required for text rendering.

Note: For Gaphor 2.7.0, `gir1.2-gtk-3.0` and `gir1.2-gtksource-4` are required `apt_packages`, although we do not use the GUI. From Gaphor 2.7.1 onwards all you need is GI-repository and Pango.

4.2 Errors

Errors are shown on the console when the documentation is built and in the document.

An error will appear in the documentation. Something like this:

Error: No diagram 'Wrong name' in model 'example' (../examples/sequence-diagram.gaphor).

CHAPTER

FIVE

JUPYTER AND SCRIPTING

One way to work with models is through the GUI. However, you may also be interested in getting more out of your models by interacting with them through Python scripts and Jupyter notebooks.

You can use scripts to:

- Explore the model, check for (in)valid conditions.
- Generate code, as is done for Gaphor's data model.
- Update a model from another source, like a CSV file.

Since Gaphor is written in Python, it also functions as a library.

5.1 Getting started

To get started, you'll need a Python console. You can use the interactive console in Gaphor, use a Jupyter notebook, although that will require setting up a Python development environment.

5.2 Query a model

The first step is to load a model. For this you'll need an `ElementFactory`. The `ElementFactory` is responsible to creating and maintaining the model. It acts as a repository for the model while you're working on it.

```
from gaphor.core.modeling import ElementFactory

element_factory = ElementFactory()
```

The module `gaphor.storage` contains everything to load and save models. Gaphor supports multiple *modeling languages*. The `ModelingLanguageService` consolidates those languages and makes it easy for the loader logic to find the appropriate classes.

Note: In versions before 2.13, an `EventManager` is required. In later versions, the `ModelingLanguageService` can be initialized without event manager.

```
from gaphor.core.eventmanager import EventManager
from gaphor.services.modelinglanguage import ModelingLanguageService
from gaphor.storage import storage

```

(continues on next page)

(continued from previous page)

```
event_manager = EventManager()

modeling_language = ModelingLanguageService(event_manager=event_manager)

with open("../models/Core.gaphor") as file_obj:
    storage.load(
        file_obj,
        element_factory,
        modeling_language,
    )
```

At this point the model is loaded in the `element_factory` and can be queried.

Note: A modeling language consists of the model elements, and diagram items. Graphical components are loaded separately. For the most basic manupilations, GTK (the GUI toolkit we use) is not required, but you may run into situations where Gaphor tries to load the GTK library.

One trick to avoid this (when generating Sphinx docs at least) is to use autodoc's mock function to mock out the GTK and GDK libraries. However, Pango needs to be installed for text rendering.

A simple query only tells you what elements are in the model. The method `ElementFactory.select()` returns an iterator. Sometimes it's easier to obtain a list directly. For those cases you can use `ElementFatory.lselect()`. Here we select the last five elements:

```
for element in element_factory.lselect()[:5]:
    print(element)
```

```
<gaphor.UML.uml.Package element 3867dda4-7a95-11ea-a112-7f953848cf85>
<gaphor.core.modeling.diagram.Diagram element 3867dda5-7a95-11ea-a112-7f953848cf85>
<gaphor.UML.classes.klass.ClassItem element 4cda498f-7a95-11ea-a112-7f953848cf85>
<gaphor.UML.classes.klass.ClassItem element 5cdae47f-7a95-11ea-a112-7f953848cf85>
<gaphor.UML.classes.klass.ClassItem element 639b48d1-7a95-11ea-a112-7f953848cf85>
```

Elements can also be queried by type and with a predicate function:

```
from gaphor import UML
for element in element_factory.select(UML.Class):
    print(element.name)
```

```
Element
Diagram
Presentation
Comment
StyleSheet
Property
Tagged
ElementChange
ValueChange
RefChange
PendingChange
ChangeKind
```

```
for diagram in element_factory.select(
    lambda e: isinstance(e, UML.Class) and e.name == "Diagram"
):
    print(diagram)
```

```
<gaphor.UML.uml.Class element 5cdae47e-7a95-11ea-a112-7f953848cf85>
```

Now, let's say we want to do some simple (pseudo-)code generation. We can iterate class attributes and write some output.

```
diagram: UML.Class

def qname(element):
    return ".".join(element.qualifiedName)

diagram = next(element_factory.select(lambda e: isinstance(e, UML.Class) and e.name ==
↪"Diagram"))

print(f"class {diagram.name}({', '.join(qname(g) for g in diagram.general)}):")
for attribute in diagram.attribute:
    if attribute.typeValue:
        # Simple attribute
        print(f"    {attribute.name}: {attribute.typeValue}")
    elif attribute.type:
        # Association
        print(f"    {attribute.name}: {qname(attribute.type)}")
```

```
class Diagram(Core.Element):
    diagramType: String
    element: Core.Element
    ownedPresentation: Core.Presentation
    qualifiedName: String
    name: String
```

To find out which relations can be queried, have a look at the *modeling language* documentation. Gaphor's data models have been built using the *UML* language.

You can find out more about a model property by printing it.

```
print(UML.Class.ownedAttribute)
```

```
<association ownedAttribute: Property[0..*] <>-> class_>
```

In this case it tells us that the type of `UML.Class.ownedAttribute` is `UML.Property`. `UML.Property.class_` is set to the owner class when `ownedAttribute` is set. It is a bidirectional relation.

5.3 Draw a diagram

Another nice feature is drawing the diagrams. At this moment this requires a function. This behavior is similar to the *diagram directive*.

```
from gaphor.core.modeling import Diagram
from gaphor.extensions.ipython import draw

d = next(element_factory.select(Diagram))
draw(d, format="svg")
```

```
<IPython.core.display.SVG object>
```

5.4 Create a diagram

(Requires Gaphor 2.13)

Now let's make something a little more fancy. We still have the core model loaded in the element factory. From this model we can create a custom diagram. With a little help of the auto-layout service, we can make it a readable diagram.

To create the diagram, we *drop elements* on the diagram. Items on a diagram represent an element in the model. We'll also drop all associations on the model. Only if both ends can connect, the association will be added.

```
from gaphor.diagram.drop import drop
from gaphor.extensions.ipython import auto_layout

temp_diagram = element_factory.create(Diagram)

for name in ["Presentation", "Diagram", "Element"]:
    element = next(element_factory.select(
        lambda e: isinstance(e, UML.Class) and e.name == name
    ))
    drop(element, temp_diagram, x=0, y=0)

# Drop all assocations, see what sticks
for association in element_factory.lselect(UML.Association):
    drop(association, temp_diagram, x=0, y=0)

auto_layout(temp_diagram)

draw(temp_diagram, format="svg")
```

```
<IPython.core.display.SVG object>
```

The diagram is not perfect, but you get the picture.

5.5 Update a model

Updating a model always starts with the element factory: that's where elements are created.

To create a UML Class instance, you can:

```
my_class = element_factory.create(UML.Class)
my_class.name = "MyClass"
```

To give it an attribute, create an attribute type (`UML.Property`) and then assign the attribute values.

```
my_attr = element_factory.create(UML.Property)
my_attr.name = "my_attr"
my_attr.typeValue = "string"
my_class.ownedAttribute = my_attr
```

Adding it to the diagram looks like this:

```
my_diagram = element_factory.create(Diagram)
drop(my_class, my_diagram, x=0, y=0)
draw(my_diagram, format="svg")
```

```
<IPython.core.display.SVG object>
```

If you save the model, your changes are persisted:

```
with open("../my-model.gaphor", "w") as out:
    storage.save(out, element_factory)
```

5.6 What else

What else is there to know…

- Gaphor supports derived associations. For example, `element.owner` points to the owner element. For an attribute that would be its containing class.
- All data models are described in the `Modeling Languages` section of the docs.
- If you use Gaphor's Console, you'll need to apply all changes in a transaction, or they will result in an error.
- If you want a comprehensive example of a code generator, have a look at Gaphor's `coder` module. This module is used to generate the code for the data models used by Gaphor.
- This page is rendered with MyST-NB. It's actually a Jupyter Notebook!

5.7 Examples

Here is another example:

5.7.1 Example: Gaphor services

In this example we're doing something a little less trivial. In Gaphor, services are defined as entry points. Each service is a class, and takes parameters with names that match other services. This allows services to depend on other services.

It looks something like this:

```
# entry point name: my_service
class MyService:
    ...

# entry point name: my_other_service
class MyOtherService:
    def __init__(self, my_service):
        ...
```

Let's first load the entry points.

```
from gaphor.entrypoint import load_entry_points

entry_points = load_entry_points("gaphor.services")

entry_points
```

```
/home/docs/checkouts/readthedocs.org/user_builds/gaphor/envs/latest/lib/python3.11/site-
↪packages/gaphor/diagram/text.py:6: PyGIWarning: Pango was imported without specifying␣
↪a version first. Use gi.require_version('Pango', '1.0') before import to ensure that␣
↪the right version gets loaded.
  from gi.repository import Pango, PangoCairo
/home/docs/checkouts/readthedocs.org/user_builds/gaphor/envs/latest/lib/python3.11/site-
↪packages/gaphor/diagram/text.py:6: PyGIWarning: PangoCairo was imported without␣
↪specifying a version first. Use gi.require_version('PangoCairo', '1.0') before import␣
↪to ensure that the right version gets loaded.
  from gi.repository import Pango, PangoCairo
```

```
{'auto_layout': gaphor.plugins.autolayout.pydot.AutoLayoutService,
 'component_registry': gaphor.services.componentregistry.ComponentRegistry,
 'console_window': gaphor.plugins.console.consolewindow.ConsoleWindow,
 'copy': gaphor.ui.copyservice.CopyService,
 'diagram_export': gaphor.plugins.diagramexport.DiagramExport,
 'diagrams': gaphor.ui.diagrams.Diagrams,
 'element_dispatcher': gaphor.core.modeling.elementdispatcher.ElementDispatcher,
 'element_editor': gaphor.ui.elementeditor.ElementEditor,
 'element_factory': gaphor.core.modeling.elementfactory.ElementFactory,
 'event_manager': gaphor.core.eventmanager.EventManager,
 'export_menu': gaphor.ui.menufragment.MenuFragment,
 'file_manager': gaphor.ui.filemanager.FileManager,
 'main_window': gaphor.ui.mainwindow.MainWindow,
```

(continues on next page)

(continued from previous page)

```
 'model_browser': gaphor.ui.modelbrowser.ModelBrowser,
 'modeling_language': gaphor.services.modelinglanguage.ModelingLanguageService,
 'properties': gaphor.services.properties.Properties,
 'recent_files': gaphor.ui.recentfiles.RecentFiles,
 'sanitizer': gaphor.UML.sanitizerservice.SanitizerService,
 'toolbox': gaphor.ui.toolbox.Toolbox,
 'tools_menu': gaphor.ui.menufragment.MenuFragment,
 'undo_manager': gaphor.services.undomanager.UndoManager,
 'xmi_export': gaphor.plugins.xmiexport.XMIExport}
```

Now let's create a component in our model for every service.

```python
from gaphor import UML
from gaphor.core.modeling import ElementFactory

element_factory = ElementFactory()

def create_component(name):
    c = element_factory.create(UML.Component)
    c.name = name
    return c

components = {name: create_component(name) for name in entry_points}
components
```

```
{'auto_layout': <gaphor.UML.uml.Component element 67d0fb4c-e866-11ed-b111-0242ac110002>,
 'component_registry': <gaphor.UML.uml.Component element 67d0fea8-e866-11ed-b111-
↪0242ac110002>,
 'console_window': <gaphor.UML.uml.Component element 67d1002e-e866-11ed-b111-
↪0242ac110002>,
 'copy': <gaphor.UML.uml.Component element 67d101aa-e866-11ed-b111-0242ac110002>,
 'diagram_export': <gaphor.UML.uml.Component element 67d102f4-e866-11ed-b111-
↪0242ac110002>,
 'diagrams': <gaphor.UML.uml.Component element 67d10416-e866-11ed-b111-0242ac110002>,
 'element_dispatcher': <gaphor.UML.uml.Component element 67d10542-e866-11ed-b111-
↪0242ac110002>,
 'element_editor': <gaphor.UML.uml.Component element 67d10696-e866-11ed-b111-
↪0242ac110002>,
 'element_factory': <gaphor.UML.uml.Component element 67d10808-e866-11ed-b111-
↪0242ac110002>,
 'event_manager': <gaphor.UML.uml.Component element 67d10920-e866-11ed-b111-0242ac110002>
↪,
 'export_menu': <gaphor.UML.uml.Component element 67d10a1a-e866-11ed-b111-0242ac110002>,
 'file_manager': <gaphor.UML.uml.Component element 67d10b1e-e866-11ed-b111-0242ac110002>,
 'main_window': <gaphor.UML.uml.Component element 67d10c22-e866-11ed-b111-0242ac110002>,
 'model_browser': <gaphor.UML.uml.Component element 67d10d3a-e866-11ed-b111-0242ac110002>
↪,
 'modeling_language': <gaphor.UML.uml.Component element 67d10e34-e866-11ed-b111-
↪0242ac110002>,
 'properties': <gaphor.UML.uml.Component element 67d10f24-e866-11ed-b111-0242ac110002>,
 'recent_files': <gaphor.UML.uml.Component element 67d11014-e866-11ed-b111-0242ac110002>,
 'sanitizer': <gaphor.UML.uml.Component element 67d1110e-e866-11ed-b111-0242ac110002>,
```

(continues on next page)

(continued from previous page)

```
 'toolbox': <gaphor.UML.uml.Component element 67d111fe-e866-11ed-b111-0242ac110002>,
 'tools_menu': <gaphor.UML.uml.Component element 67d112ee-e866-11ed-b111-0242ac110002>,
 'undo_manager': <gaphor.UML.uml.Component element 67d1144c-e866-11ed-b111-0242ac110002>,
 'xmi_export': <gaphor.UML.uml.Component element 67d11546-e866-11ed-b111-0242ac110002>}
```

With all components mapped, we can create dependencies between those components, based on the constructor parameter names.

```
import inspect

for name, cls in entry_points.items():
    for param_name in inspect.signature(cls).parameters:
        if param_name not in components:
            continue

        dep = element_factory.create(UML.Usage)
        dep.client = components[name]
        dep.supplier = components[param_name]
```

With all elements in the model, we can create a diagram. Let's drop the components and dependencies on the diagram and let auto-layout do its magic.

To make the dependency look good, we have to add a style sheet. If you create a new diagram via the GUI, this element is automatically added.

```
from gaphor.core.modeling import Diagram, StyleSheet
from gaphor.diagram.drop import drop

element_factory.create(StyleSheet)
diagram = element_factory.create(Diagram)

for element in element_factory.lselect():
    drop(element, diagram, x=0, y=0)
```

Last step is to layout and draw the diagram.

```
from gaphor.extensions.ipython import auto_layout, draw

auto_layout(diagram)

draw(diagram, format="svg")
```

```
<IPython.core.display.SVG object>
```

That's all. As you can see from the diagram, a lot of services rely on `EventManager`.

CHAPTER

SIX

STEREOTYPES

In UML, stereotypes are way to extend the application of the UML language to new domains. For example: SysML started as a profile for UML.

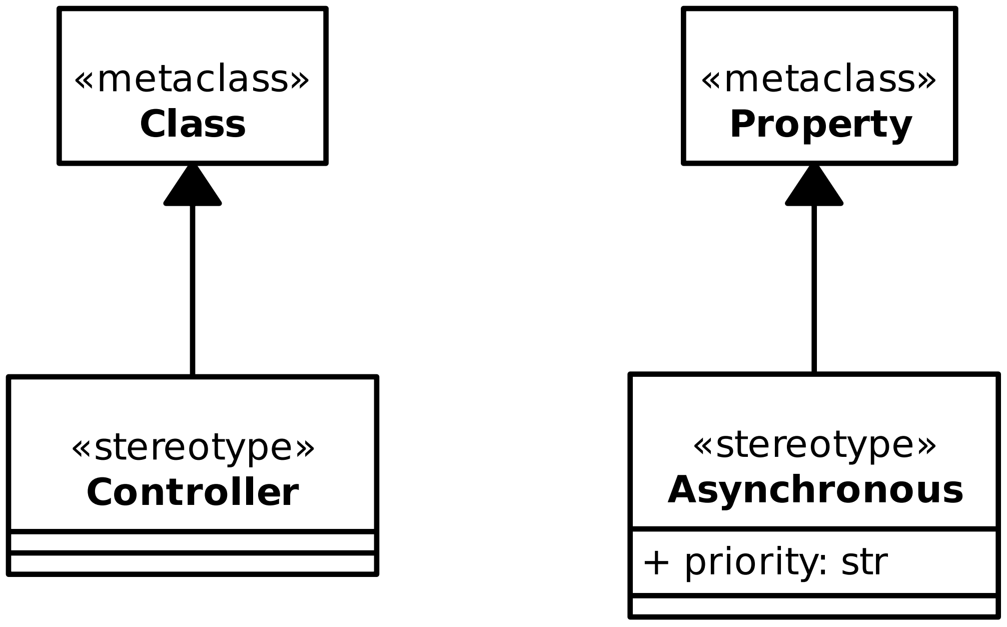
Gaphor supports stereotypes too. They're *the* way for you to adapt your models to your specific needs.

The UML, SysML, RAAML and other models used in Gaphor – the code is generated from Gaphor model files – make use of stereotypes to provide specific information used when generating the data model code.

To create a stereotype, ensure the UML Profile is active and open the *Profile* section of the toolbox. First add a *Metaclass* to your diagram. Next add a *Stereotype*, and connect both with a *Extension*. The `«metaclass»` stereotype will only show once the *Extension* is connected both class and stereotype.

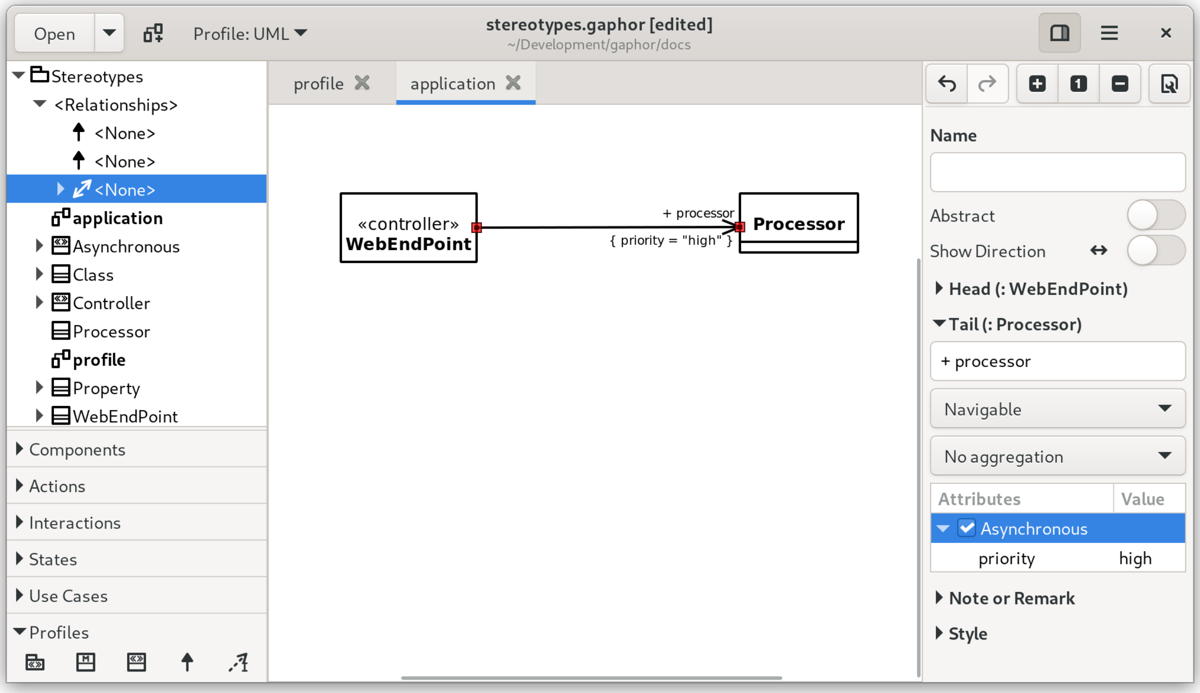
Note: The class names in the metaclass should be a class name from the UML model, such as `Class`, `Interface`, `Property`, `Association`. Or even `Element` if you want to use the stereotype on all elements.

Your stereotype declaration may look something like this:



The `Asynchronous` stereotype has a property `priority`. This property can be proved a value once the stereotype is applied to a *Property*, such as an association end.

When a stereotype can be applied to a model element, a *Stereotype* section will appear in the editor.



Attributes	Value
✓ Asynchronous	
priority	high

CHAPTER

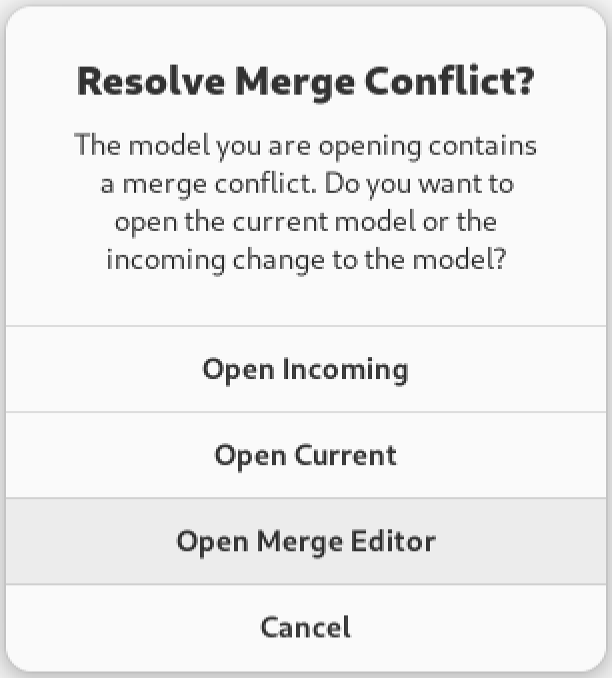
SEVEN

RESOLVE MERGE CONFLICTS

Suppose you're working on a model. If you create a change, while someone else has also made changes, there's a fair chance you'll end up with a merge conflict.

Gaphor tries to make the changes to a model as small as possible: all elements are stored in the same order. However, since a Gaphor model is a persisted graph of objects, merging changes is not as simple as opening a text editor.

From Gaphor 2.18 onwards Gaphor is also capable of merging models. Once a merge conflict has been detected, Gaphor will offer the option to open the current model, the incoming model or merge changes manually via the Merge Editor.



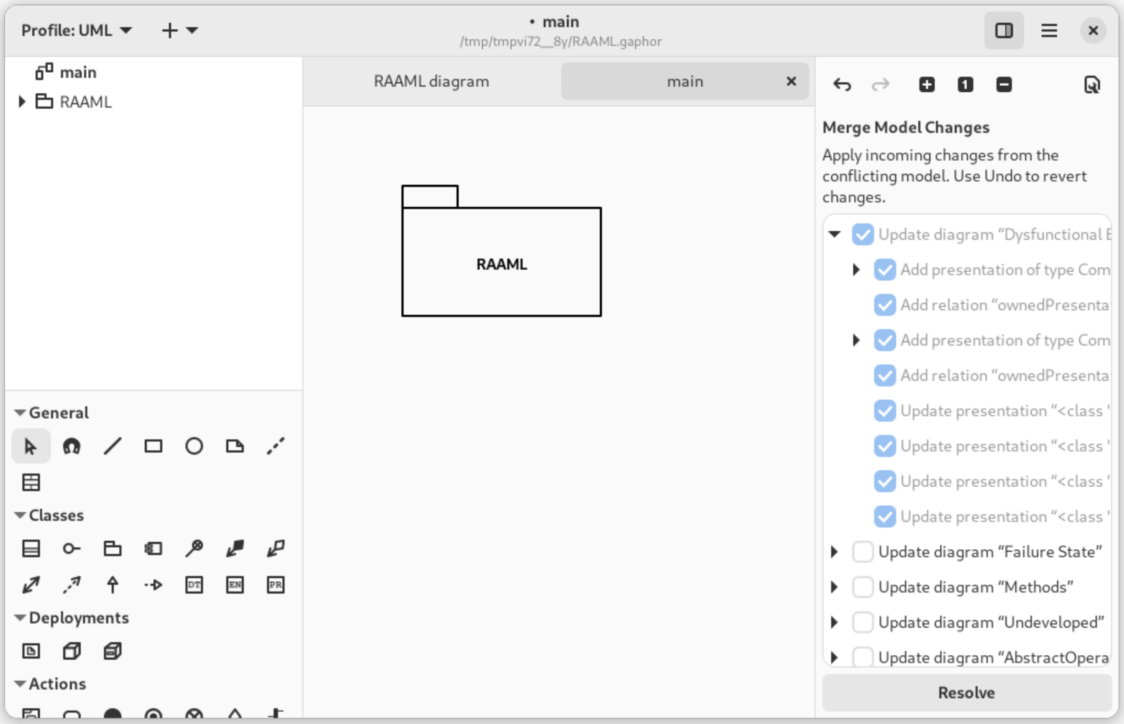
Resolve Merge Conflict?

The model you are opening contains a merge conflict. Do you want to open the current model or the incoming change to the model?

If you choose *Open Merge Editor*, both models will be loaded. The current model remains as is. In addition, the changes made to the incoming model are calculated. Those changes are stored as *pending change* objects in the model.

Tip: Pending changes are part of the model, you can save the model with changes and resolve those at a later point.

The Merge Editor is shown on the right side, replacing the (normal) Property Editor.



Merge Model Changes

Apply incoming changes from the conflicting model. Use Undo to revert changes.

- ✓ Update diagram "Dysfunctional E
  - ✓ Add presentation of type Com
  - ✓ Add relation "ownedPresenta
  - ✓ Add presentation of type Com
  - ✓ Add relation "ownedPresenta
  - ✓ Update presentation "<class '
  - ✓ Update presentation "<class '
  - ✓ Update presentation "<class '
  - ✓ Update presentation "<class '
- ☐ Update diagram "Failure State"
- ☐ Update diagram "Methods"
- ☐ Update diagram "Undeveloped"
- ☐ Update diagram "AbstractOpera

Merge actions are grouped by diagram, where possible. When you apply a change, all changes listed as children are also applied. Once changes are applied, they can only be reverted by undoing the change (hit *Undo*).

Note: The Merge Editor replaces the Property Editor, as long as there are pending changes in the model.

It is concidered good practice to resolve the merge conflict before you continue modeling.

When all conflicts have been resolved, press *Resolve* to finish merge conflict resolution.

CHAPTER

EIGHT

GAPHOR ON LINUX

Gaphor can be installed as Flatpak and AppImage on Linux, some distributions provide packages. Check out the Gaphor download page for details.

Older releases are available from GitHub.

CI builds are also available.

8.1 Development Environment

There are two ways to set up a development environment:

1. *GNOME Builder*, ideal for "drive by" contributions.
2. *A local environment*.

8.1.1 GNOME Builder

Open GNOME Builder 43 or newer, clone the repository. Check if the *Build Profile* is set to `org.gaphor.Gaphor.json`. If so, hit the *Run* button to start the application.

8.1.2 A Local Environment

To set up a development environment with Linux, you first need a fairly new Linux distribution version. For example, the latest Ubuntu LTS or newer, Arch, Debian Testing, SUSE Tumbleweed, or similar. Gaphor depends on newer versions of GTK, and we don't test for backwards compatibility. You will also need the latest stable version of Python. In order to get the latest stable version without interfering with your system-wide Python version, we recommend that you install pyenv.

Install the pyenv prerequisites first, and then install pyenv:

```
curl https://pyenv.run | bash
```

Make sure you follow the instruction at the end of the installation script to install the commands in your shell's rc file. Next install the latest version of Python by executing:

```
pyenv install 3.x.x
```

Where 3.x.x is replaced by the latest stable version of Python (pyenv should let you tab-complete available versions).

Next install the Gaphor prerequisites by installing the gobject introspection and cairo build dependencies, for example, in Ubuntu execute:

```
sudo apt-get install -y python3-dev python3-gi python3-gi-cairo
gir1.2-gtk-4.0 libgirepository1.0-dev libcairo2-dev libgtksourceview-5-dev
```

Install Poetry using pipx:

```
pipx install poetry
```

Clone the repository.

```
cd gaphor
# activate latest python for this project
pyenv local 3.x.x # 3.x.x is the version you installed earlier
poetry env use 3.x # ensures poetry /consistently/ uses latest major release
poetry config virtualenvs.in-project true
poetry install
poetry run gaphor
```

NOTE: Gaphor requires GTK 4. It works best with GTK >=4.8 and libadwaita >=1.2.

8.2 Create a Flatpak Package

The main method that Gaphor is packaged for Linux is with a Flatpak package. Flatpak is a software utility for software deployment and package management for Linux. It offer a sandbox environment in which users can run application software in isolation from the rest of the system.

We distribute the official Flatpak using Flathub, and building of the image is done at the Gaphor Flathub repository.

1. Install Flatpak
2. Install flatpak-builder

   ```
   sudo apt-get install flatpak-builder
   ```

3. Install the GNOME SDK

   ```
   flatpak install flathub org.gnome.Sdk 43
   ```

4. Clone the Flathub repository and install the necessary SDK:

   ```
   git clone https://github.com/flathub/org.gaphor.Gaphor.git
   cd org.gaphor.Gaphor
   make setup
   ```

5. Build Gaphor Flatpak

   ```
   make
   ```

6. Install the Flatpak

   ```
   make install
   ```

8.3 Create an AppImage Package

AppImage is a format for distributing portable software on Linux without needing superuser permissions to install the application. The AppImage file is one executable which contains both Gaphor and Python. It allows Gaphor to be run on any AppImage supported platform without installation or root access.

We build our AppImage by first bundling Gaphor with PyInstaller and then converting it in to an AppImage.

In order for the built AppImage to be compatible with older versions of Linux, we build the release versions using Docker with an older LTS version of Ubuntu.

```
poetry run poe package
cd _packaging/appimage
make dist
```

8.4 Linux Distribution Packages

Examples of Gaphor and Gaphas RPM spec files can be found in PLD Linux repository:

- https://github.com/pld-linux/python-gaphas
- https://github.com/pld-linux/gaphor

There is also an Arch User Repository (AUR) for Gaphor available for Arch users.

Please, do not hesitate to contact us if you need help to create a Linux package for Gaphor or Gaphas.

CHAPTER

NINE

GAPHOR ON MACOS

The latest release of Gaphor can be downloaded from the Gaphor download page. Gaphor can also be installed as a Homebrew cask.

Older releases are available from GitHub.

CI builds are also available.

9.1 Development Environment

To setup a development environment with macOS:

1. Install Homebrew
2. Open a terminal and execute:

```
brew install python3 gobject-introspection gtk4 gtksourceview5 libadwaita adwaita-icon-
↪theme graphviz
```

Install Poetry using pipx:

```
pipx install poetry
```

Clone the repository.

```
cd gaphor
poetry config virtualenvs.in-project true
poetry install
poetry run gaphor
```

If PyGObject does not compile and complains about a missing `ffi.h` file, set the following environment variable and run `poetry install` again:

```
export PKG_CONFIG_PATH=/usr/local/opt/libffi/lib/pkgconfig
poetry install
```

9.2 Packaging for macOS

In order to create an exe installation package for macOS, we utilize PyInstaller which analyzes Gaphor to find all the dependencies and bundle them in to a single folder.

1. Follow the instructions for settings up a development environment above
2. Open a terminal and execute the following from the repository directory:

```
poetry install --with packaging
poetry run poe package
```

CHAPTER

TEN

GAPHOR ON WINDOWS

Gaphor can be installed as with our installer. Check out the Gaphor download page for details.

Older releases are available from GitHub.

CI builds are also available.

10.1 Development Environment

10.1.1 Choco

We recommend using Chocolately as a package manager in Windows.

To install it, open PowerShell as an administrator, then execute:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.Net.WebClient).
↪DownloadString('https://community.chocolatey.org/install.ps1'))
```

To run local scripts in follow-on steps, also execute

```
Set-ExecutionPolicy RemoteSigned
```

This allows for local PowerShell scripts to run without signing, but still requires signing for remote scripts.

10.1.2 Git

To setup a development environment in Windows first install Git by executing as an administrator:

```
choco install git
```

10.1.3 MSYS2

The development environment in the next step needs MSYS2 installed to provide some Linux command line tools in Windows.

Keep PowerShell open as administrator and install MSYS2:

```
choco install msys2
```

10.1.4 GTK and Python with gvsbuild

gvsbuild provides a Python script helps you build the GTK library stack for Windows using Visual Studio. By compiling GTK with Visual Studio, we can then use a standard Python development environment in Windows.

First we will install the gvsbuild dependencies:

1. Visual C++ build tools workload for Visual Studio 2022 Build Tools
2. Python

Install Visual Studio 2022

With your admin PowerShell terminal:

```
choco install visualstudio2022-workload-vctools
```

Install the Latest Python

In Windows, The full installer contains all the Python components and is the best option for developers using Python for any kind of project.

For more information on how to use the official installer, please see the full installer instructions. The default installation options should be fine for use with Gaphor.

1. Install the latest Python version using the official installer.
2. Open a PowerShell terminal as a normal user and check the python version:

   ```
   py -3.11 --version
   ```

Install Graphviz

Graphviz is used by Gaphor for automatic diagram formatting.

1. Install from Chocolately with administrator PowerShell:

   ```
   choco install graphviz
   ```

2. Restart your PowerShell terminal as a normal user and check that the dot command is available:

   ```
   dot -?
   ```

Install pipx

From the regular user PowerShell terminal execute:

```
py -3.11 -m pip install --user pipx
py -3.11 -m pipx ensurepath
```

Install gvsbuild

From the regular user PowerShell terminal execute:

```
pipx install gvsbuild
```

Build GTK

In the same PowerShell terminal, execute:

```
gvsbuild build --enable-gi --py-wheel gobject-introspection gtk4 libadwaita␣
↪gtksourceview5 pygobject pycairo adwaita-icon-theme hicolor-icon-theme
```

Grab a coffee, the build will take a few minutes to complete.

10.1.5 Setup Gaphor

In the same PowerShell terminal, clone the repository:

```
cd (to the location you want to put Gaphor)
git clone https://github.com/gaphor/gaphor.git
cd gaphor
```

Install Poetry

```
pipx install poetry
poetry config virtualenvs.in-project true
```

Add GTK to your environmental variables:

```
$env:Path = $env:Path + ";C:\gtk-build\gtk\x64\release\bin"
$env:LIB = "C:\gtk-build\gtk\x64\release\lib"
$env:INCLUDE = "C:\gtk-build\gtk\x64\release\include;C:\gtk-build\gtk\x64\release\
↪include\cairo;C:\gtk-build\gtk\x64\release\include\glib-2.0;C:\gtk-build\gtk\x64\
↪release\include\gobject-introspection-1.0;C:\gtk-build\gtk\x64\release\lib\glib-2.0\
↪include;"
```

You can also edit your account’s Environmental Variables to persist across PowerShell sessions.

Install Gaphor’s dependencies

```
poetry install
```

Reinstall PyGObject and pycairo using gvsbuild wheels

```
poetry run pip install --force-reinstall (Resolve-Path C:\gtk-build\build\x64\release\
↪pygobject\dist\PyGObject*.whl)
poetry run pip install --force-reinstall (Resolve-Path C:\gtk-build\build\x64\release\
↪pycairo\dist\pycairo*.whl)
```

Launch Gaphor!

```
poetry run gaphor
```

10.1.6 Debugging using Visual Studio Code

Start a new PowerShell terminal, and set current directory to the project folder:

```
cd (to the location you put gaphor)
```

Ensure that path environment variable is set:

```
$env:Path = "C:\gtk-build\gtk\x64\release\bin;" + $env:Path
```

Start Visual Studio Code:

```
code .
```

To start the debugger, execute the following steps:

1. Open `__main__.py` file from `gaphor` folder
2. Add a breakpoint on line `main(sys.argv)`
3. In the menu, select Run → Start debugging
4. Choose Select module from the list
5. Enter `gaphor` as module name

Visual Studio Code will start the application in debug mode, and will stop at main.

10.2 Packaging for Windows

In order to create an exe installation package for Windows, we utilize PyInstaller which analyzes Gaphor to find all the dependencies and bundle them in to a single folder. We then use a custom bash script that creates a Windows installer using NSIS and a portable installer using 7-Zip. To install them, open PowerShell as an administrator, then execute:

```
choco install nsis 7zip
```

Then build your installer using:

```
poetry install --only main,packaging,automation
poetry build
poetry run poe package
poetry run poe win-installer
```

CHAPTER

ELEVEN

GAPHOR IN A CONTAINER

Instead of setting up a development environment locally, the easiest way to contribute to the project is using GitHub Codespaces.

11.1 GitHub Codespaces

Follow these steps to open Gaphor in a Codespace:

1. Navigate to https://github.com/gaphor/gaphor
2. Click the Code drop-down menu and select the **Open with Codespaces** option.
3. Select **+ New codespace** at the bottom on the pane.

For more info, check out the GitHub documentation.

CHAPTER

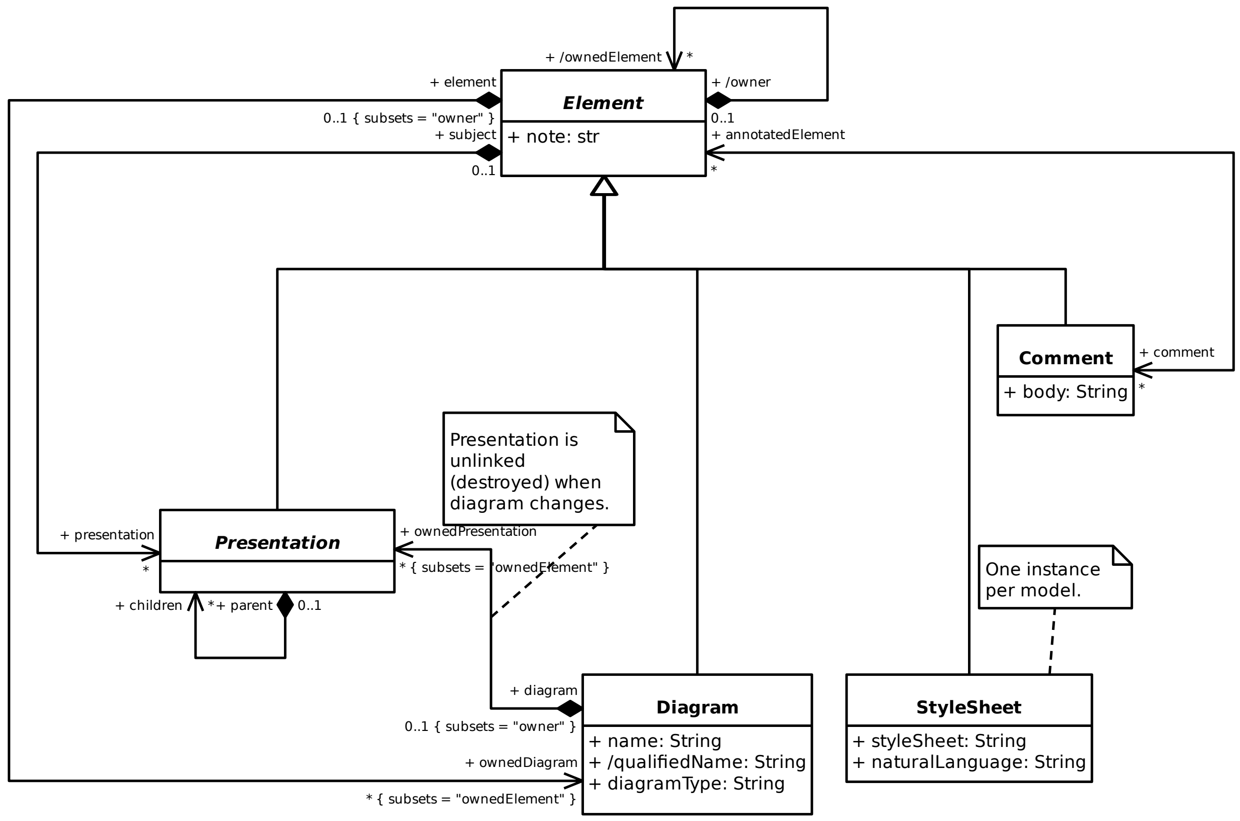
TWELVE

MODELING LANGUAGE CORE

The Core modeling language is the the basis for any other language.

The `Element` class acts as the root for all gaphor domain classes. `Diagram` and `Presentation` form the basis for everything you see in a diagram.

All data models in Gaphor are generated from actual Gaphor model files. This allows us to provide you nice diagrams of Gaphor's internal model.



The `Element` base class provides event notification and integrates with the model repository (internally known as `ElementFactory`). Bi-directional relationships are also possible, as well as derived relations.

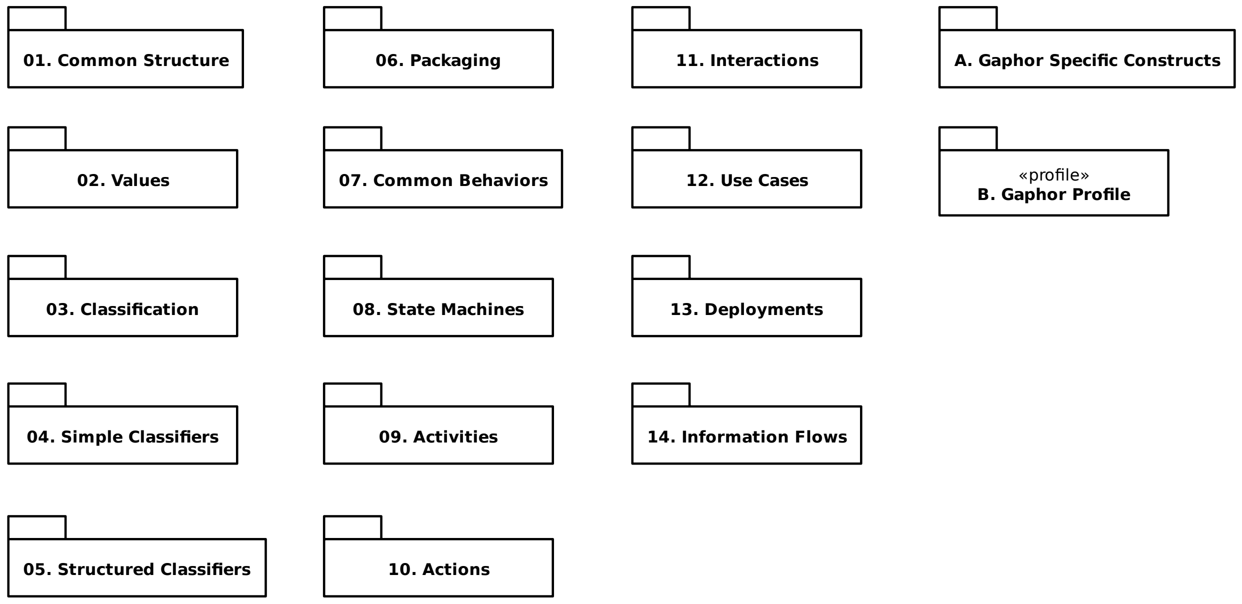
CHAPTER

THIRTEEN

UNIFIED MODELING LANGUAGE

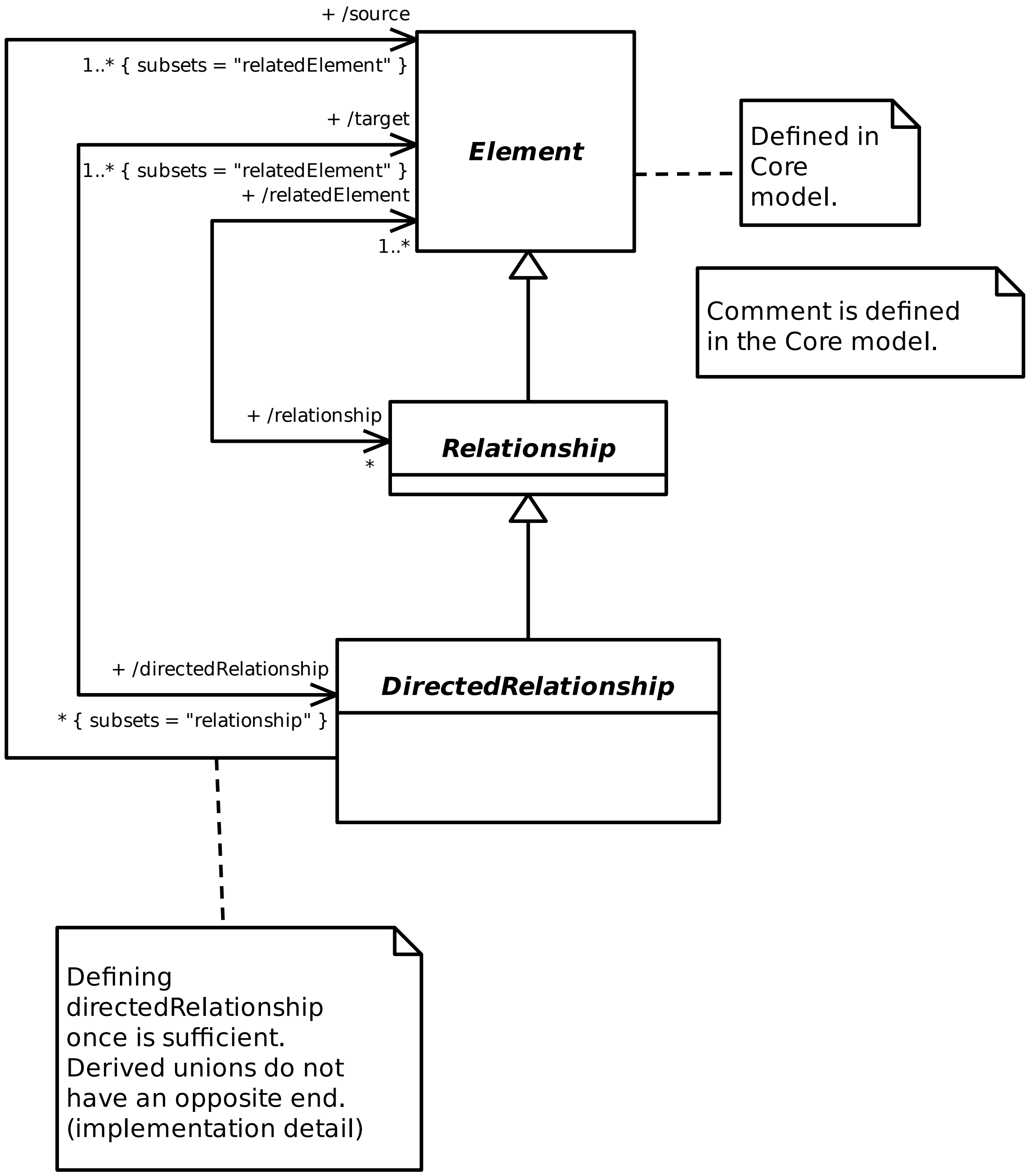
The UML model is the most extensive model in Gaphor. It is used as a base language for *SysML*, *RAAML*, and *C4*.

Gaphor follows the official UML 2.5.1 data model. Where changes have been made a comment has been added to the model. In particular where *m:n* relationships subset *1:n* relationships.



13.1 01. Common Structure

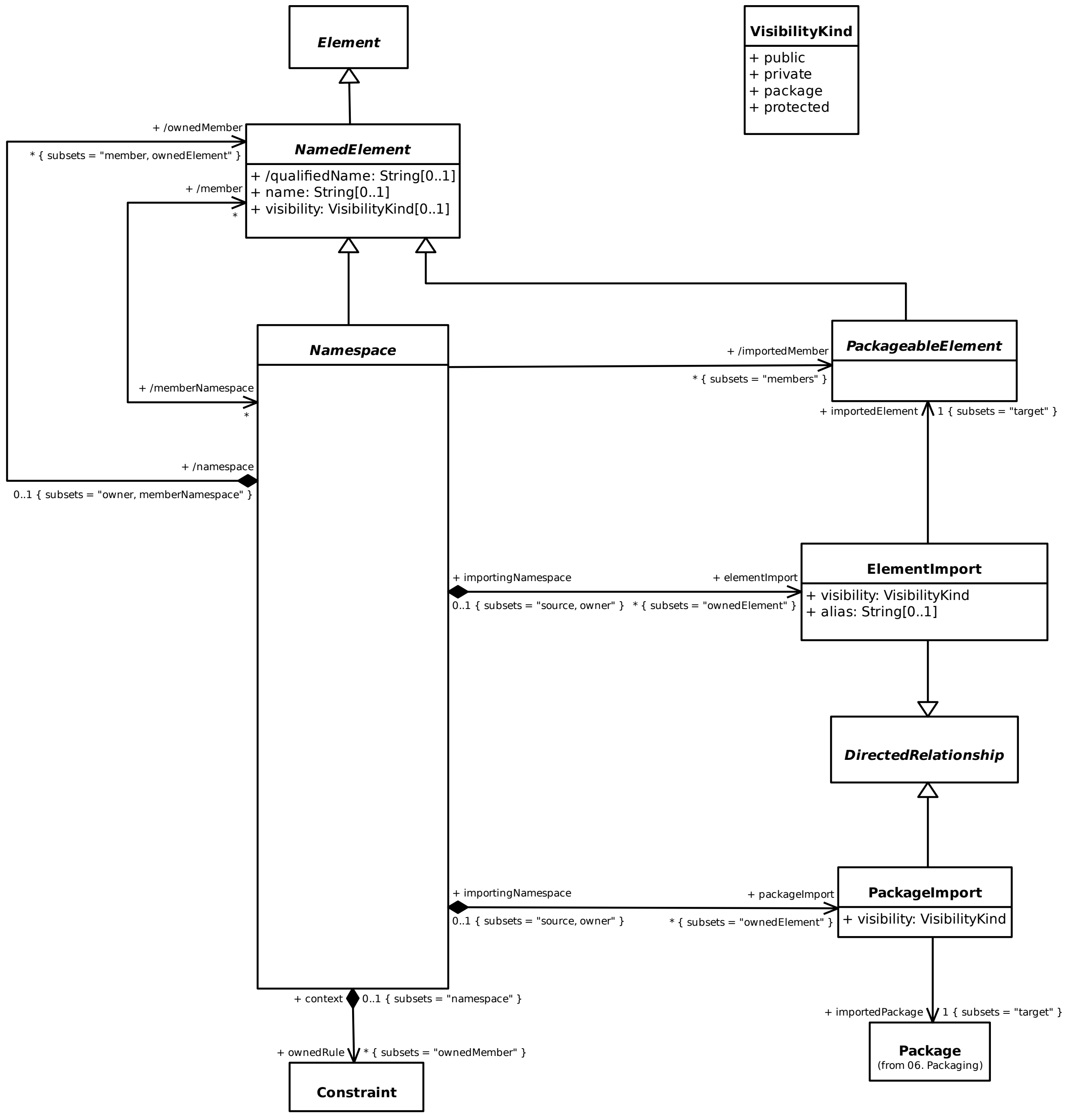
13.1.1 1. Root


+ /source
1..* { subsets = "relatedElement" }
+ /target
1..* { subsets = "relatedElement" }
+ /relatedElement
1..*
Element
Defined in Core model.
Comment is defined in the Core model.
+ /relationship
*
Relationship
+ /directedRelationship
* { subsets = "relationship" }
DirectedRelationship
Defining directedRelationship once is sufficient. Derived unions do not have an opposite end. (implementation detail)

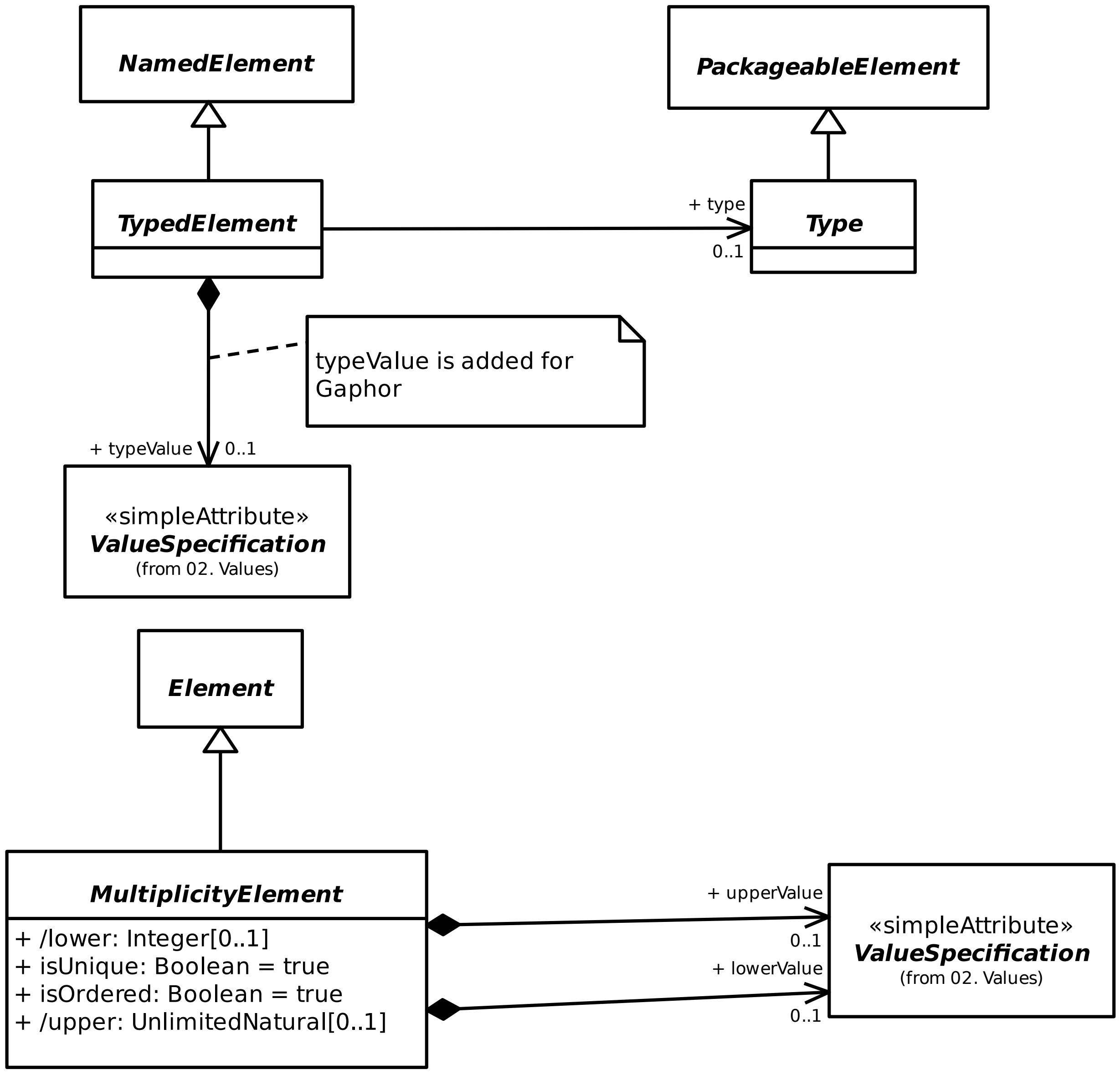
13.1.2 2. Templates

Not implemented.

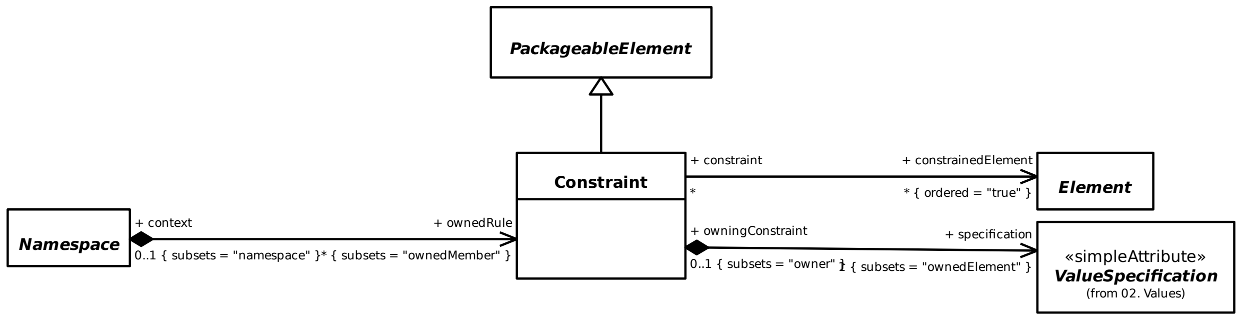
13.1.3 3. Namespaces



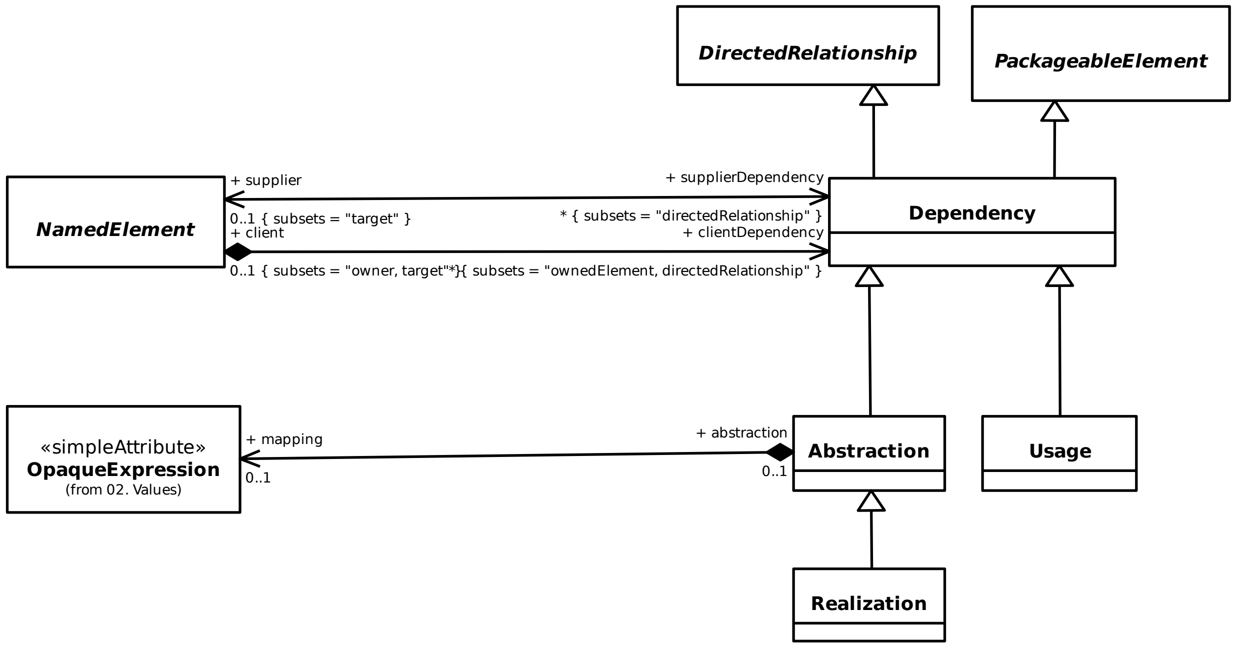
13.1.4 4. Types and Multiplicity



13.1.5 5. Constraints

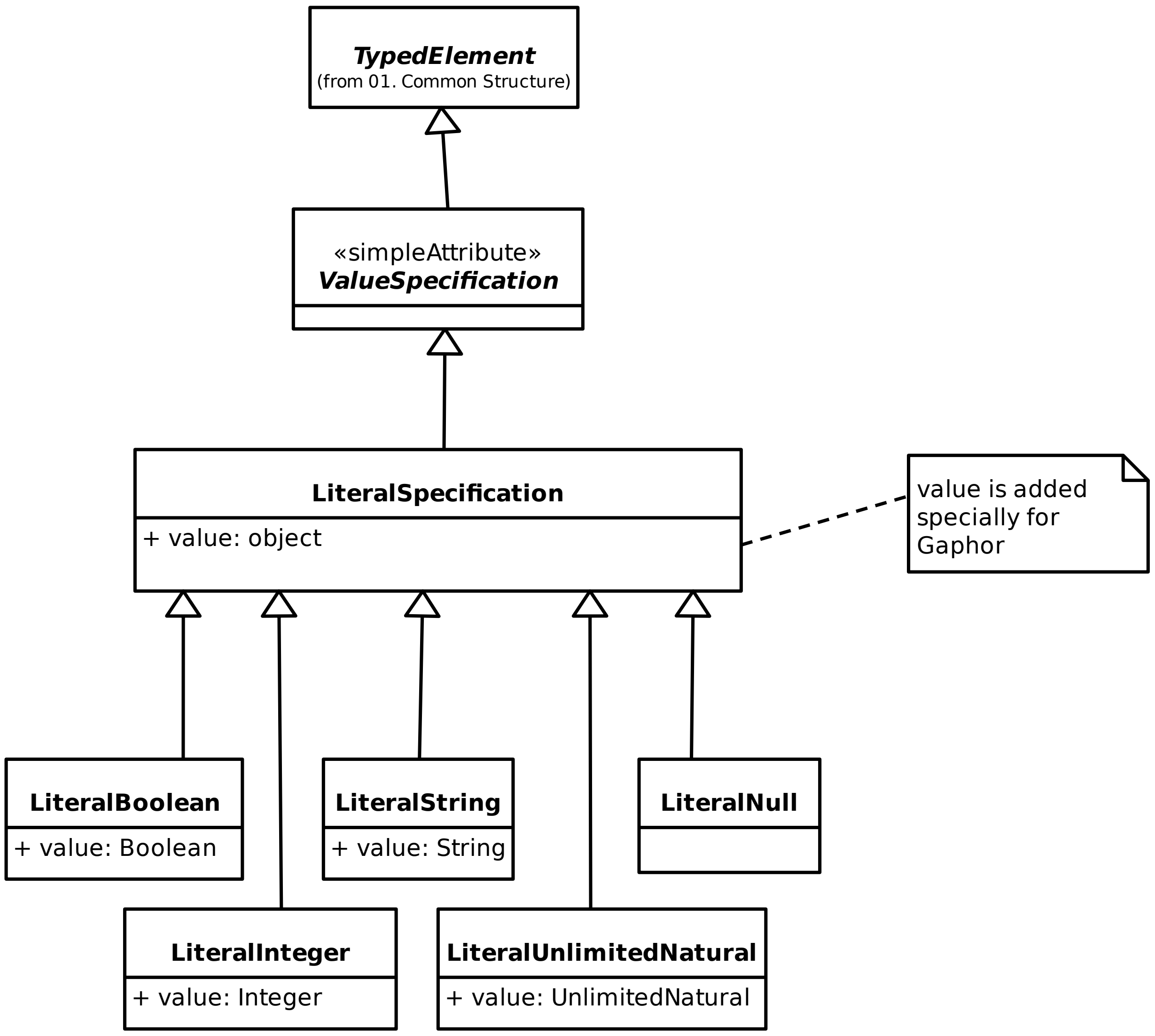


13.1.6 6. Dependencies

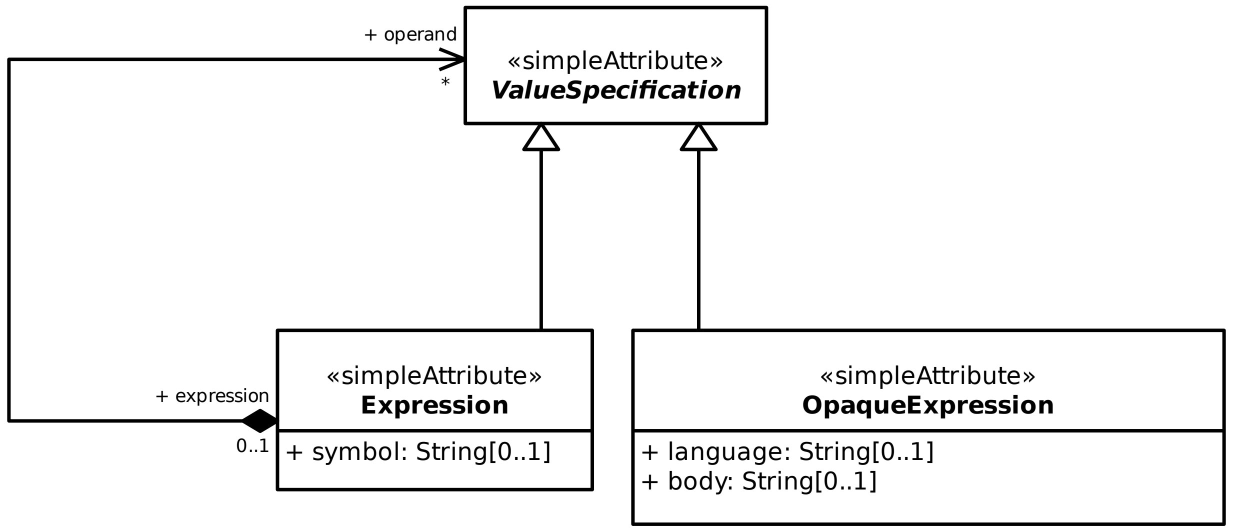


13.2 02. Values

13.2.1 1. Literals

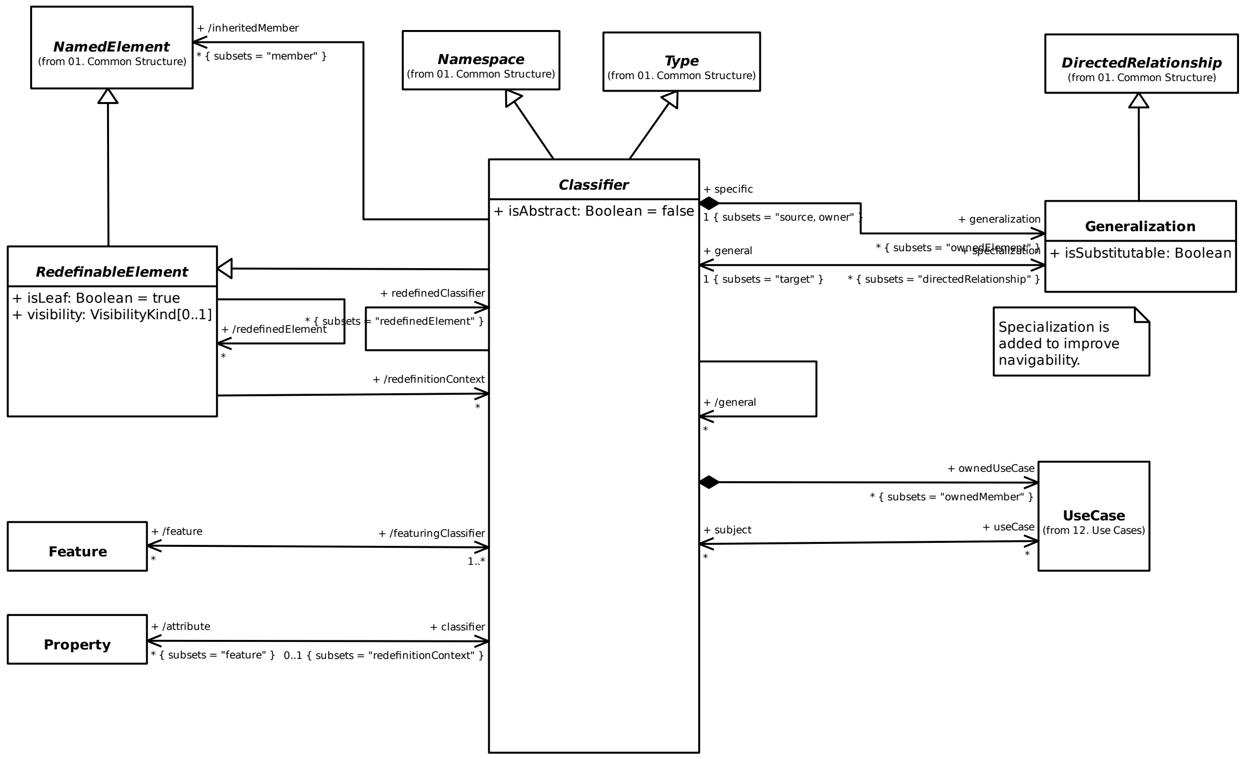


13.2.2 2. Expressions

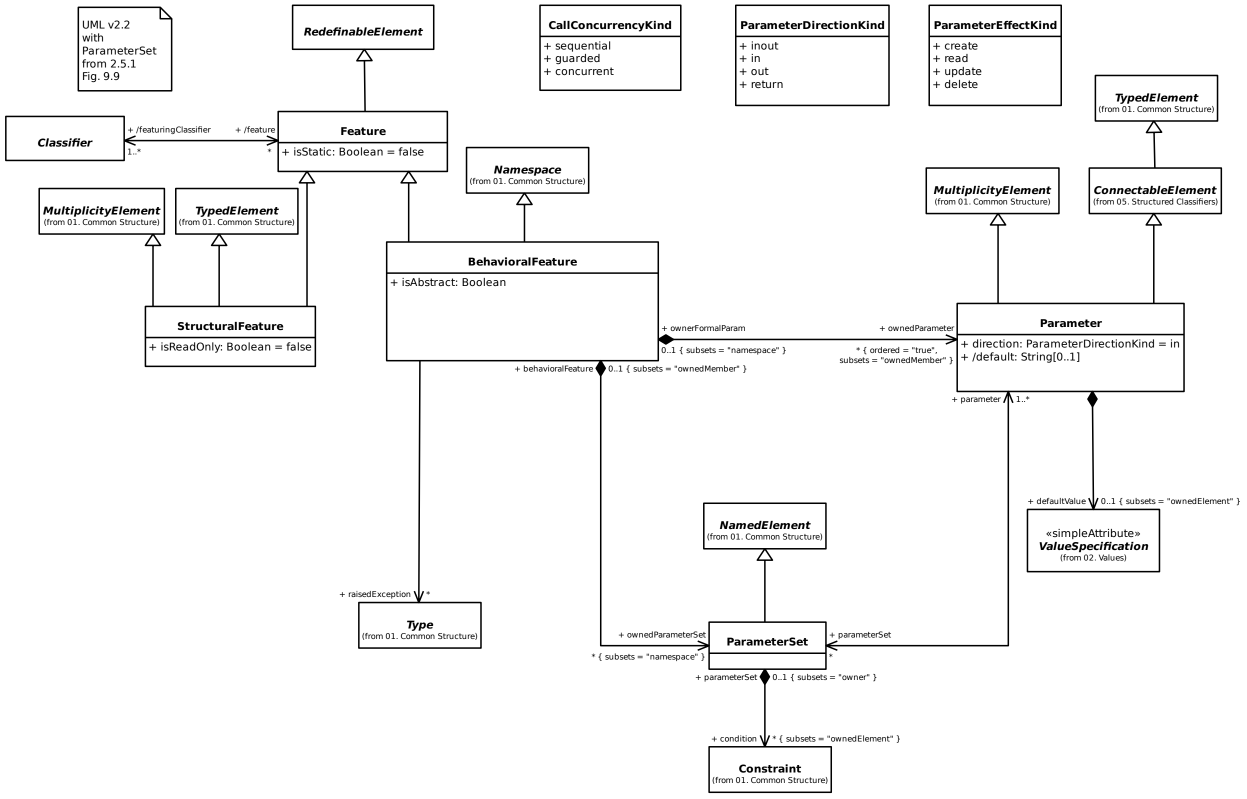


13.3 03. Classification

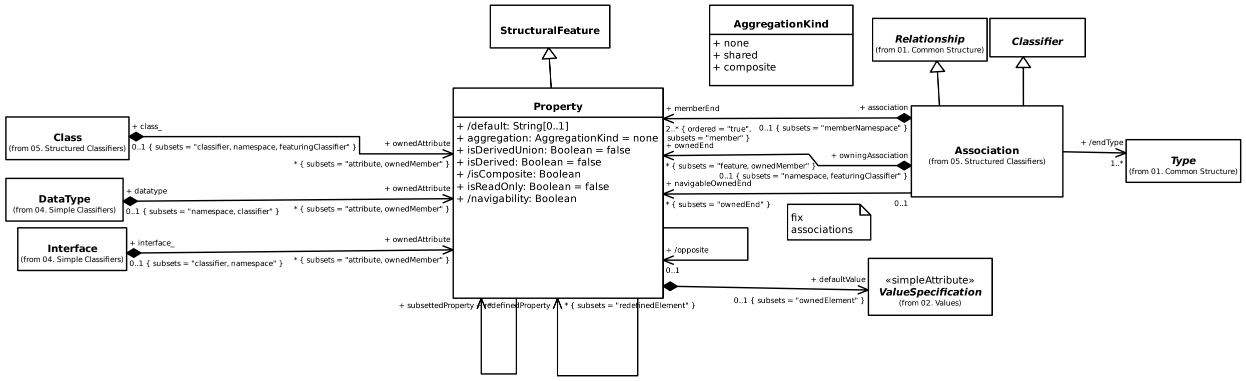
13.3.1 1. Classifiers


NamedElement
(from 01. Common Structure)
+ /inheritedMember
* { subsets = "member" }
Namespace
(from 01. Common Structure)
Type
(from 01. Common Structure)
DirectedRelationship
(from 01. Common Structure)
Classifier
+ isAbstract: Boolean = false
+ specific
1 { subsets = "source, owner" }
+ generalization
* { subsets = "ownedElement" }
Generalization
+ isSubstitutable: Boolean
+ general
1 { subsets = "target" }
+ specialization
* { subsets = "directedRelationship" }
Specialization is added to improve navigability.
RedefinableElement
+ isLeaf: Boolean = true
+ visibility: VisibilityKind[0..1]
+ redefinedClassifier
* { subsets = "redefinedElement" }
+ /redefinedElement
*
+ /redefinitionContext
*
+ /general
*
+ ownedUseCase
* { subsets = "ownedMember" }
UseCase
(from 12. Use Cases)
+ subject
*
+ useCase
*
Feature
+ /feature
*
+ /featuringClassifier
1..*
Property
+ /attribute
* { subsets = "feature" }
+ classifier
0..1 { subsets = "redefinitionContext" }

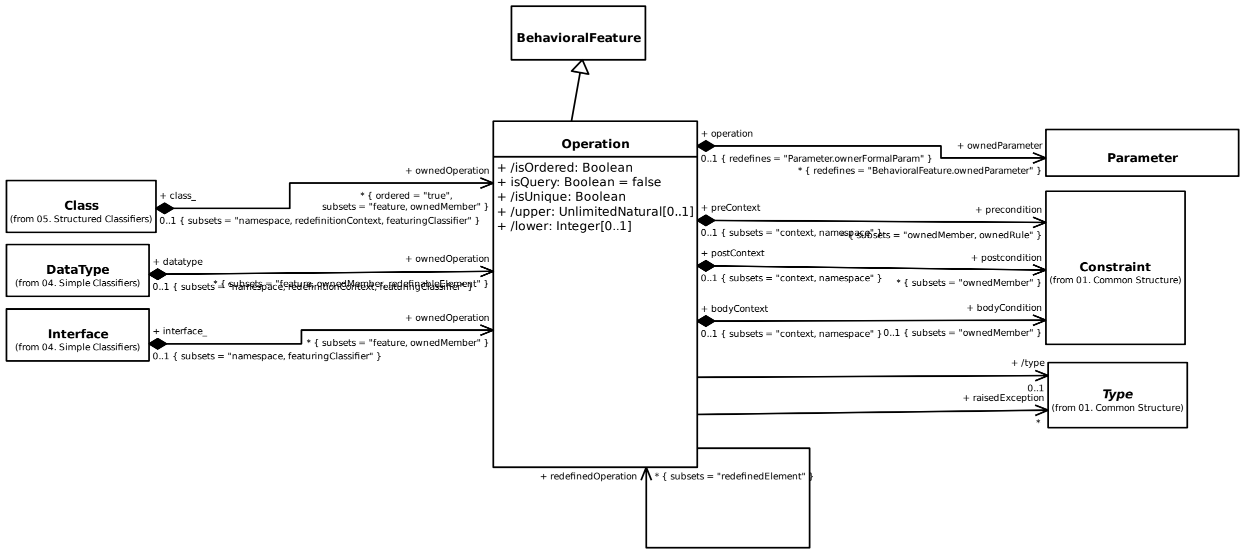
13.3.2 3. Features



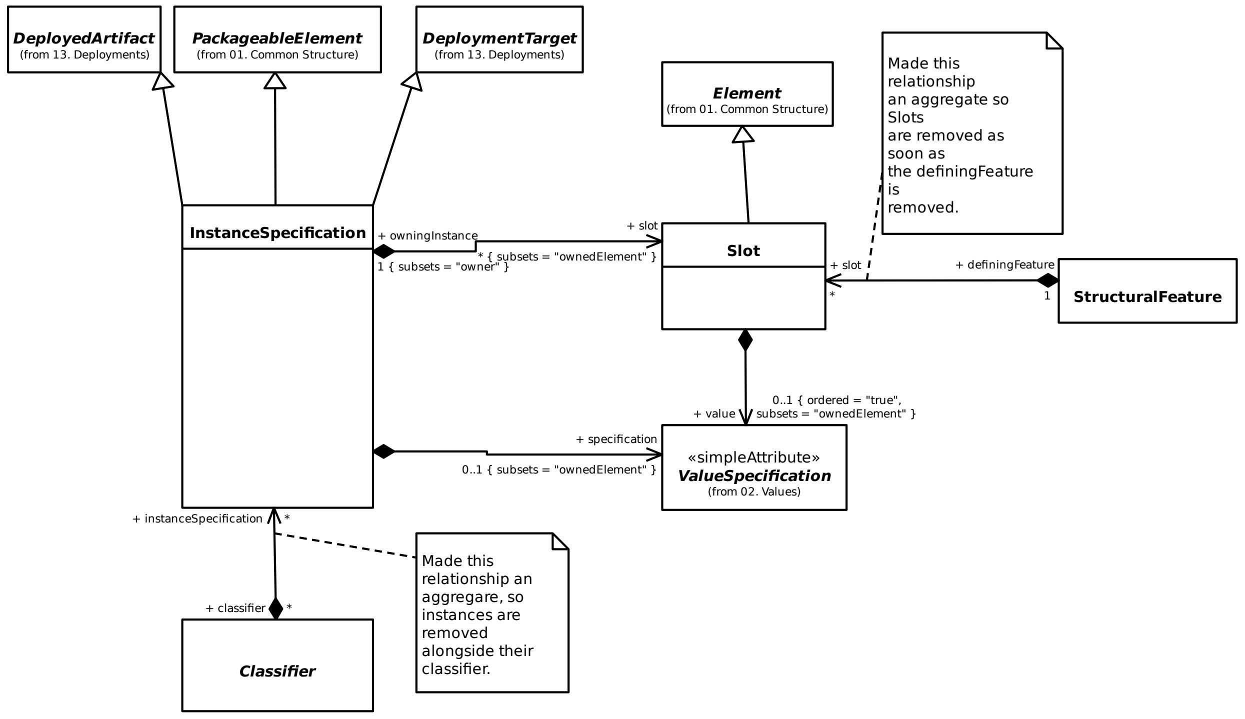
13.3.3 4. Properties



13.3.4 5. Operations

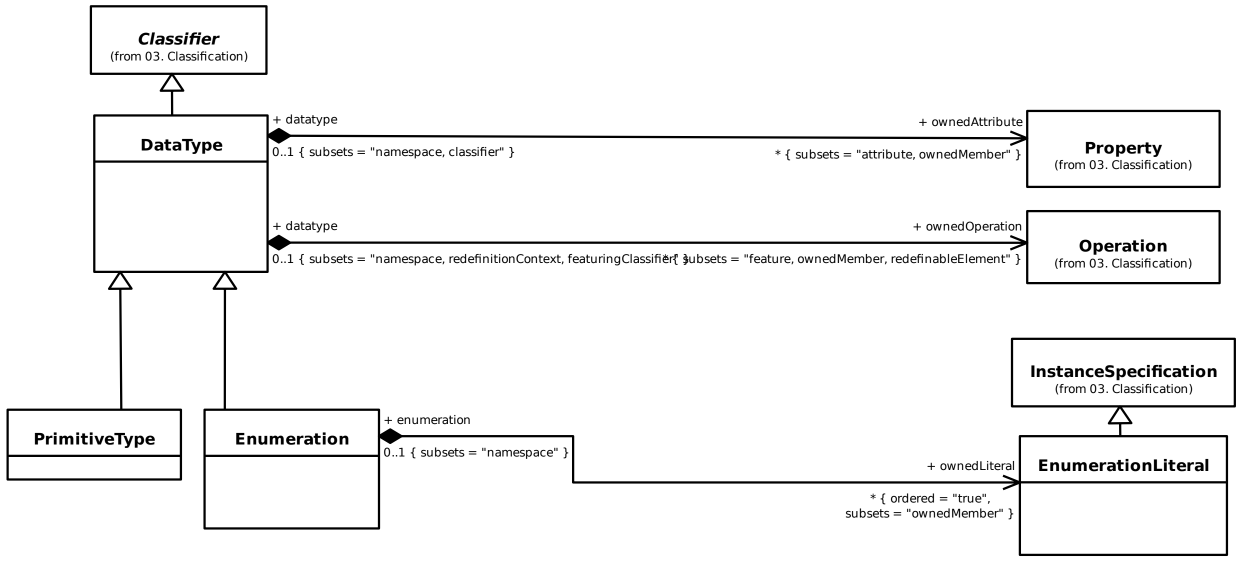

BehavioralFeature
Operation
+ /isOrdered: Boolean
+ isQuery: Boolean = false
+ /isUnique: Boolean
+ /upper: UnlimitedNatural[0..1]
+ /lower: Integer[0..1]
Class
(from 05. Structured Classifiers)
+ class_
0..1 { subsets = "namespace, redefinitionContext, featuringClassifier" }
+ ownedOperation
* { ordered = "true", subsets = "feature, ownedMember" }
DataType
(from 04. Simple Classifiers)
+ datatype
0..1 { subsets = "namespace, redefinitionContext, featuringClassifier" }
+ ownedOperation
* { subsets = "feature, ownedMember, redefinableElement" }
Interface
(from 04. Simple Classifiers)
+ interface_
0..1 { subsets = "namespace, featuringClassifier" }
+ ownedOperation
* { subsets = "feature, ownedMember" }
+ operation
0..1 { redefines = "Parameter.ownerFormalParam" }
+ ownedParameter
* { redefines = "BehavioralFeature.ownedParameter" }
Parameter
+ preContext
0..1 { subsets = "context, namespace" }
+ precondition
* { subsets = "ownedMember, ownedRule" }
+ postContext
0..1 { subsets = "context, namespace" }
+ postcondition
* { subsets = "ownedMember" }
+ bodyContext
0..1 { subsets = "context, namespace" }
+ bodyCondition
0..1 { subsets = "ownedMember" }
Constraint
(from 01. Common Structure)
+ /type
0..1
+ raisedException
*
Type
(from 01. Common Structure)
+ redefinedOperation
* { subsets = "redefinedElement" }

13.3.5 7. Instances

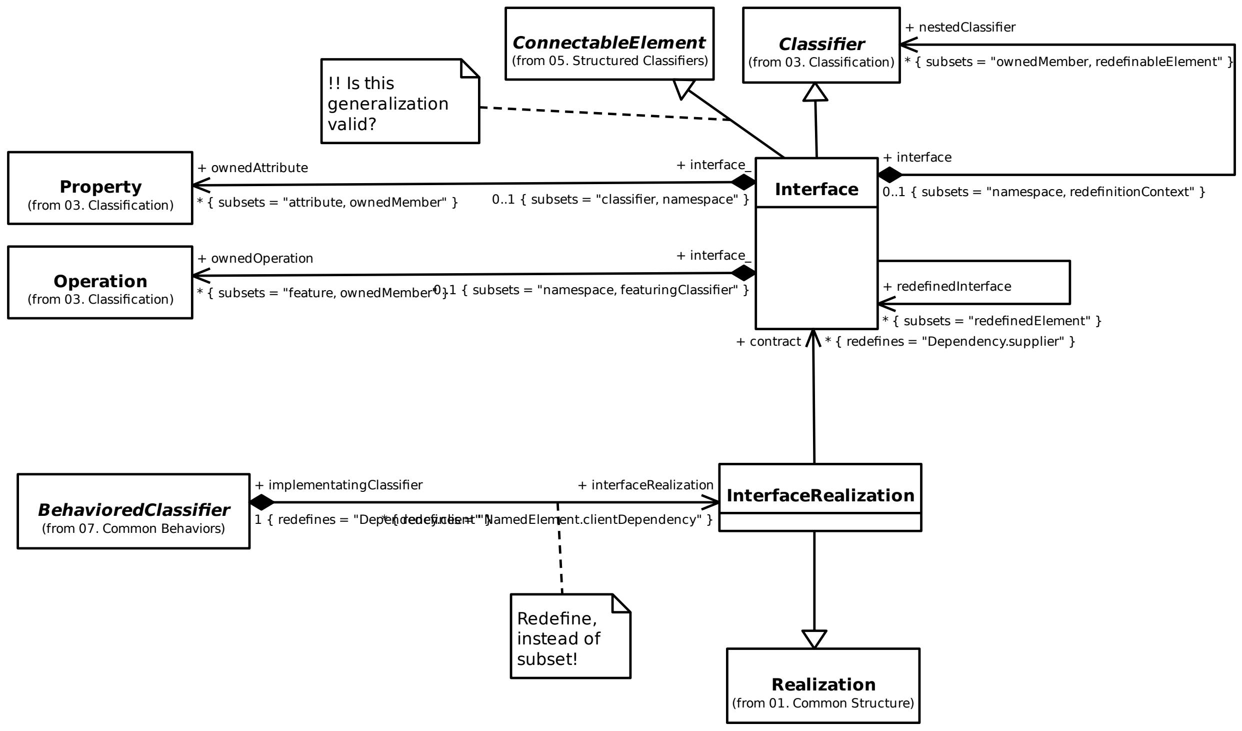

DeployedArtifact
(from 13. Deployments)
PackageableElement
(from 01. Common Structure)
DeploymentTarget
(from 13. Deployments)
Element
(from 01. Common Structure)
Made this relationship an aggregate so Slots are removed as soon as the definingFeature is removed.
InstanceSpecification
+ owningInstance
1 { subsets = "owner" }
+ slot
* { subsets = "ownedElement" }
Slot
+ slot
*
+ definingFeature
1
StructuralFeature
+ value
0..1 { ordered = "true", subsets = "ownedElement" }
+ specification
0..1 { subsets = "ownedElement" }
«simpleAttribute»
ValueSpecification
(from 02. Values)
+ instanceSpecification
*
+ classifier
*
Classifier
Made this relationship an aggregare, so instances are removed alongside their classifier.

13.4 04. Simple Classifiers

13.4.1 1. Data Types

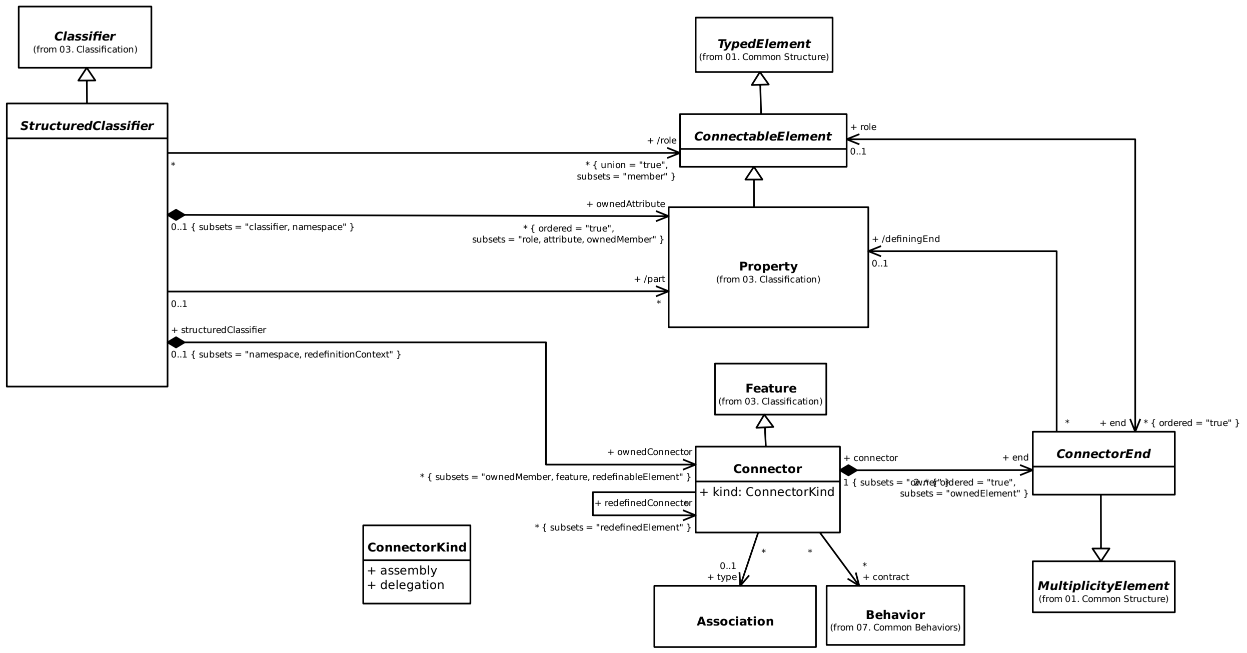

Classifier
(from 03. Classification)
DataType
+ datatype
0..1 { subsets = "namespace, classifier" }
+ ownedAttribute
* { subsets = "attribute, ownedMember" }
Property
(from 03. Classification)
+ datatype
0..1 { subsets = "namespace, redefinitionContext, featuringClassifier" }
+ ownedOperation
* { subsets = "feature, ownedMember, redefinableElement" }
Operation
(from 03. Classification)
InstanceSpecification
(from 03. Classification)
PrimitiveType
Enumeration
+ enumeration
0..1 { subsets = "namespace" }
+ ownedLiteral
* { ordered = "true", subsets = "ownedMember" }
EnumerationLiteral

13.4.2 3. Interfaces

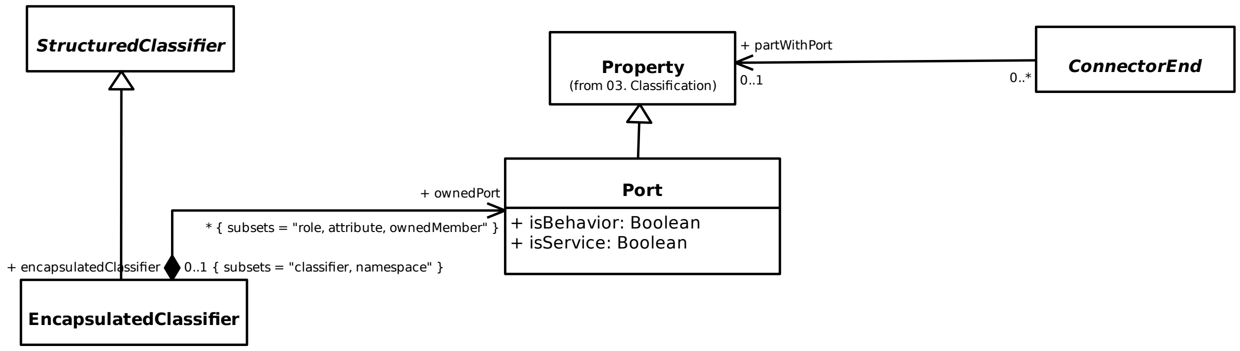

ConnectableElement
(from 05. Structured Classifiers)
Classifier
(from 03. Classification)
+ nestedClassifier
* { subsets = "ownedMember, redefinableElement" }
!! Is this generalization valid?
Property
(from 03. Classification)
+ ownedAttribute
* { subsets = "attribute, ownedMember" }
+ interface_
0..1 { subsets = "classifier, namespace" }
Interface
+ interface
0..1 { subsets = "namespace, redefinitionContext" }
Operation
(from 03. Classification)
+ ownedOperation
* { subsets = "feature, ownedMember" }
+ interface_
0..1 { subsets = "namespace, featuringClassifier" }
+ redefinedInterface
* { subsets = "redefinedElement" }
+ contract
* { redefines = "Dependency.supplier" }
BehavioredClassifier
(from 07. Common Behaviors)
+ implementatingClassifier
1 { redefines = "Dependency.client" }
+ interfaceRealization
* { redefines = "NamedElement.clientDependency" }
InterfaceRealization
Redefine, instead of subset!
Realization
(from 01. Common Structure)

13.5 05. Structured Classifiers

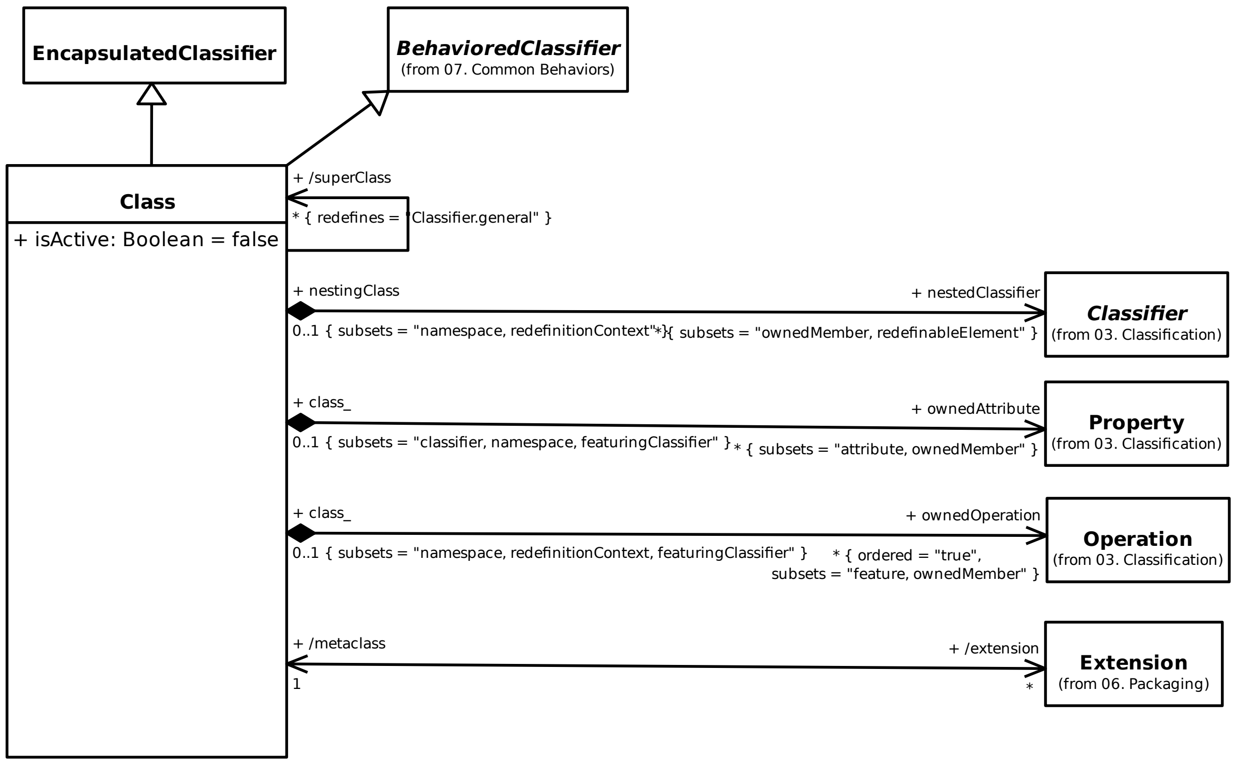
13.5.1 1. Structured Classifiers



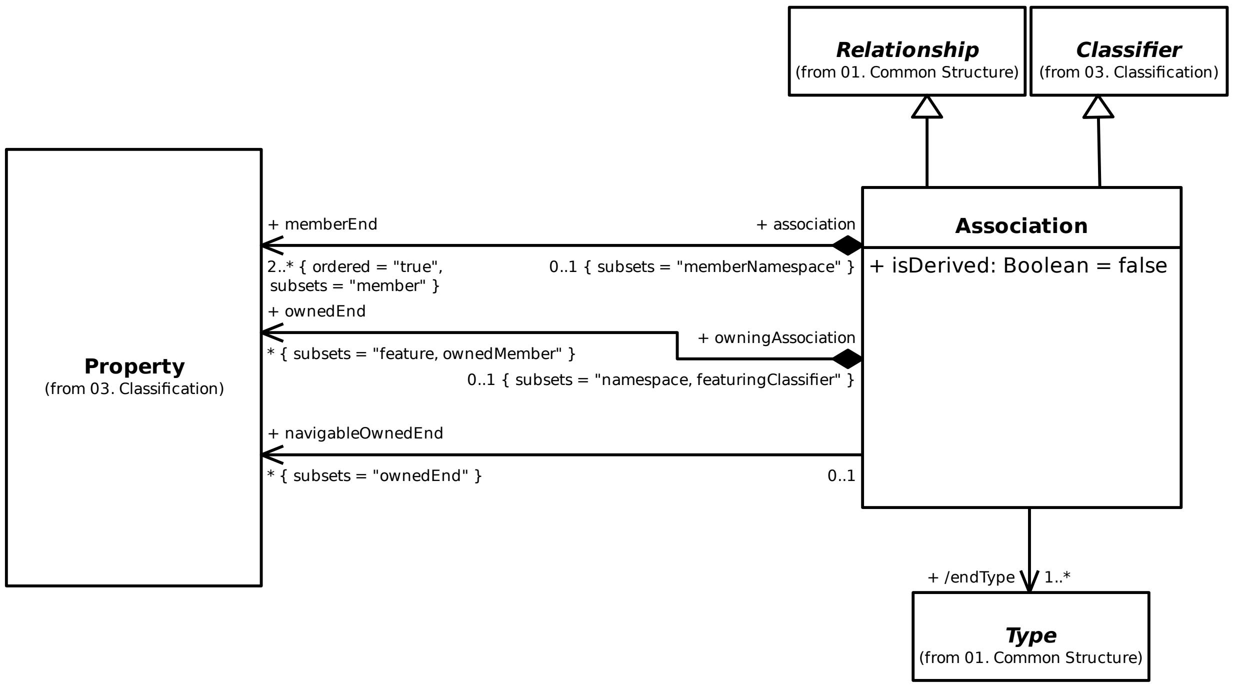
13.5.2 2. Encapsulated Classifiers



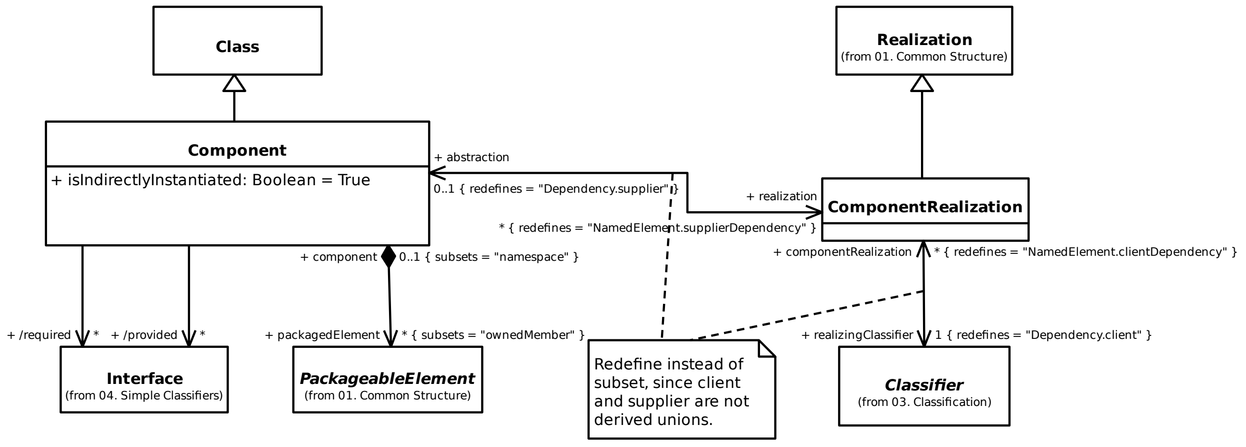
13.5.3 3. Classes


EncapsulatedClassifier
BehavioredClassifier
(from 07. Common Behaviors)
Class
+ isActive: Boolean = false
+ /superClass
* { redefines = "Classifier.general" }
+ nestingClass
0..1 { subsets = "namespace, redefinitionContext" }
+ nestedClassifier
* { subsets = "ownedMember, redefinableElement" }
Classifier
(from 03. Classification)
+ class_
0..1 { subsets = "classifier, namespace, featuringClassifier" }
+ ownedAttribute
* { subsets = "attribute, ownedMember" }
Property
(from 03. Classification)
+ class_
0..1 { subsets = "namespace, redefinitionContext, featuringClassifier" }
+ ownedOperation
* { ordered = "true", subsets = "feature, ownedMember" }
Operation
(from 03. Classification)
+ /metaclass
1
+ /extension
*
Extension
(from 06. Packaging)

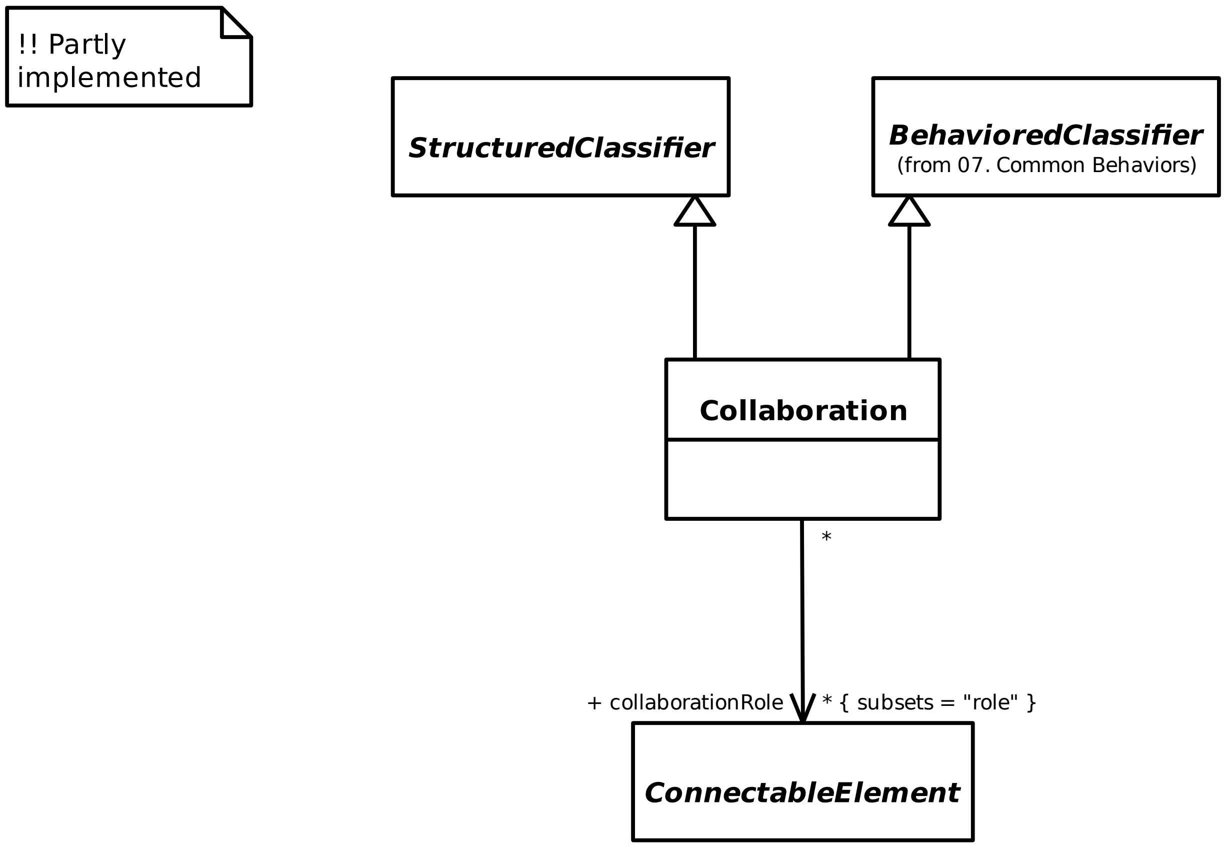
13.5.4 4. Associations


Relationship
(from 01. Common Structure)
Classifier
(from 03. Classification)
Association
+ isDerived: Boolean = false
Property
(from 03. Classification)
+ memberEnd
2..* { ordered = "true", subsets = "member" }
+ association
0..1 { subsets = "memberNamespace" }
+ ownedEnd
* { subsets = "feature, ownedMember" }
+ owningAssociation
0..1 { subsets = "namespace, featuringClassifier" }
+ navigableOwnedEnd
* { subsets = "ownedEnd" }
0..1
+ /endType
1..*
Type
(from 01. Common Structure)

13.5.5 5. Components

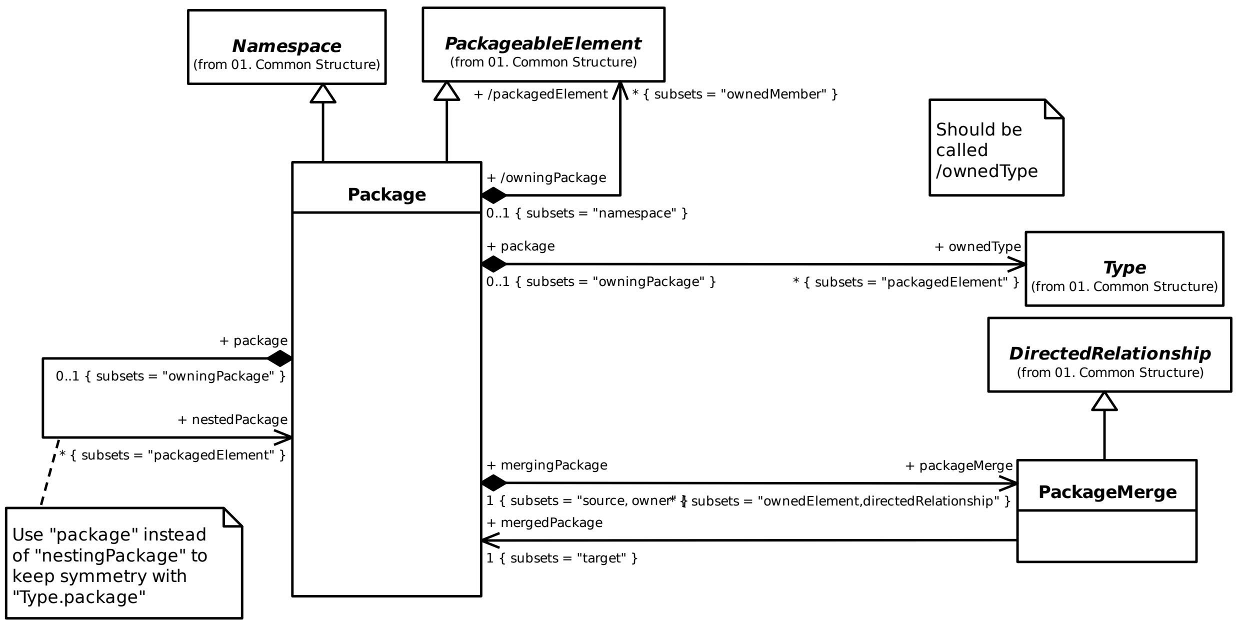

Class
Realization
(from 01. Common Structure)
Component
+ isIndirectlyInstantiated: Boolean = True
+ abstraction
0..1 { redefines = "Dependency.supplier" }
+ realization
ComponentRealization
* { redefines = "NamedElement.supplierDependency" }
+ componentRealization
* { redefines = "NamedElement.clientDependency" }
+ component
0..1 { subsets = "namespace" }
+ /required
*
+ /provided
*
+ packagedElement
* { subsets = "ownedMember" }
+ realizingClassifier
1 { redefines = "Dependency.client" }
Interface
(from 04. Simple Classifiers)
PackageableElement
(from 01. Common Structure)
Redefine instead of subset, since client and supplier are not derived unions.
Classifier
(from 03. Classification)

13.5.6 6. Collaborations

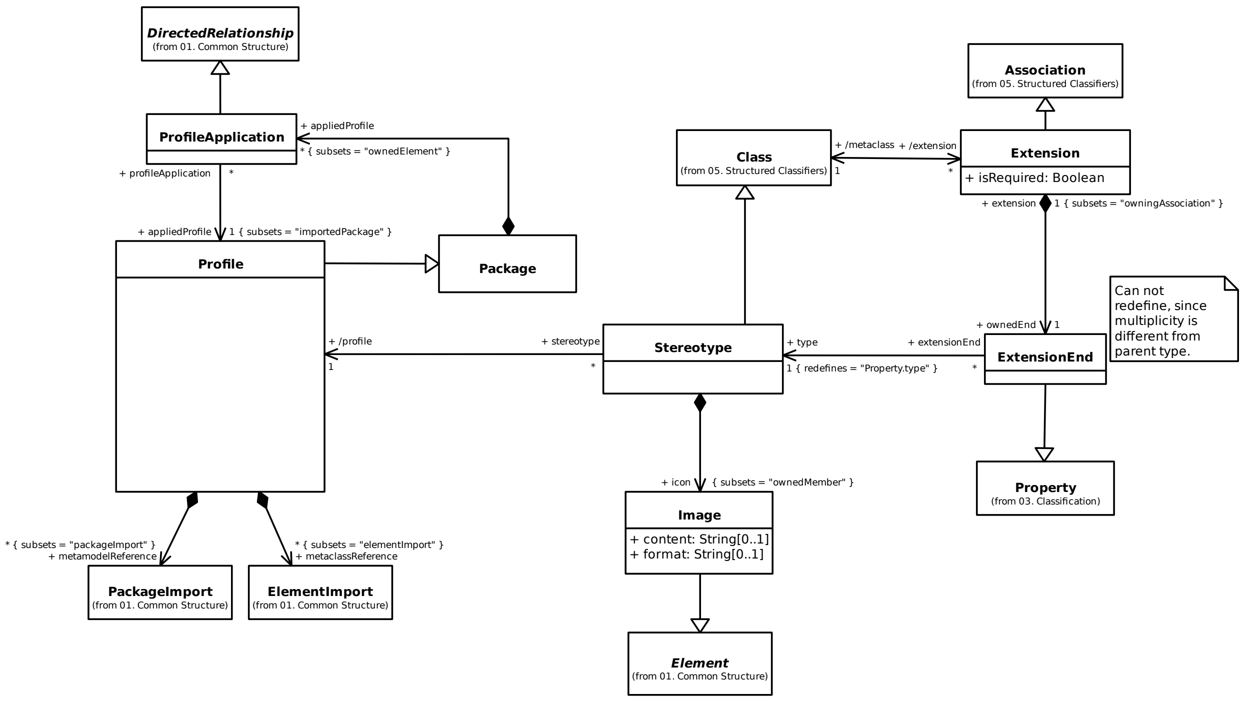


13.6 06. Packaging

13.6.1 1. Packages

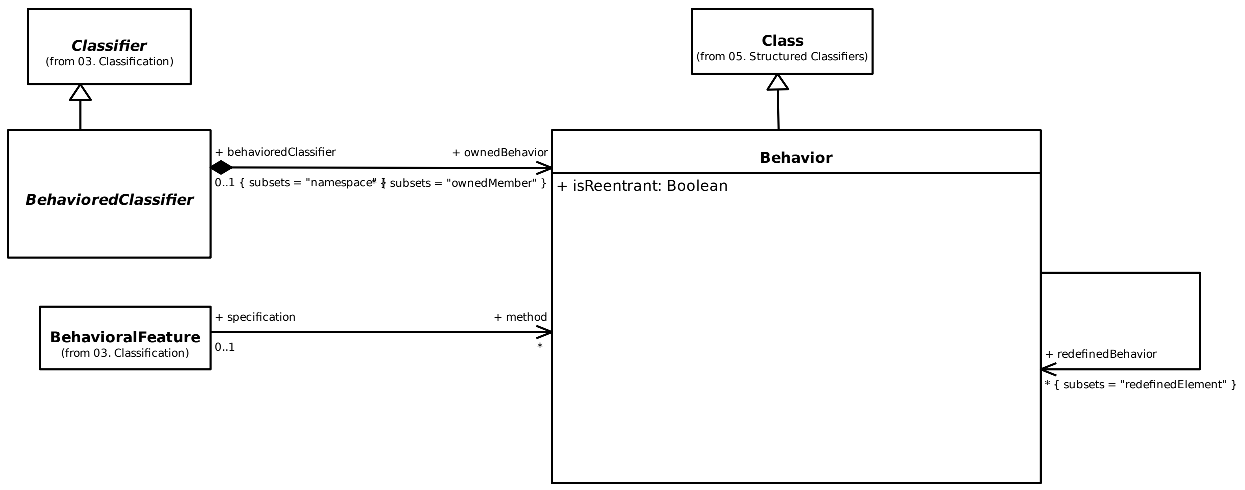

Namespace
(from 01. Common Structure)
PackageableElement
(from 01. Common Structure)
+ /packagedElement
* { subsets = "ownedMember" }
Should be called /ownedType
Package
+ /owningPackage
0..1 { subsets = "namespace" }
+ package
0..1 { subsets = "owningPackage" }
+ ownedType
* { subsets = "packagedElement" }
Type
(from 01. Common Structure)
DirectedRelationship
(from 01. Common Structure)
+ package
0..1 { subsets = "owningPackage" }
+ nestedPackage
* { subsets = "packagedElement" }
Use "package" instead of "nestingPackage" to keep symmetry with "Type.package"
+ mergingPackage
1 { subsets = "source, owner" }
+ packageMerge
* { subsets = "ownedElement,directedRelationship" }
+ mergedPackage
1 { subsets = "target" }
PackageMerge

13.6.2 2. Profiles

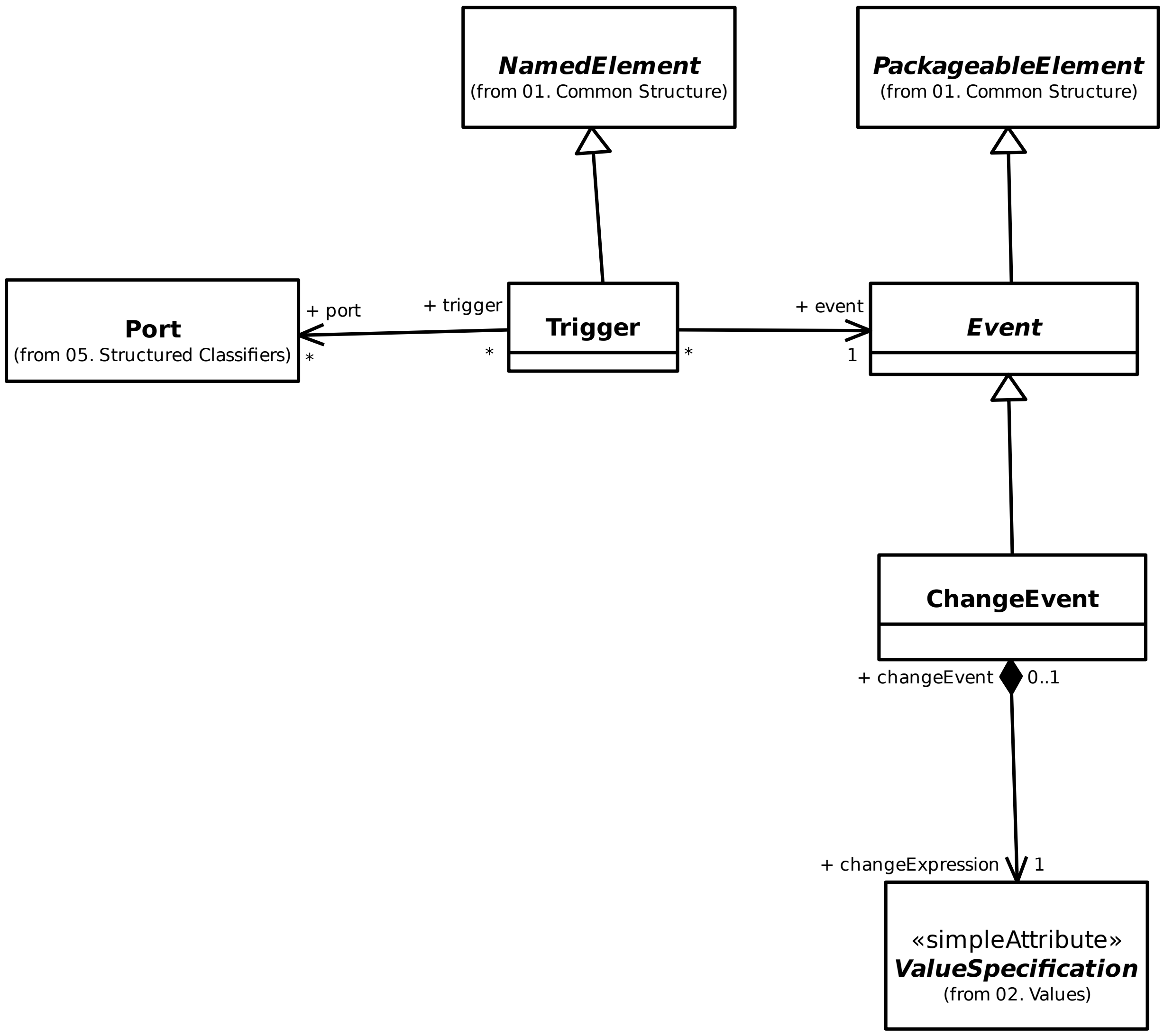

DirectedRelationship
(from 01. Common Structure)
ProfileApplication
+ appliedProfile
* { subsets = "ownedElement" }
+ profileApplication
*
+ appliedProfile
1 { subsets = "importedPackage" }
Profile
Package
+ /profile
1
+ stereotype
*
* { subsets = "packageImport" }
+ metamodelReference
* { subsets = "elementImport" }
+ metaclassReference
PackageImport
(from 01. Common Structure)
ElementImport
(from 01. Common Structure)
Association
(from 05. Structured Classifiers)
Class
(from 05. Structured Classifiers)
+ /metaclass
1
+ /extension
*
Extension
+ isRequired: Boolean
+ extension
1 { subsets = "owningAssociation" }
Can not redefine, since multiplicity is different from parent type.
+ ownedEnd
1
Stereotype
+ type
1 { redefines = "Property.type" }
+ extensionEnd
*
ExtensionEnd
+ icon
{ subsets = "ownedMember" }
Image
+ content: String[0..1]
+ format: String[0..1]
Property
(from 03. Classification)
Element
(from 01. Common Structure)

13.7 07. Common Behaviors

13.7.1 1. Behaviors

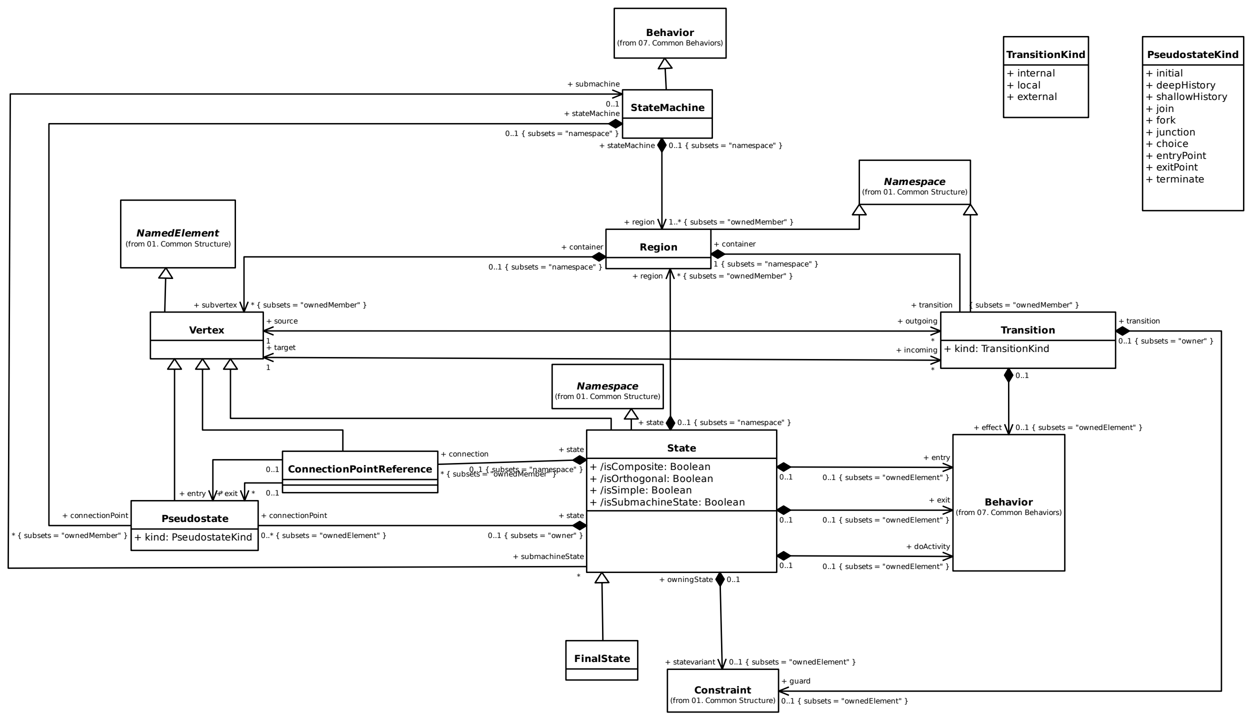

Classifier
(from 03. Classification)
Class
(from 05. Structured Classifiers)
BehavioredClassifier
+ behavioredClassifier
0..1 { subsets = "namespace" }
+ ownedBehavior
* { subsets = "ownedMember" }
Behavior
+ isReentrant: Boolean
BehavioralFeature
(from 03. Classification)
+ specification
0..1
+ method
*
+ redefinedBehavior
* { subsets = "redefinedElement" }

13.7.2 2. Events



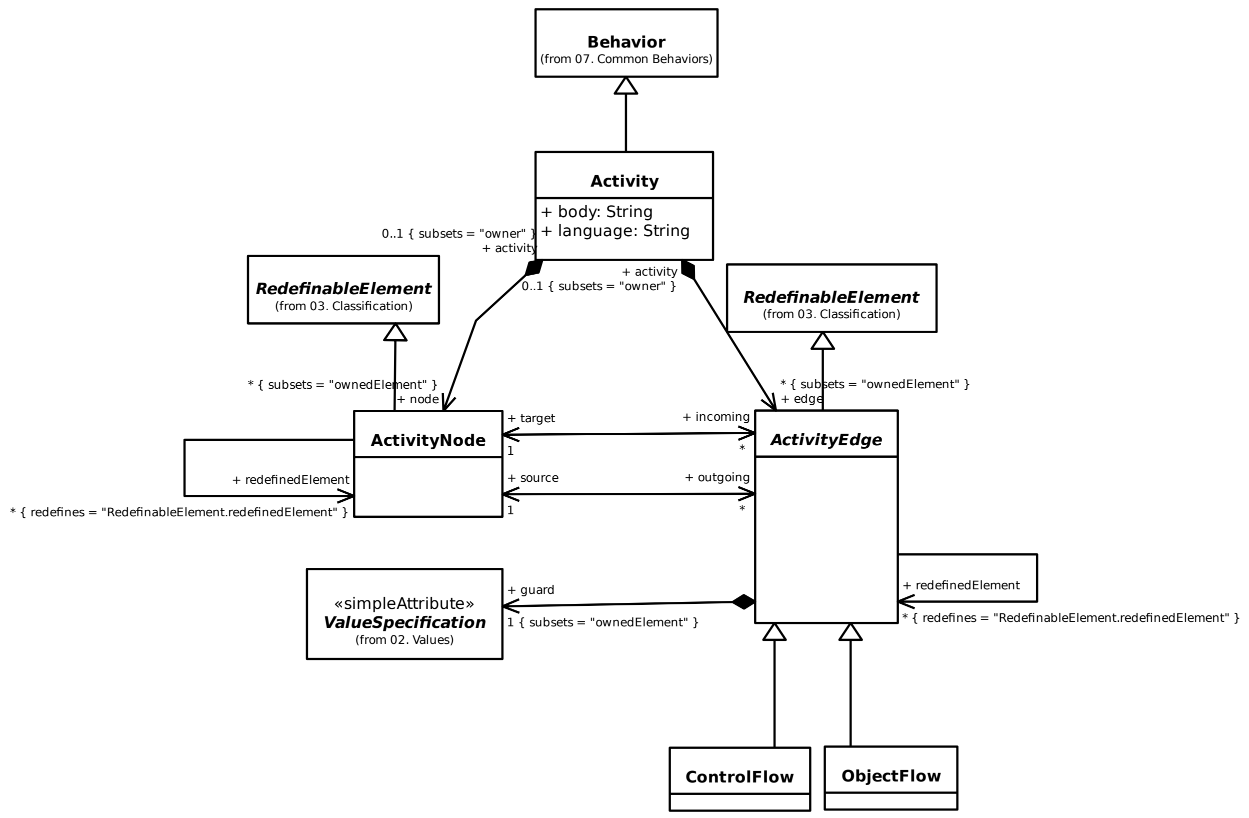
13.8 08. State Machines

13.8.1 1. Behavior State Machines

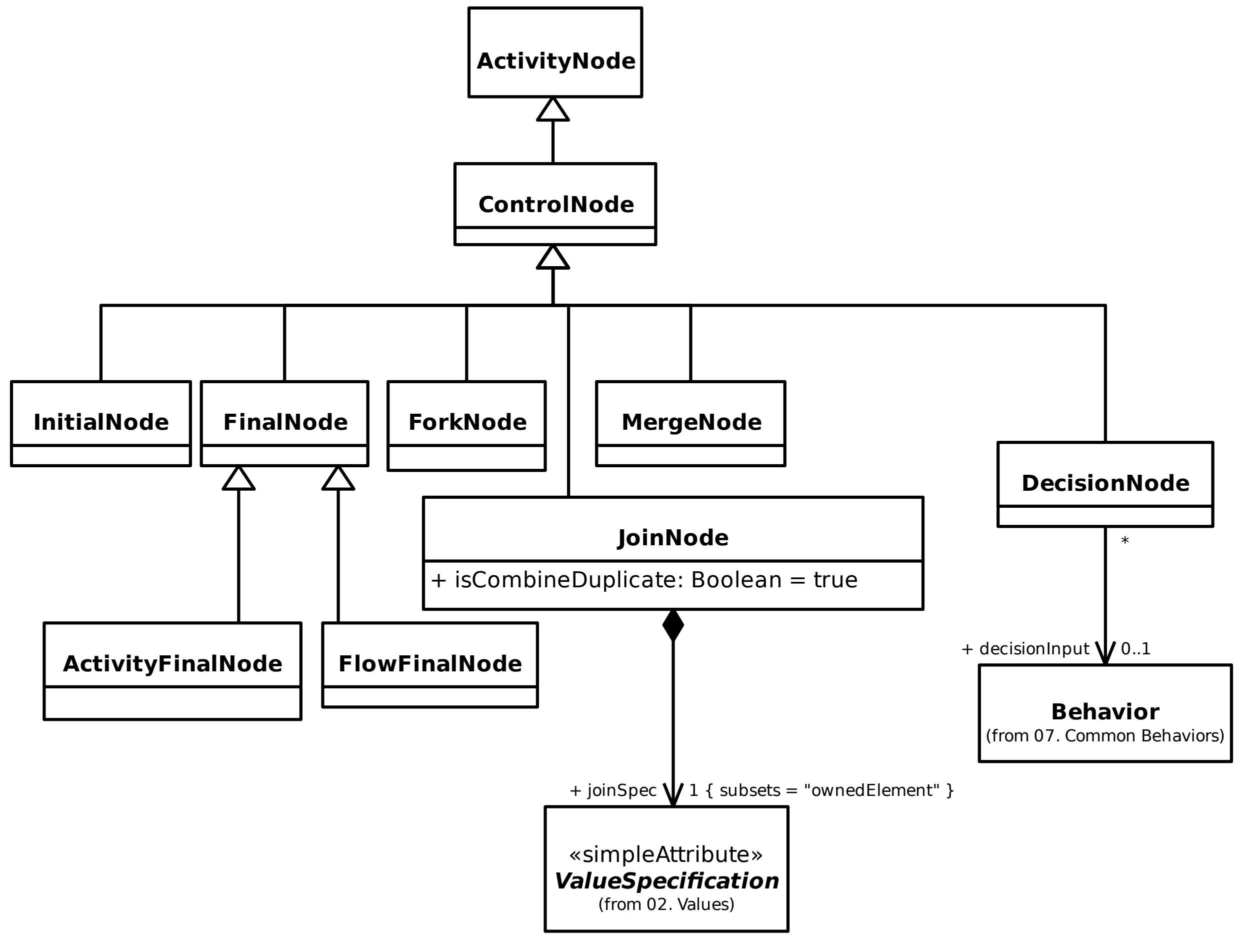

Behavior
(from 07. Common Behaviors)
TransitionKind
+ internal
+ local
+ external
PseudostateKind
+ initial
+ deepHistory
+ shallowHistory
+ join
+ fork
+ junction
+ choice
+ entryPoint
+ exitPoint
+ terminate
+ submachine
0..1
+ stateMachine
StateMachine
0..1 { subsets = "namespace" }
+ stateMachine
0..1 { subsets = "namespace" }
Namespace
(from 01. Common Structure)
NamedElement
(from 01. Common Structure)
+ region
1..* { subsets = "ownedMember" }
+ container
Region
+ container
0..1 { subsets = "namespace" }
1 { subsets = "namespace" }
+ region
* { subsets = "ownedMember" }
+ subvertex
* { subsets = "ownedMember" }
+ transition
{ subsets = "ownedMember" }
+ source
+ outgoing
Vertex
1
+ target
1
Transition
+ kind: TransitionKind
+ transition
0..1 { subsets = "owner" }
*
+ incoming
*
0..1
Namespace
(from 01. Common Structure)
+ state
0..1 { subsets = "namespace" }
+ effect
0..1 { subsets = "ownedElement" }
+ connection
+ state
State
+ /isComposite: Boolean
+ /isOrthogonal: Boolean
+ /isSimple: Boolean
+ /isSubmachineState: Boolean
0..1
ConnectionPointReference
0..1 { subsets = "namespace" }
* { subsets = "ownedMember" }
+ entry
0..1
0..1 { subsets = "ownedElement" }
+ entry
+ exit
*
*
0..1
+ exit
Behavior
(from 07. Common Behaviors)
+ connectionPoint
Pseudostate
+ kind: PseudostateKind
* { subsets = "ownedMember" }
+ connectionPoint
0..* { subsets = "ownedElement" }
+ state
0..1 { subsets = "owner" }
0..1
0..1 { subsets = "ownedElement" }
+ submachineState
*
+ doActivity
0..1
0..1 { subsets = "ownedElement" }
+ owningState
0..1
FinalState
+ statevariant
0..1 { subsets = "ownedElement" }
Constraint
(from 01. Common Structure)
+ guard
0..1 { subsets = "ownedElement" }

13.9 09. Activities

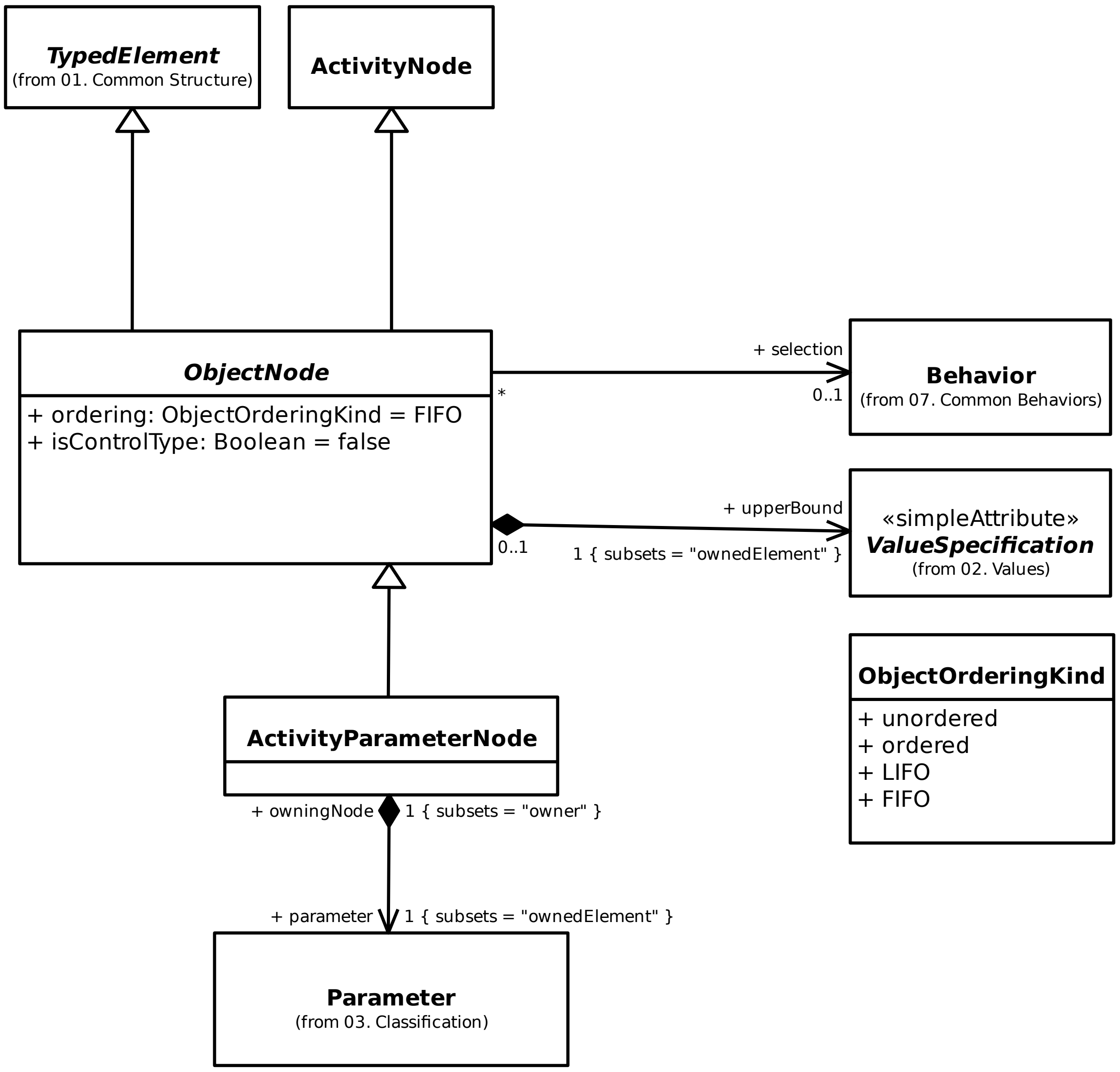
13.9.1 1. Activities


Behavior
(from 07. Common Behaviors)
Activity
+ body: String
+ language: String
0..1 { subsets = "owner" }
+ activity
+ activity
0..1 { subsets = "owner" }
RedefinableElement
(from 03. Classification)
RedefinableElement
(from 03. Classification)
* { subsets = "ownedElement" }
+ node
* { subsets = "ownedElement" }
+ edge
ActivityNode
ActivityEdge
+ target
1
+ incoming
*
+ source
1
+ outgoing
*
+ redefinedElement
* { redefines = "RedefinableElement.redefinedElement" }
+ redefinedElement
* { redefines = "RedefinableElement.redefinedElement" }
«simpleAttribute»
ValueSpecification
(from 02. Values)
+ guard
1 { subsets = "ownedElement" }
ControlFlow
ObjectFlow

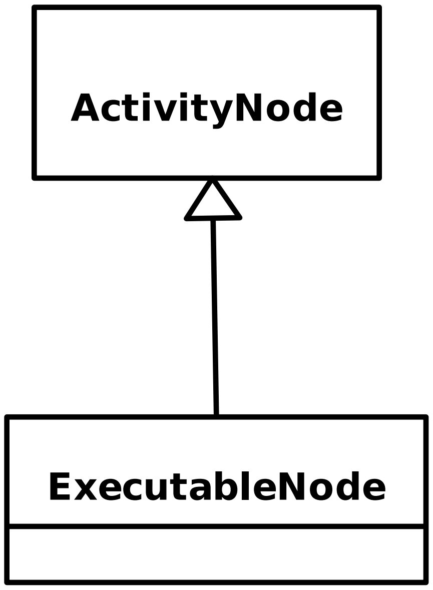
13.9.2 2. Control Nodes



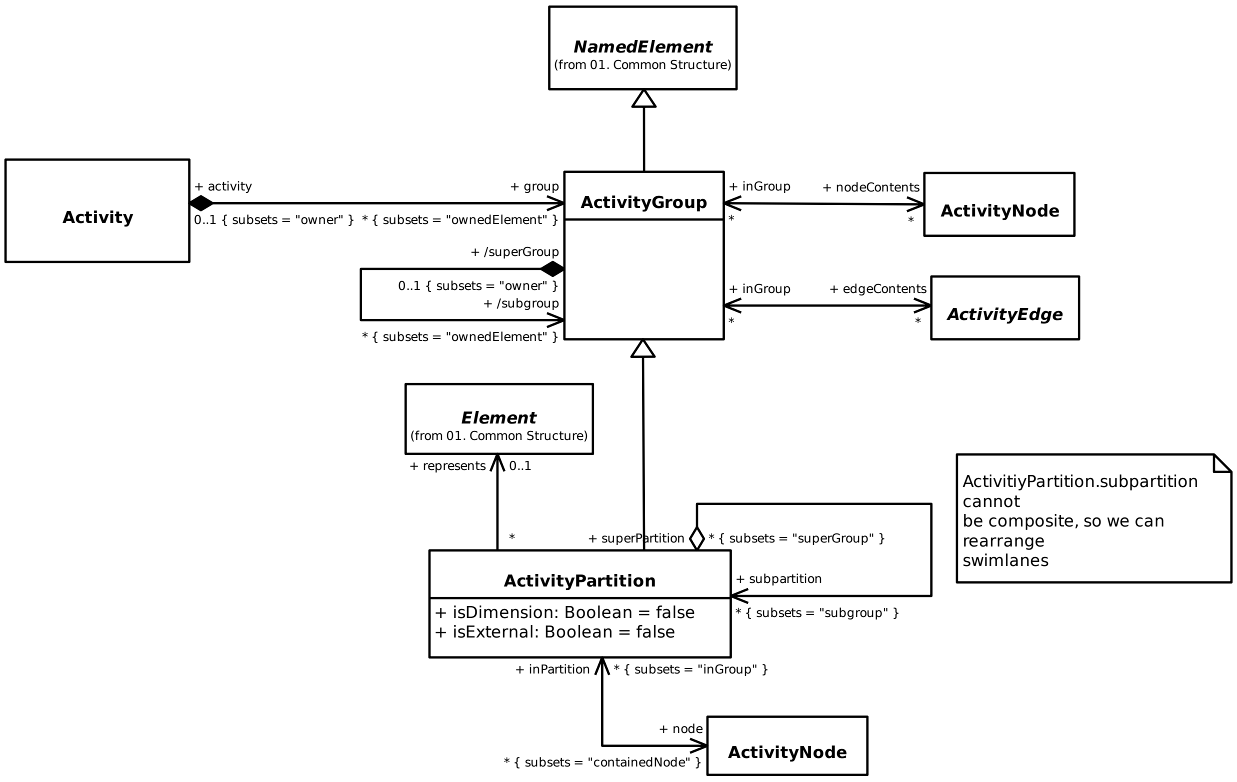
13.9.3 3. Object Nodes



13.9.4 4. Executable Nodes

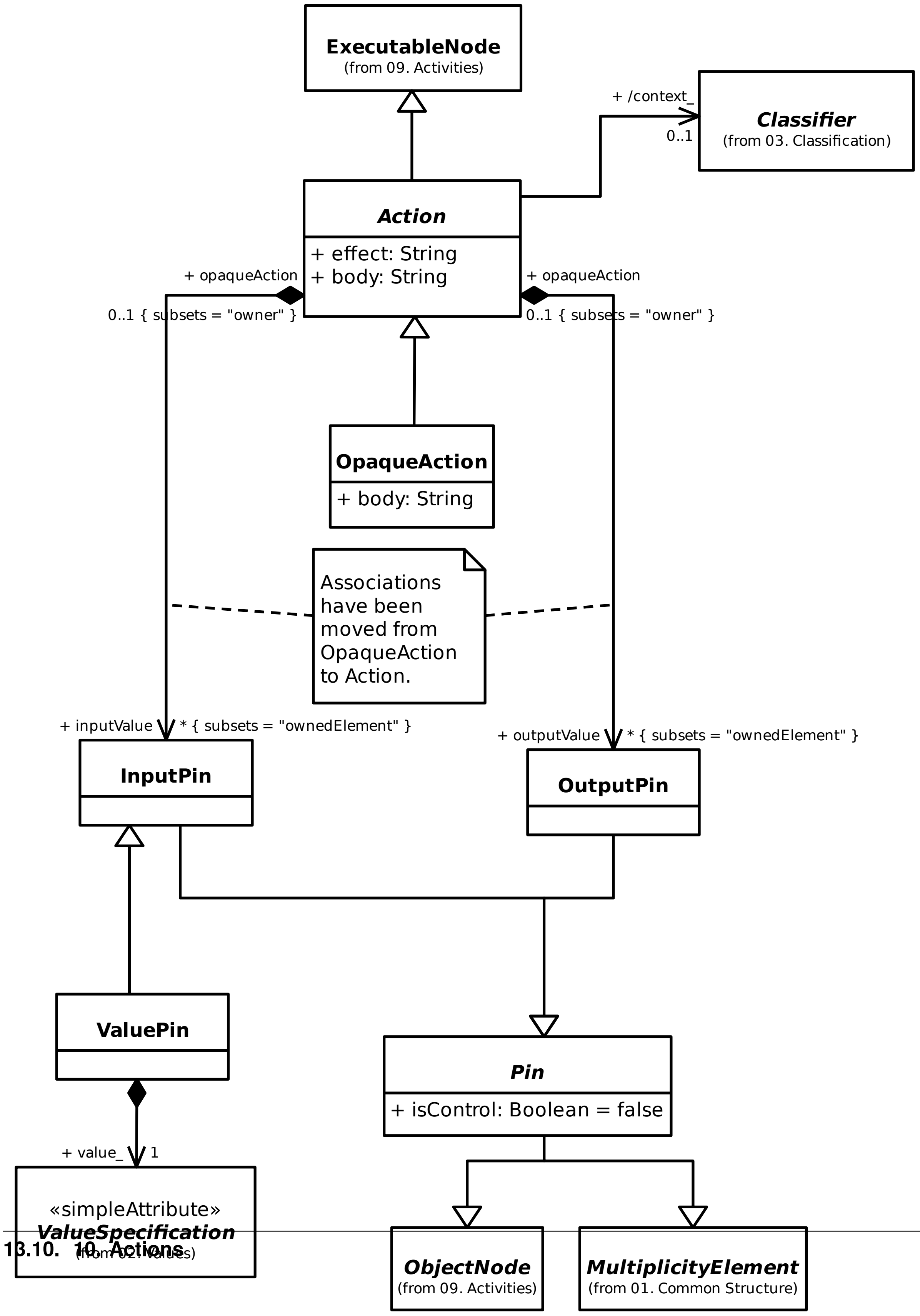


13.9.5 5. Activity Groups

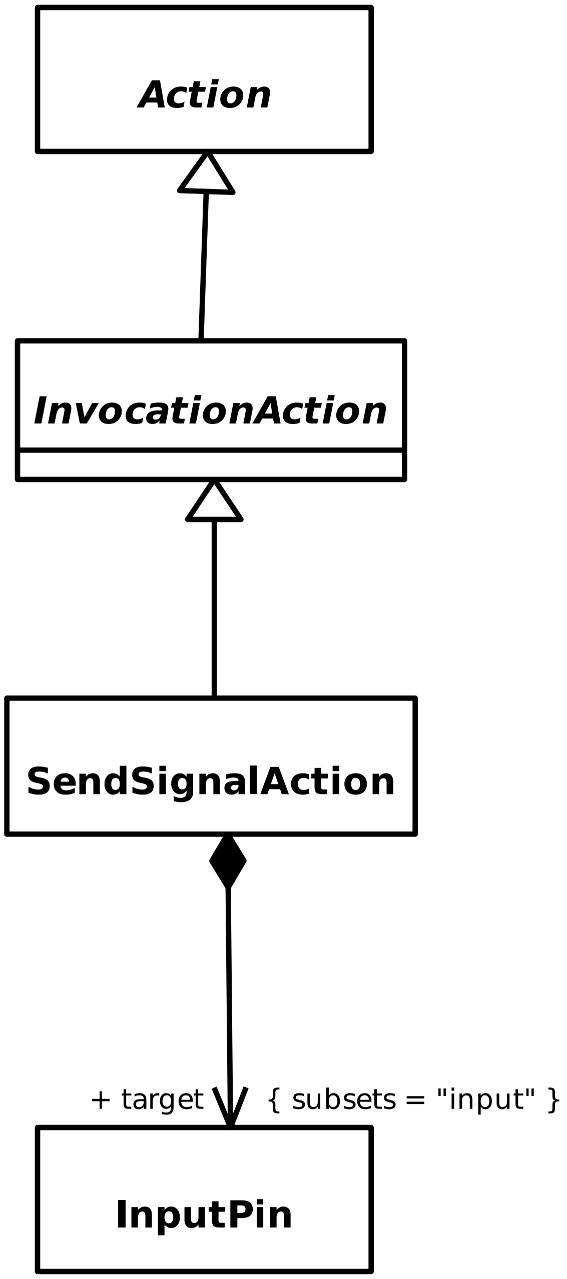

NamedElement
(from 01. Common Structure)
Activity
+ activity
0..1 { subsets = "owner" }
+ group
* { subsets = "ownedElement" }
ActivityGroup
+ /superGroup
0..1 { subsets = "owner" }
+ /subgroup
* { subsets = "ownedElement" }
+ inGroup
*
+ nodeContents
*
ActivityNode
+ inGroup
*
+ edgeContents
*
ActivityEdge
Element
(from 01. Common Structure)
+ represents
0..1
*
+ superPartition
* { subsets = "superGroup" }
+ subpartition
* { subsets = "subgroup" }
ActivityPartition
+ isDimension: Boolean = false
+ isExternal: Boolean = false
+ inPartition
* { subsets = "inGroup" }
+ node
* { subsets = "containedNode" }
ActivityNode
ActivitiyPartition.subpartition cannot
be composite, so we can rearrange
swimlanes

13.10 10. Actions

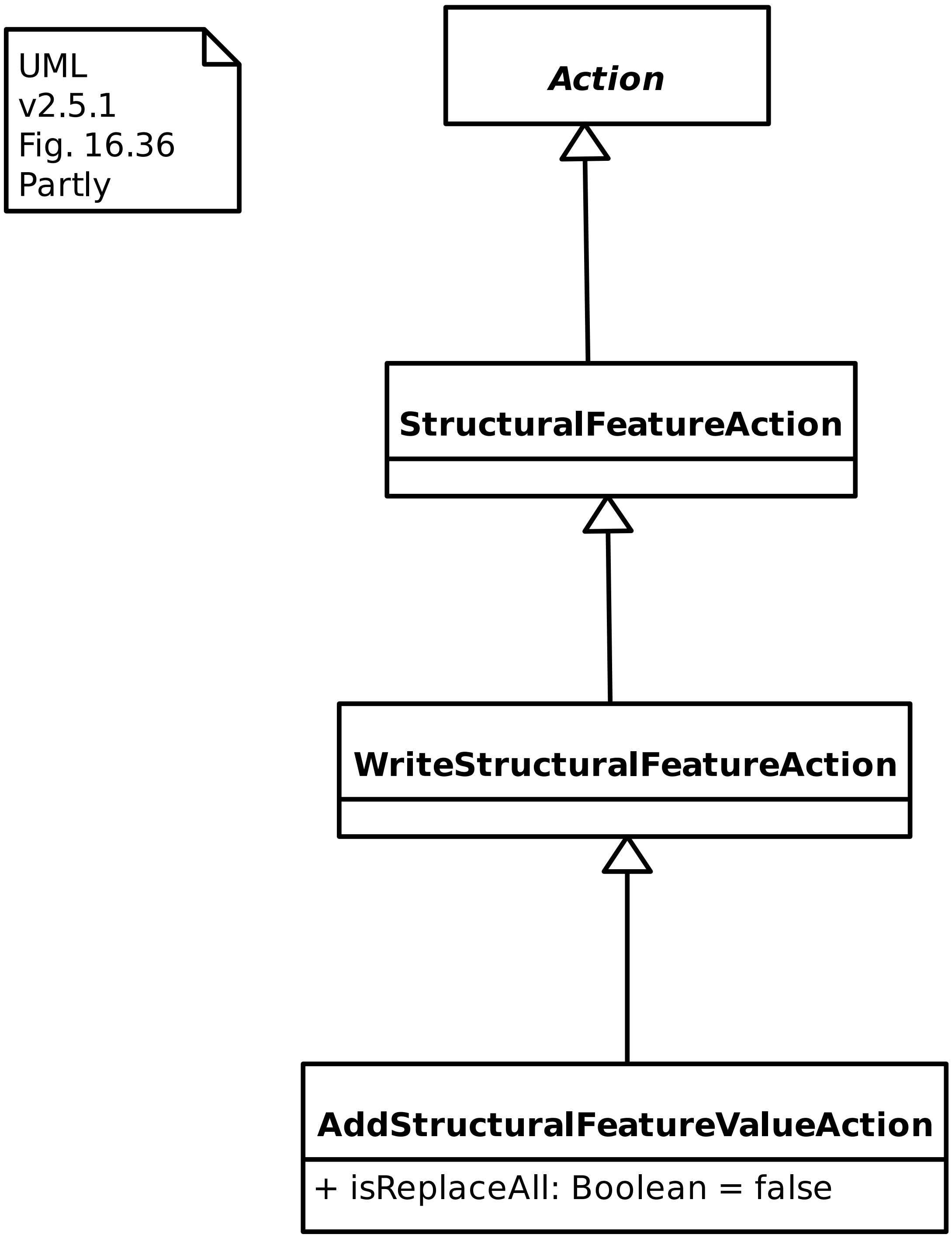
13.10.1 1. Actions



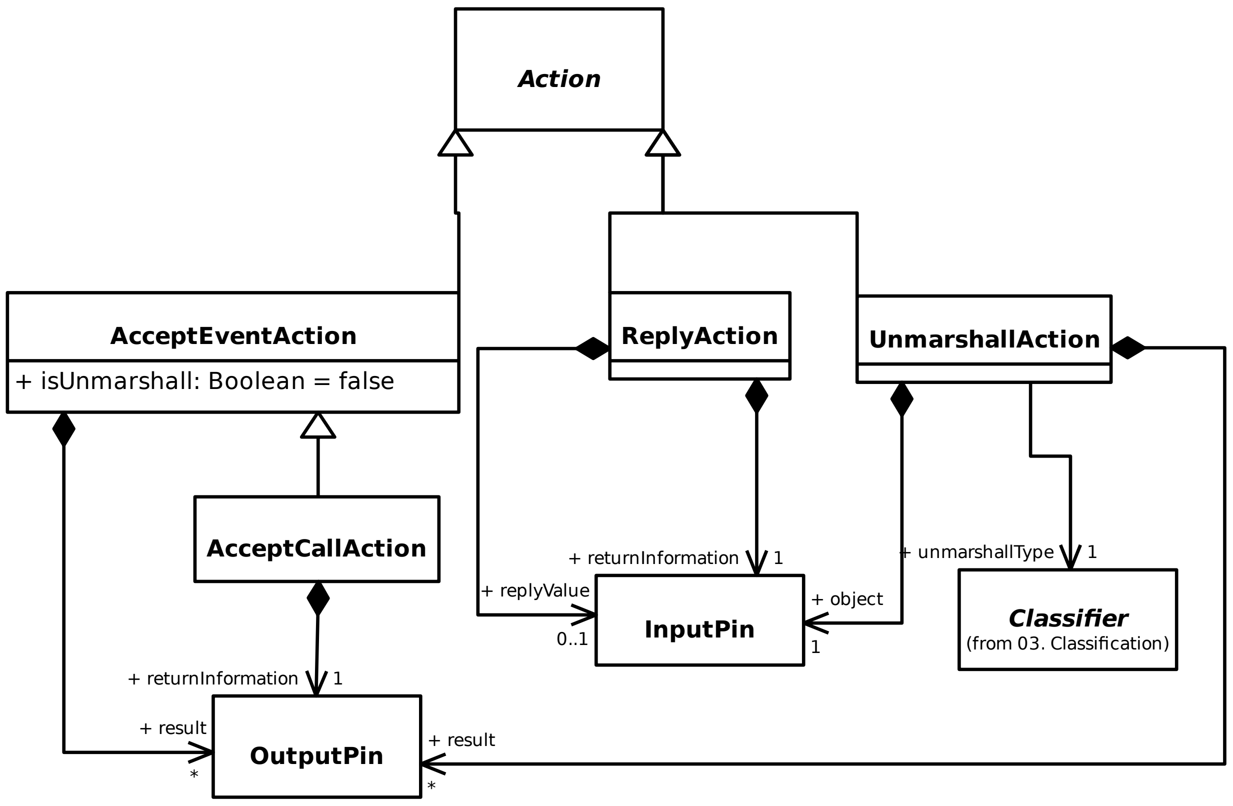
13.10.2 2. Invocation Actions



13.10.3 7. Structural Feature Actions

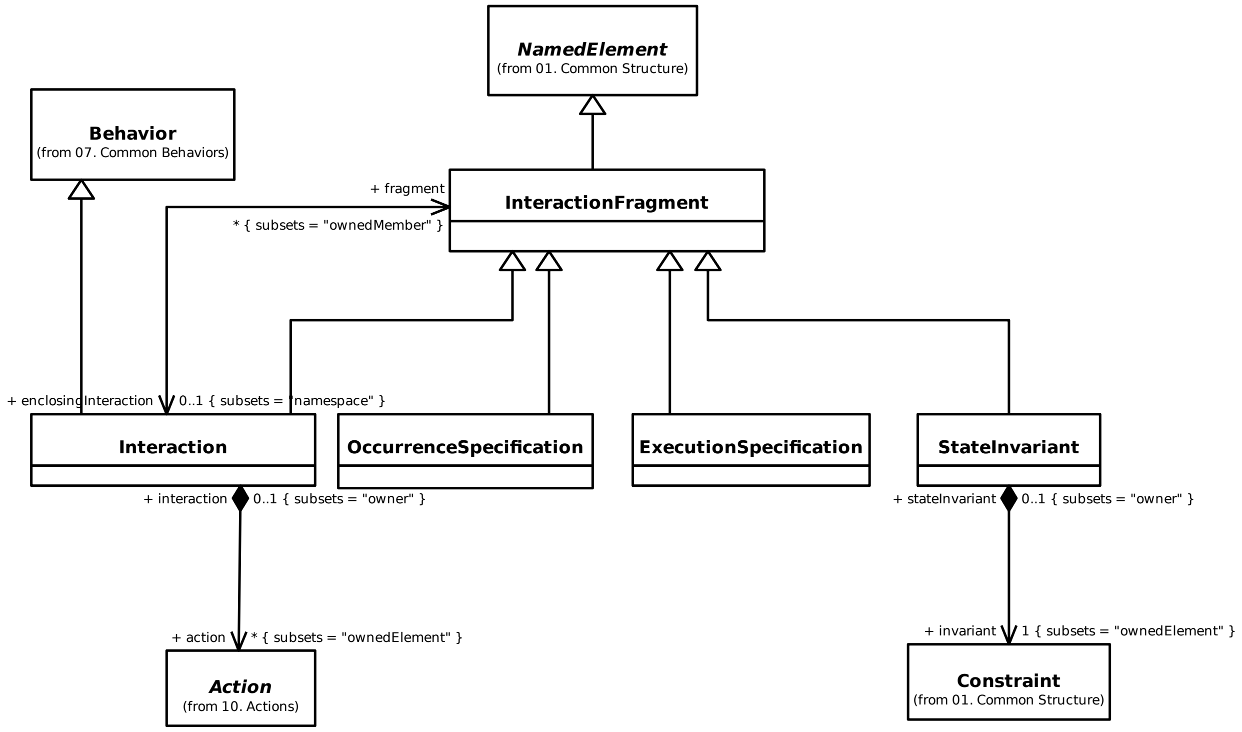

UML
v2.5.1
Fig. 16.36
Partly
Action
StructuralFeatureAction
WriteStructuralFeatureAction
AddStructuralFeatureValueAction
+ isReplaceAll: Boolean = false

13.10.4 9. Accept Event Actions

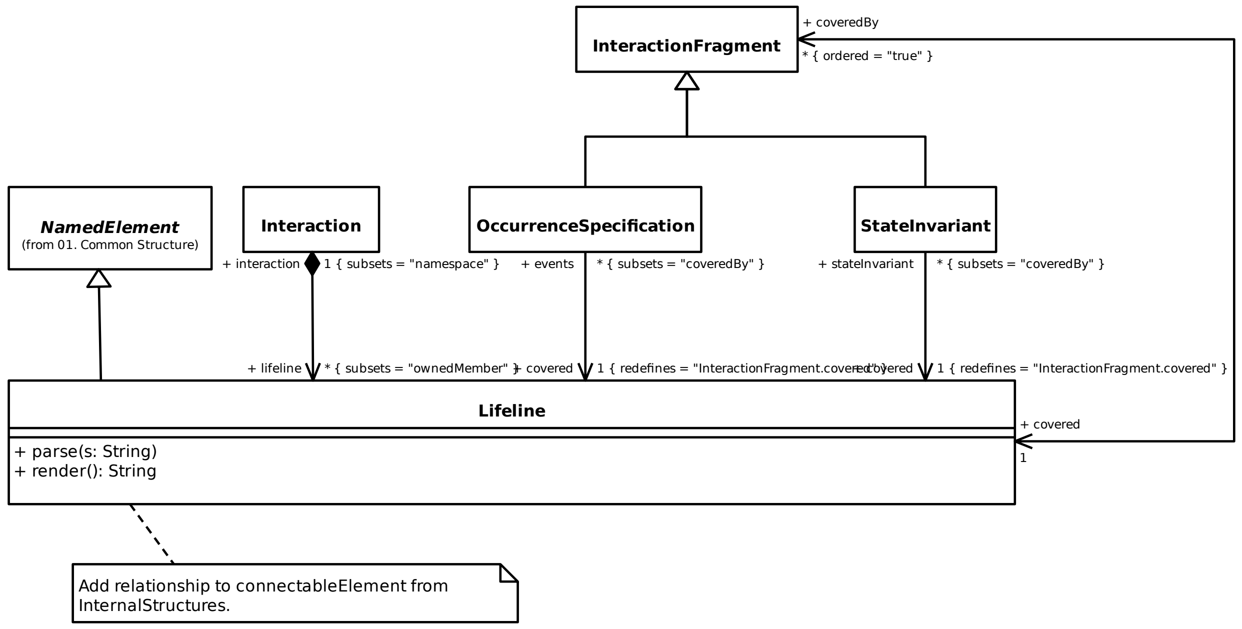


13.11 11. Interactions

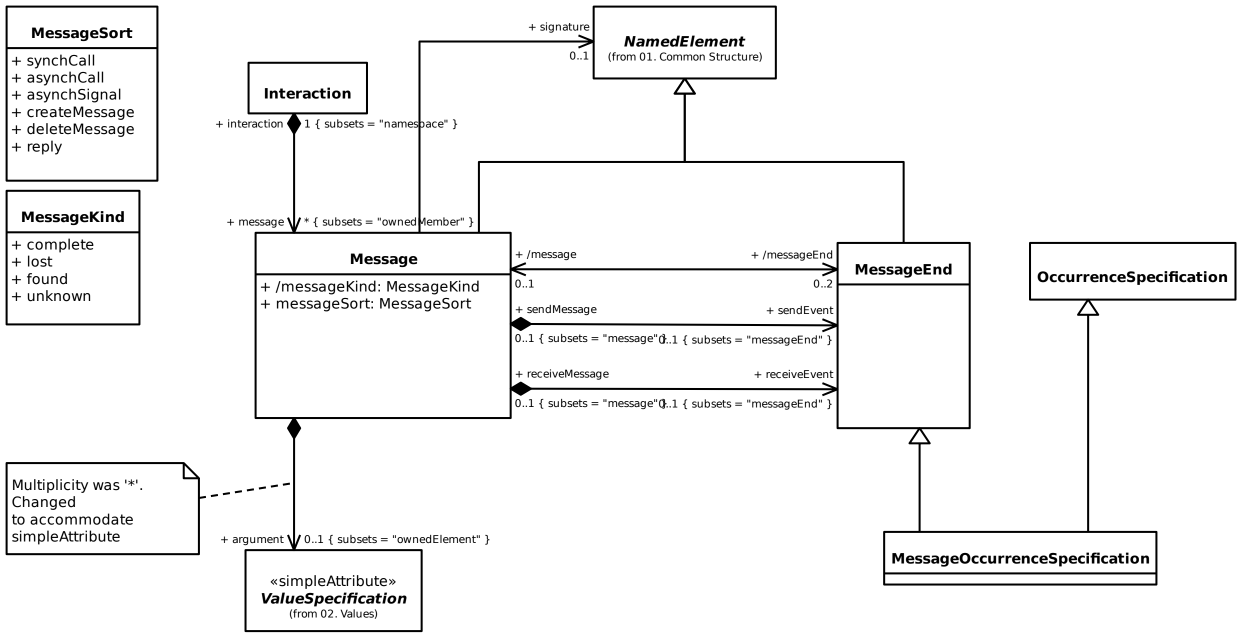
13.11.1 1. Interactions


NamedElement
(from 01. Common Structure)
Behavior
(from 07. Common Behaviors)
+ fragment
* { subsets = "ownedMember" }
InteractionFragment
+ enclosingInteraction
0..1 { subsets = "namespace" }
Interaction
OccurrenceSpecification
ExecutionSpecification
StateInvariant
+ interaction
0..1 { subsets = "owner" }
+ stateInvariant
0..1 { subsets = "owner" }
+ action
* { subsets = "ownedElement" }
Action
(from 10. Actions)
+ invariant
1 { subsets = "ownedElement" }
Constraint
(from 01. Common Structure)

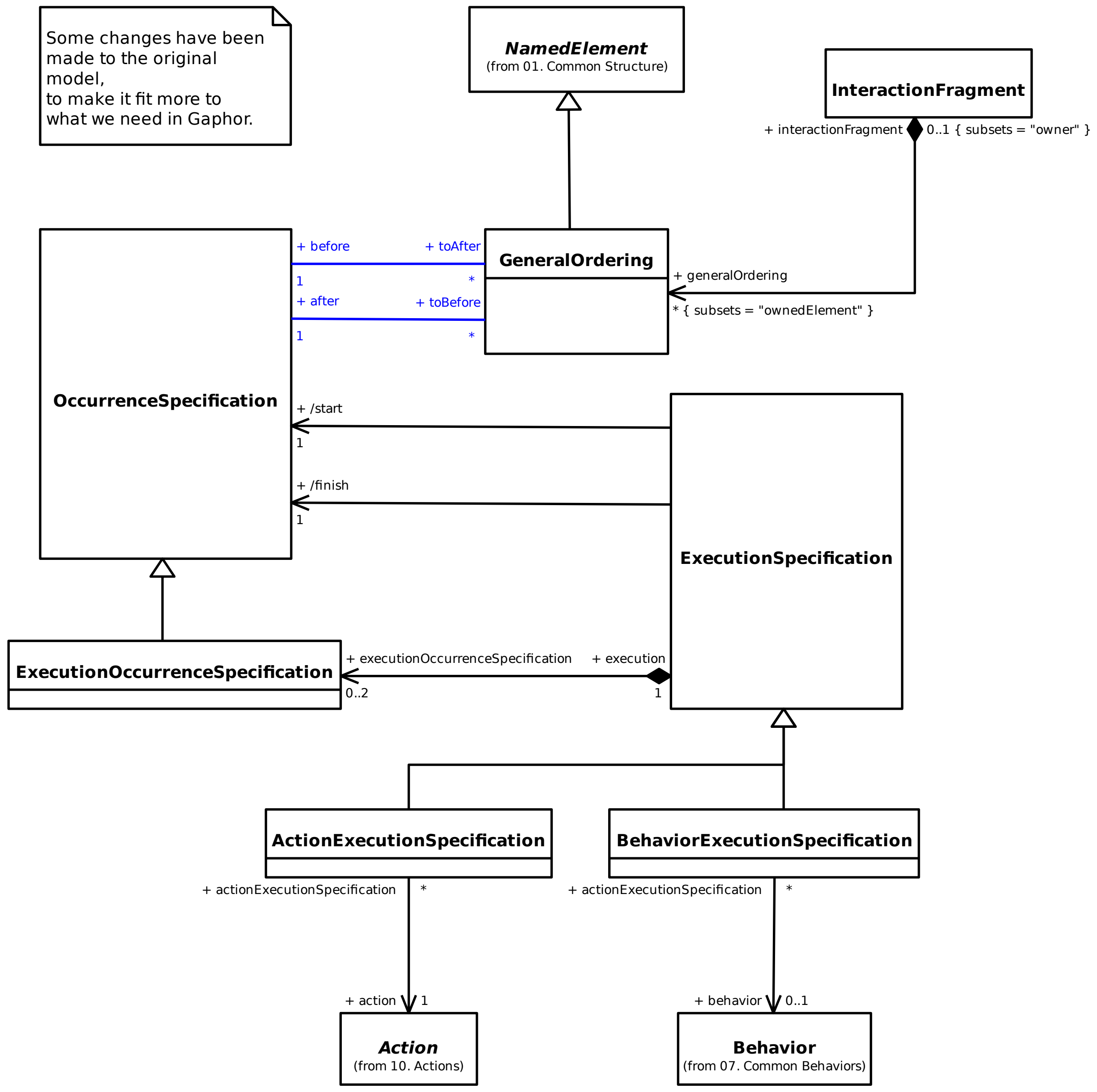
13.11.2 2. Lifelines


InteractionFragment
+ coveredBy
* { ordered = "true" }
NamedElement
(from 01. Common Structure)
Interaction
OccurrenceSpecification
StateInvariant
+ interaction
1 { subsets = "namespace" }
+ events
* { subsets = "coveredBy" }
+ stateInvariant
* { subsets = "coveredBy" }
+ lifeline
* { subsets = "ownedMember" }
+ covered
1 { redefines = "InteractionFragment.covered" }
+ covered
1 { redefines = "InteractionFragment.covered" }
Lifeline
+ parse(s: String)
+ render(): String
+ covered
1
Add relationship to connectableElement from InternalStructures.

13.11.3 3. Messages

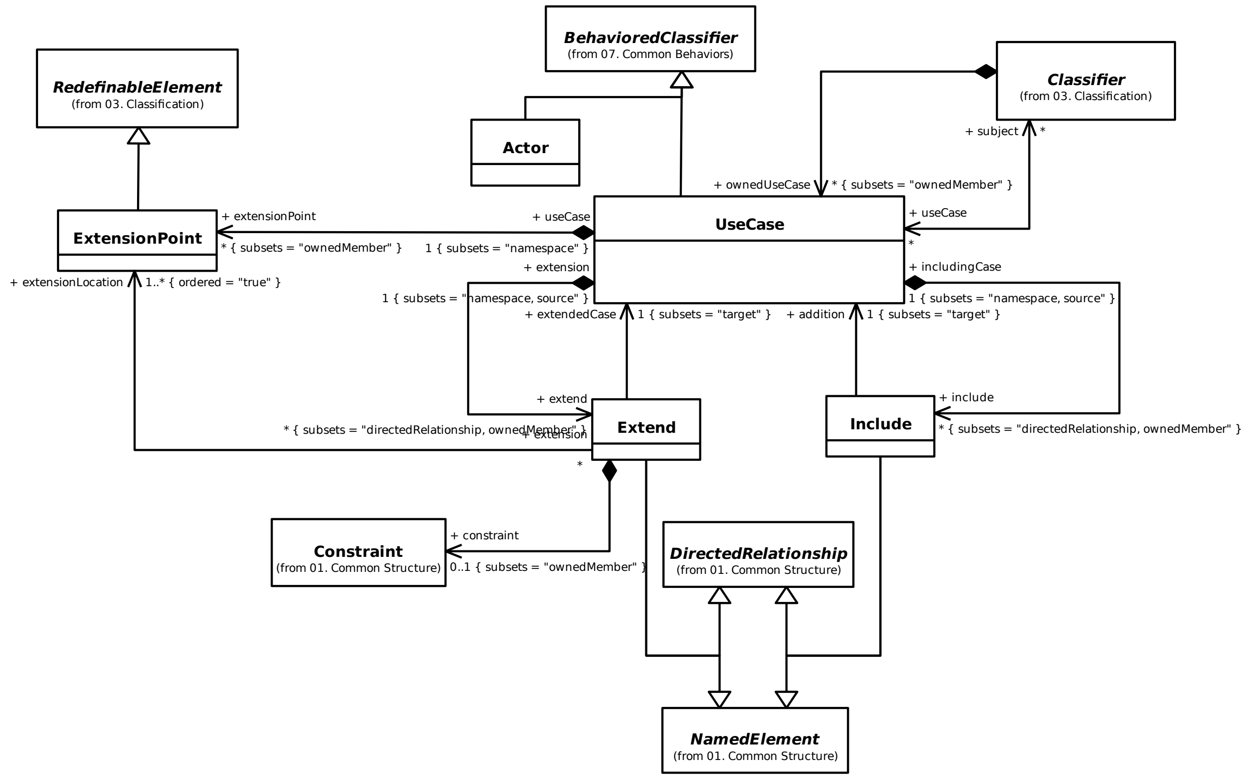

MessageSort
+ synchCall
+ asynchCall
+ asynchSignal
+ createMessage
+ deleteMessage
+ reply
MessageKind
+ complete
+ lost
+ found
+ unknown
Interaction
+ interaction
1 { subsets = "namespace" }
+ message
* { subsets = "ownedMember" }
+ signature
0..1
NamedElement
(from 01. Common Structure)
Message
+ /messageKind: MessageKind
+ messageSort: MessageSort
+ /message
0..1
+ /messageEnd
0..2
+ sendMessage
0..1 { subsets = "message" }
+ sendEvent
0..1 { subsets = "messageEnd" }
+ receiveMessage
0..1 { subsets = "message" }
+ receiveEvent
0..1 { subsets = "messageEnd" }
MessageEnd
OccurrenceSpecification
MessageOccurrenceSpecification
Multiplicity was '*'. Changed to accommodate simpleAttribute
+ argument
0..1 { subsets = "ownedElement" }
«simpleAttribute»
ValueSpecification
(from 02. Values)

13.11.4 4. Occurrences


Some changes have been made to the original model, to make it fit more to what we need in Gaphor.
NamedElement
(from 01. Common Structure)
InteractionFragment
+ interactionFragment
0..1 { subsets = "owner" }
GeneralOrdering
+ generalOrdering
* { subsets = "ownedElement" }
+ before
+ toAfter
1
*
+ after
+ toBefore
1
*
OccurrenceSpecification
+ /start
1
+ /finish
1
ExecutionSpecification
ExecutionOccurrenceSpecification
+ executionOccurrenceSpecification
+ execution
0..2
1
ActionExecutionSpecification
BehaviorExecutionSpecification
+ actionExecutionSpecification
*
+ actionExecutionSpecification
*
+ action
1
+ behavior
0..1
Action
(from 10. Actions)
Behavior
(from 07. Common Behaviors)

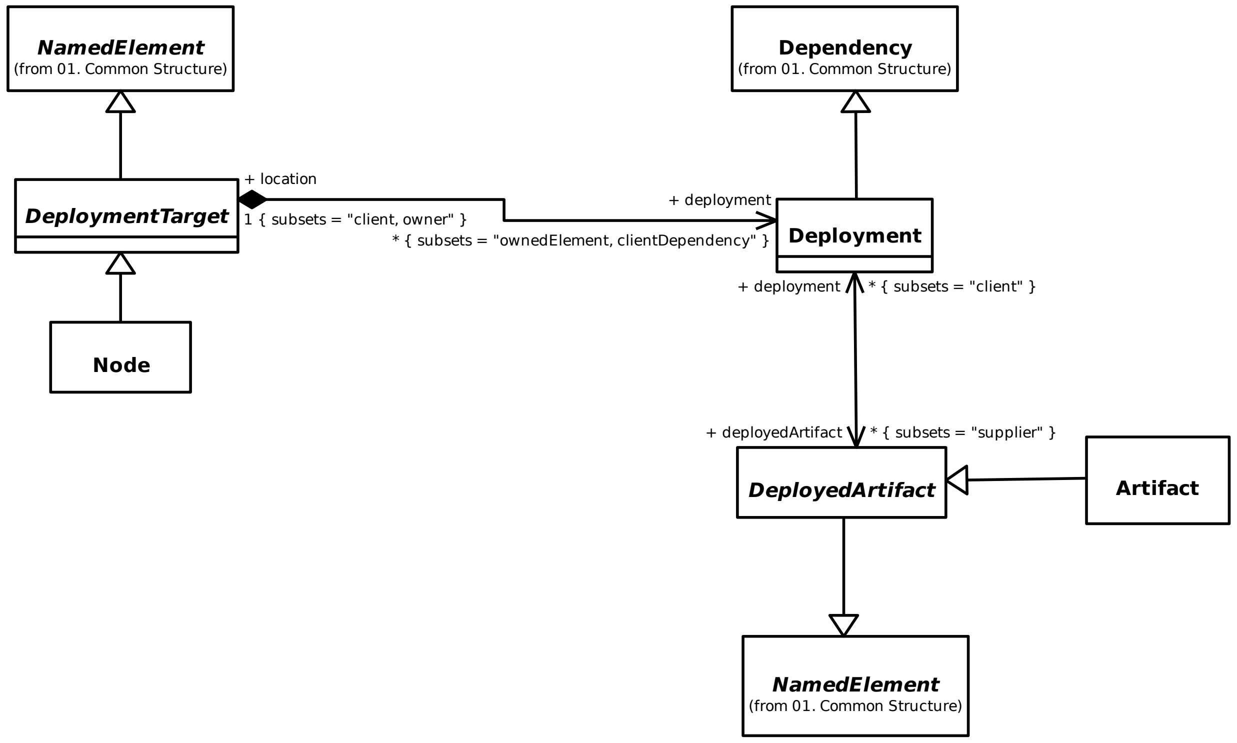
13.12 12. Use Cases

13.12.1 UseCases

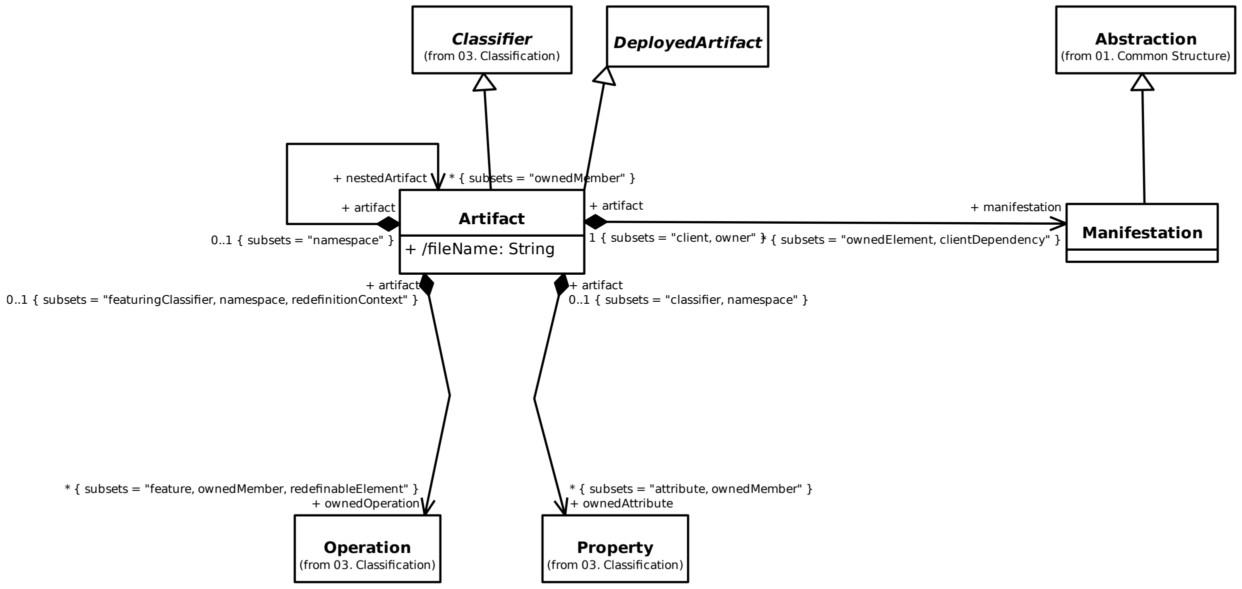

BehavioredClassifier
(from 07. Common Behaviors)
RedefinableElement
(from 03. Classification)
Classifier
(from 03. Classification)
Actor
+ subject
*
+ ownedUseCase
* { subsets = "ownedMember" }
ExtensionPoint
+ extensionPoint
* { subsets = "ownedMember" }
+ useCase
1 { subsets = "namespace" }
UseCase
+ useCase
*
+ extension
1 { subsets = "namespace, source" }
+ includingCase
1 { subsets = "namespace, source" }
+ extensionLocation
1..* { ordered = "true" }
+ extendedCase
1 { subsets = "target" }
+ addition
1 { subsets = "target" }
+ extend
Extend
+ extension
* { subsets = "directedRelationship, ownedMember" }
Include
+ include
* { subsets = "directedRelationship, ownedMember" }
*
Constraint
(from 01. Common Structure)
+ constraint
0..1 { subsets = "ownedMember" }
DirectedRelationship
(from 01. Common Structure)
NamedElement
(from 01. Common Structure)

13.13 13. Deployments

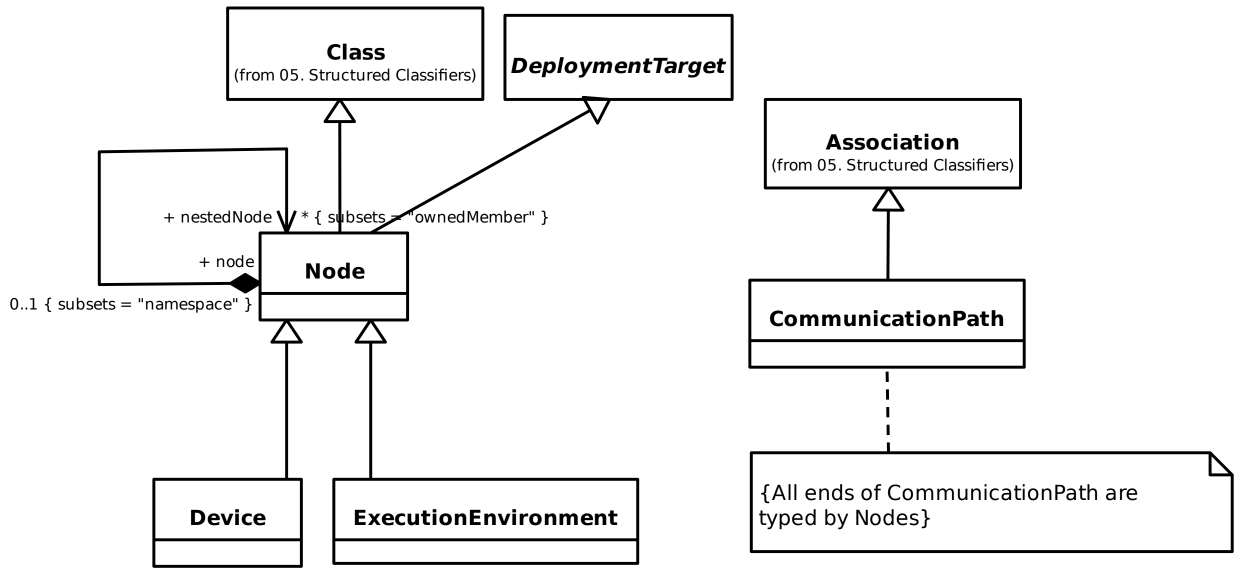
13.13.1 1. Deployments


NamedElement
(from 01. Common Structure)
Dependency
(from 01. Common Structure)
DeploymentTarget
+ location
1 { subsets = "client, owner" }
+ deployment
* { subsets = "ownedElement, clientDependency" }
Deployment
+ deployment
* { subsets = "client" }
Node
+ deployedArtifact
* { subsets = "supplier" }
DeployedArtifact
Artifact
NamedElement
(from 01. Common Structure)

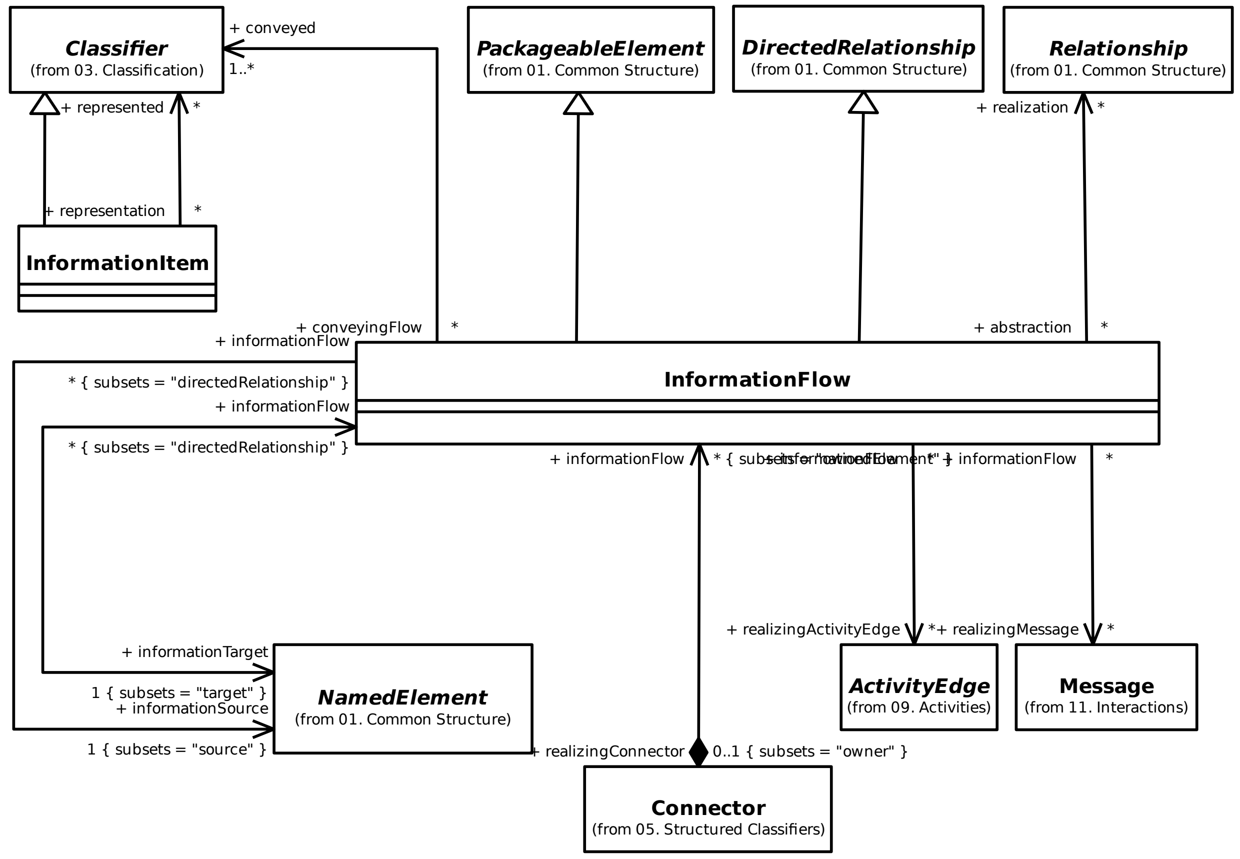
13.13.2 2. Artifacts


Classifier
(from 03. Classification)
DeployedArtifact
Abstraction
(from 01. Common Structure)
+ nestedArtifact
* { subsets = "ownedMember" }
+ artifact
0..1 { subsets = "namespace" }
Artifact
+ /fileName: String
+ artifact
1 { subsets = "client, owner" }
+ manifestation
* { subsets = "ownedElement, clientDependency" }
Manifestation
+ artifact
0..1 { subsets = "featuringClassifier, namespace, redefinitionContext" }
+ artifact
0..1 { subsets = "classifier, namespace" }
* { subsets = "feature, ownedMember, redefinableElement" }
+ ownedOperation
* { subsets = "attribute, ownedMember" }
+ ownedAttribute
Operation
(from 03. Classification)
Property
(from 03. Classification)

13.13.3 3. Nodes

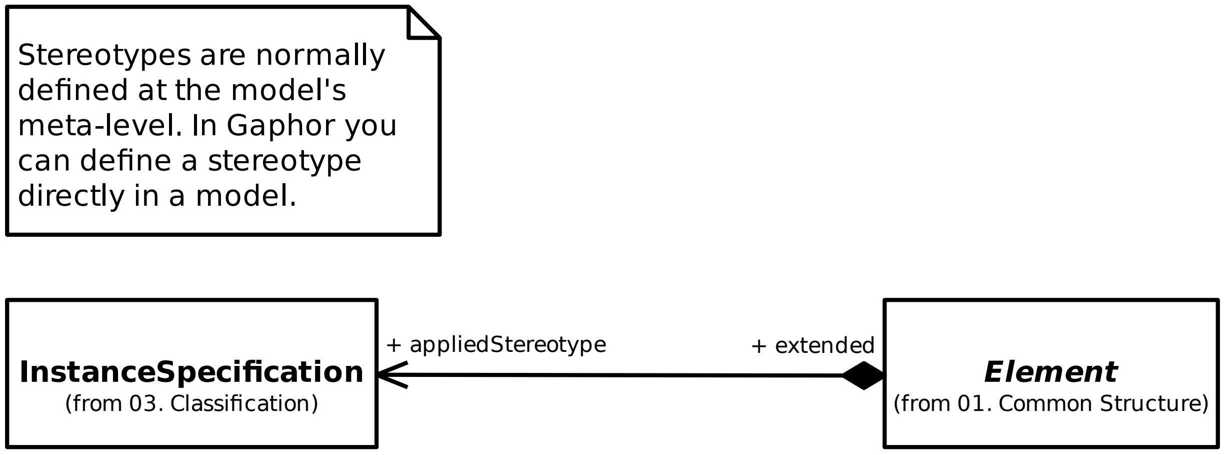

Class
(from 05. Structured Classifiers)
DeploymentTarget
Association
(from 05. Structured Classifiers)
+ nestedNode
* { subsets = "ownedMember" }
+ node
0..1 { subsets = "namespace" }
Node
CommunicationPath
Device
ExecutionEnvironment
{All ends of CommunicationPath are typed by Nodes}

13.14 14. Information Flows



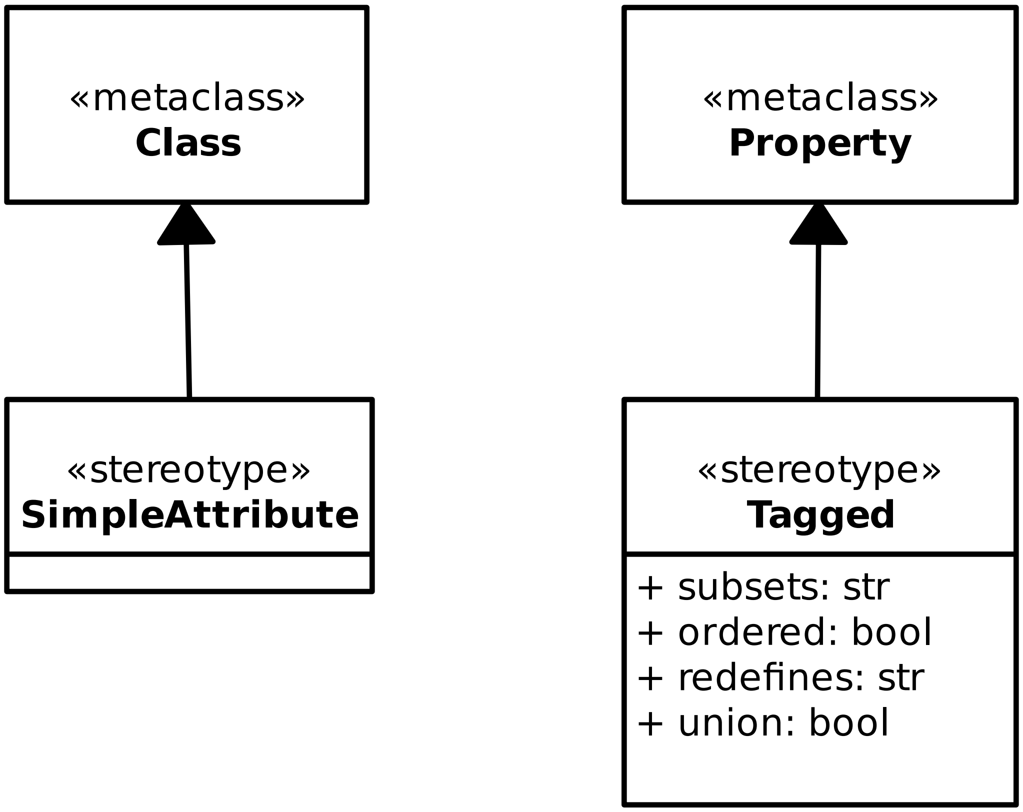
13.15 A. Gaphor Specific Constructs

13.15.1 1. Stereotype Applications


Stereotypes are normally defined at the model's meta-level. In Gaphor you can define a stereotype directly in a model.
InstanceSpecification
(from 03. Classification)
+ appliedStereotype
+ extended
Element
(from 01. Common Structure)

13.16 B. Gaphor Profile

In order to provide extra information to the diagram elements (mainly association ends), the Gaphor model has been extended with stereotypes.

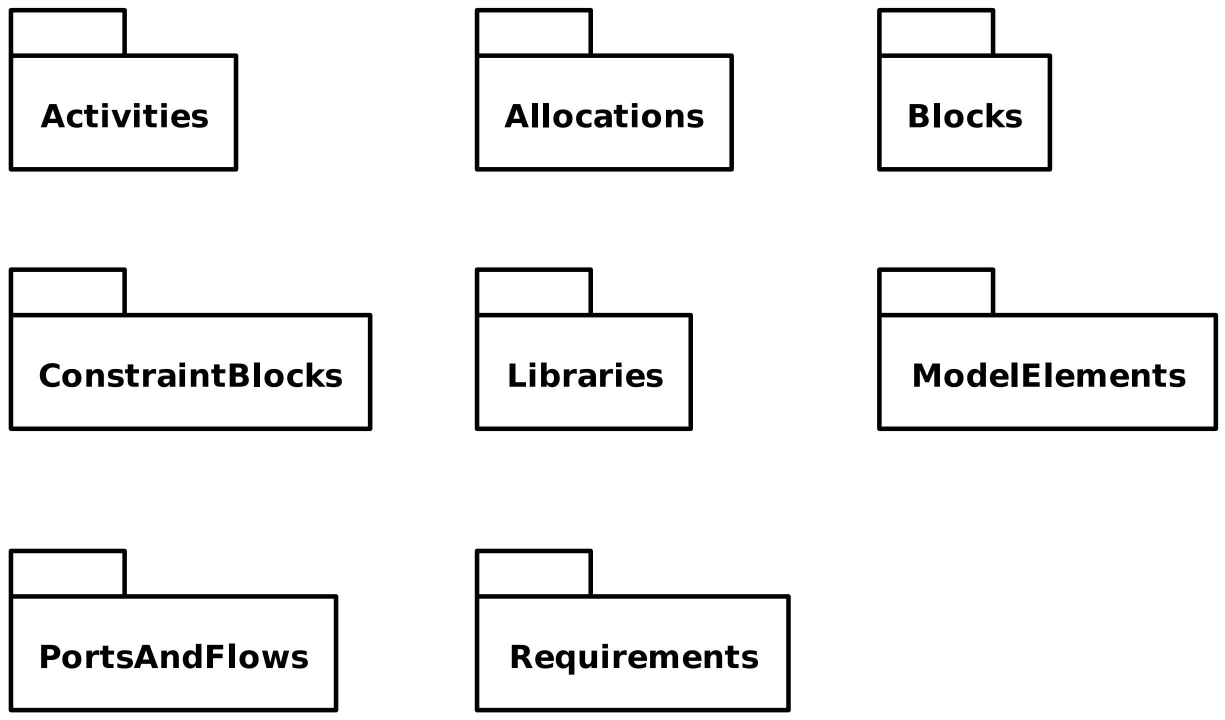


CHAPTER

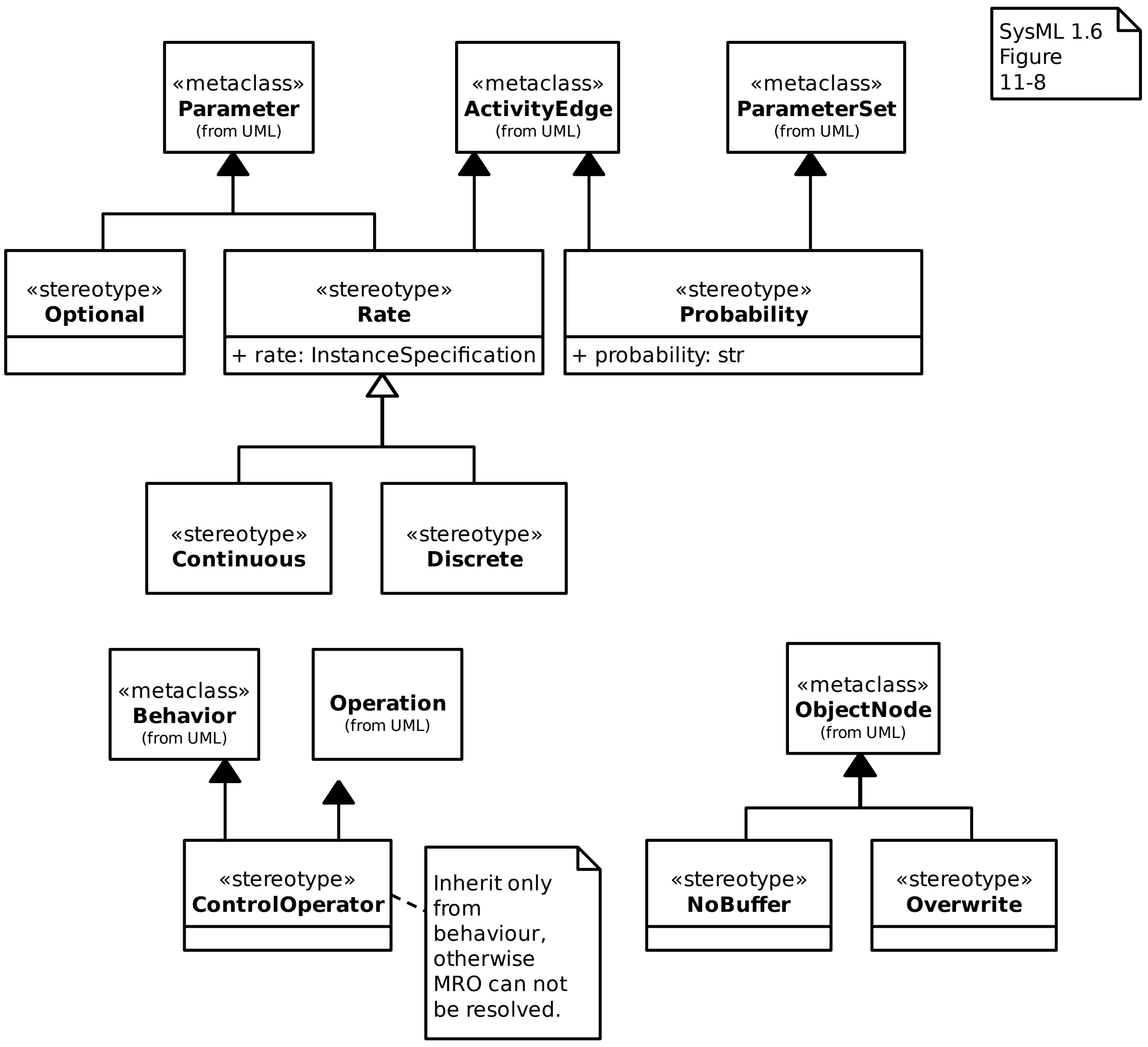
FOURTEEN

SYSTEMS MODELING LANGUAGE

Gaphor implements part of the SysML 1.6 specification.

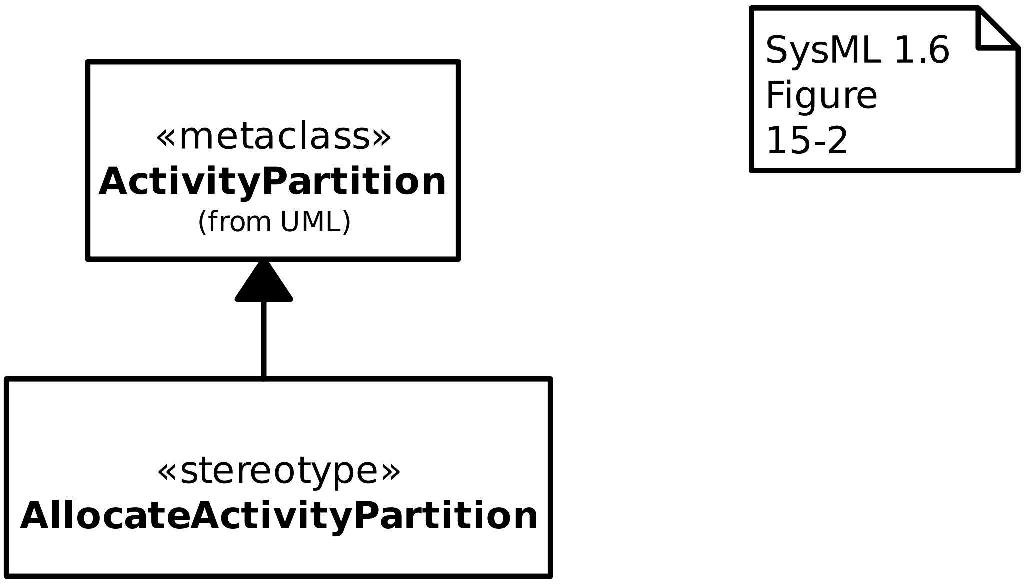


14.1 Activities

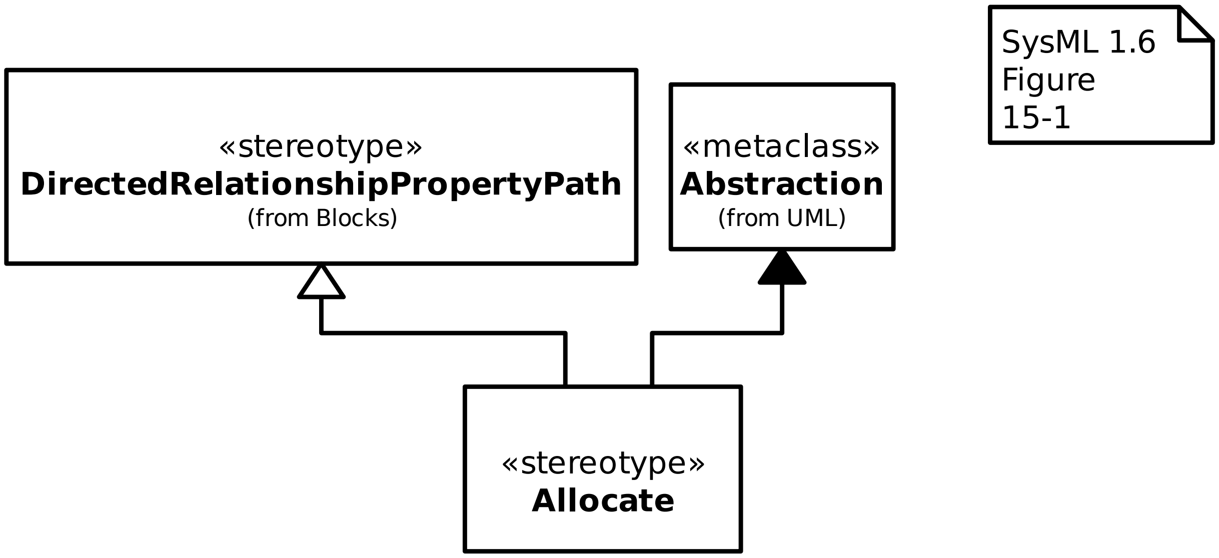

SysML 1.6 Figure 11-8
«metaclass» Parameter (from UML)
«metaclass» ActivityEdge (from UML)
«metaclass» ParameterSet (from UML)
«stereotype» Optional
«stereotype» Rate
+ rate: InstanceSpecification
«stereotype» Probability
+ probability: str
«stereotype» Continuous
«stereotype» Discrete
«metaclass» Behavior (from UML)
Operation (from UML)
«metaclass» ObjectNode (from UML)
«stereotype» ControlOperator
Inherit only from behaviour, otherwise MRO can not be resolved.
«stereotype» NoBuffer
«stereotype» Overwrite

14.2 Allocations

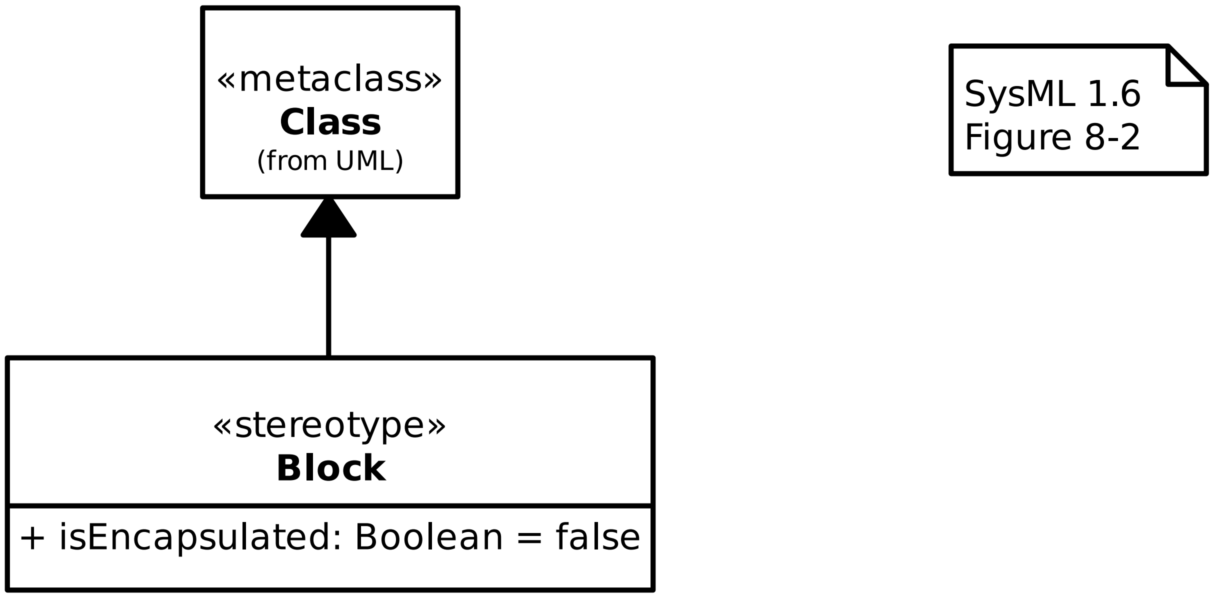
14.2.1 AllocatedActivityPartition



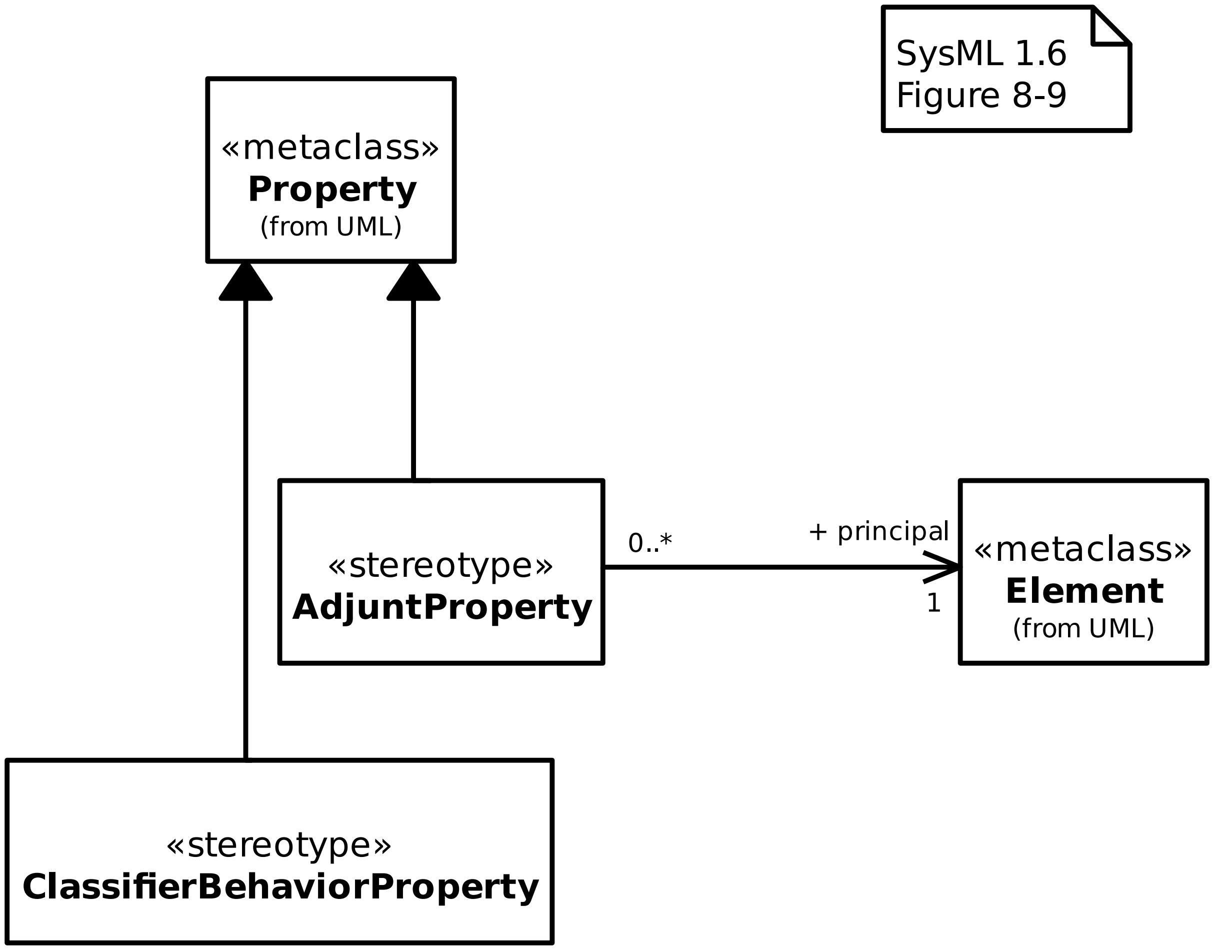
14.2.2 Allocation



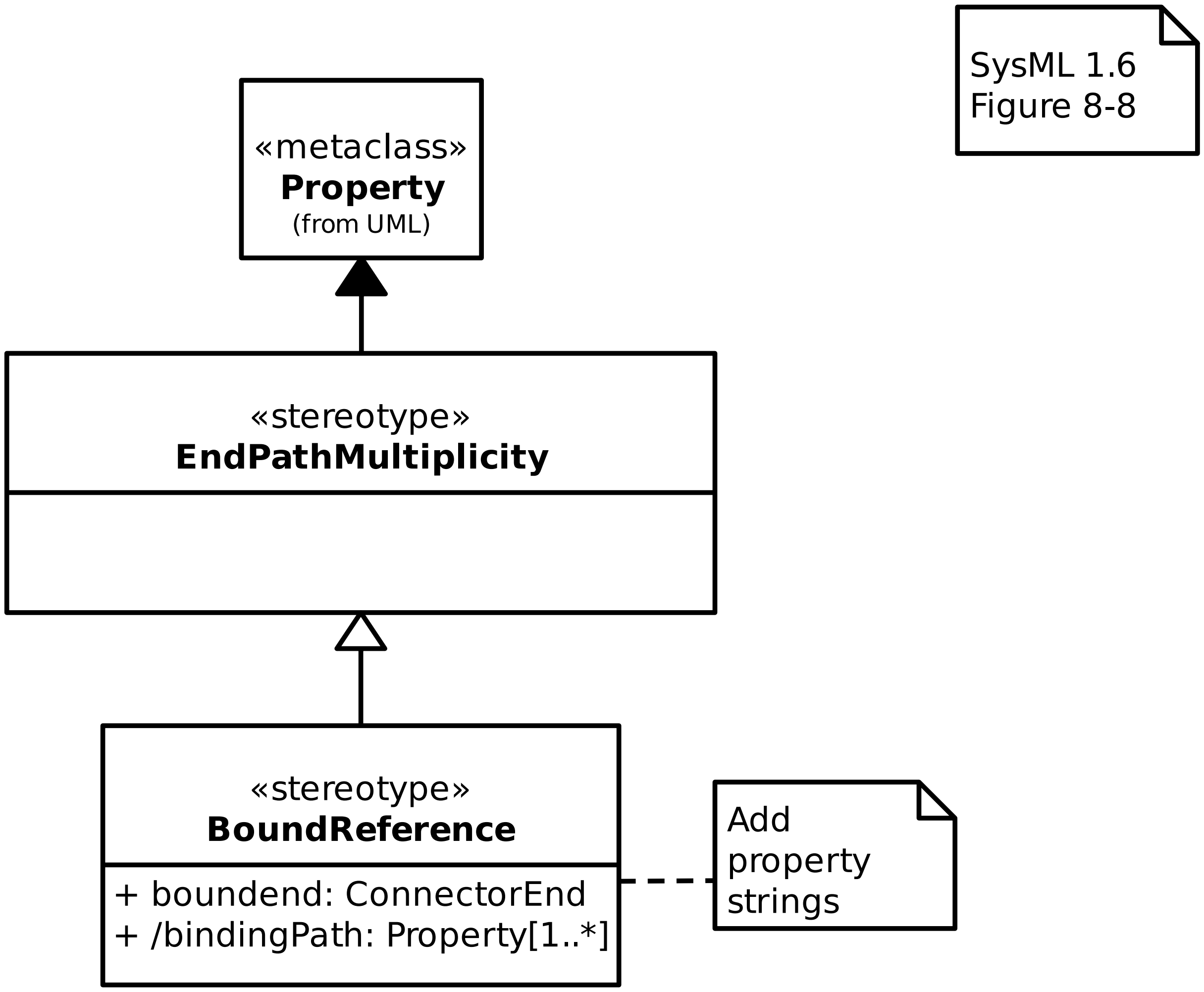
14.3 Blocks


«metaclass»
Class
(from UML)
«stereotype»
Block
+ isEncapsulated: Boolean = false
SysML 1.6
Figure 8-2

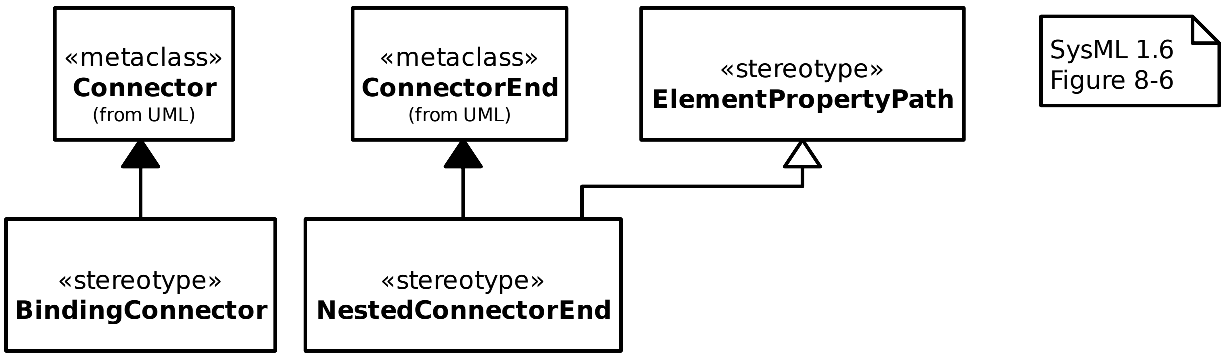
14.3.1 Adjunt and Classifier Behavior Properties



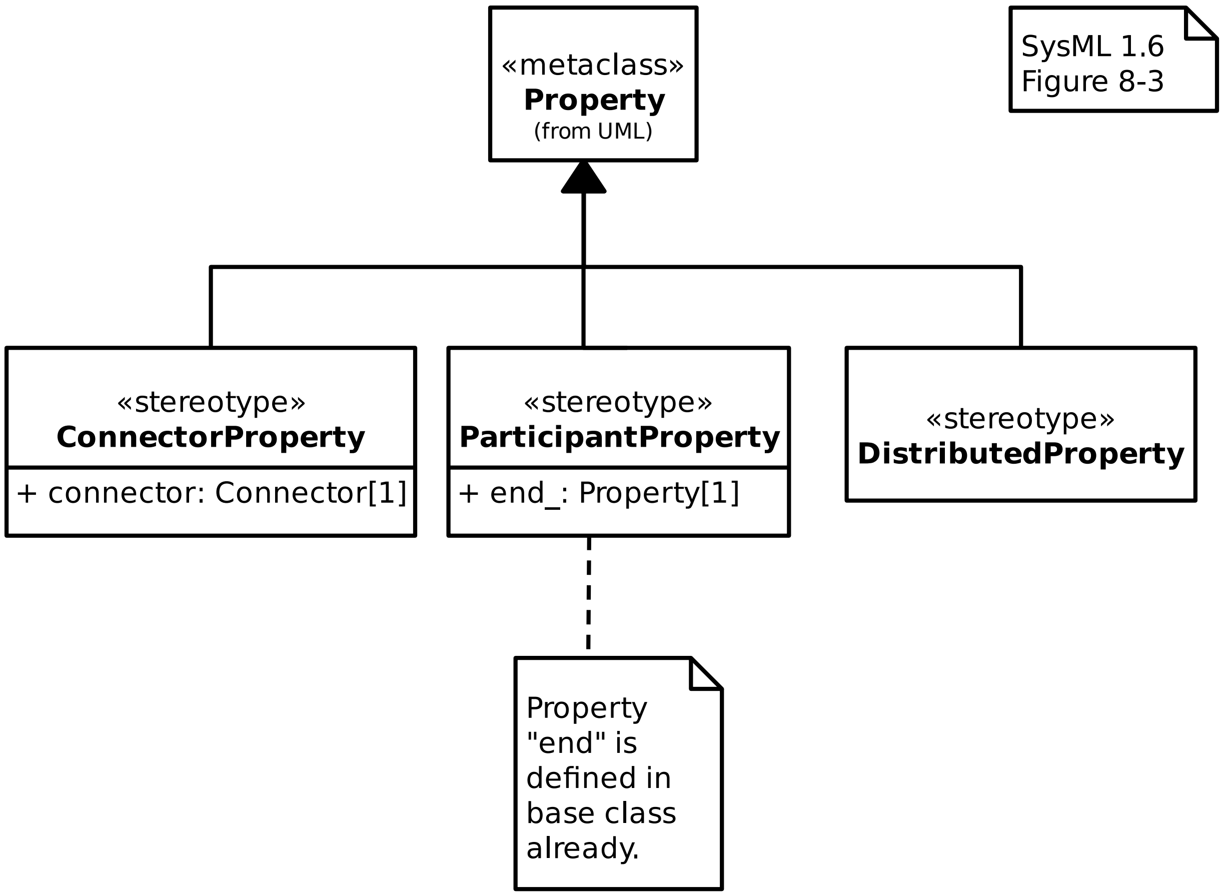
14.3.2 Bound References



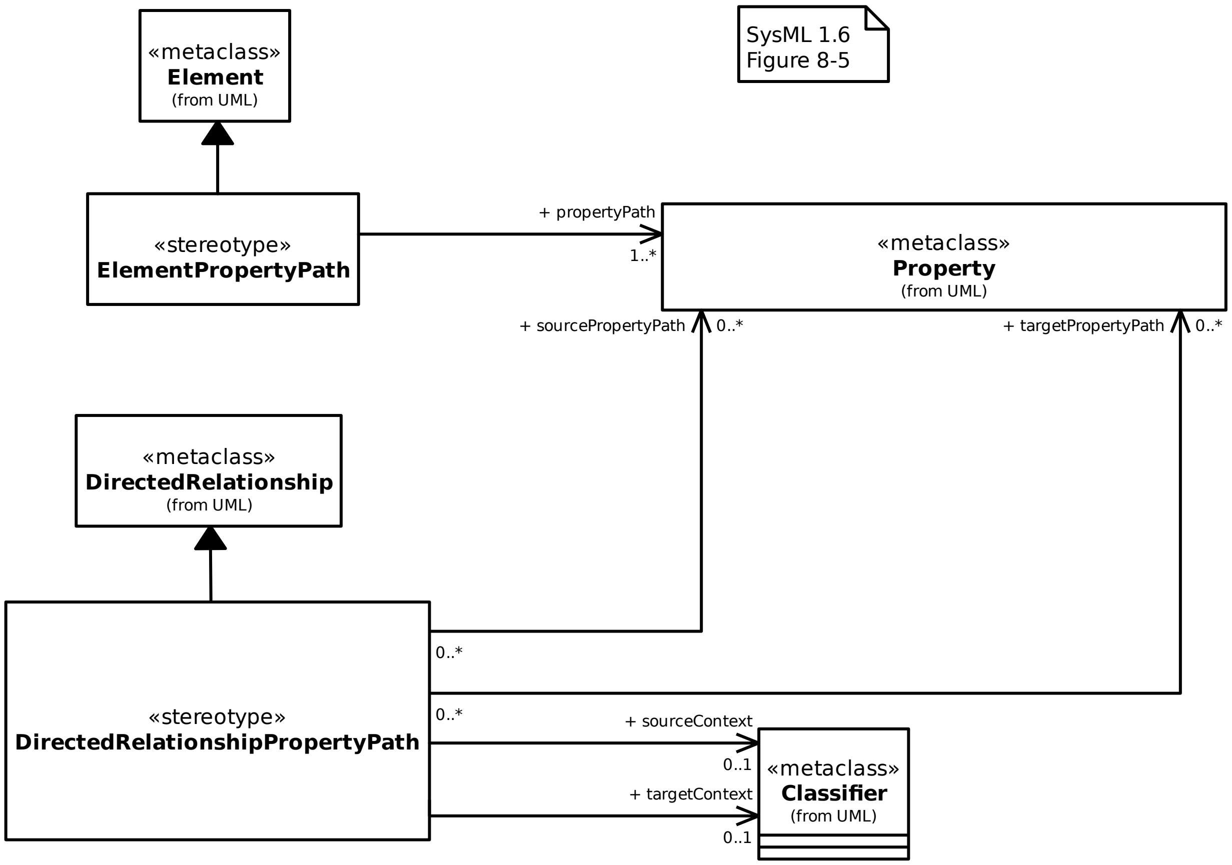
14.3.3 Connector Ends



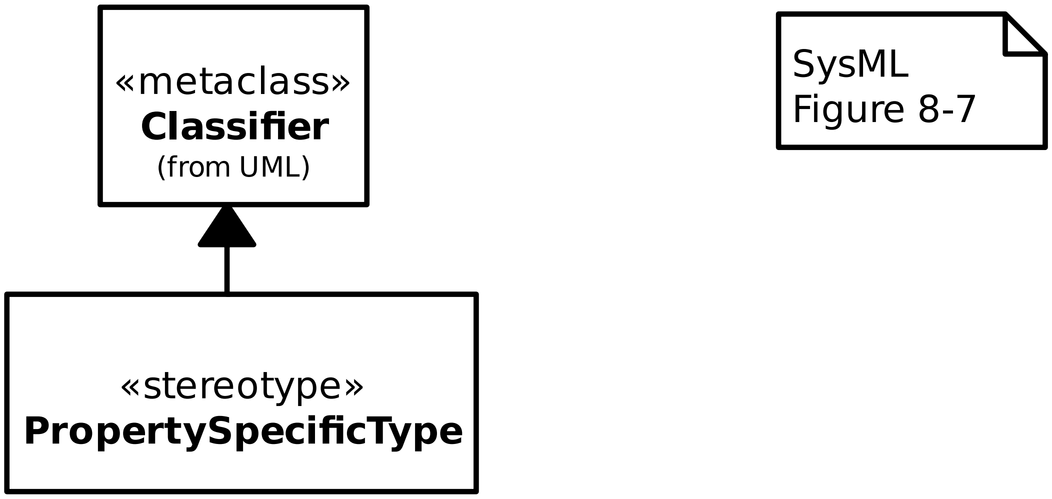
14.3.4 Properties



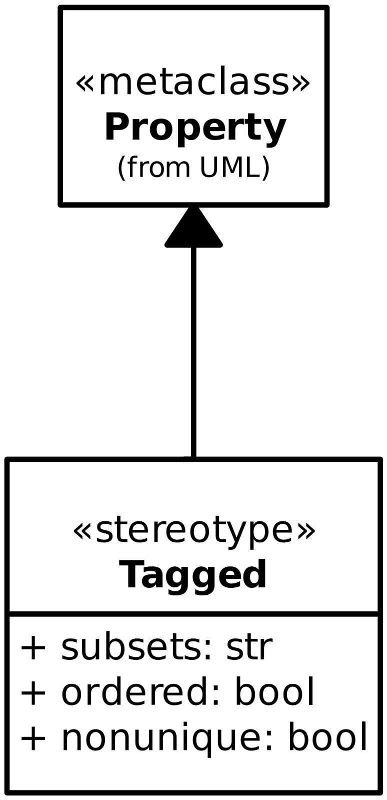
14.3.5 Property Paths



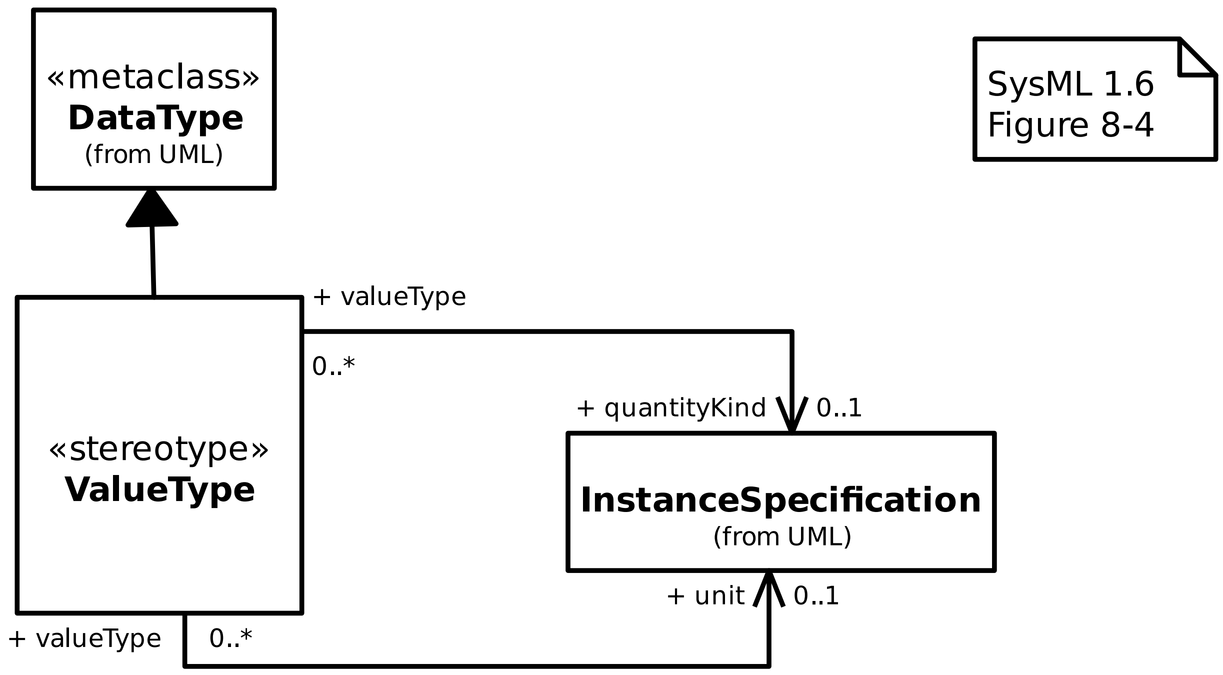
14.3.6 Property-Specific Types



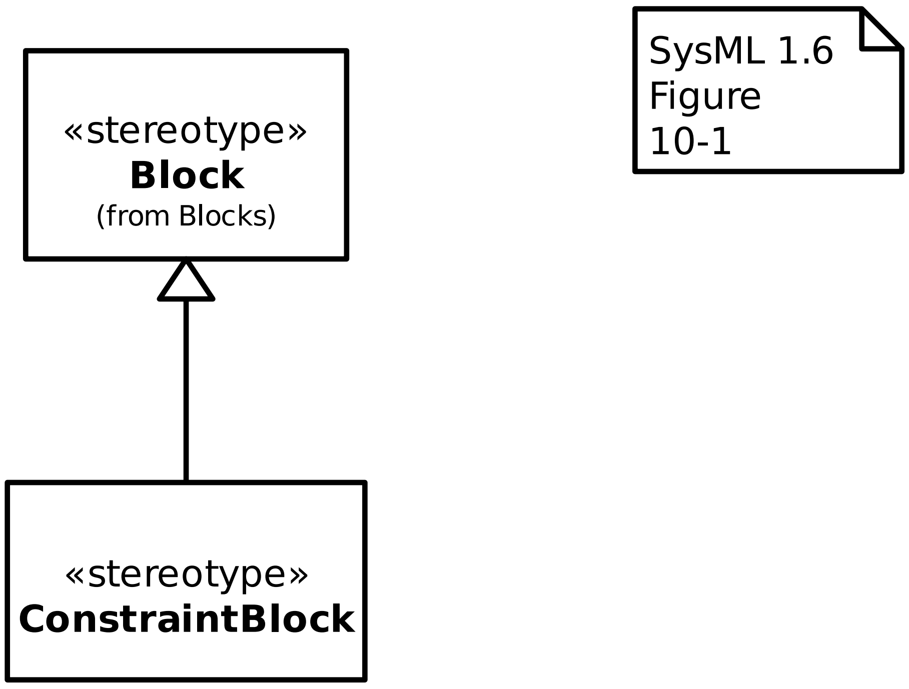
14.3.7 Property Strings



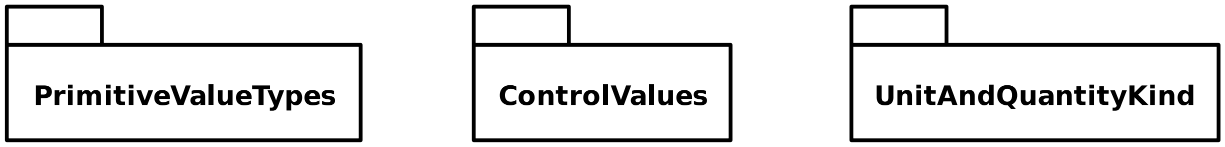
14.3.8 Value Types



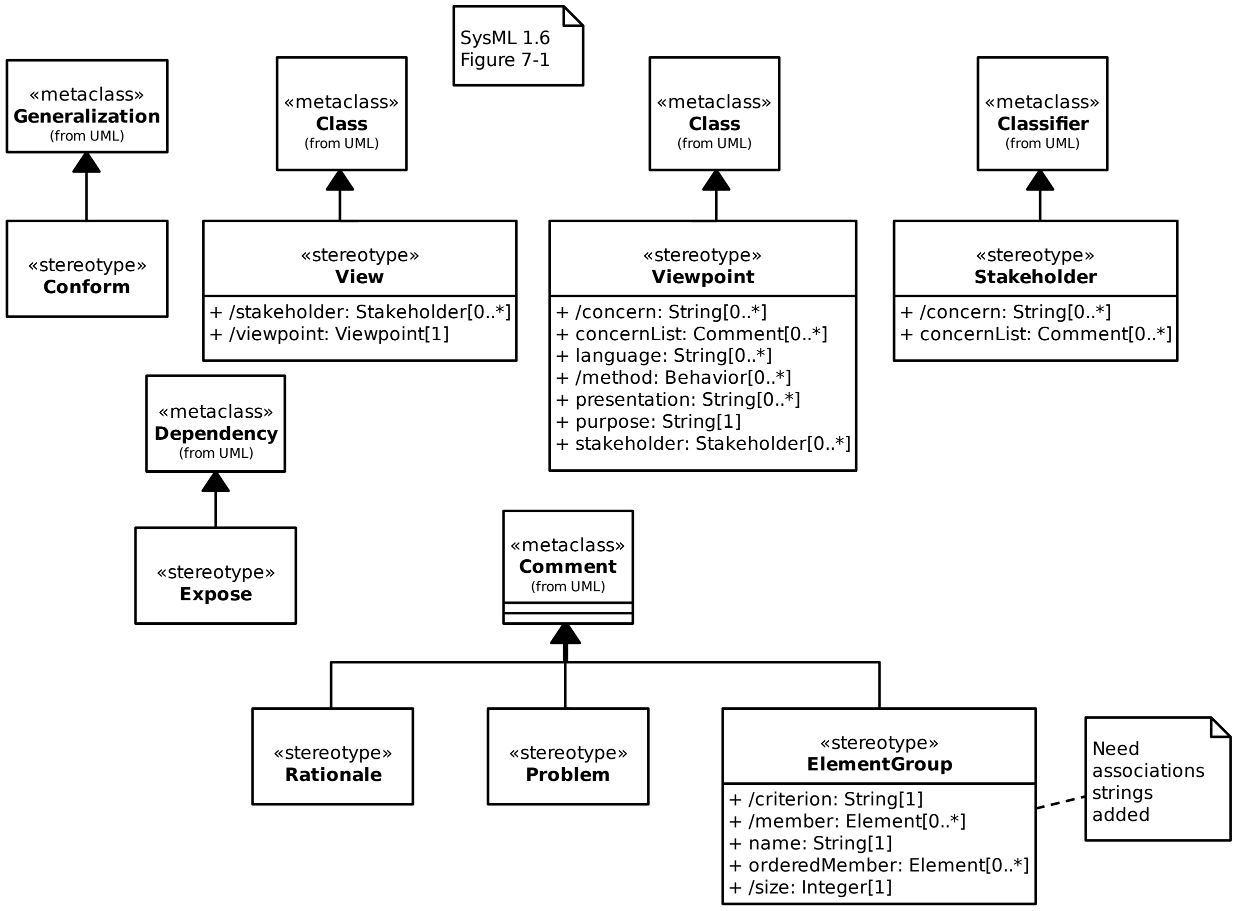
14.4 ConstraintBlocks



14.5 Libraries

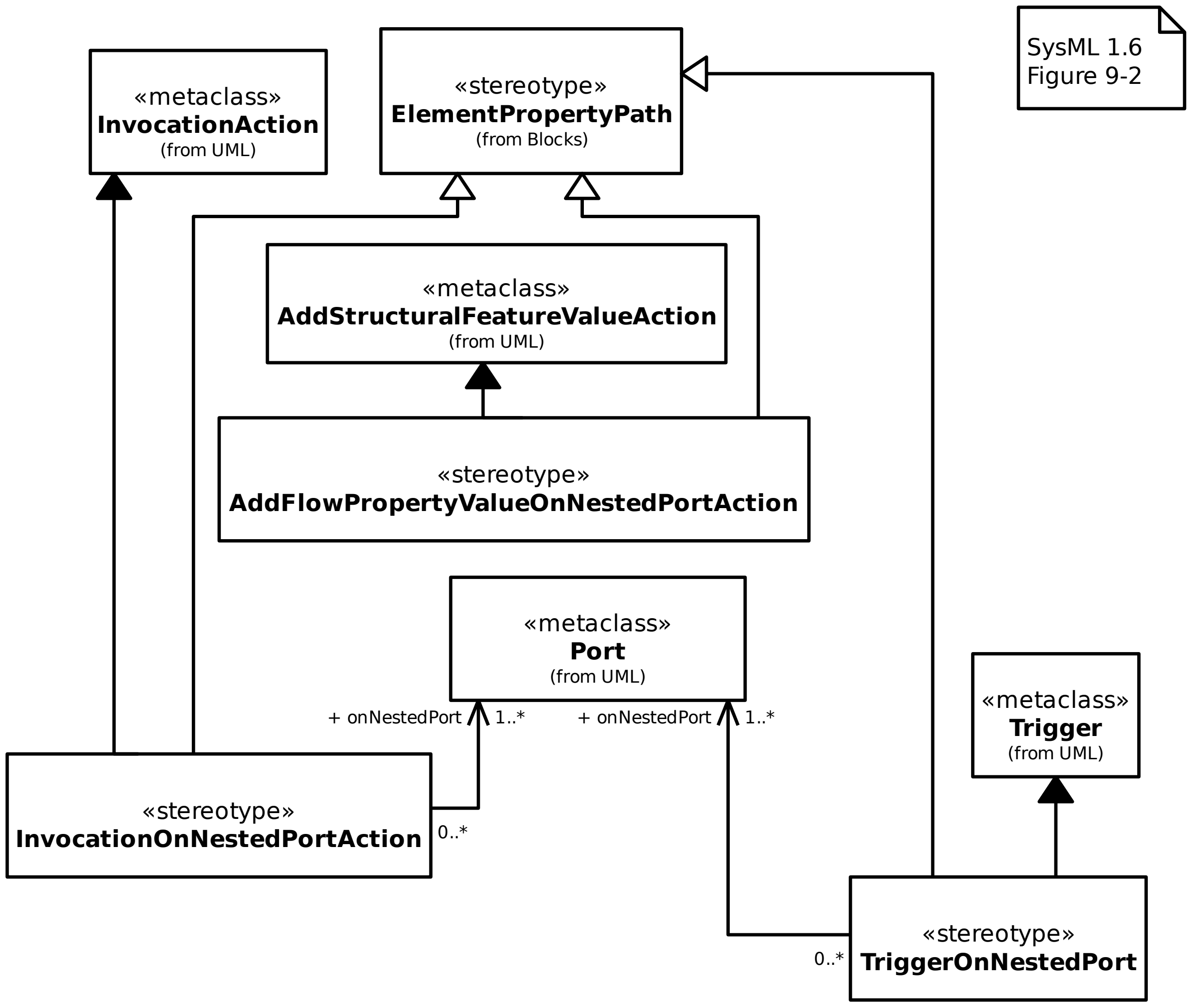


14.6 ModelElements

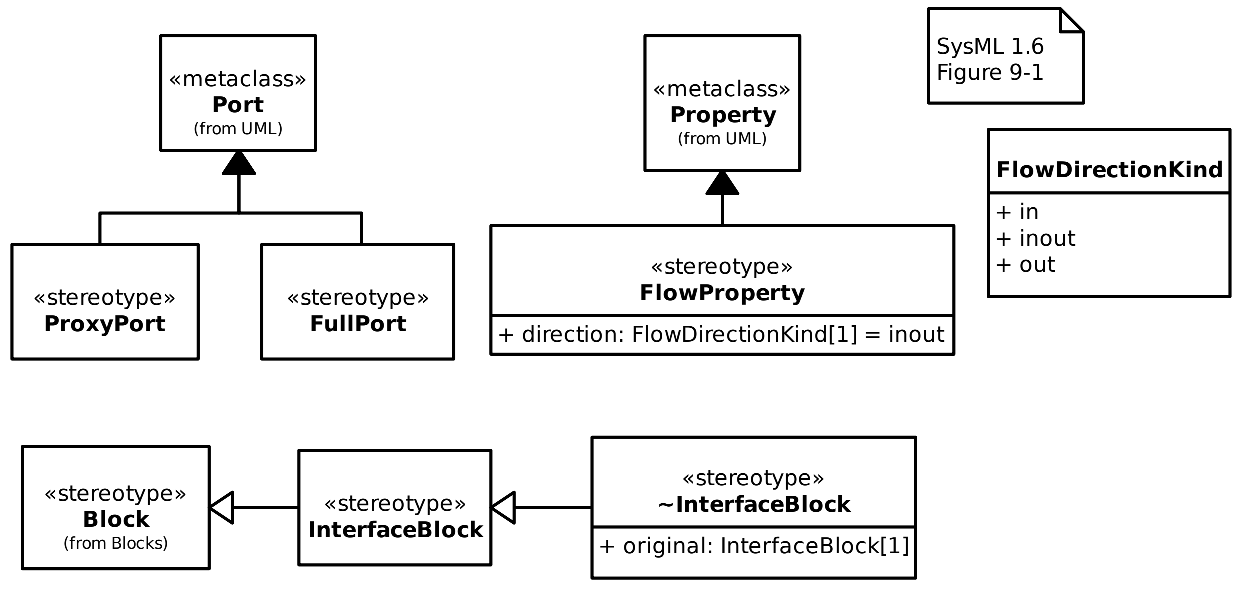

SysML 1.6 Figure 7-1
«metaclass» Generalization (from UML)
«stereotype» Conform
«metaclass» Class (from UML)
«stereotype» View
+ /stakeholder: Stakeholder[0..*]
+ /viewpoint: Viewpoint[1]
«metaclass» Class (from UML)
«stereotype» Viewpoint
+ /concern: String[0..*]
+ concernList: Comment[0..*]
+ language: String[0..*]
+ /method: Behavior[0..*]
+ presentation: String[0..*]
+ purpose: String[1]
+ stakeholder: Stakeholder[0..*]
«metaclass» Classifier (from UML)
«stereotype» Stakeholder
+ /concern: String[0..*]
+ concernList: Comment[0..*]
«metaclass» Dependency (from UML)
«stereotype» Expose
«metaclass» Comment (from UML)
«stereotype» Rationale
«stereotype» Problem
«stereotype» ElementGroup
+ /criterion: String[1]
+ /member: Element[0..*]
+ name: String[1]
+ orderedMember: Element[0..*]
+ /size: Integer[1]
Need associations strings added

14.7 PortsAndFlows

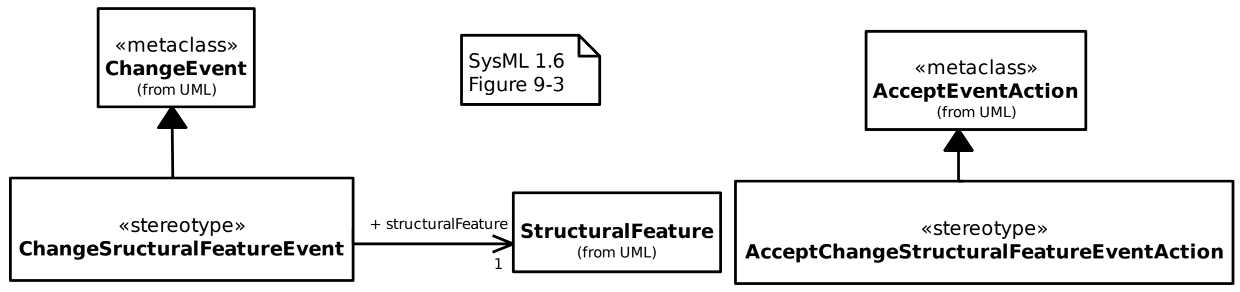
14.7.1 Actions on Nested Ports



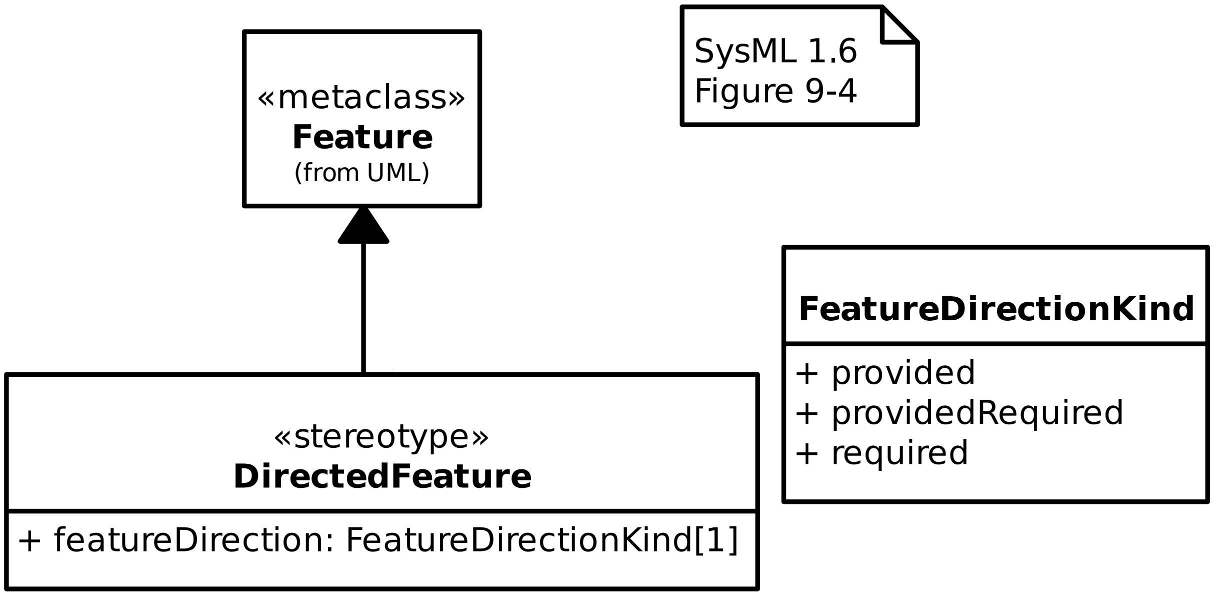
14.7.2 Port Stereotypes



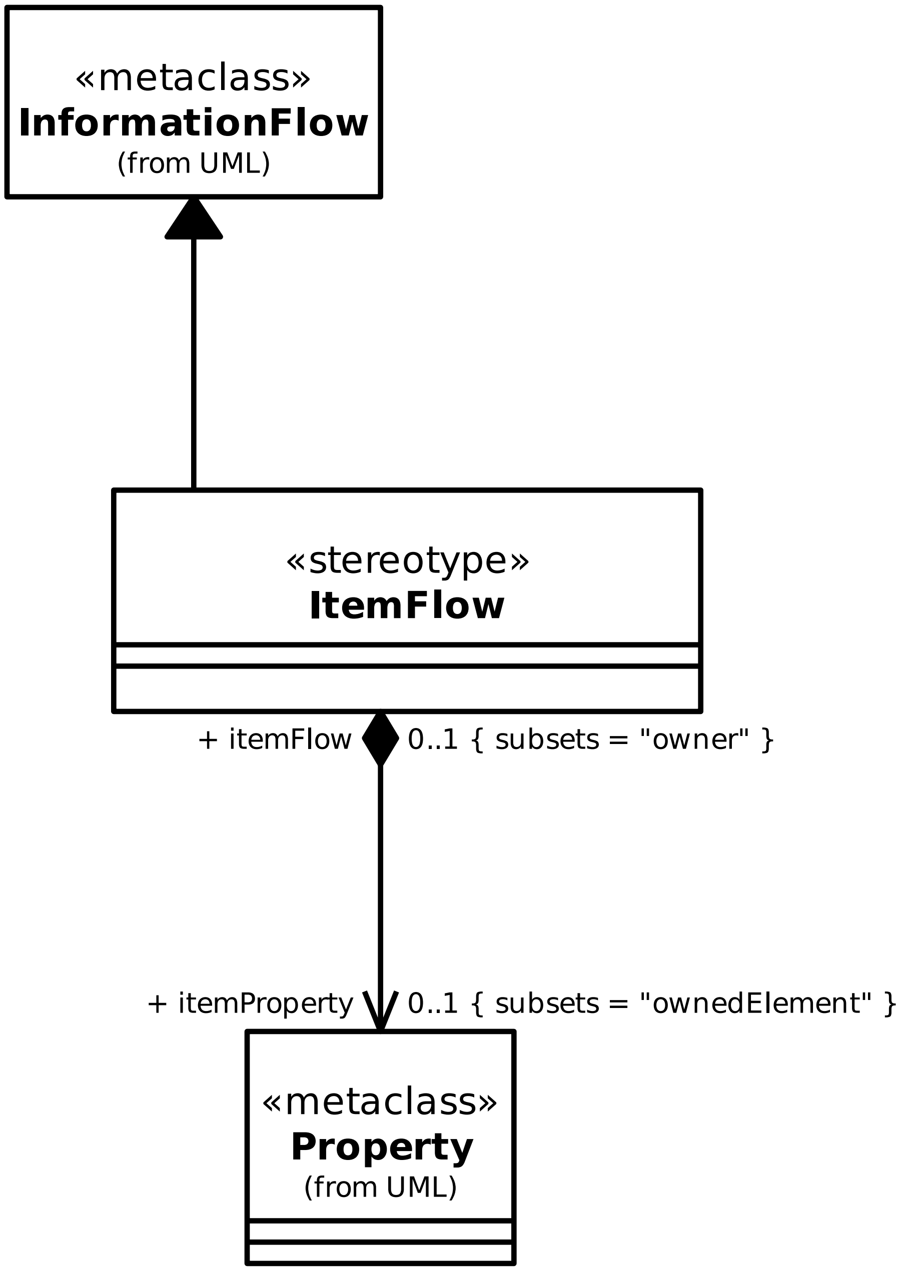
14.7.3 Property Value Change Events



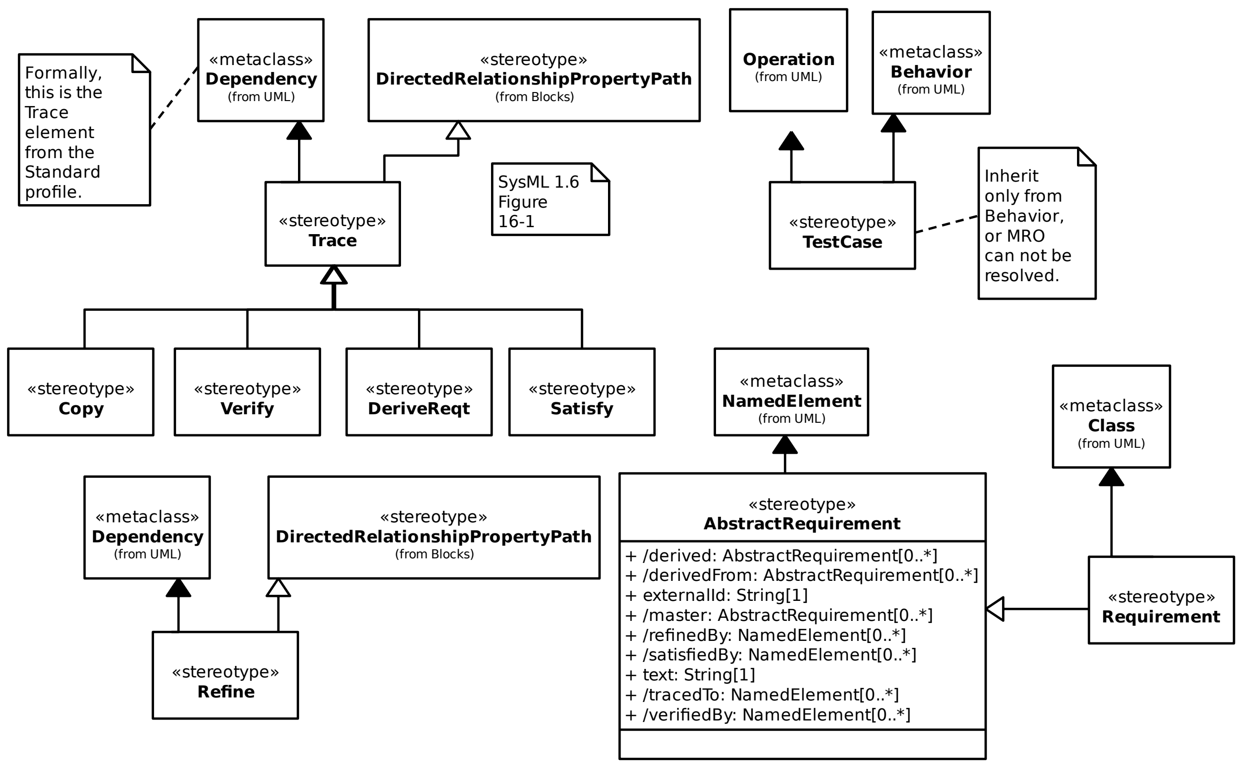
14.7.4 Provided and Required Features



14.7.5 Item Flow



14.8 Requirements

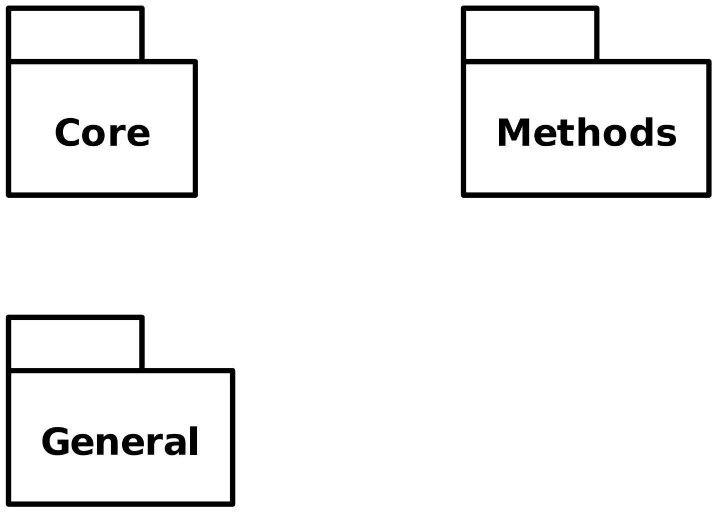

Formally, this is the Trace element from the Standard profile.
«metaclass» Dependency (from UML)
«stereotype» DirectedRelationshipPropertyPath (from Blocks)
Operation (from UML)
«metaclass» Behavior (from UML)
«stereotype» Trace
SysML 1.6 Figure 16-1
«stereotype» TestCase
Inherit only from Behavior, or MRO can not be resolved.
«stereotype» Copy
«stereotype» Verify
«stereotype» DeriveReqt
«stereotype» Satisfy
«metaclass» NamedElement (from UML)
«metaclass» Class (from UML)
«metaclass» Dependency (from UML)
«stereotype» DirectedRelationshipPropertyPath (from Blocks)
«stereotype» AbstractRequirement
+ /derived: AbstractRequirement[0..*]
+ /derivedFrom: AbstractRequirement[0..*]
+ externalId: String[1]
+ /master: AbstractRequirement[0..*]
+ /refinedBy: NamedElement[0..*]
+ /satisfiedBy: NamedElement[0..*]
+ text: String[1]
+ /tracedTo: NamedElement[0..*]
+ /verifiedBy: NamedElement[0..*]
«stereotype» Requirement
«stereotype» Refine

CHAPTER

FIFTEEN

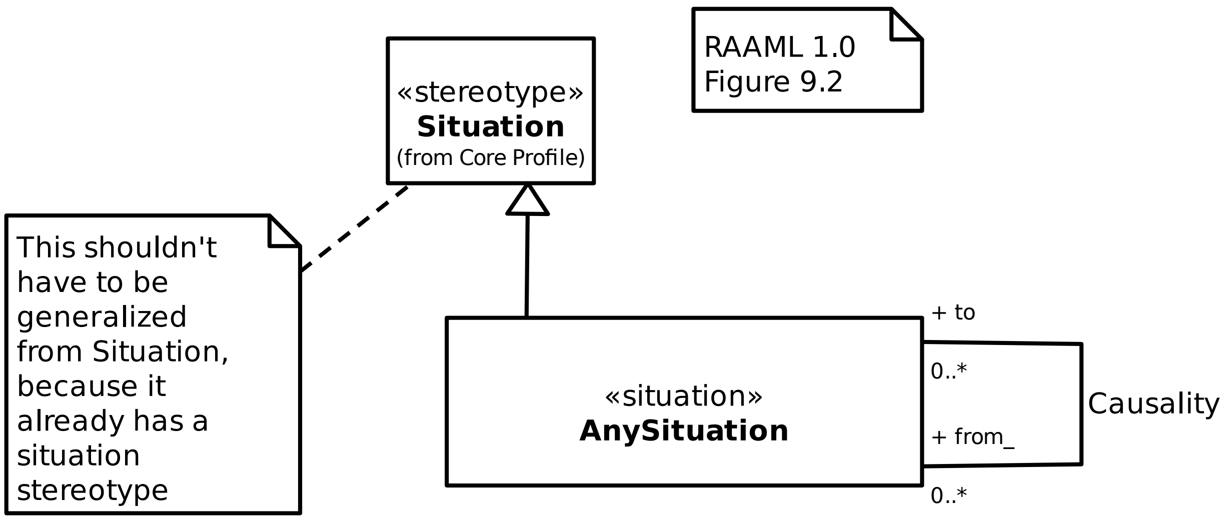
RISK ANALYSIS AND ASSESSMENT MODELING LANGUAGE

Gaphor implements parts of the RAAML 1.0 specification.

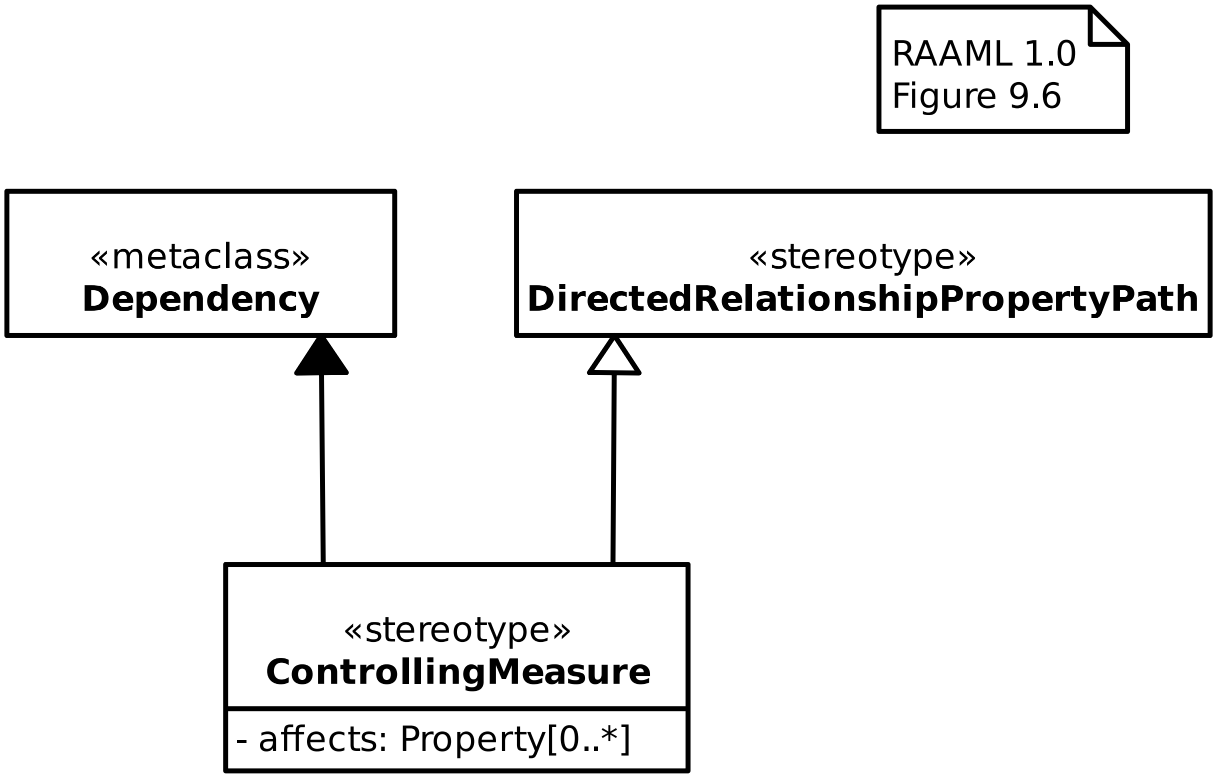


15.1 Core

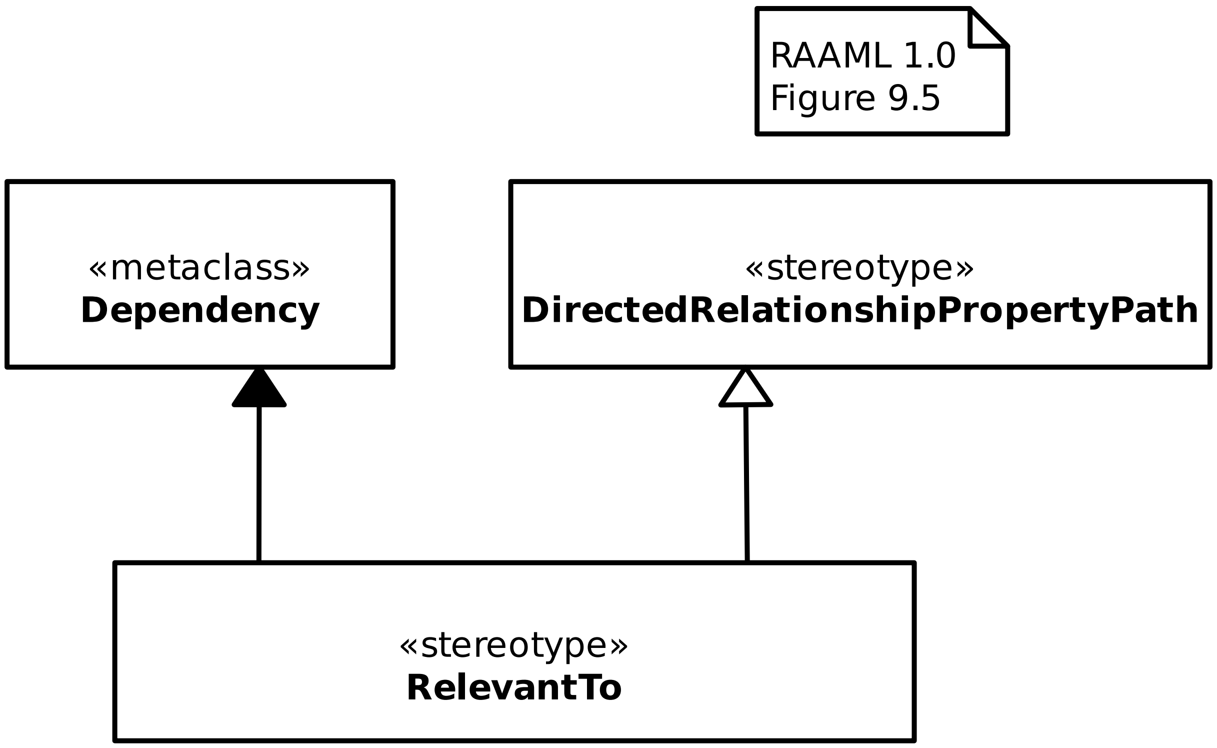
15.1.1 Core Library/Any Situation



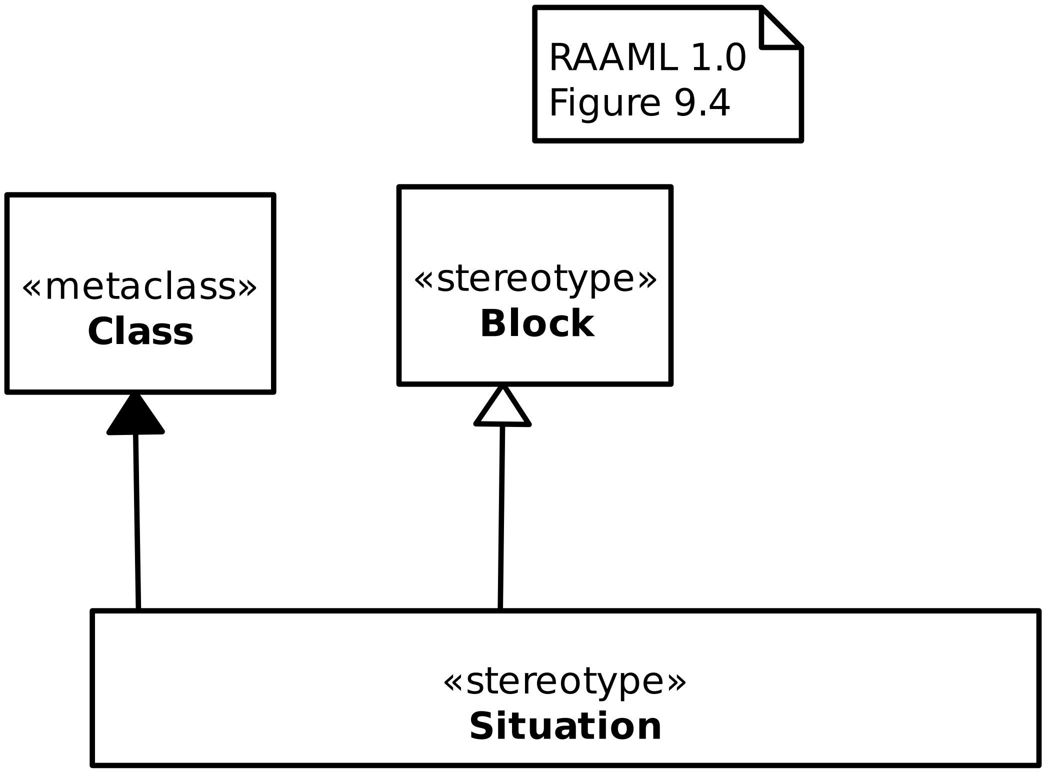
15.1.2 Core Profile/Controlling Measure



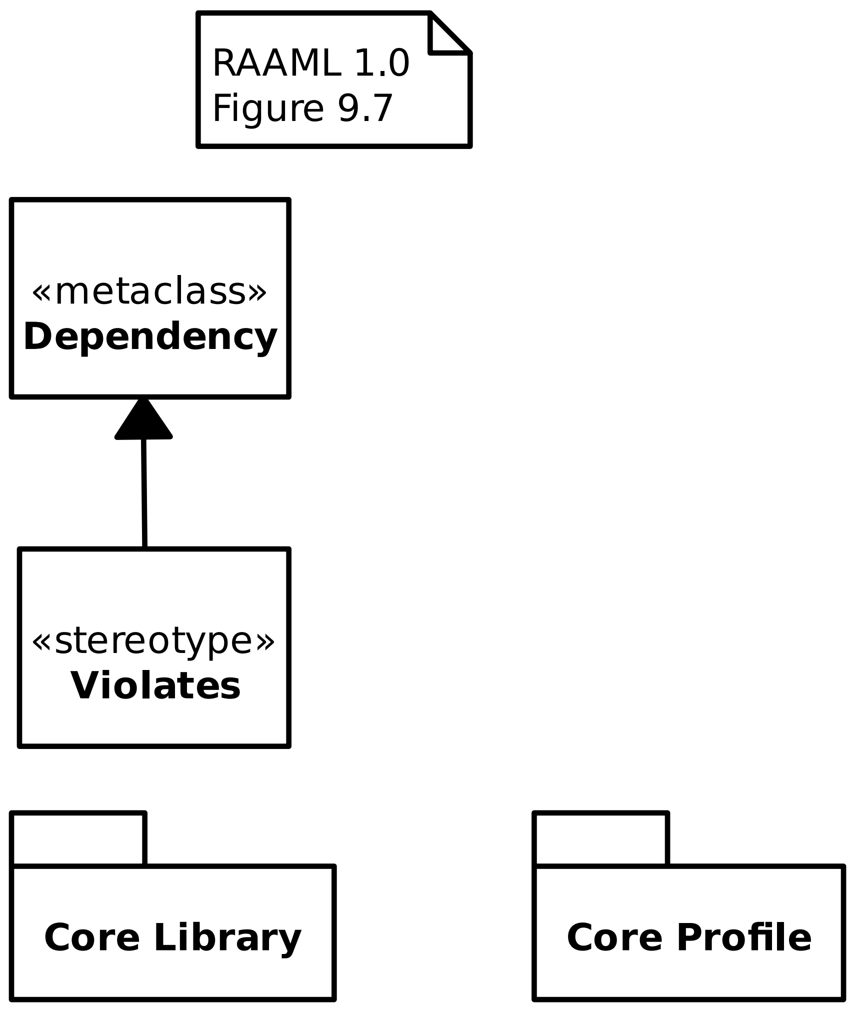
15.1.3 Core Profile/Relevant To



15.1.4 Core Profile/Situation

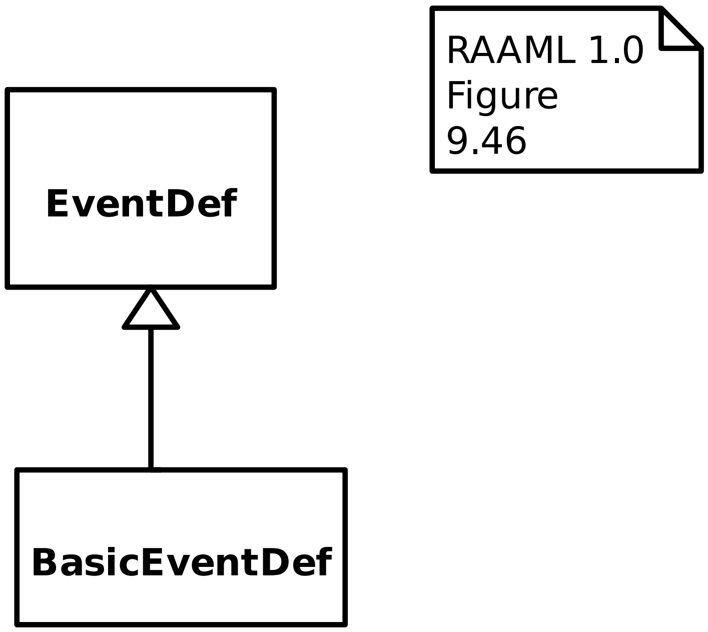


15.1.5 Core Profile/Violates

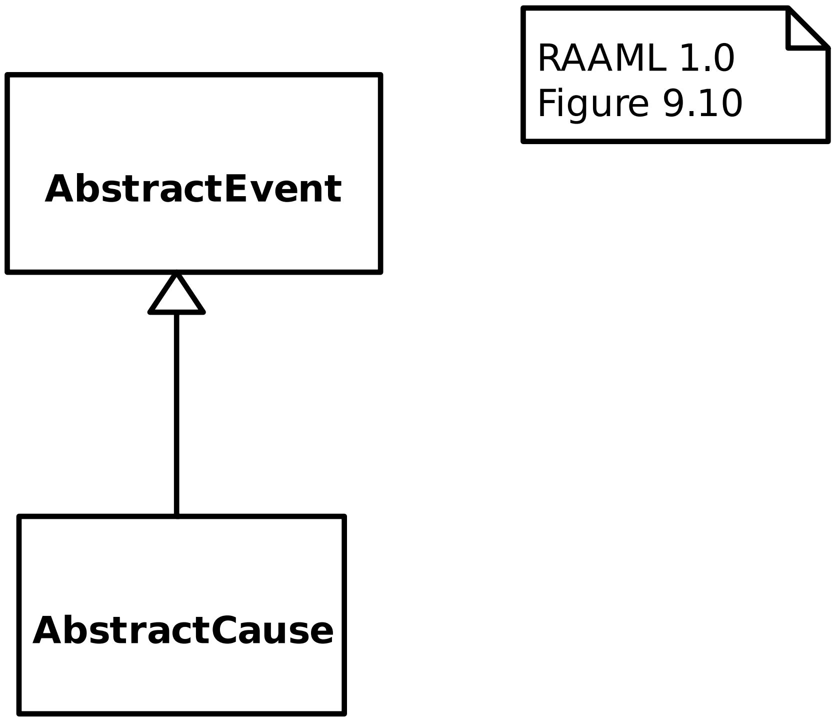


15.2 General

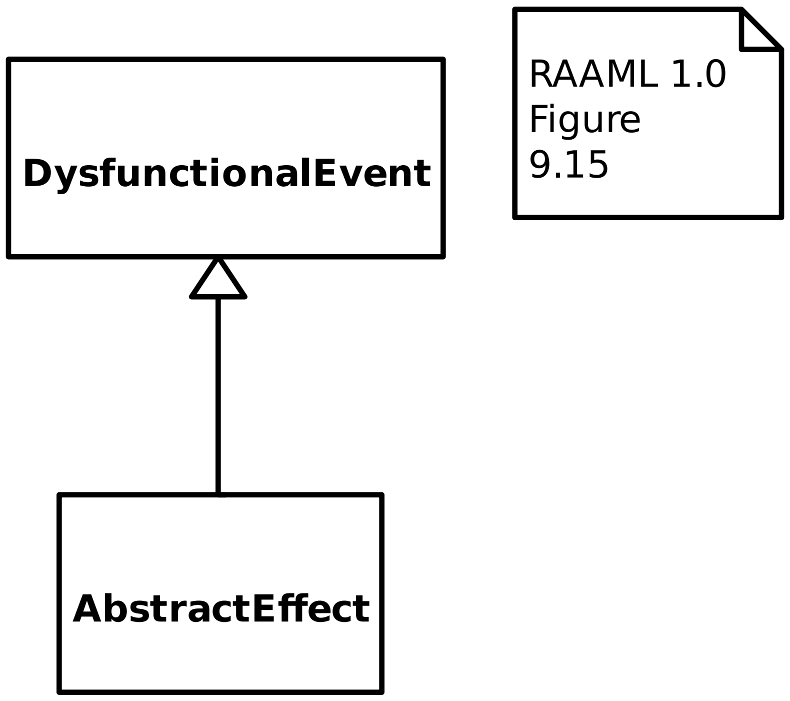
15.2.1 Basic Event



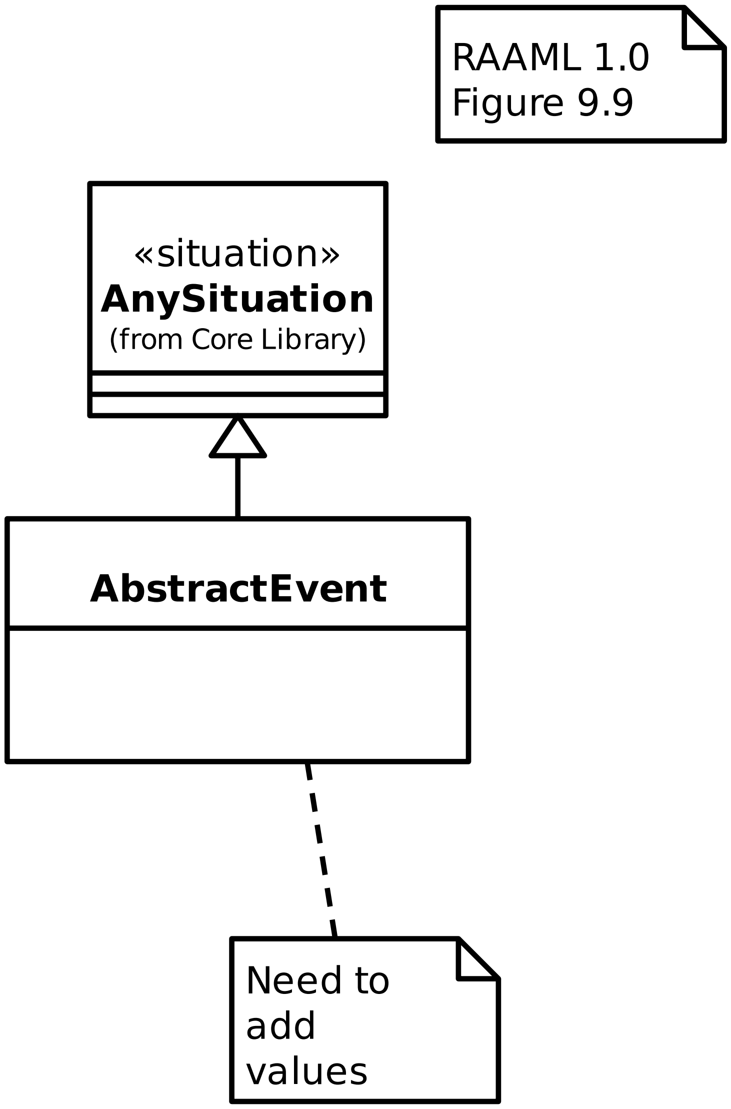
15.2.2 General Concepts Library/Abstract Cause



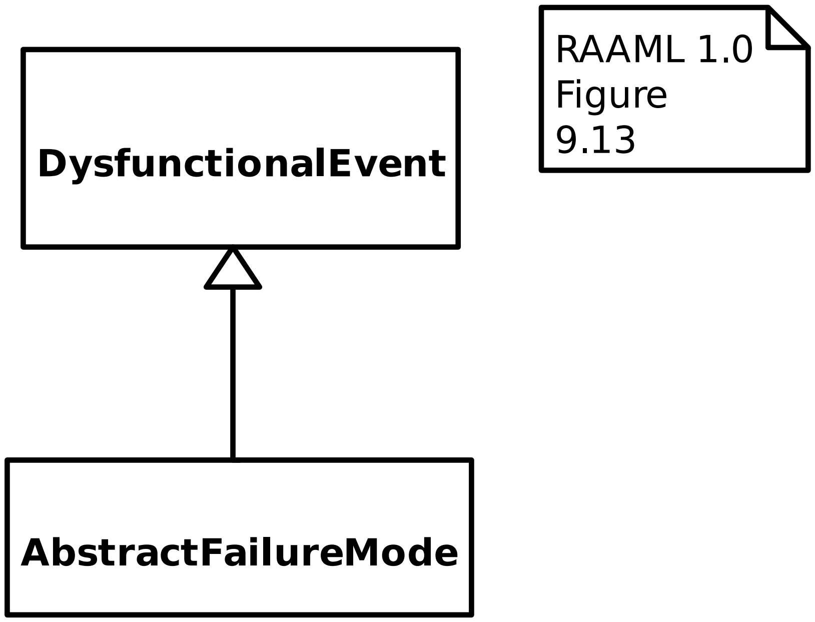
15.2.3 General Concepts Library/Abstract Effect



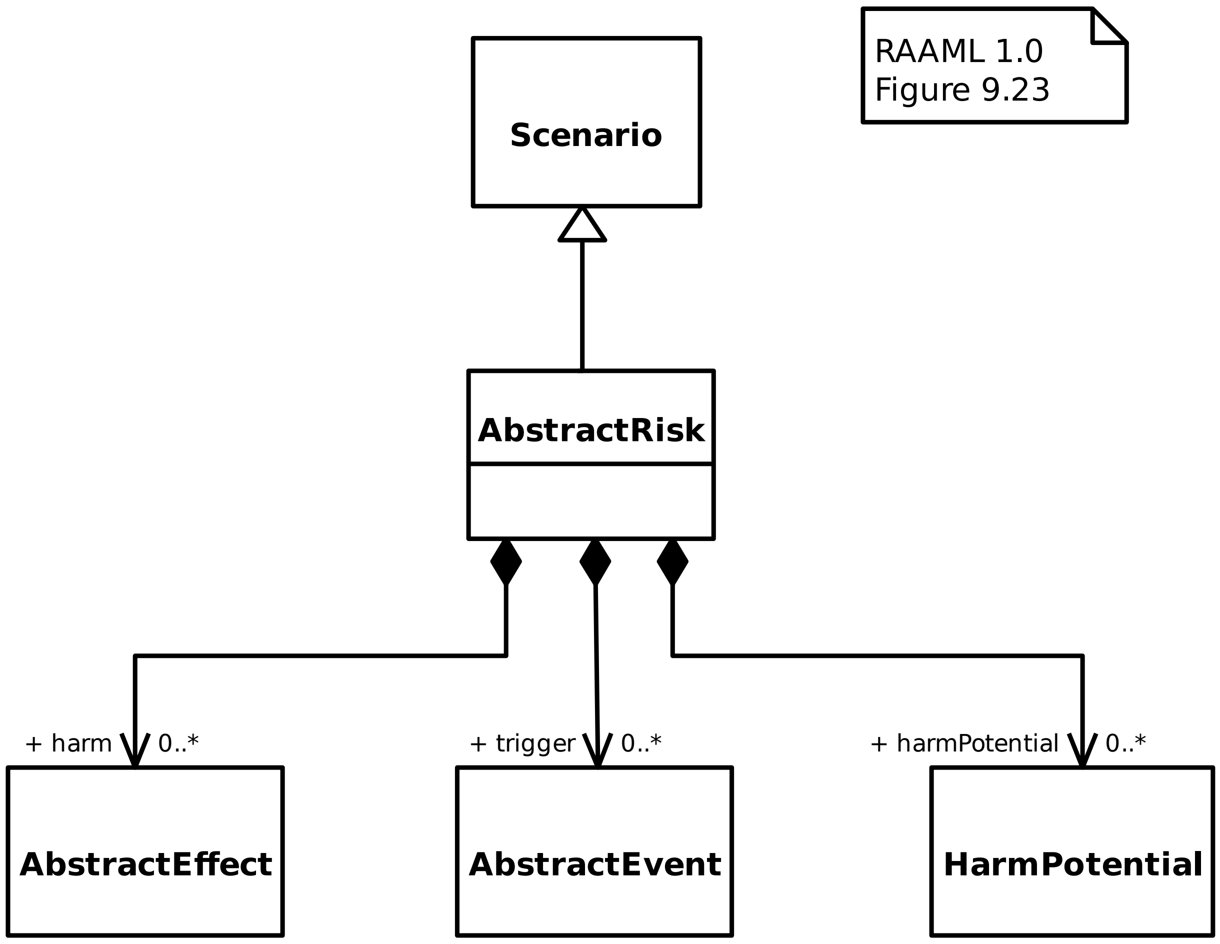
15.2.4 General Concepts Library/Abstract Event



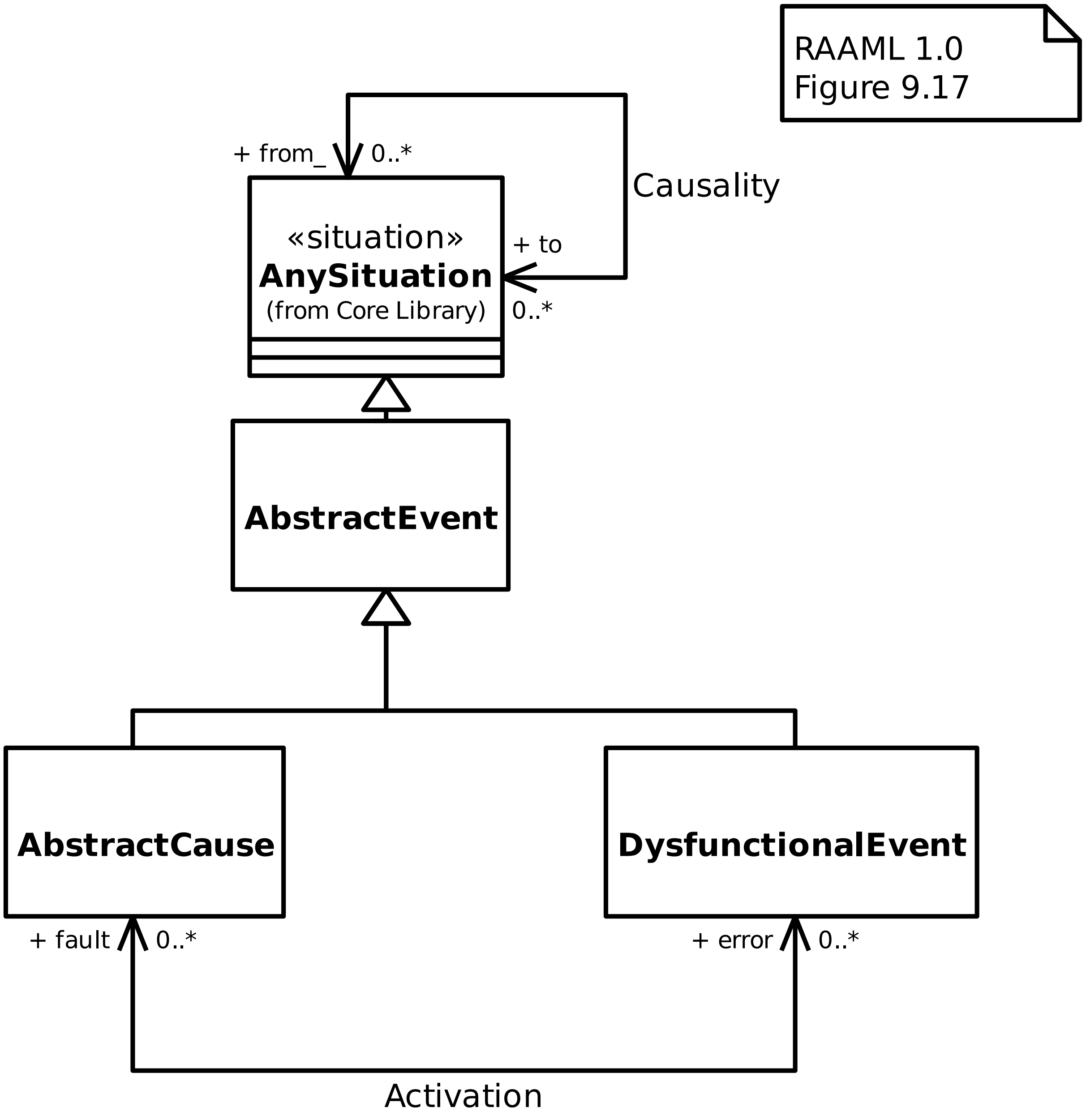
15.2.5 General Concepts Library/Abstract Failure Mode



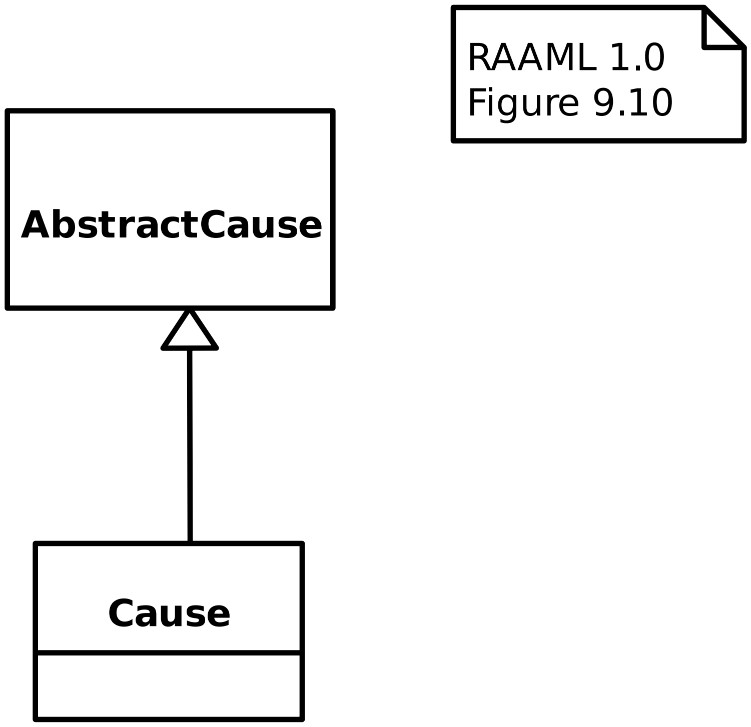
15.2.6 General Concepts Library/Abstract Risk



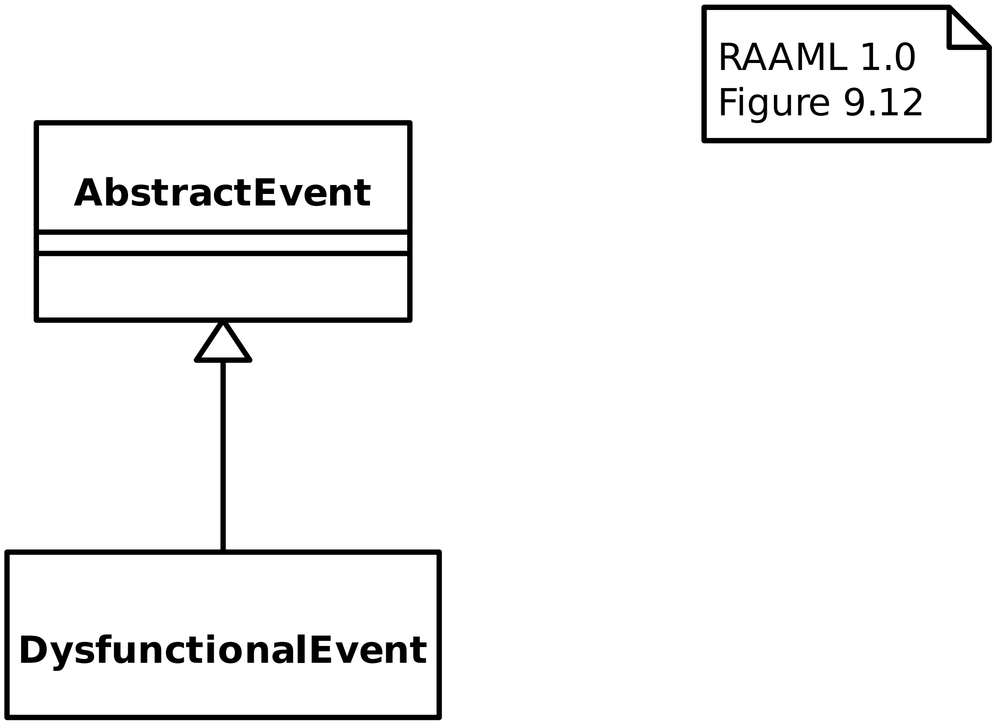
15.2.7 General Concepts Library/Activation



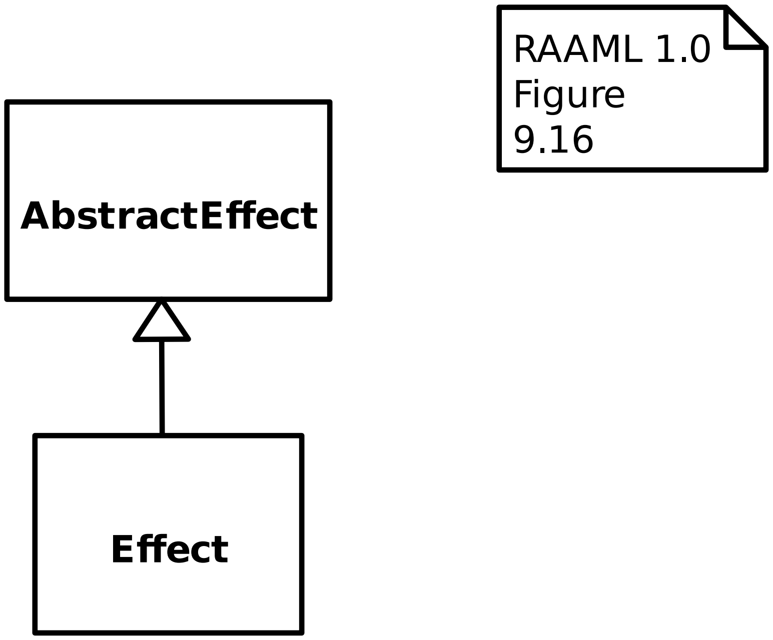
15.2.8 General Concepts Library/Cause



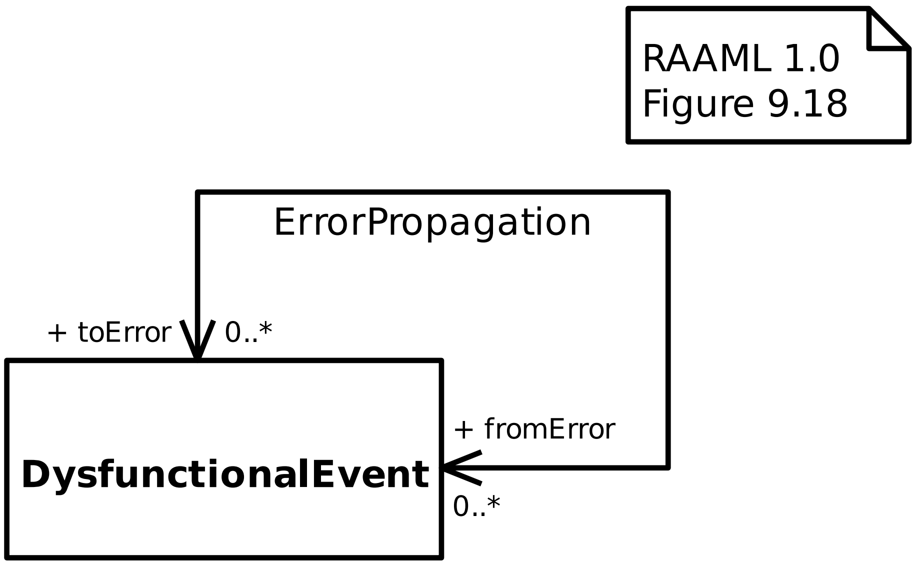
15.2.9 General Concepts Library/Dysfunctional Event



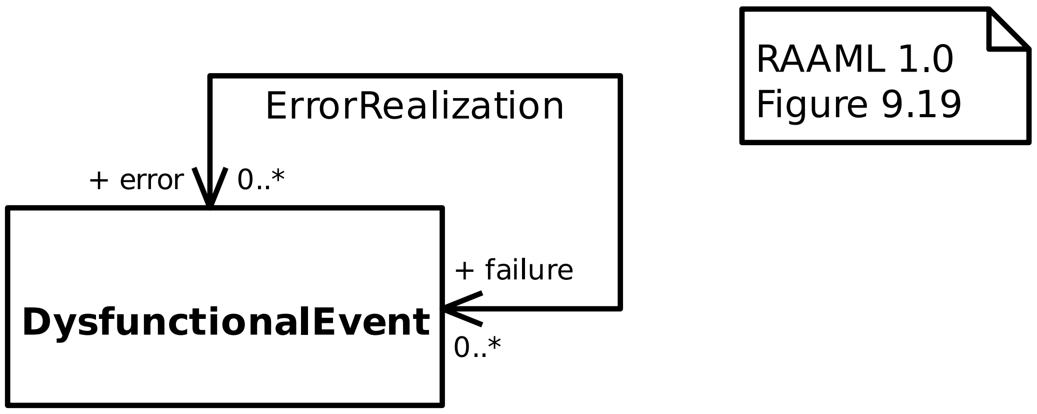
15.2.10 General Concepts Library/Effect



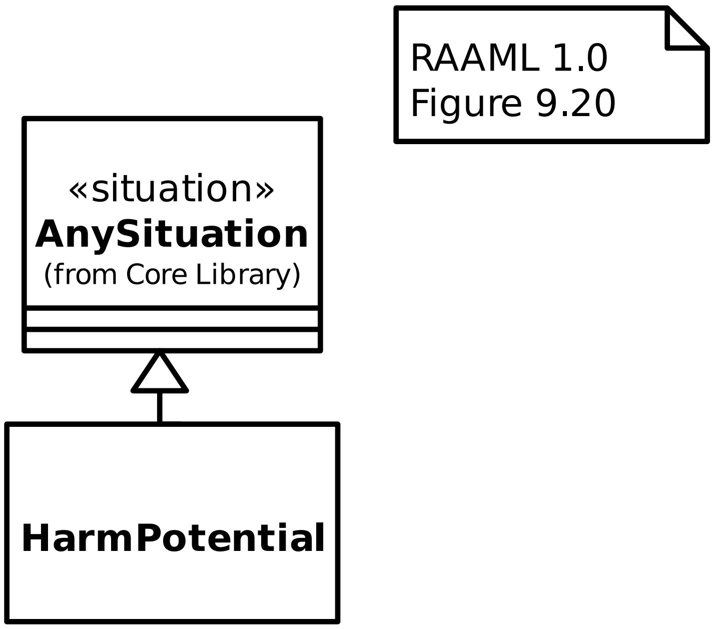
15.2.11 General Concepts Library/Error Propagation



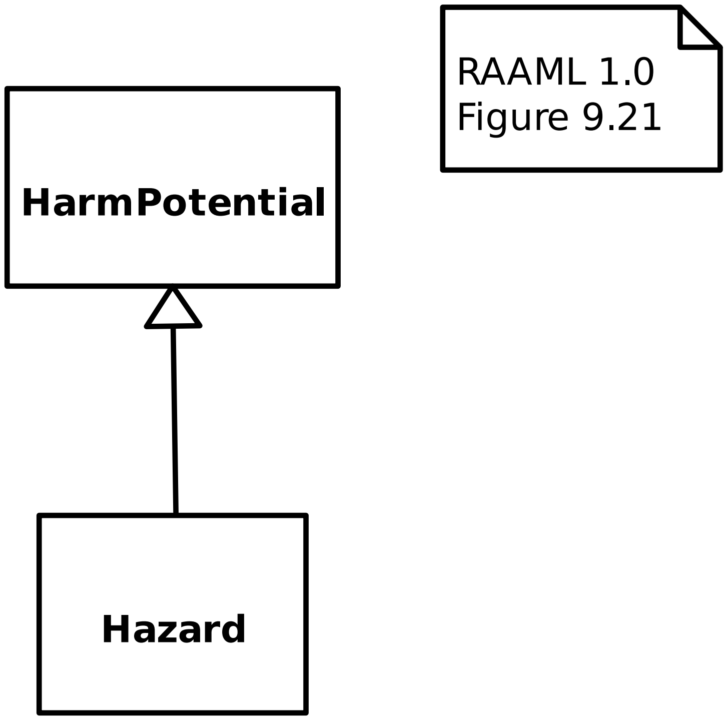
15.2.12 General Concepts Library/Error Realization



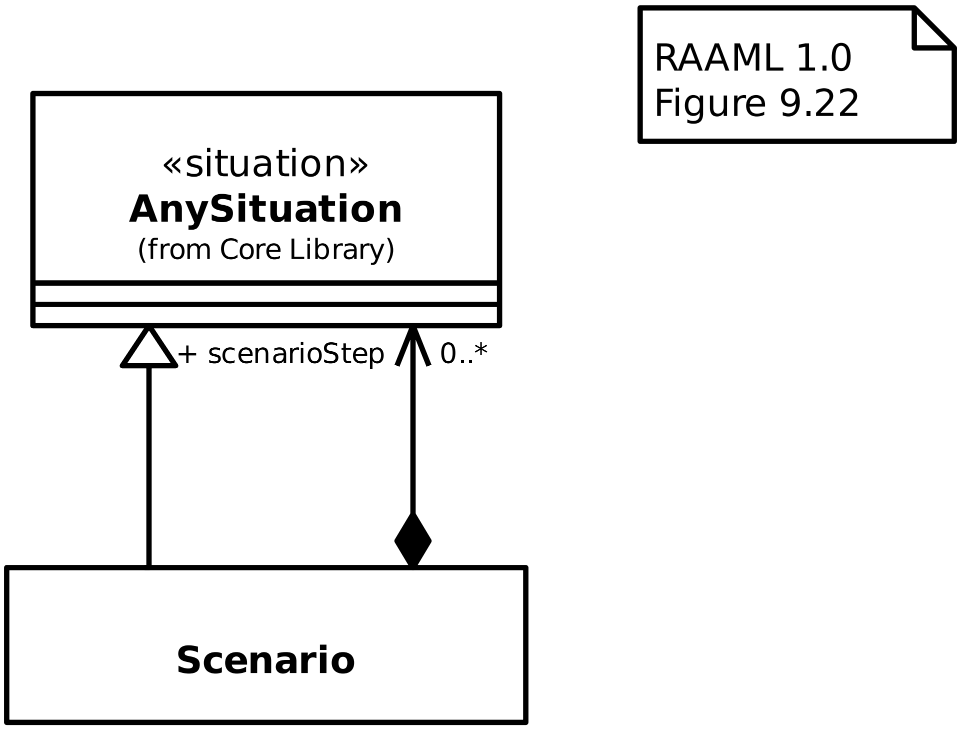
15.2.13 General Concepts Library/Harm Potential



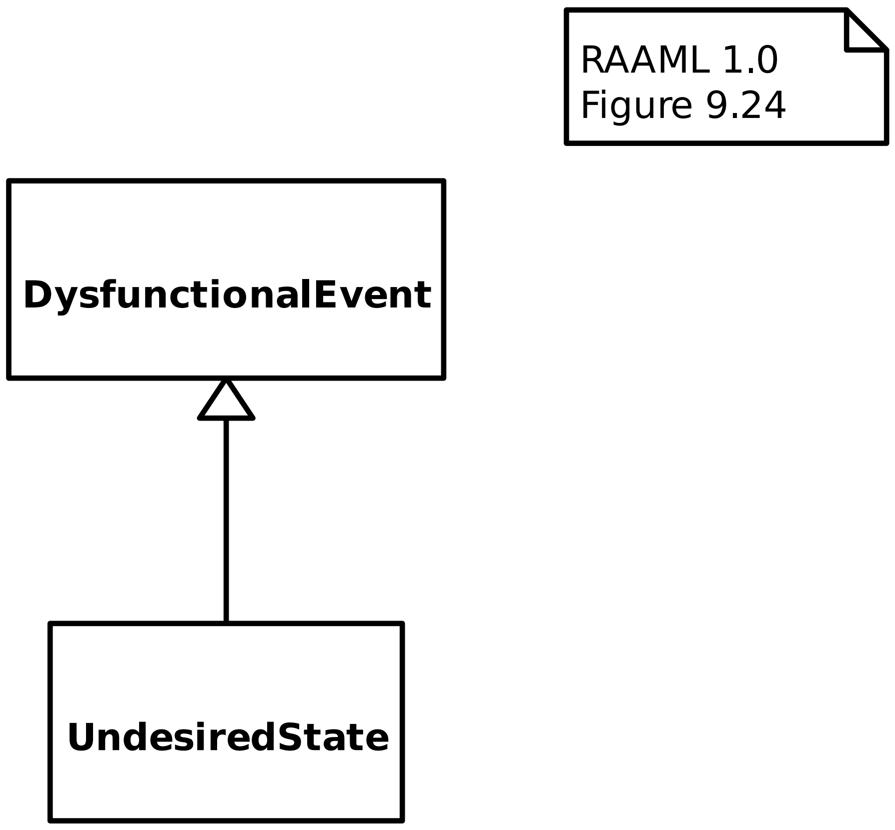
15.2.14 General Concepts Library/Hazard



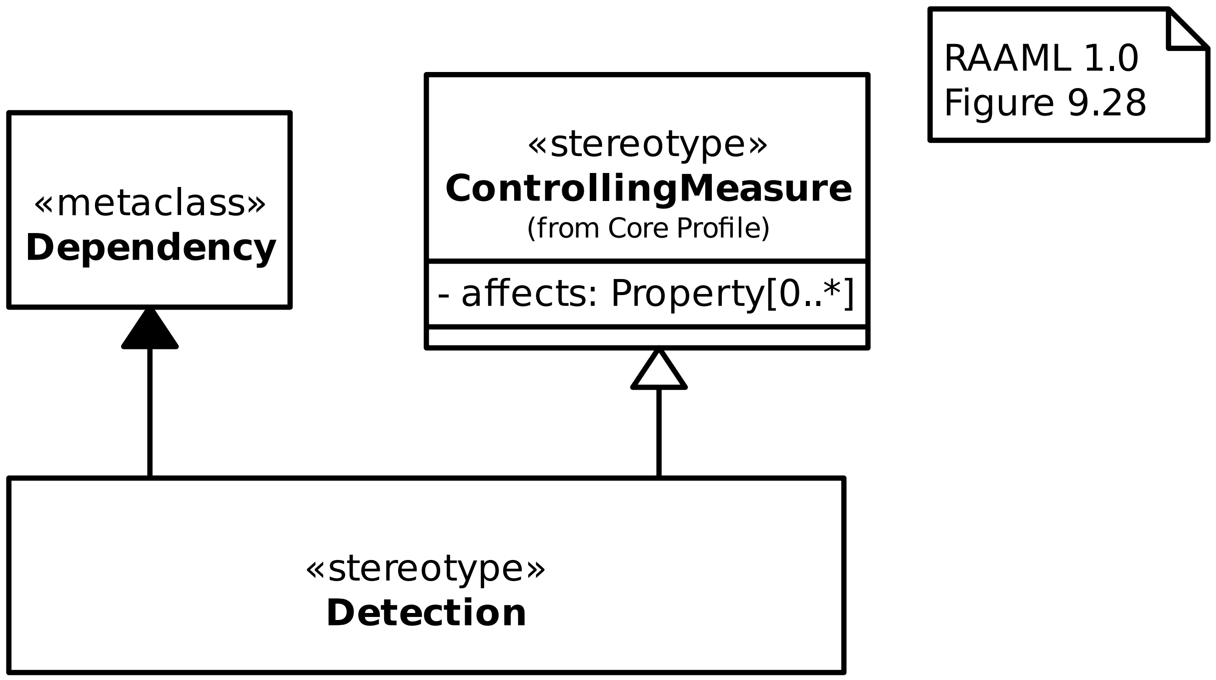
15.2.15 General Concepts Library/Scenario



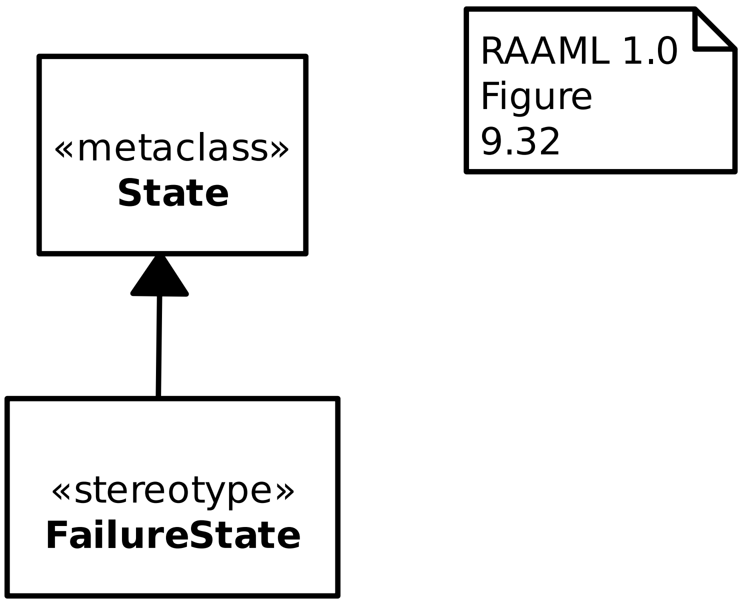
15.2.16 General Concepts Library/Undesired State



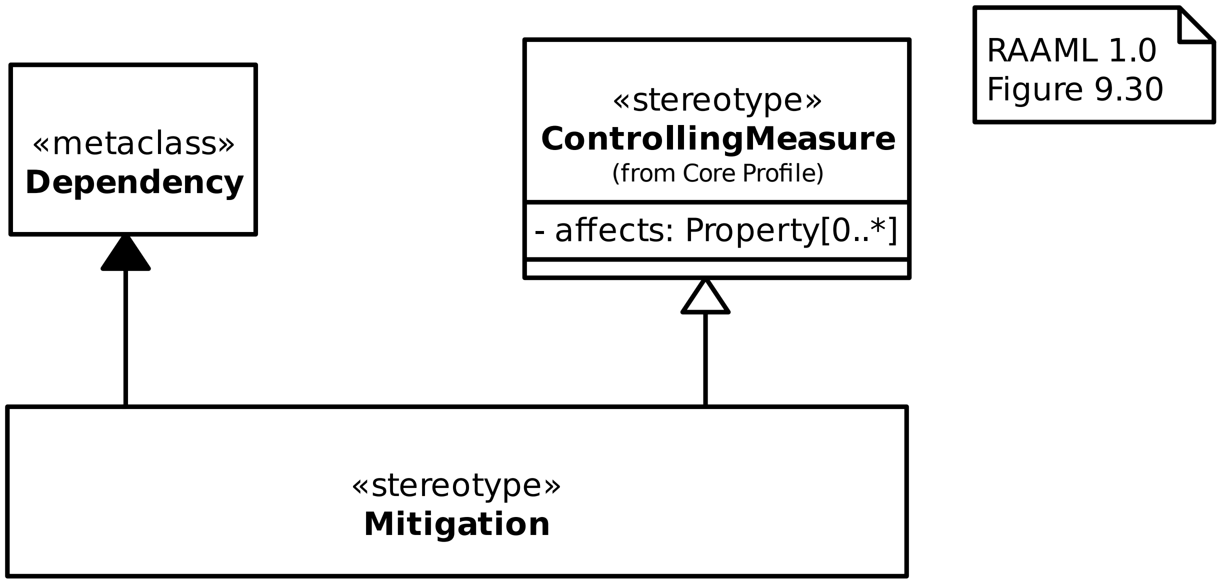
15.2.17 General Concepts Profile/Detection



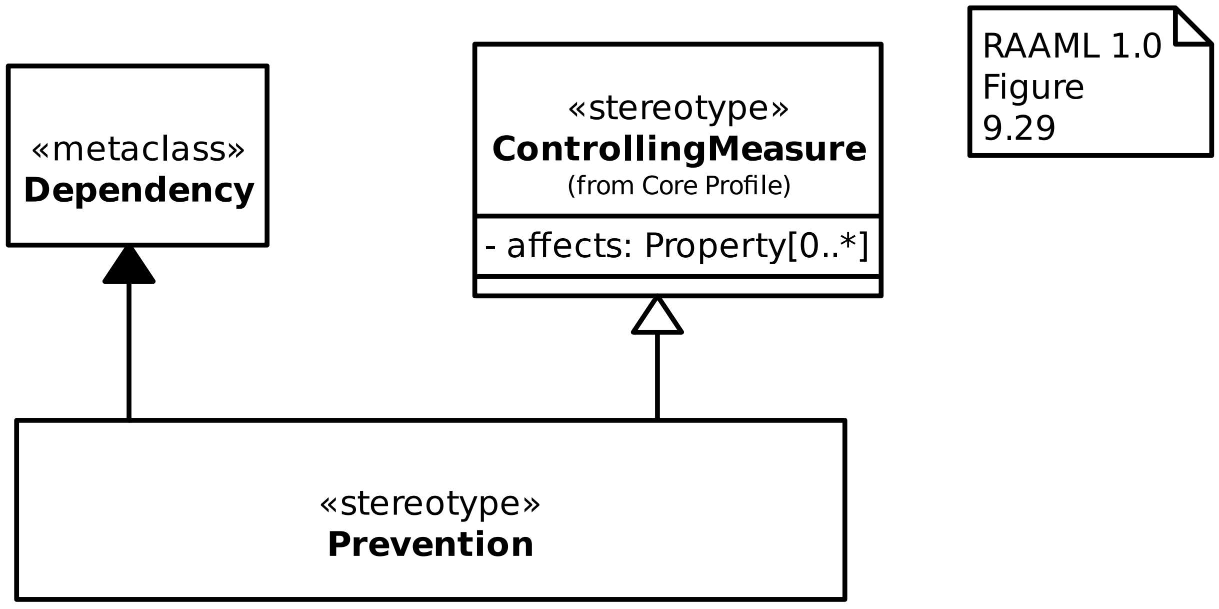
15.2.18 General Concepts Profile/Failure State



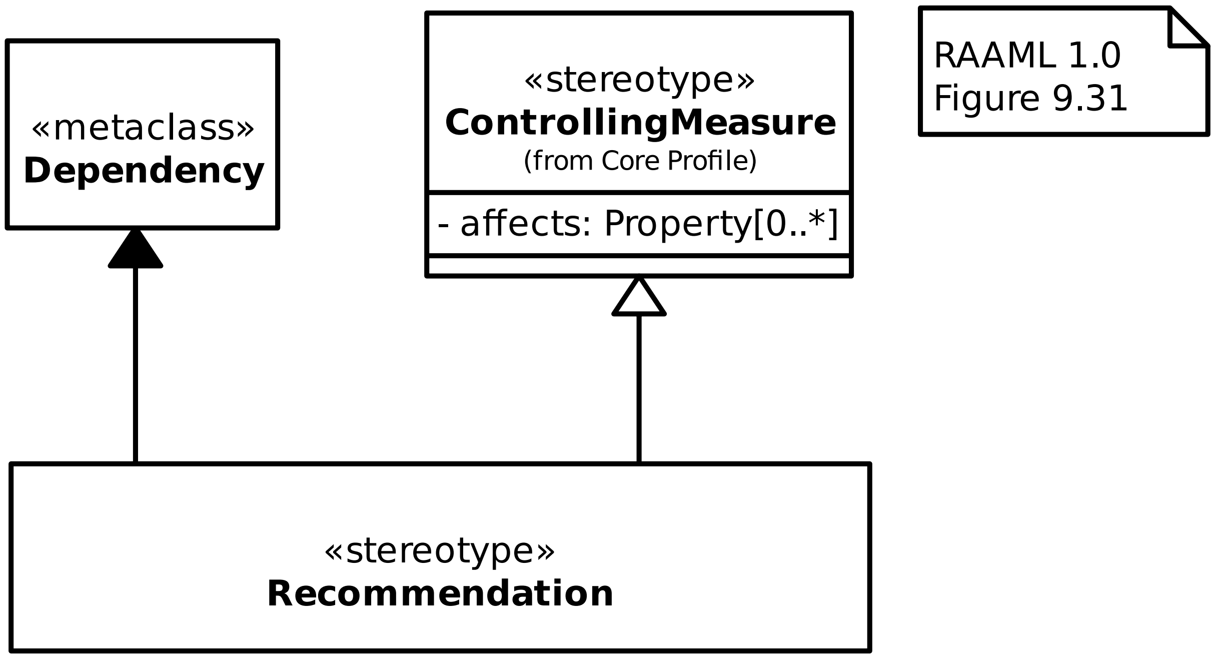
15.2.19 General Concepts Profile/Mitigation



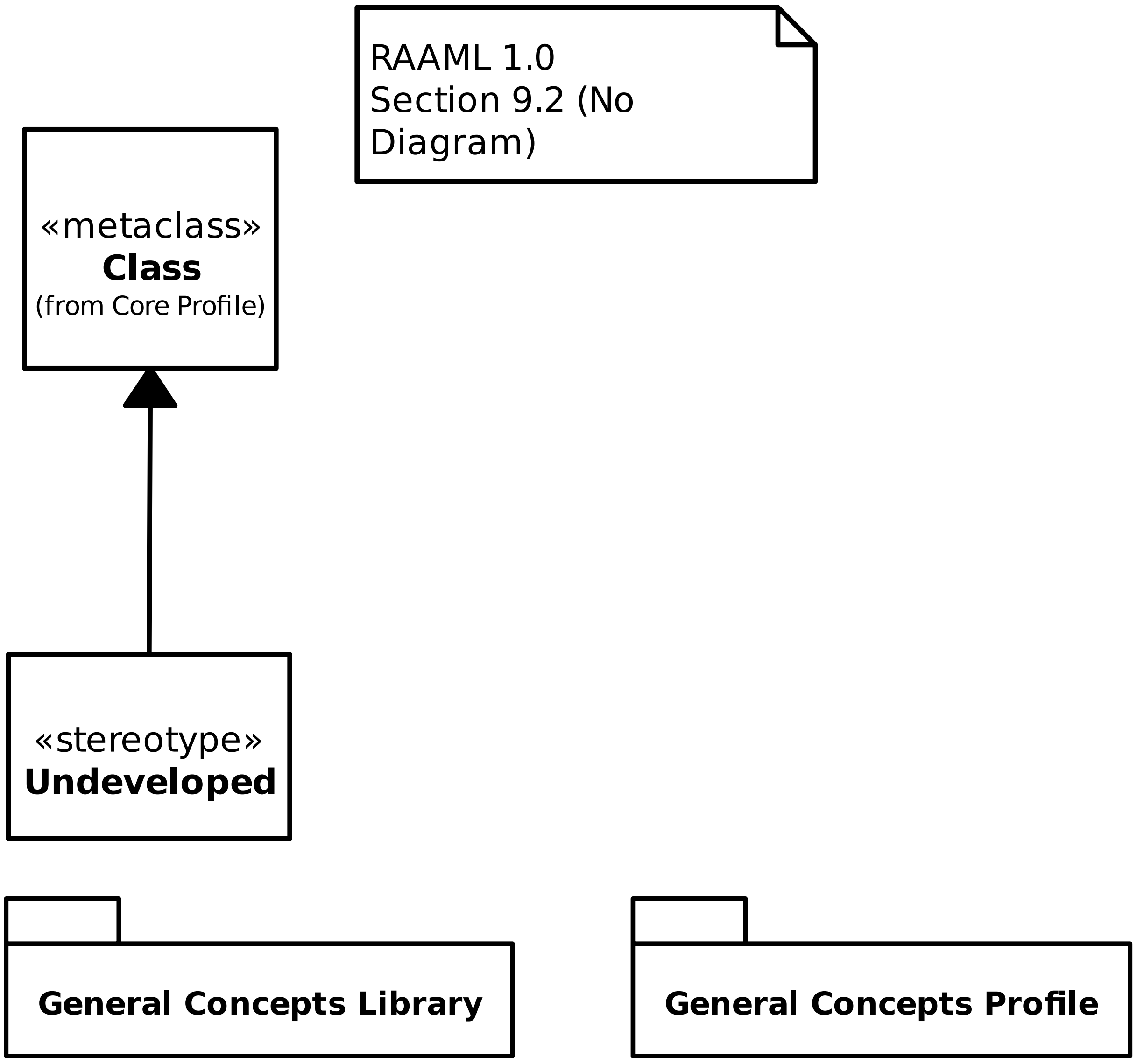
15.2.20 General Concepts Profile/Prevention



15.2.21 General Concepts Profile/Recommendation

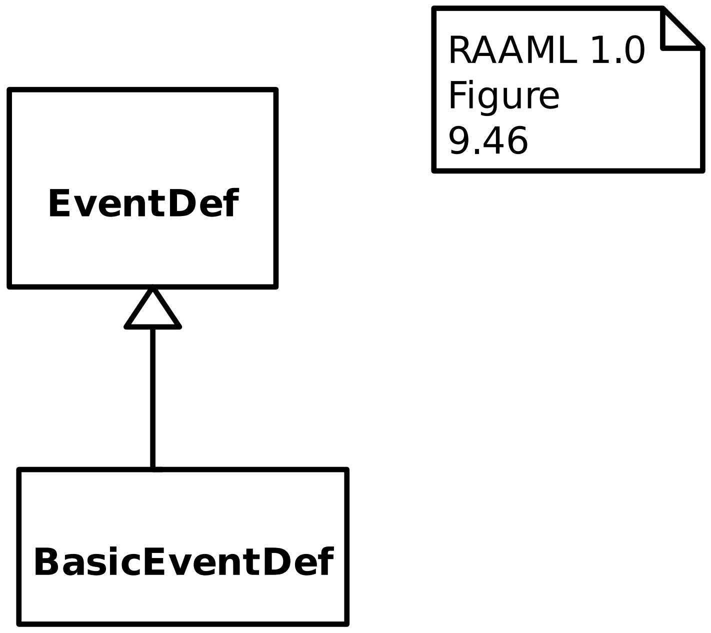


15.2.22 General Concepts Profile/Undeveloped

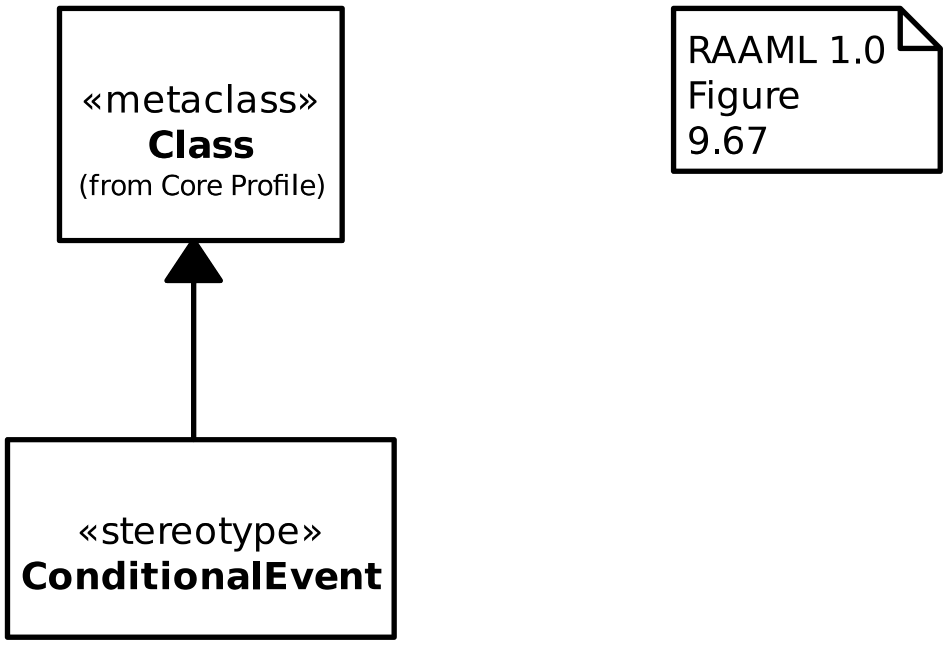


15.3 Methods

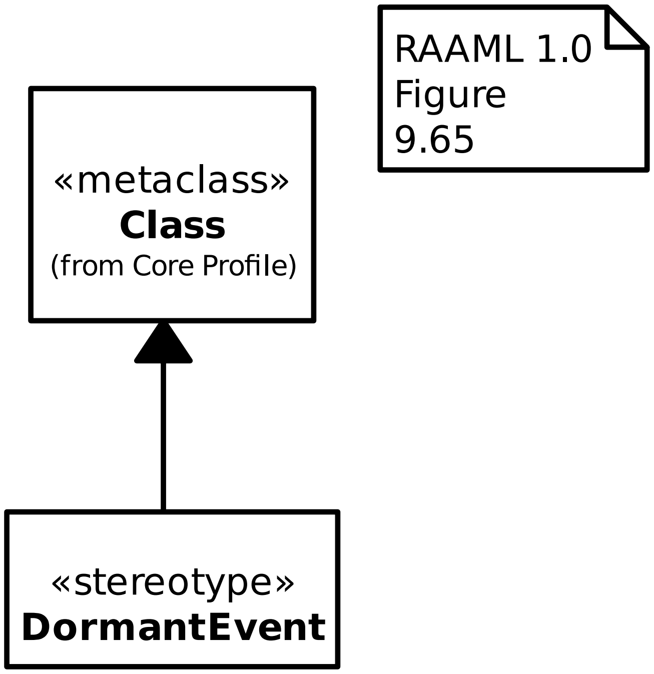
15.3.1 FTA/FTA Library/Events/Basic Event



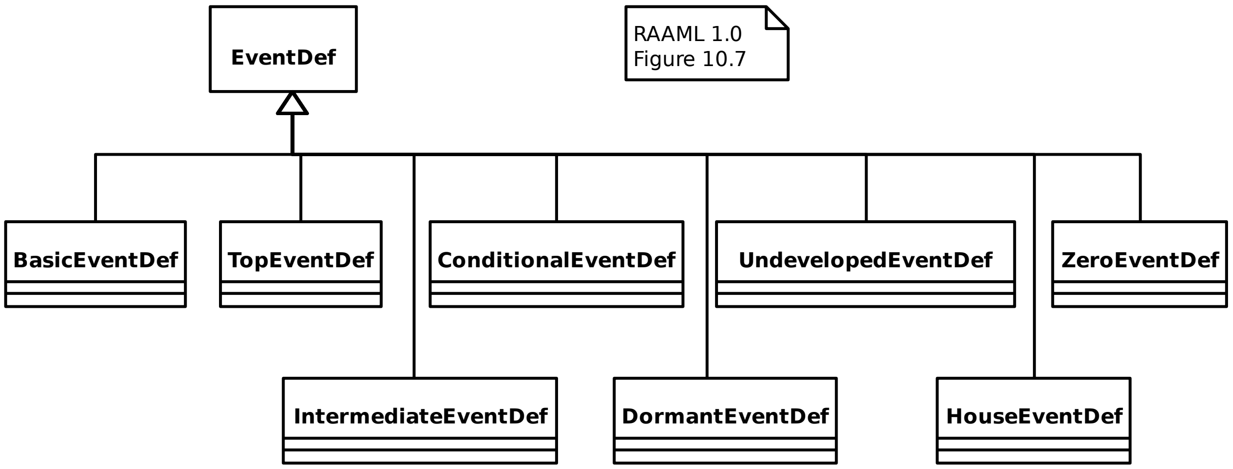
15.3.2 FTA/FTA Library/Events/Conditional Event



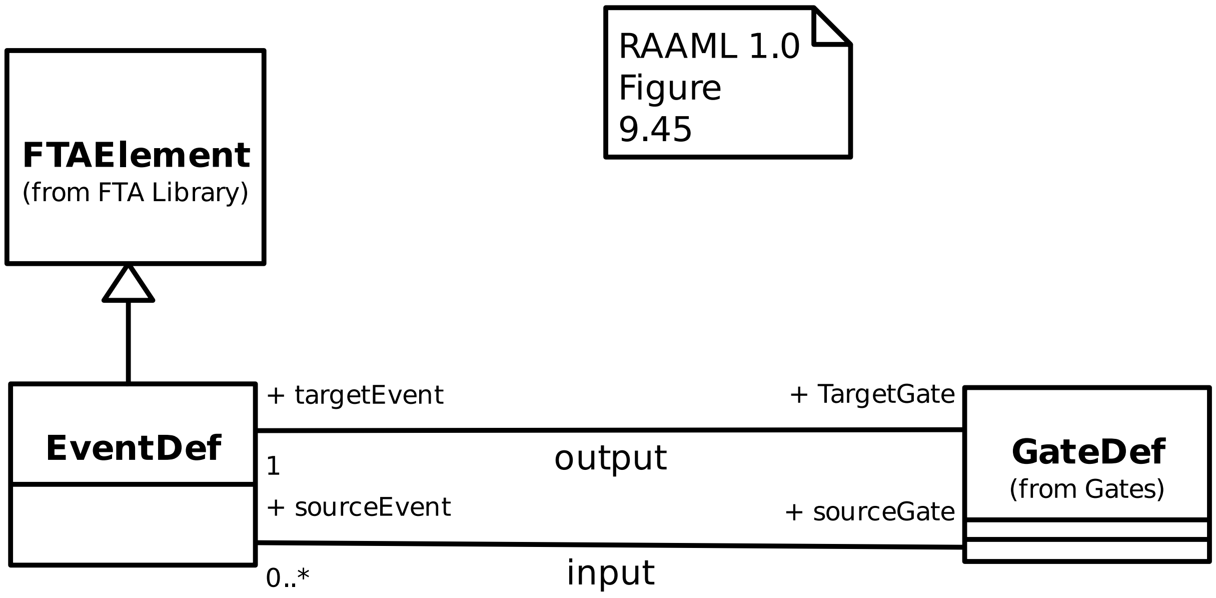
15.3.3 FTA/FTA Library/Events/Dormant Event



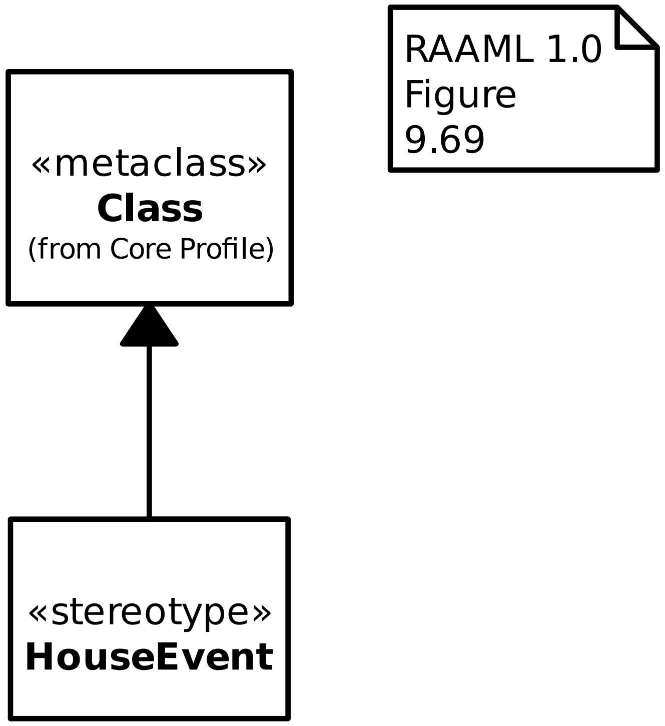
15.3.4 FTA/FTA Library/Events/Events



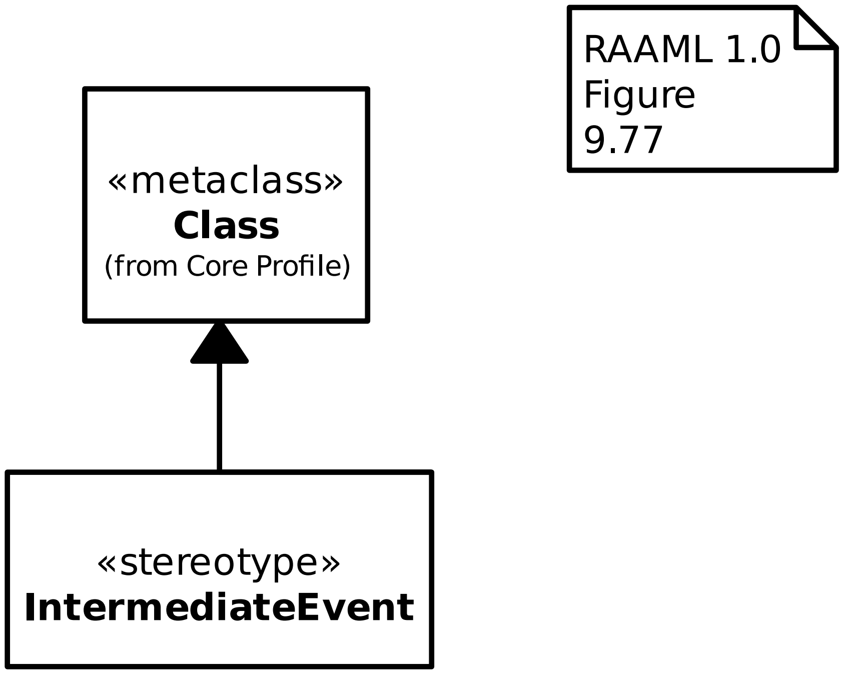
15.3.5 FTA/FTA Library/Events/Event



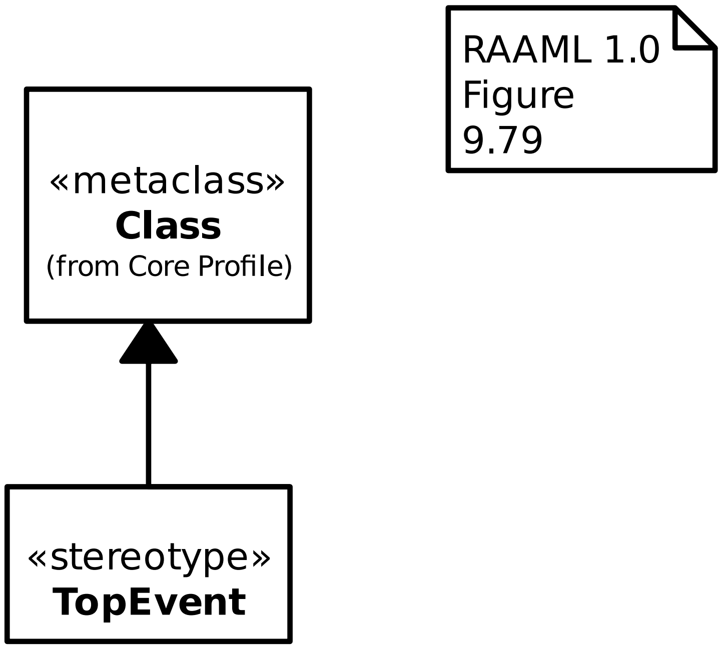
15.3.6 FTA/FTA Library/Events/House Event



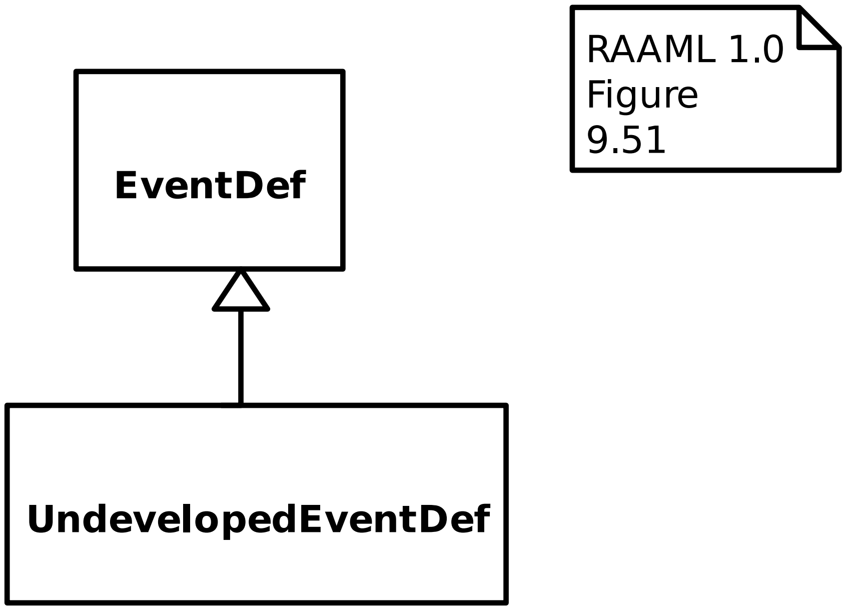
15.3.7 FTA/FTA Library/Events/Intermediate Event



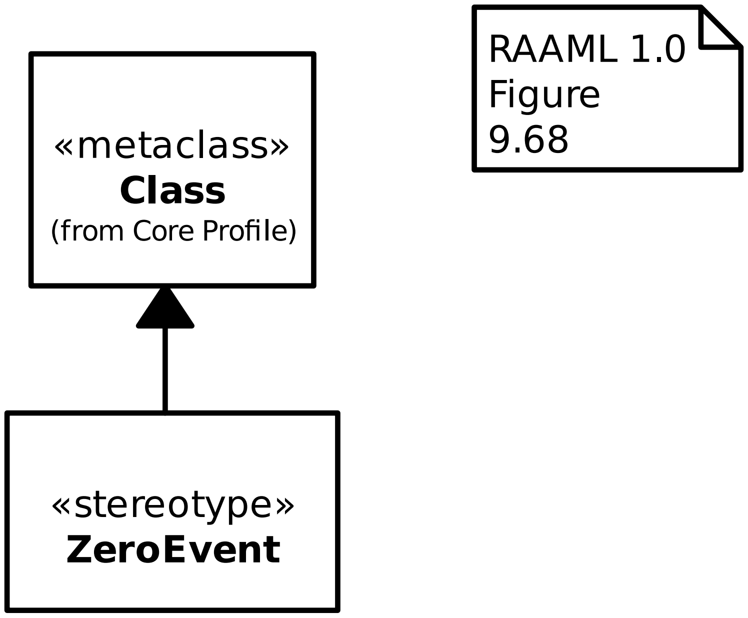
15.3.8 FTA/FTA Library/Events/Top Event



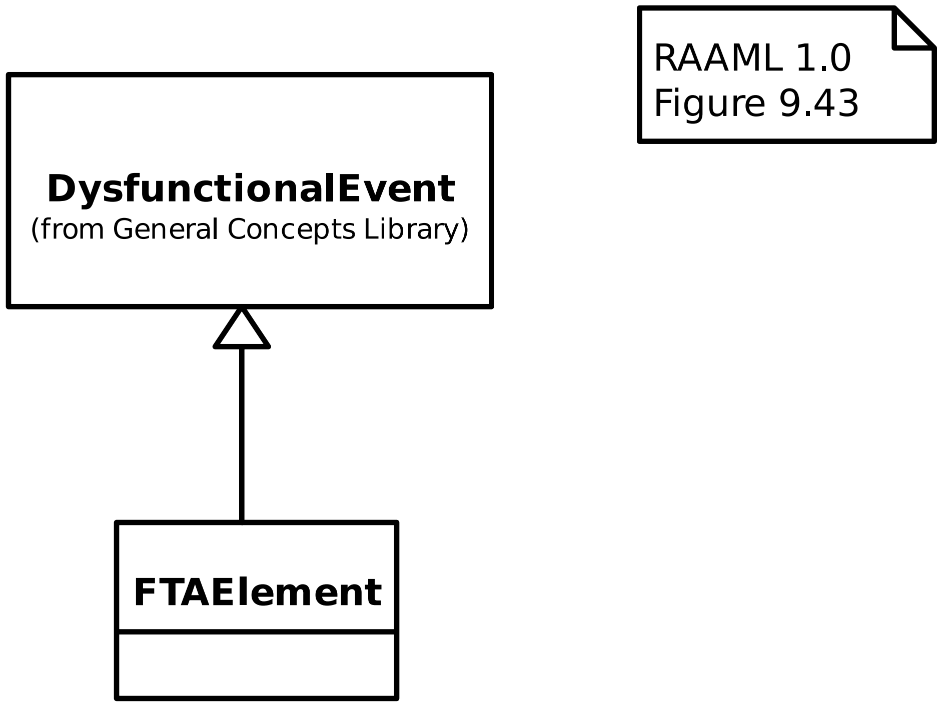
15.3.9 FTA/FTA Library/Events/Undeveloped Event



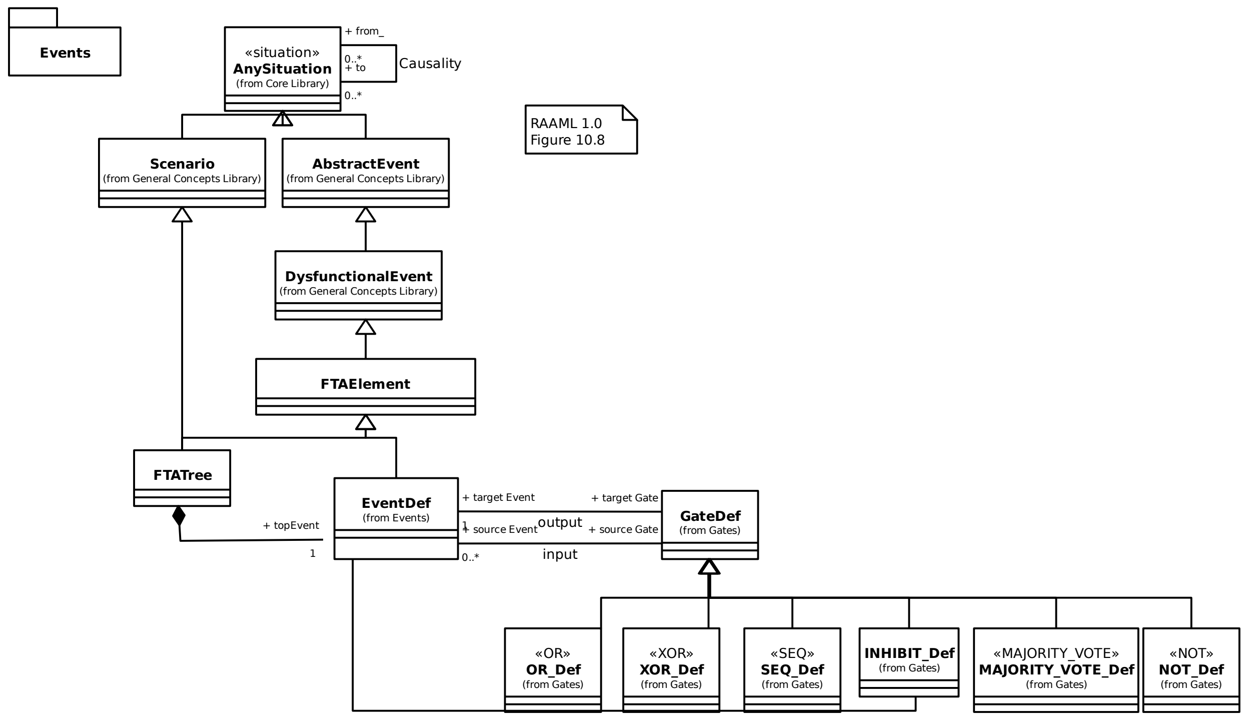
15.3.10 FTA/FTA Library/Events/Zero Event



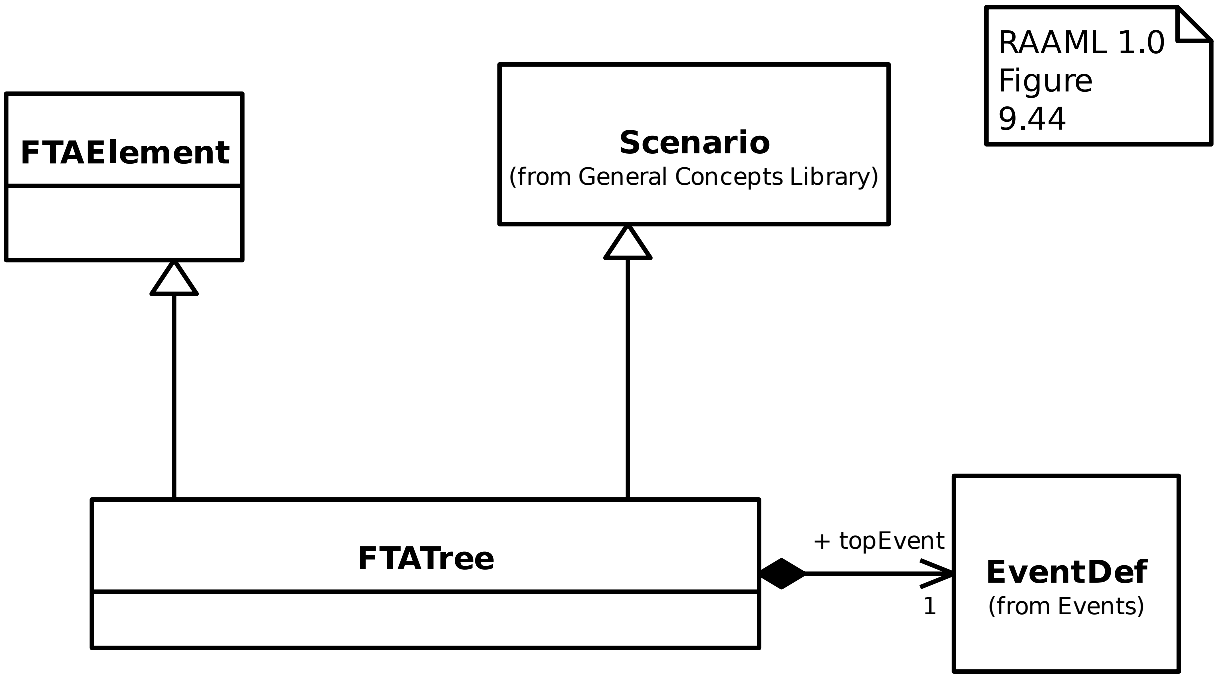
15.3.11 FTA/FTA Library/FTA Element



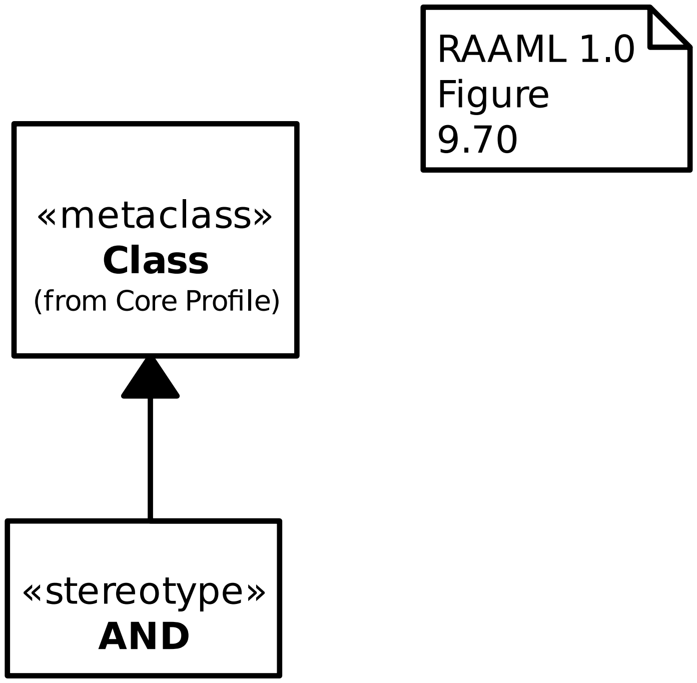
15.3.12 FTA/FTA Library/FTA Library



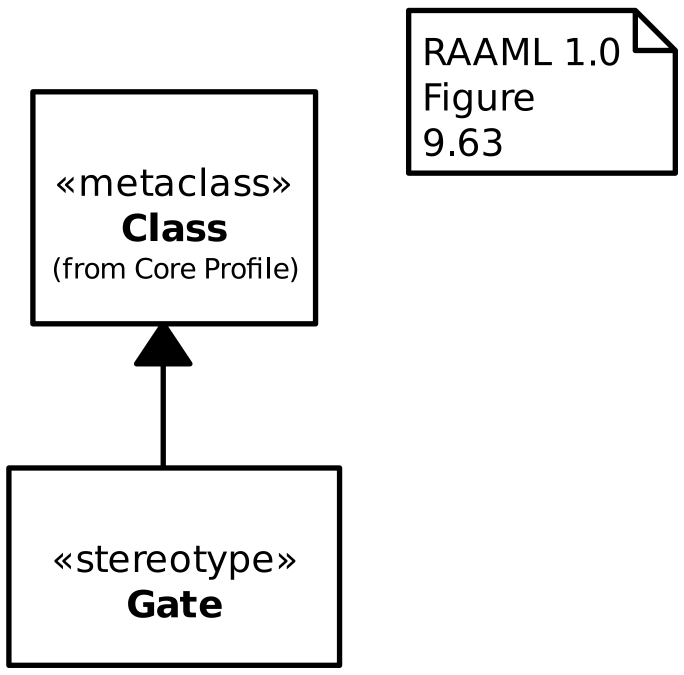
15.3.13 FTA/FTA Library/FTA Tree



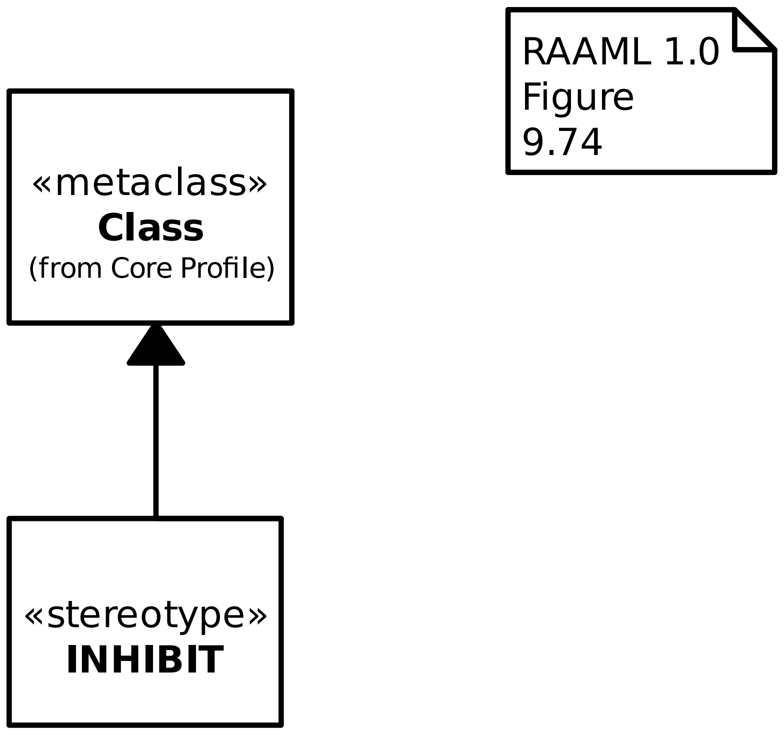
15.3.14 FTA/FTA Library/Gates/AND



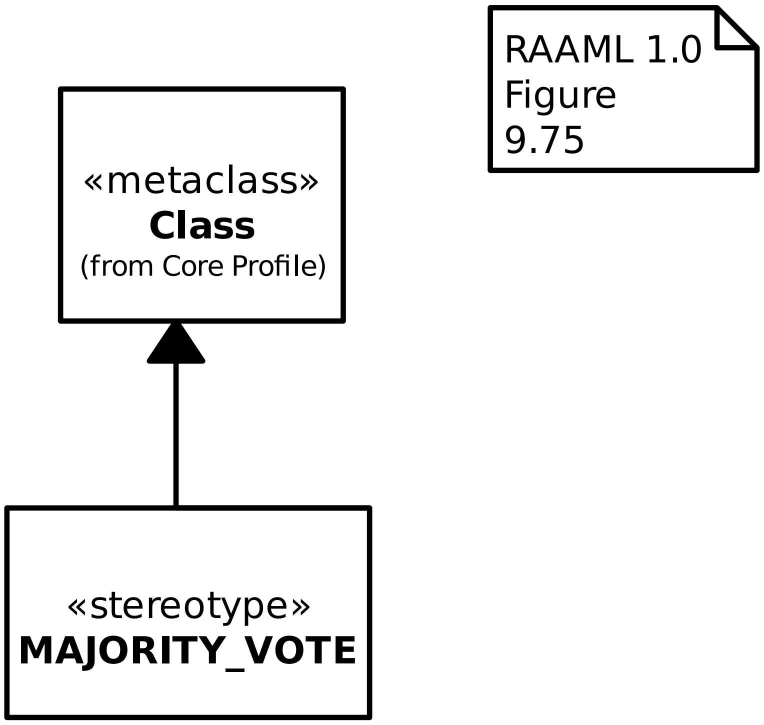
15.3.15 FTA/FTA Library/Gates/Gate



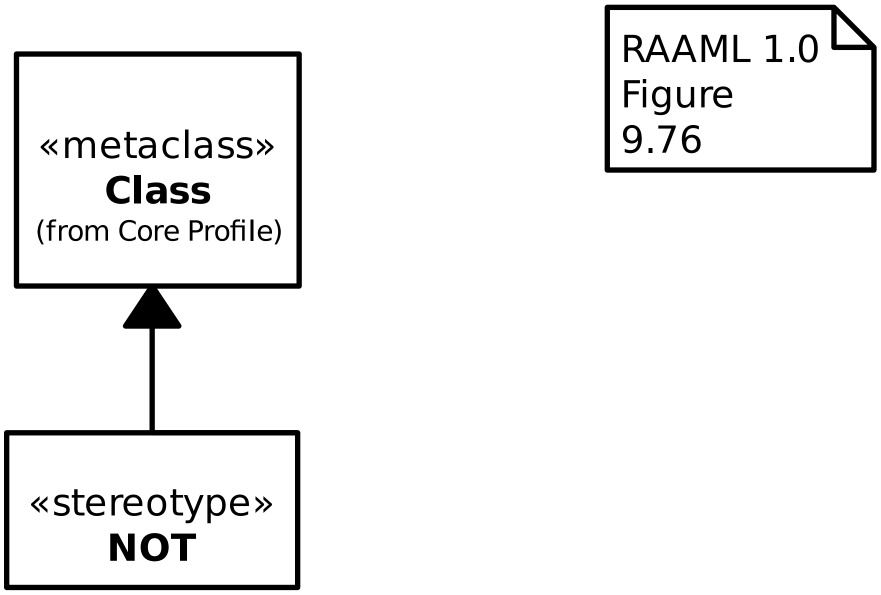
15.3.16 FTA/FTA Library/Gates/INHIBIT



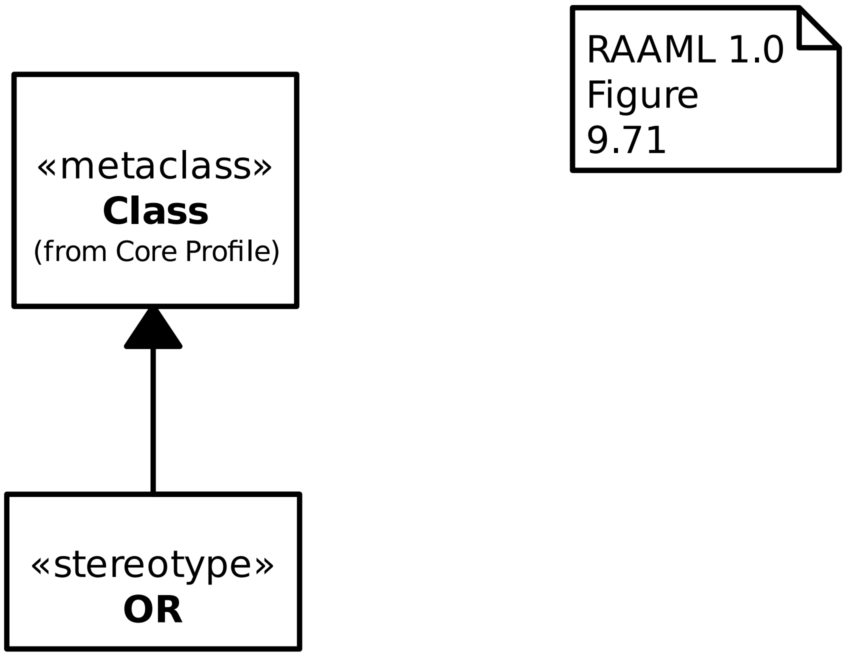
15.3.17 FTA/FTA Library/Gates/MAJORITY_VOTE



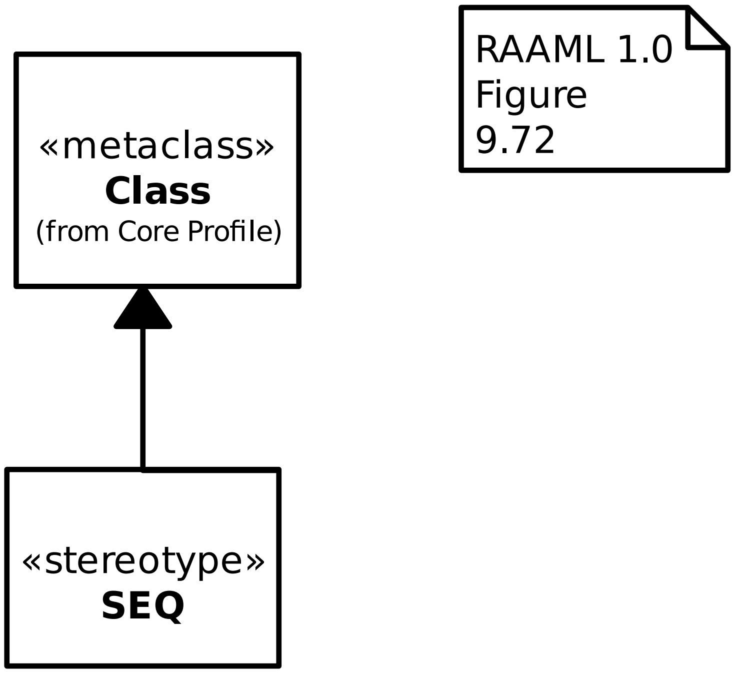
15.3.18 FTA/FTA Library/Gates/NOT



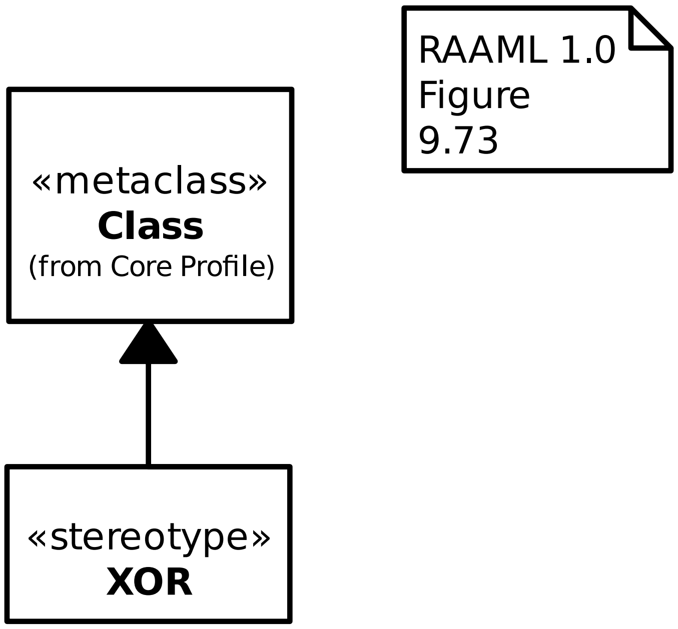
15.3.19 FTA/FTA Library/Gates/OR



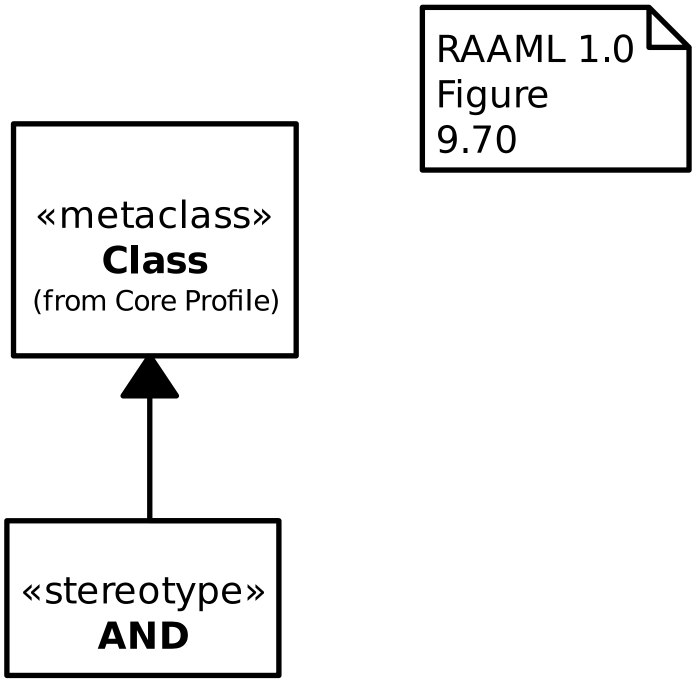
15.3.20 FTA/FTA Library/Gates/SEQ



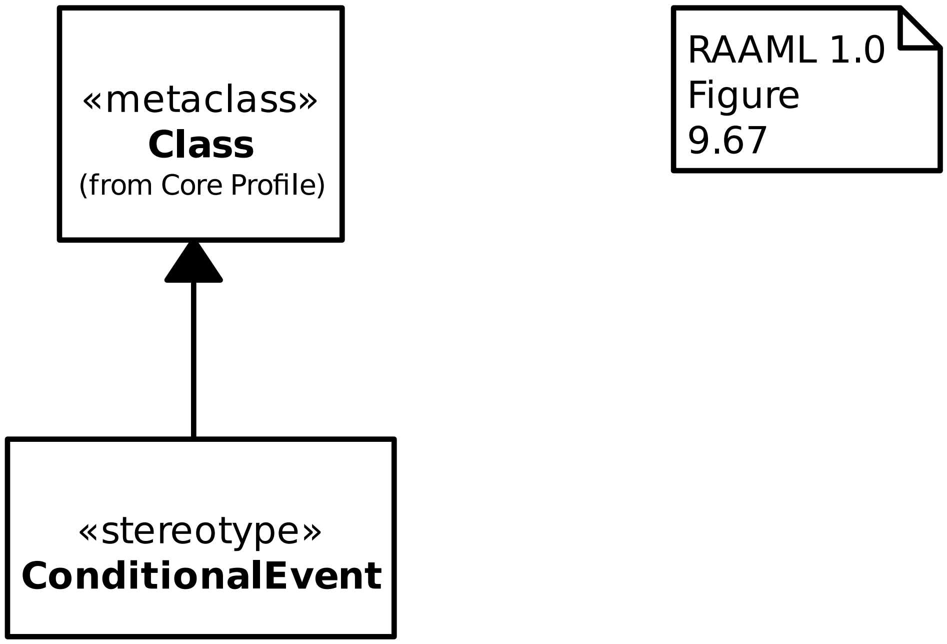
15.3.21 FTA/FTA Library/Gates/XOR



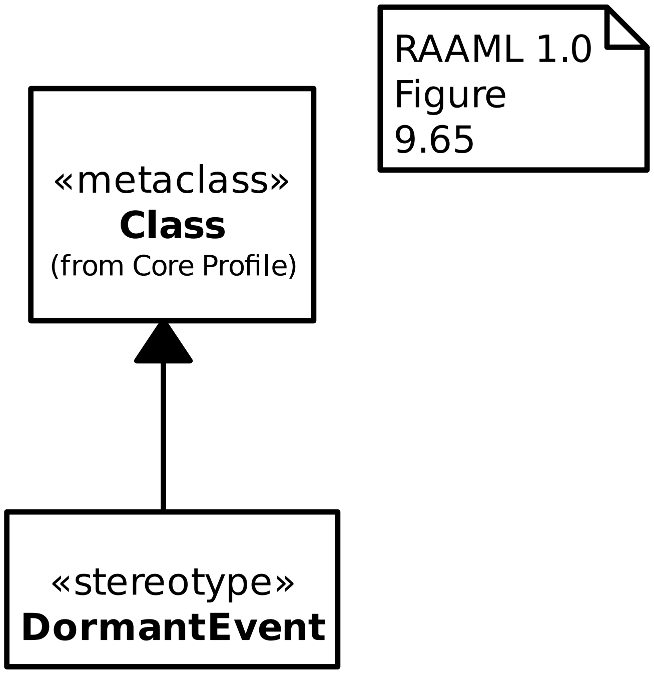
15.3.22 FTA/FTA Profile/AND



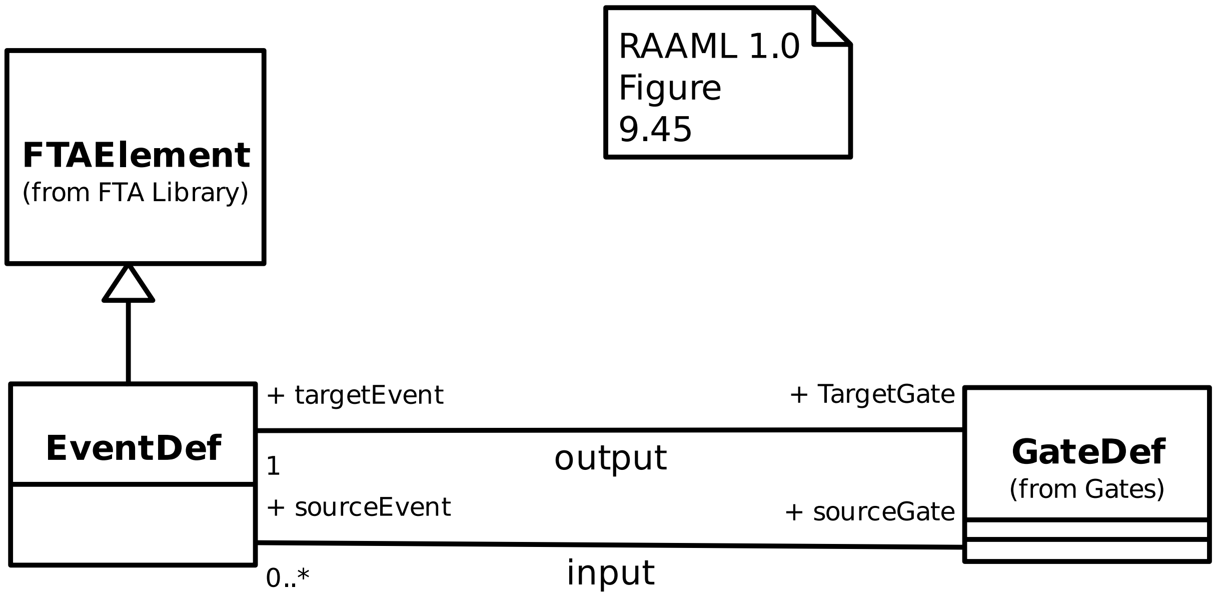
15.3.23 FTA/FTA Profile/Conditional Event



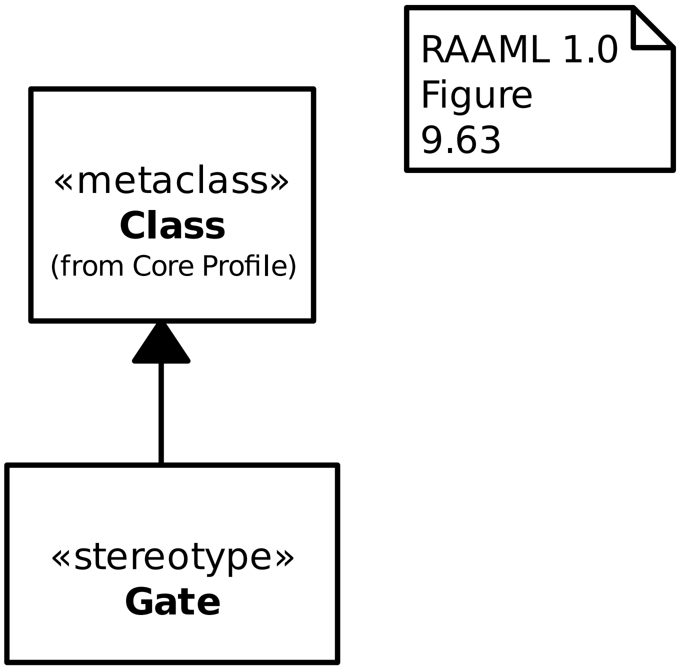
15.3.24 FTA/FTA Profile/Dormant Event



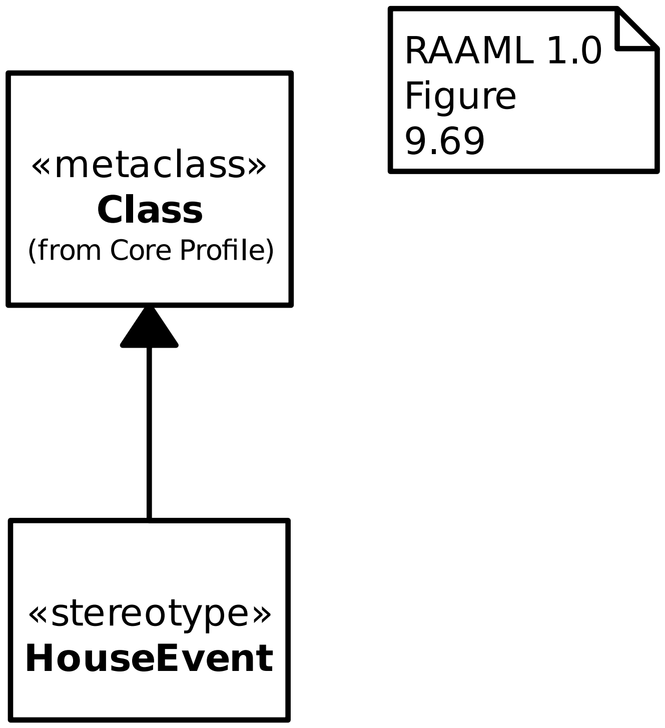
15.3.25 FTA/FTA Profile/Event



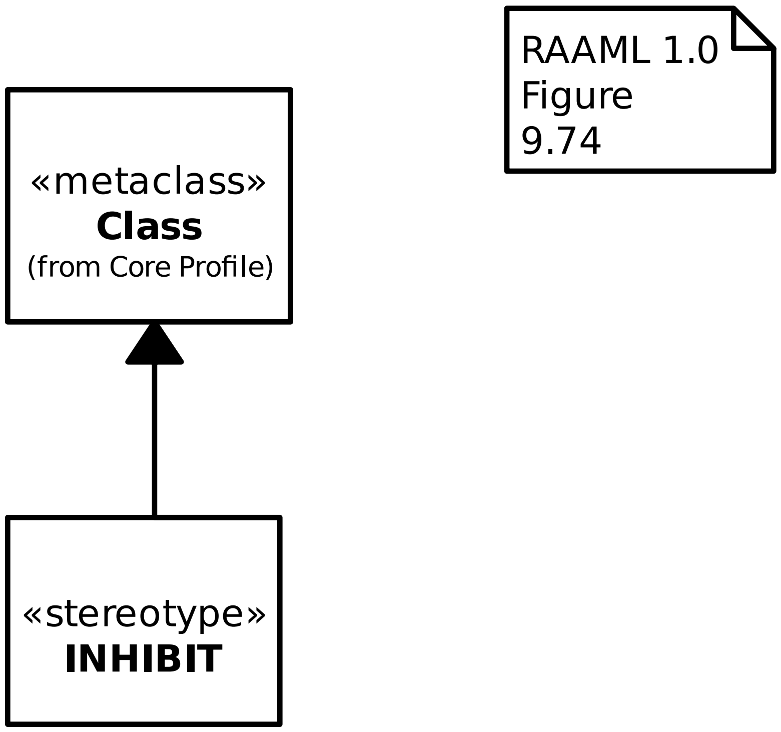
15.3.26 FTA/FTA Profile/Gate



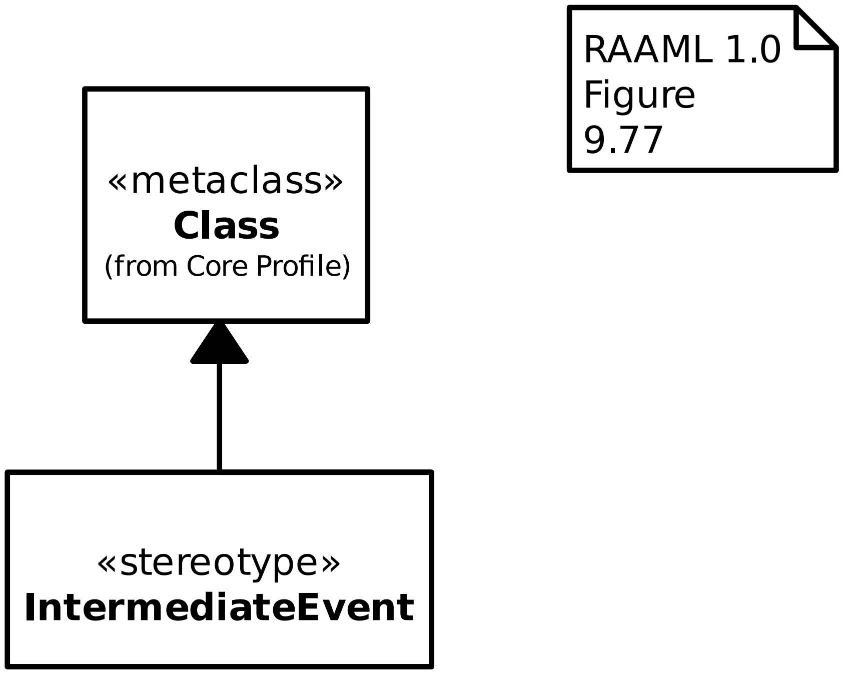
15.3.27 FTA/FTA Profile/House Event



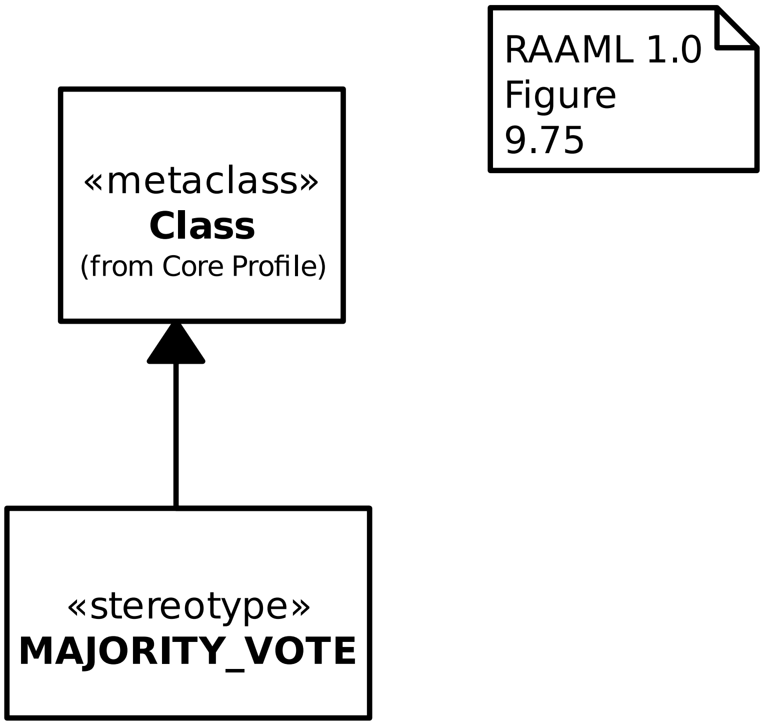
15.3.28 FTA/FTA Profile/INHIBIT



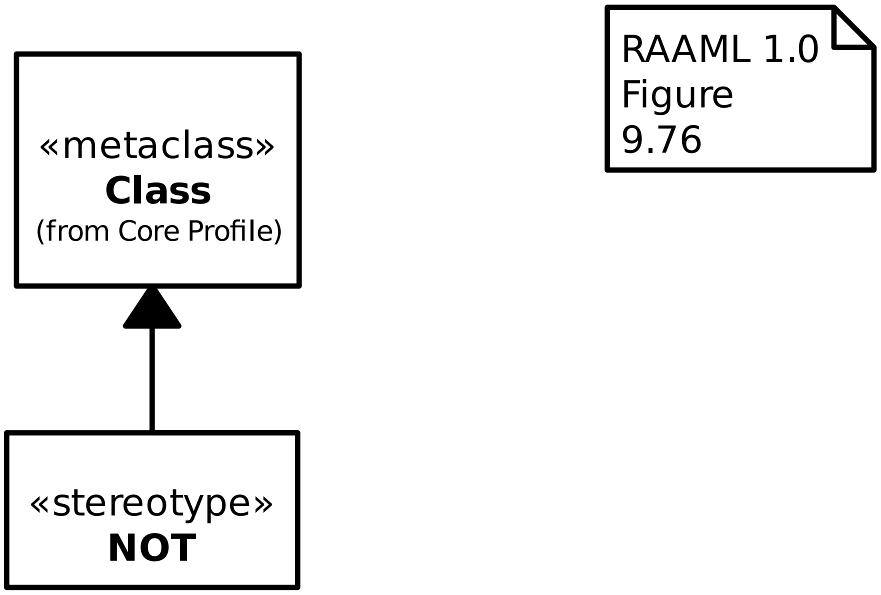
15.3.29 FTA/FTA Profile/Intermediate Event



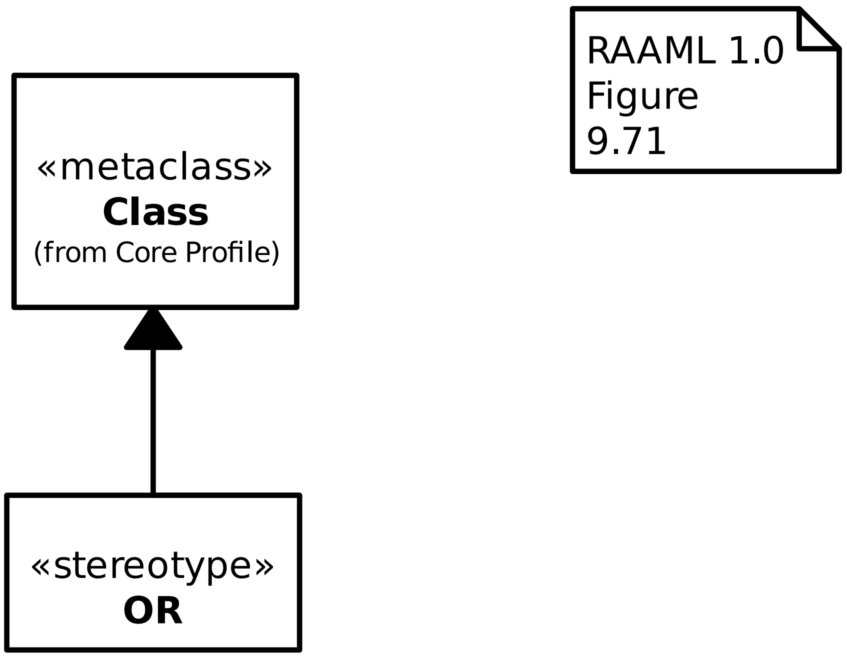
15.3.30 FTA/FTA Profile/MAJORITY_VOTE



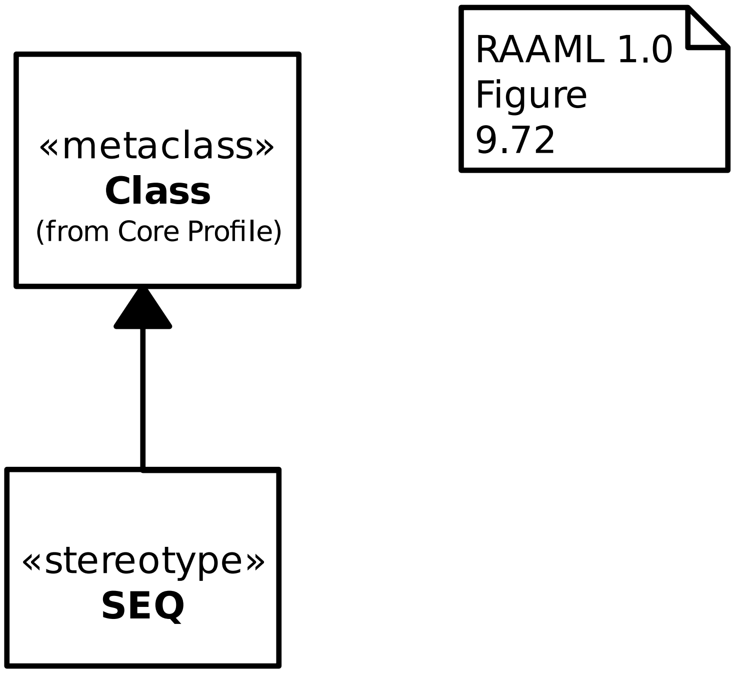
15.3.31 FTA/FTA Profile/NOT



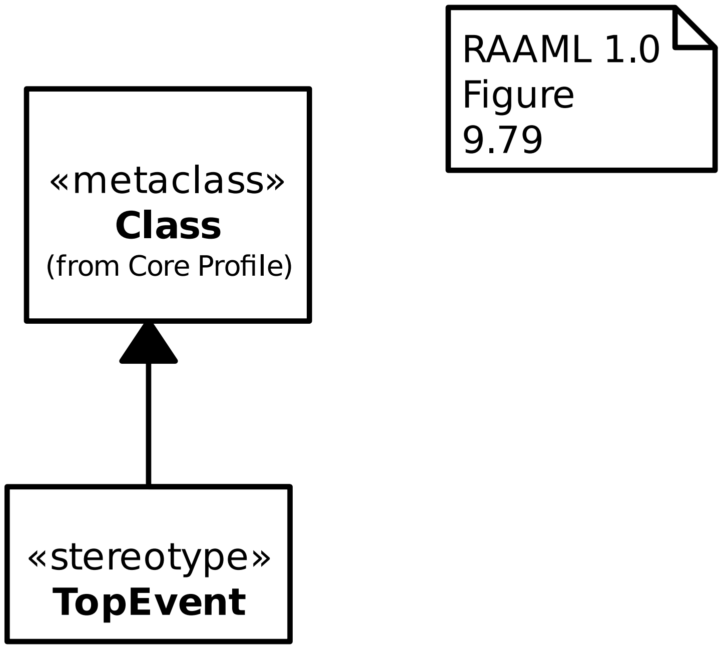
15.3.32 FTA/FTA Profile/OR



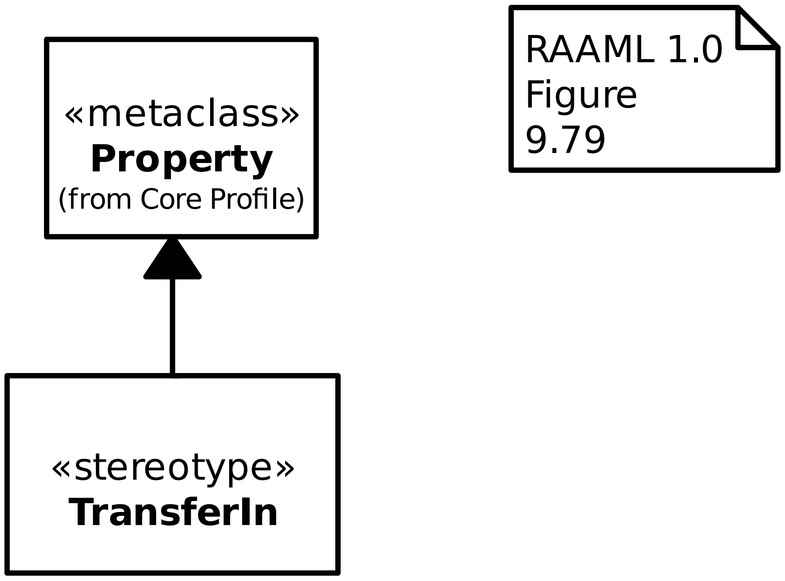
15.3.33 FTA/FTA Profile/SEQ



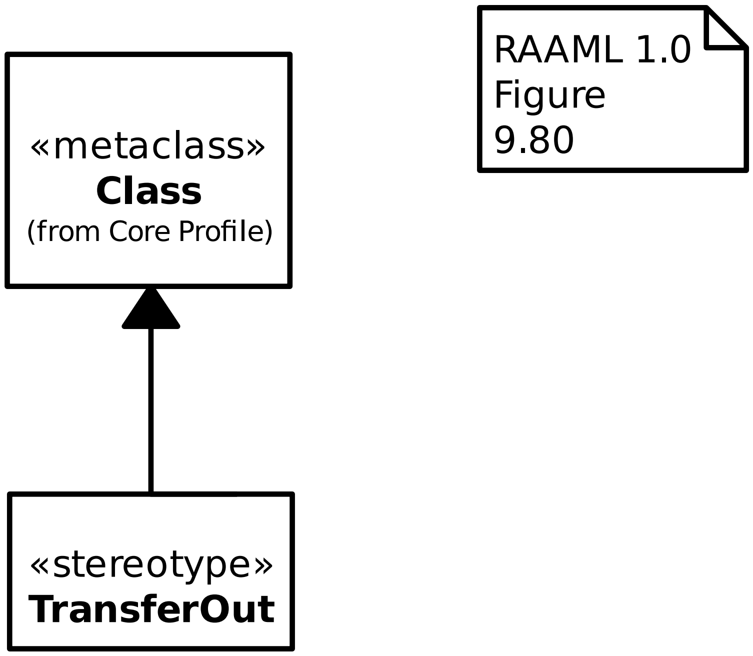
15.3.34 FTA/FTA Profile/Top Event



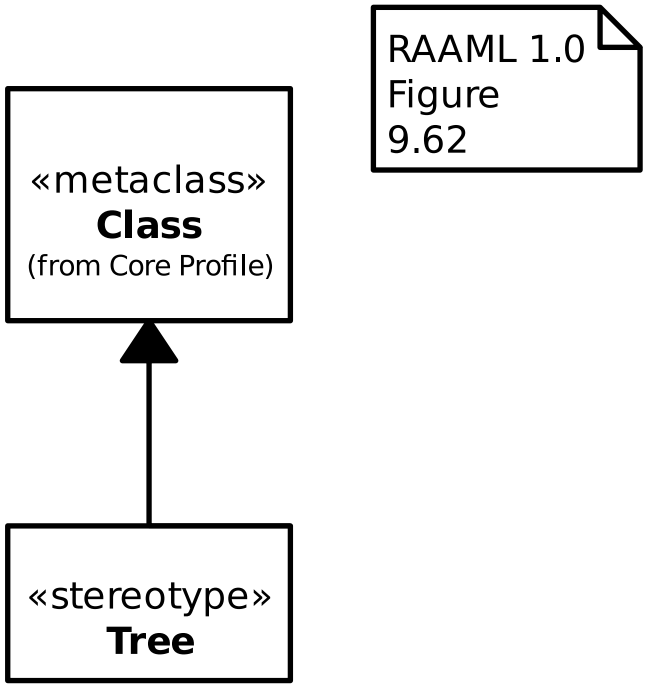
15.3.35 FTA/FTA Profile/Transfer In



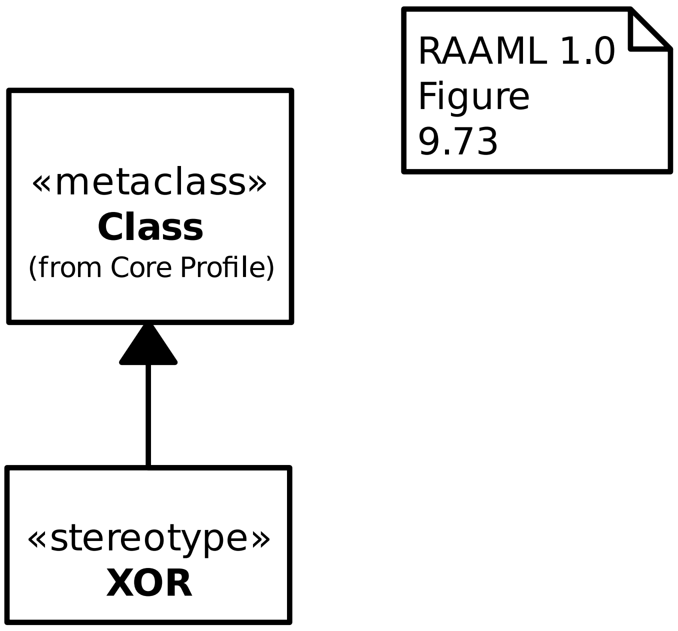
15.3.36 FTA/FTA Profile/Transfer Out



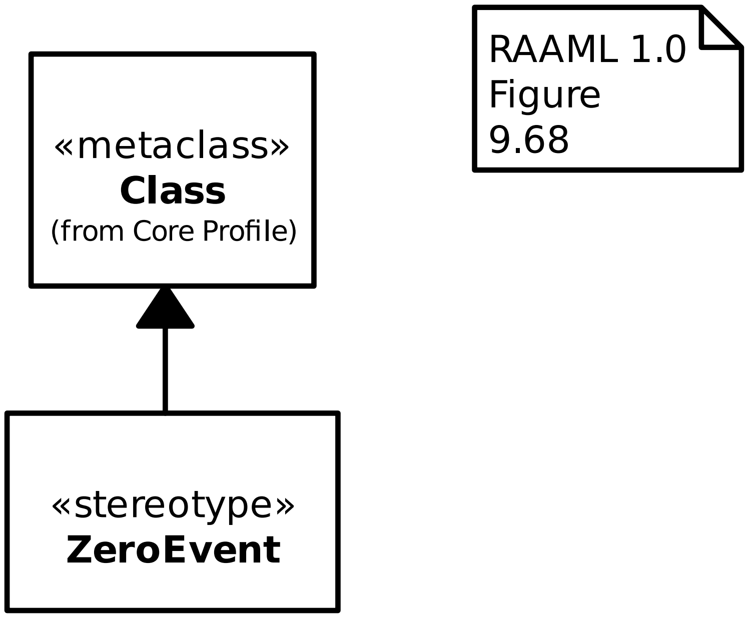
15.3.37 FTA/FTA Profile/Tree



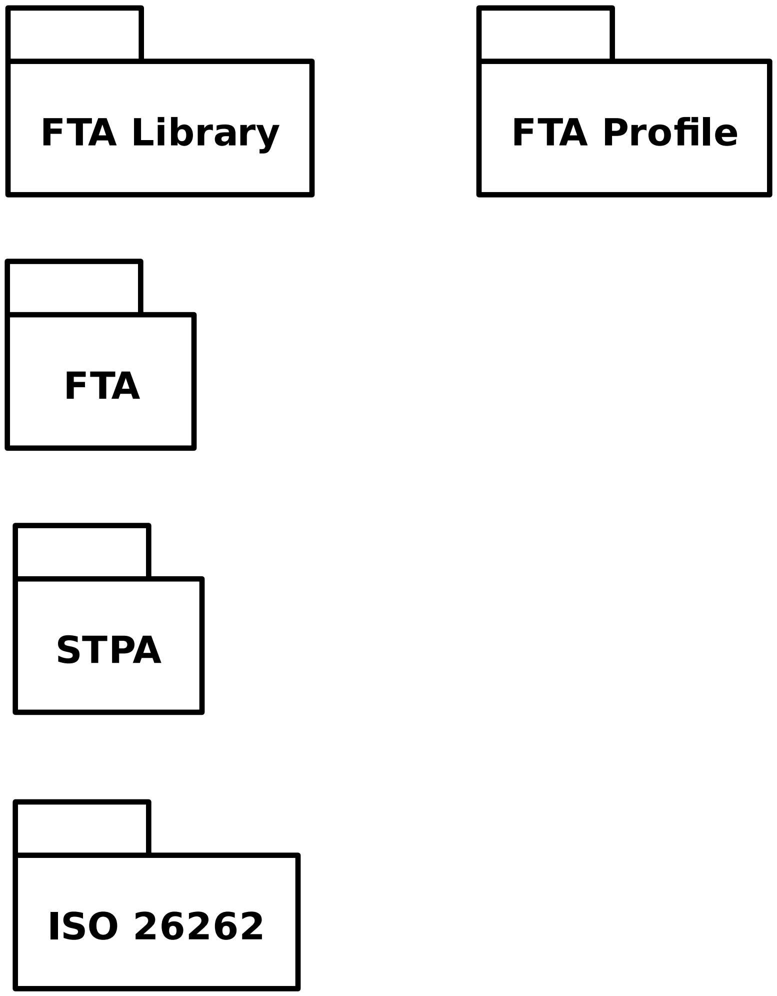
15.3.38 FTA/FTA Profile/XOR



15.3.39 FTA/FTA Profile/Zero Event



15.3.40 FTA/FTA



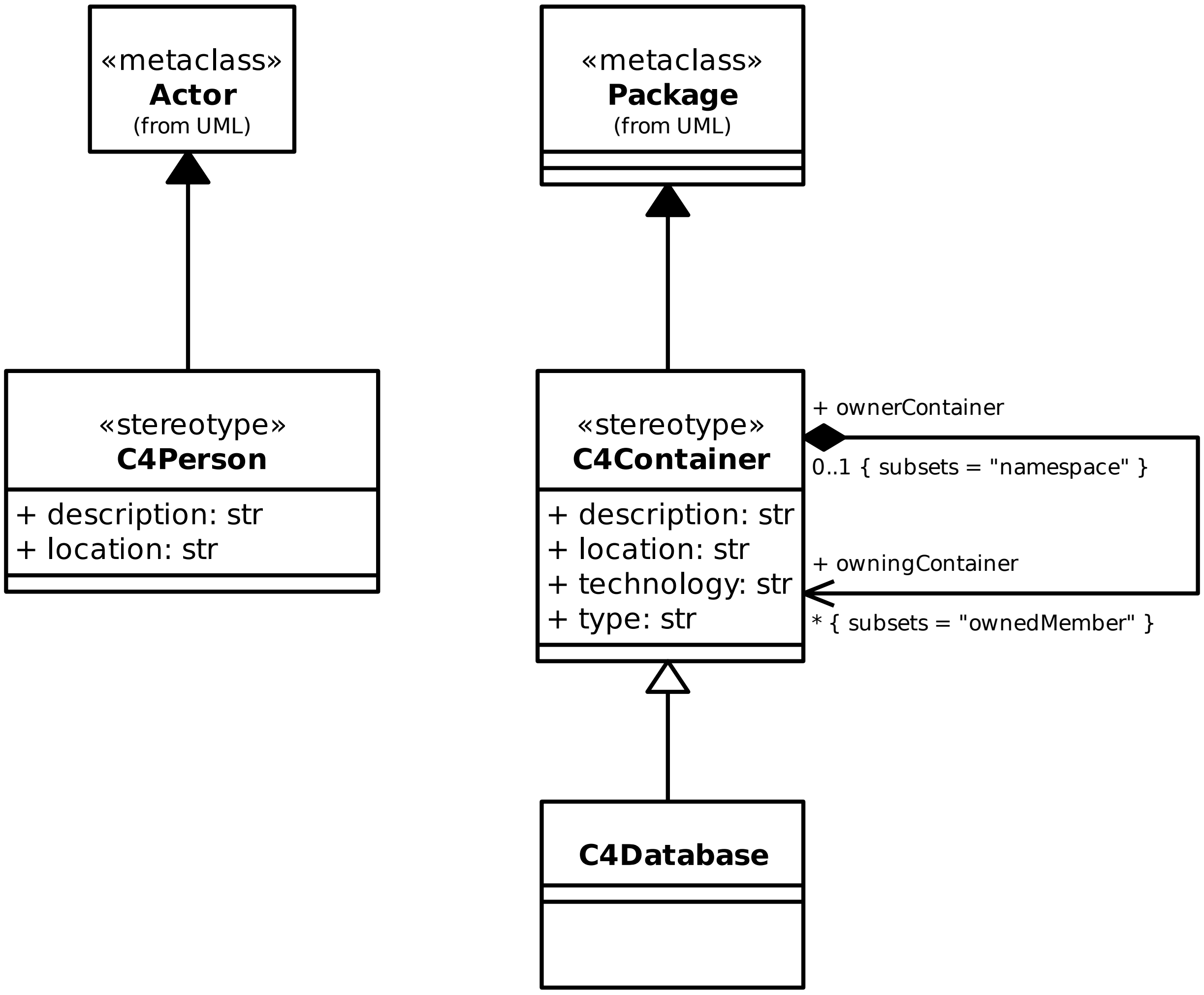
CHAPTER

SIXTEEN

THE C4 MODEL

The C4 model is a simple visual language to describe the static structure of a software system.

It's based on the *UML* language.



CHAPTER

SEVENTEEN

DESIGN PRINCIPLES

Gaphor has been around for quite a few years. In those years we (the Gaphor developers) learned a few things on how to build it. Gaphor tries to be easily accessible for novice users as well as a useful tool for more experienced users.

Gaphor is not your average editor. It's a modeling environment. This implies there is a language underpinning the models. Languages adhere to rules and Gaphor tries to follow those rules.

Usability is very important. When you're new to Gaphor, it should be easy to find your way around. Minimal knowledge of UML should at least allow you to create a class diagram.

17.1 Guidance

To help users, Gaphor should provide guidance where it can.

17.1.1 Help with relationships

The diagram has a feature that it grays out all elements a relationship can not connect to. This helps you to decide where a relation can connect to. You can still mix different elements, but we try to make it as simple as possible to make consistent models.

17.1.2 Keep the model in sync

An important part of modeling is to design a system in abstractions and be able to explain those to others. As systems become more complicated, it's important to have the design (model) layed out in diagrams.

Gaphor goes through great lengths to keep the model in sync with the diagrams. In doing so, unused elements can be automatically removed from the model if they're no longer shown in any diagram.

17.2 Out of your way

When modeling, you should be busy with your problem or solution domain, not with the tool. Gaphor tries to stay out of your way as much as possible. It does not try to nag you with error messages, because the model is not "correct".

17.2.1 Avoid dialogs

In doing the right thing, and staying out of the way of users, Gaphor avoids the use of dialogs as much as possible.

Gaphor should allow you to do the sensible thing (see above) and not get you out of your flow with all sorts of questions.

17.2.2 Notify on changes

When Gaphor is doing something that is not directly visible, you'll see a notification, for example, an element that's indirectly removed from the model. It will not interrupt you with dialogs, but only provide a small in-app notification. If the change is undesired, hit `undo`.

17.2.3 Balanced

Although Gaphor implements quite a bit of the UML 2 model, it's not complete. We try to find the right balance in features to suite both expert and novice modellers.

17.3 Continuity

A model that is created should be usable in the future. Gaphor does acknowledge that. We care about compatibility.

17.3.1 Backwards compatibility

Gaphor is capable of loading models going back to Gaphor 1.0. It's important for a tool to always allow older models to be loaded.

17.3.2 Multi-platform

We put a lot of effort in making Gaphor run on all major platforms: Windows, macOS, and Linux. Having Gaphor available on all platforms is essential if the model needs to be shared. It would be awful if you need to run one specific operating system in order to open a model.

So far, we do not support the fourth major platform (web). Native applications provide a better user experience (once installed). But this may change.

17.4 User interaction

Gaphor is originally written on Linux. It uses GTK as it's user interface toolkit. This sort of implies that Gaphor follows the GNOME Human Interface Guidelines (HIG). Gaphor is also a multi-platform application. We try to stay close to the GNOME HIG, but try not to introduce concepts that are not available on Windows and macOS.

User interface components are not generated. We found that UI generation (like many enterprise modeling tools do) provides an awful user experience. We want users to use Gaphor on a regular basis, so we aim for it to be a tool that's pleasant to look at and easy to work with.

17.5 What else?

- **Idempotency** Allow the same operation to be applied multiple times. This should not affect the result.
- **Event Driven** Gaphor is a user application. It acts to user events. The application uses an internal event dispatches (event bus) to distribute events to interested parties. Everyone should be able to listen to events.

CHAPTER

EIGHTEEN

FRAMEWORK

18.1 Overview

Gaphor is built in a light, service oriented fashion. The application is split in a series of services, such as a file, event, and undo managers. Those services are loaded based on entry points defined in the `pyproject.toml` file. To learn more about the architecture, please see the description about the *Service Oriented Architecture*.

18.2 Event driven

Parts of Gaphor communicate with each other through events. Whenever something important happens, for example, an attribute of a model element changes, an event is sent. When other parts of the application are interested in a change, they register an event handler for that event type. Events are emitted though a central broker so you do not have to register on every individual element that can send an event they are interested in. For example, a diagram item could register an event rule and then check if the element that sent the event is actually the event the item is representing. For more information see the full description of the *event system*.

18.3 Transactional

Gaphor is *transactional*, which means it keeps track of the functions it performs as a series of transactions. The transactions work by sending an event when a transaction starts and sending another when a transaction ends. This allows, for example, the undo manager to keep a running log of the previous transactions so that a transaction can be reversed if the undo button is pressed.

18.4 Main Components

The main portion of Gaphor that executes first is called the `Application`. Gaphor can have multiple models open at any time. Each model is kept in a `Session`. Only one Application instance is active. Each session will load its own services defined as *gaphor.services*.

The most notable services are:

18.4.1 event_manager

This is the central component used for event dispatching. Every service that does something with events (both sending and receiving) depends on this component.

18.4.2 file_manager

Loading and saving a model is done through this service.

18.4.3 element_factory

The *data model* itself is maintained in the element factory (`gaphor.core.modeling.elementfactory`). This service is used to create model elements, as well as to lookup elements or query for a set of elements.

18.4.4 undo_manager

One of the most appreciated services. It allows users to make a mistake every now and then!

The undo manager is transactional. Actions performed by a user are only stored if a transaction is active. If a transaction is completed (committed) a new undo action is stored. Transactions can also be rolled back, in which case all changes are played back directly. For more information see the full description of the *undo manager*.

CHAPTER

NINETEEN

SERVICE ORIENTED ARCHITECTURE

Gaphor has a service oriented architecture. What does this mean? Well, Gaphor is built as a set of small islands (services). Each island provides a specific piece of functionality. For example, we use separate services to load/save models, provide the menu structure, and to handle the undo system.

We define services as entry points in the `pyproject.toml`. With entry points, applications can register functionality for specific purposes. We also group entry points in to *entry point groups*. For example, we use the console_scripts entry point group to start an application from the command line.

19.1 Services

Gaphor is modeled around the concept of services. Each service can be registered with the application and then it can be used by other services or other objects living within the application.

Each service should implement the Service interface. This interface defines one method:

```
shutdown(self)
```

Which is called when a service needs to be cleaned up.

We allow each service to define its own methods, as long as the service is implemented too.

Services should be defined as entry points in the `pyproject.toml` file.

Typically, a service does some work in the background. Services can also expose actions that can be invoked by users. For example, the *Ctrl-z* key combo (undo) is implemented by the UndoManager service.

A service can also depend on another services. Service initialization resolves these dependencies. To define a service dependency, just add it to the constructor by its name defined in the entry point:

```
class MyService(Service):

    def __init__(self, event_manager, element_factory):
        self.event_manager = event_manager
        self.element_factory = element_factory
        event_manager.subscribe(self._element_changed)

    def shutdown(self):
        self.event_manager.unsubscribe(self._element_changed)

    @event_handler(ElementChanged)
    def _element_changed(self, event):
```

Services that expose actions should also inherit from the ActionProvider interface. This interface does not require any additional methods to be implemented. Action methods should be annotated with an `@action` annotation.

19.2 Example: ElementFactory

A nice example of a service in use is the ElementFactory. It is one of the core services.

The UndoManager depends on the events emitted by the ElementFactory. When an important events occurs, like an element is created or destroyed, that event is emitted. We then use an event handler for ElementFactory that stores the add/remove signals in the undo system. Another example of events that are emitted are with `UML.Elements`. Those classes, or more specifically, the properties, send notifications every time their state changes.

19.3 Entry Points

Gaphor uses a main entry point group called `gaphor.services`.

Services are used to perform the core functionality of the application while breaking the functions in to individual components. For example, the element factory and undo manager are both services.

Plugins can also be created to extend Gaphor beyond the core functionality as an add-on. For example, a plugin could be created to connect model data to other applications. Plugins are also defined as services. For example a new XMI export plugin would be defined as follows in the `pyproject.toml`:

```
[tool.poetry.plugins."gaphor.services"]
"xmi_export" = "gaphor.plugins.xmiexport:XMIExport"
```

19.4 Interfaces

Each service (and plugin) should implement the `gaphor.abc.Service` interface:

class `gaphor.abc.`**`Service`**

Base interface for all services in Gaphor.

abstract **`shutdown`**`()` → None

Shutdown the services, free resources.

Another more specialized service that also inherits from `gaphor.abc.Service`, is the UI Component service. Services that use this interface are used to define windows and user interface functionality. A UI component should implement the `gaphor.ui.abc.UIComponent` interface:

class `gaphor.ui.abc.`**`UIComponent`**

A user interface component.

abstract **`close`**`()`

Close the UI component.

The component can decide to hide or destroy the UI components.

abstract **`open`**`()`

Create and display the UI components (windows).

shutdown()

Shut down this component.

It's not supposed to be opened again.

Typically, a service and UI component would like to present some actions to the user, by means of menu entries. Every service and UI component can advertise actions by implementing the `gaphor.abc.ActionProvider` interface:

class gaphor.abc.**ActionProvider**

An action provider is a special service that provides actions via `@action` decorators on its methods (see gaphor/action.py).

19.5 Example plugin

A small example is provided by means of the Hello world plugin. Take a look at the files at GitHub. The example plugin needs to be updated to support versions 1.0.0 and later of Gaphor.

The pyproject.toml file contains a plugin:

```
[tool.poetry.plugins."gaphor.services"]
"helloworld" = "gaphor_helloworld_plugin:HelloWorldPlugin"
```

This refers to the class `HelloWorldPlugin` in package/module gaphor_plugins_helloworld.

Here is a stripped version of the hello world plugin:

```python
from gaphor.abc import Service, ActionProvider
from gaphor.core import _, action

class HelloWorldPlugin(Service, ActionProvider):     # 1.

    def __init__(self, tools_menu):                  # 2.
        self.tools_menu = tools_menu
        tools_menu.add_actions(self)                 # 3.

    def shutdown(self):                              # 4.
        self.tools_menu.remove_actions(self)

    @action(                                         # 5.
        name="helloworld",
        label=_("Hello world"),
        tooltip=_("Every application..."),
    )
    def helloworld_action(self):
        main_window = self.main_window
        pass  # gtk code left out
```

1. As stated before, a plugin should implement the `Service` interface. It also implements `ActionProvider`, saying it has some actions to be performed by the user.
2. The menu entry will be part of the "Tools" extension menu. This extension point is created as a service. Other services can also be passed as dependencies. Services can get references to other services by defining them as arguments of the constructor.
3. All action defined in this service are registered.

4. Each service has a `shutdown()` method. This allows the service to perform some cleanup when it's shut down.
5. The action that can be invoked. The action is defined and will be picked up by `add_actions()` method (see 3.)

CHAPTER

TWENTY

EVENT SYSTEM

The event system in Gaphor provides an API to *handle* events and to *subscribe* to events.

In Gaphor we manage event handler subscriptions through the `EventManager` service. Gaphor is highly event driven:

- Changes in the loaded model are emitted as events
- Changes on diagrams are emitted as events
- Changes in the UI are emitted as events

Although Gaphor depends heavily on GTK for its user interface, Gaphor is using its own event dispatcher. Events can be structured in hierarchies. For example, an `AttributeUpdated` event is a subtype of `ElementUpdated`. If we are interested in all changes to elements, we can also register `ElementUpdated` and receive all `AttributeUpdated` events as well.

`class gaphor.core.eventmanager.`**`EventManager`**

The Event Manager.

`handle``(`**events: object*`)` → None

Send event notifications to registered handlers.

`priority_subscribe``(`*handler: Callable[[object], None]*`)` → None

Register a handler.

Priority handlers are executed directly. They should not raise other events, cause that can cause a problem in the exection order.

It's basically to make sure that all events are recorded by the undo manager.

`shutdown``()` → None

Shutdown the services, free resources.

`subscribe``(`*handler: Callable[[object], None]*`)` → None

Register a handler.

Handlers are triggered (executed) when specific events are emitted through the handle() method.

`unsubscribe``(`*handler: Callable[[object], None]*`)` → None

Unregister a previously registered handler.

Under the hood events are handled by the Generics library. For more information about how the Generic library handles events see the Generic documentation.

CHAPTER

TWENTYONE

MODELING LANGUAGES

Since version 2.0, Gaphor supports the concept of Modeling languages. This allows for development of separate modeling languages separate from the Gaphor core application.

The main language was, and will be UML. Gaphor now also supports a subset of SysML, RAAML and the C4 model.

A modeling language in Gaphor is defined by a class implementing the `gaphor.abc.ModelingLanguage` abstract base class. The modeling language should be registered as a `gaphor.modelinglanguage` entry point.

The `ModelingLanguage` interface is fairly minimal. It allows other services to look up elements and diagram items, as well as a toolbox, and diagram types. However, the responsibilities of a modeling language do not stop there. Parts of functionality will be implemented by registering handlers to a set of generic functions.

But let's not get ahead of ourselves. What is the functionality a modeling language implementation can offer?

- A data model (elements) and diagram items
- Diagram types
- A toolbox definition
- *Connectors*, allow diagram items to connect
- *Copy/paste* behavior when element copying is not trivial, for example with more than one element is involved
- *Grouping*, allow elements to be nested in one another
- *Dropping*, allow elements to be dragged from the tree view onto a diagram
- *Automatic cleanup rules* to keep the model consistent

The first three by functionalities are exposed by the `ModelingLanguage` class. The other functionalities can be extended by adding handlers to the respective generic functions.

Modeling languages can also provide new UI components. Those components are not loaded directly when you import a modeling language package. Instead they should be imported via the `gaphor.modules` entrypoint.

- *Editor pages*, shown in the collapsible pane on the right side
- *Instant (diagram) editor popups*
- Special diagram interactions

class gaphor.abc.**ModelingLanguage**

A model provider is a special service that provides an entrypoint to a model implementation, such as UML, SysML, RAAML.

abstract property diagram_types: Iterable[DiagramType]

Iterate diagram types.

abstract lookup_element(*name: str*) → type[Element] | None

Look up a model element type by (class) name.

abstract property name: str

Human-readable name of the modeling language.

abstract property toolbox_definition: ToolboxDefinition

Get structure for the toolbox.

21.1 Connectors

Connectors are used to connect one element to another.

Connectors should adhere to the `ConnectorProtocol`. Normally you would inherit from `BaseConnector`.

class gaphor.diagram.connectors.**BaseConnector**(*element: Presentation[Element]*, *line: Presentation[Element]*)

Connection adapter for Gaphor diagram items.

Line item `line` connects with a handle to a connectable item `element`.

Parameters

- **line** (*Presentation*) – connecting item
- **element** (*Presentation*) – connectable item

The following methods are required to make this work:

- `allow()`: is the connection allowed at all (during mouse movement for example).
- `connect()`: Establish a connection between element and line. Also takes care of disconnects, if required (e.g. 1:1 relationships)
- `disconnect()`: Break connection, called when dropping a handle on a point where it can not connect.

By convention the adapters are registered by (element, line) – in that order.

allow(*handle: Handle*, *port: Port*) → bool

Determine if items can be connected.

Returns *True* if connection is allowed.

connect(*handle: Handle*, *port: Port*) → bool

Connect to an element. Note that at this point the line may be connected to some other, or the same element. The connection at model level also still exists.

Returns *True* if a connection is established.

disconnect(*handle: Handle*) → None

Disconnect model level connections.

get_connected(*handle: Handle*) → Optional[Presentation[Element]]

Get item connected to a handle.

21.2 Copy and paste

Copy and paste works out of the box for simple items: one diagram item with one model element (the `subject`). It leveages the `load()` and `save()` methods of the elements to ensure all relevant data is copied.

Sometimes items need more than one model element to work. For example an Association: it has two association ends.

In those specific cases you need to implement your own copy and paste functions. To create such a thing you'll need to create two functions: one for copying and one for pasting.

`gaphor.diagram.copypaste.`**`copy`**`(`*obj: Element*`)` → Iterator[tuple[Id, Opaque]]

Create a copy of an element (or list of elements). The returned type should be distinct, so the *paste()* function can properly dispatch.

`gaphor.diagram.copypaste.`**`paste`**`(`*copy_data: T*, *diagram: Diagram*, *lookup: Callable[[str], Element | None]*`)` → Iterator[Element]

Paste previously copied data. Based on the data type created in the `copy()` function, try to duplicate the copied elements. Returns the newly created item or element

`gaphor.diagram.copypaste.`**`paste_link`**`(`*copy_data: CopyData*, *diagram: Diagram*, *lookup: Callable[[str], Element | None]*`)` → set[Presentation]:

Create a copy of the Presentation element, but try to link the underlying model element. A shallow copy.

`gaphor.diagram.copypaste.`**`paste_full`**`(`*copy_data: CopyData*, *diagram: Diagram*, *lookup: Callable[[str], Element | None]*`)` → set[Presentation]:

Create a copy of both Presentation and model element. A deep copy.

To serialize the copied elements and deserialize them again, there are two functions available:

`gaphor.diagram.copypaste.`**`serialize`**`(`*value*`)`

Return a serialized version of a value. If the `value` is an element, it's referenced.

`gaphor.diagram.copypaste.`**`deserialize`**`(`*ser*, *lookup*`)`

Deserialize a value previously serialized with `serialize()`. The `lookup` function is used to resolve references to other elements.

21.3 Grouping

Grouping is done by dragging one item on top of another, in a diagram or in the tree view.

`gaphor.diagram.group.`**`group`**`(`*parent: Element*, *element: Element*`)` → bool

Group an element in a parent element. The grouping can be based on ownership, but other types of grouping are also possible.

`gaphor.diagram.group.`**`ungroup`**`(`*parent: Element*, *element: Element*`)` → bool

Remove the grouping from an element. The function needs to check if the provided *parent* node is the right one.

`gaphor.diagram.group.`**`can_group`**`(`*parent_type: Type[Element]*, *element_or_type: Type[Element] | Element*`)` → bool

This function tries to determine if grouping is possible, without actually performing a group operation. This is not 100% accurate.

21.4 Dropping

Dropping is performed by dragging an element from the tree view and drop it on a diagram. This is an easy way to extend a diagram with already existing model elements.

`gaphor.diagram.drop.`**`drop`**(*element: Element*, *diagram: Diagram*, *x: float*, *y: float*) → Presentation | None

The drop function creates a new presentation for an element on the diagram. For relationships, a drop only works if both connected elements are present in the same diagram.

The big difference with dragging an element from the toolbox, is that dragging from the toolbox will actually place a new `Presentation` element on the diagram. `drop` works the other way around: it starts with a model element and creates an accompanying `Presentation`.

21.5 Automated model cleanup

Gaphor wants to keep the model in sync with the diagrams.

A little dispatch function is used to determine if a model element can be removed.

`gaphor.diagram.deletable.`**`deletable`**(*element: Element*) → bool

Determine if a model element can safely be removed.

21.6 Editor property pages

The editor page is constructed from snippets. For example: almost each element has a name, so there is a UI snippet that allows you to edit a name.

Each property page (snippet) should inherit from `PropertyPageBase`.

`class` `gaphor.diagram.propertypages.`**`PropertyPageBase`**

A property page which can display itself in a notebook.

`abstract construct`()

Create the page (Gtk.Widget) that belongs to the Property page.

Returns the page's toplevel widget (Gtk.Widget).

21.7 Instant (diagram) editor popups

When you double click on an item in a diagram, a popup can show up so you can easily change the name.

By default this works for any named element. You can register your own inline editor function if you need to.

`gaphor.diagram.instanteditors.`**`instant_editor`**(*item: Item*, *view*, *event_manager*, *pos: Optional[Tuple[int, int]] = None*) → bool

Show a small editor popup in the diagram. Makes for easy editing without resorting to the Element editor.

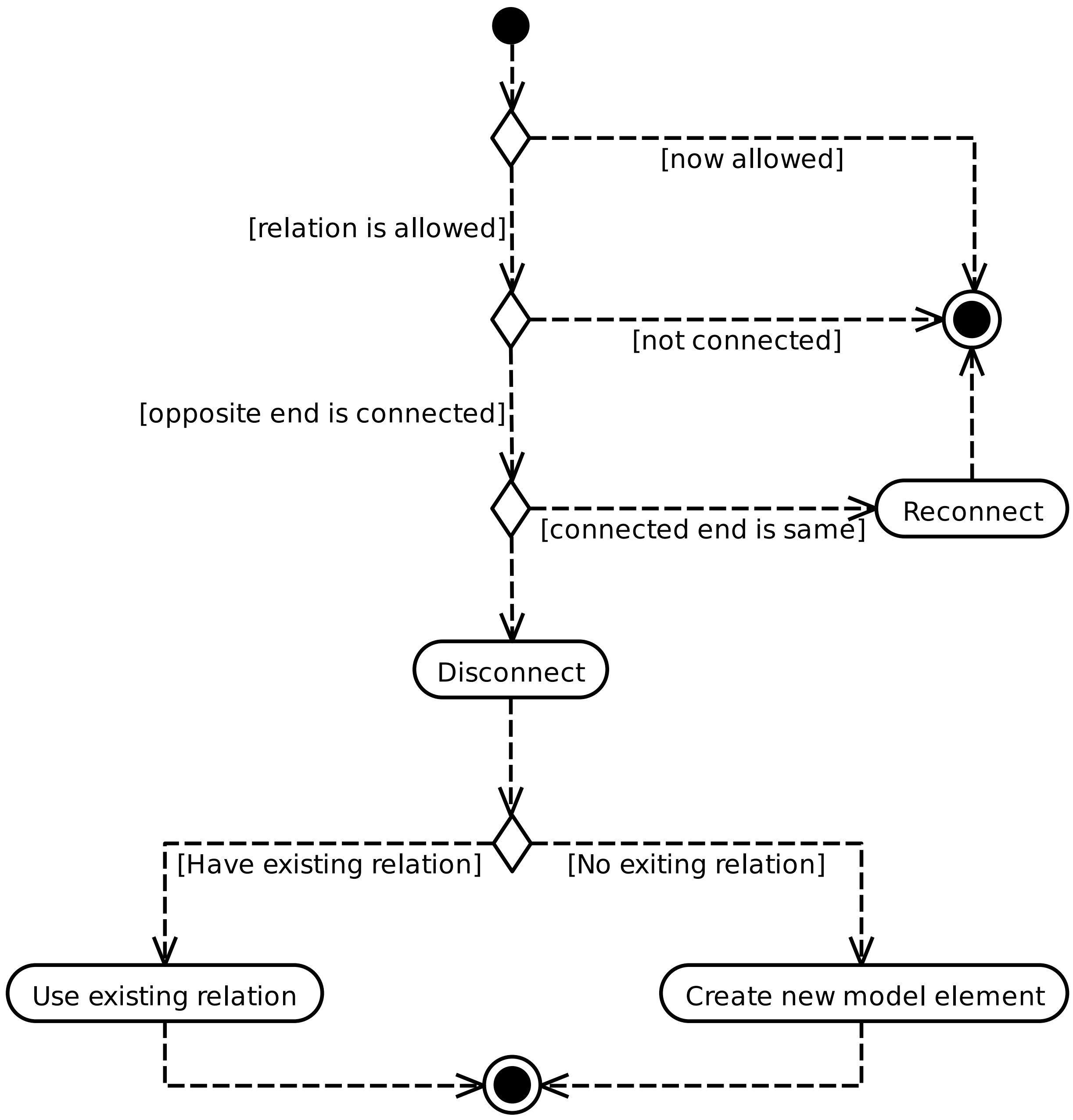
In case of a mouse press event, the mouse position (relative to the element) are also provided.

CHAPTER

TWENTYTWO

CONNECTION PROTOCOL

In Gaphor, if a connection is made on a diagram between an element and a relationship, the connection is also made at semantic level (the model). From a GUI point of view, a button release event is what kicks of the decision whether the connection is allowed.



The check if a connection is allowed should also check if it is valid to create a relation to/from the same element (like associations, but not generalizations).

CHAPTER

TWENTYTHREE

FILE FORMAT

The root element of Gaphor models is the `Gaphor` tag, all other elements are contained in this. The Gaphor element delimits the beginning and the end of an Gaphor model.

The idea is to keep the file format as simple and extensible as possible: UML elements (including Diagram) are at the top level with no nesting. A UML element can have two tags: references (`ref`) and values (`val`). References are used to point to other UML elements. Values have a value inside (an integer or a string).

Since many references are bi-directional, you'll find both ends defined in the file (e.g. `Package.ownedType` - `Actor.package`, and `Diagram.ownedPresentation` and `UseCaseItem.diagram`).

```xml
<?xml version="1.0" ?>
<Gaphor version="1.0" gaphor_version="0.3">
  <Package id="1">
    <ownedClassifier>
      <reflist>
        <ref refid="2"/>
        <ref refid="3"/>
        <ref refid="4"/>
      </reflist>
    </ownedClassifier>
  </Package>
  <Diagram id="2">
    <package>
      <ref refid="1"/>
    </package>
    <ownedPresentation>
      <reflist>
        <ref refid="5"/>
        <ref refid="6"/>
      </reflist>
    </ownedPresentation>
  </Diagram>
  <ActorItem id="5">
    <matrix>
      <val>(1.0, 0.0, 0.0, 1.0, 147.0, 132.0)</val>
    </matrix>
    <width>
      <val>38.0</val>
    </width>
    <height>
      <val>60.0</val>
```

(continues on next page)

(continued from previous page)

```xml
    </height>
    <diagram>
      <ref refid="2"/>
    </diagram>
    <subject>
      <ref refid="3"/>
    </subject>
  </ActorItem>
  <UseCaseItem id="6">
    <matrix>
      <val>(1.0, 0.0, 0.0, 1.0, 341.0, 144.0)</val>
    </matrix>
    <width>
      <val>98.0</val>
    </width>
    <height>
      <val>30.0</val>
    </height>
    <diagram>
      <ref refid="2"/>
    </diagram>
    <subject>
      <ref refid="4"/>
    </subject>
  </UseCaseItem>
  <Actor id="3">
    <name>
      <val>Actor></val>
    </name>
    <package>
      <ref refid="1"/>
    </package>
  </Actor>
  <UseCase id="4">
    <package>
      <ref refid="1"/>
    </package>
  </UseCase>
</Gaphor>
```

CHAPTER

TWENTYFOUR

UNDO MANAGER

Undo is a required feature in modern applications. Gaphor is no exception. Having an undo function in place means you can change the model and easily revert to an older state.

24.1 Overview of Transactions

The recording and playback of changes in Gaphor is handled by the the Undo Manager. The Undo Manager works transactionally. A transaction must succeed or fail as a complete unit. If the transaction fails in the middle, it is rolled back. In Gaphor this is achieved by the `transaction` module, which provides a context manager `Transaction` and a decorator called `@transactional`.

When transactions take place, they emit event notifications on the key transaction milestones so that other services can make use of the events. The event notifications are for the begin of the transaction, and the commit of the transaction if it is successful or the rollback of the transaction if it fails.

24.2 Start of a Transaction

1. A `Transaction` object is created.
2. `TransactionBegin` event is emitted.
3. The `UndoManager` instantiates a new `ActionStack` which is the transaction object, and adds the undo action to the stack.

Nested transactions are supported to allow a transaction to be added inside of another transaction that is already in progress.

24.3 Successful Transaction

1. A `TransactionCommit` event is emitted
2. The `UndoManager` closes and stores the transaction.

24.4 Failed Transaction

1. A `TransactionRollback` event is emitted.
2. The `UndoManager` plays back all the recorded actions, but does not store it.

24.5 References

- A Framework for Undoing Actions in Collaborative Systems
- Undoing Actions in Collaborative Work: Framework and Experience
- Implementing a Selective Undo Framework in Python

CHAPTER

TWENTYFIVE

TRANSACTION SUPPORT

Transaction support is located in module gaphor.transaction:

```
>>> from gaphor import transaction

>>> import sys, logging
>>> transaction.log.addHandler(logging.StreamHandler(sys.stdout))
```

Do some basic initialization, so event emission will work. Since the transaction decorator does not know about the active user session (window), it emits it's events via a global list of subscribers:

```
>>> from gaphor.core.eventmanager import EventManager
>>> event_manager = EventManager()
>>> transaction.subscribers.add(event_manager.handle)
```

The Transaction class is used mainly to signal the begin and end of a transaction. This is done by the TransactionBegin, TransactionCommit and TransactionRollback events:

```
>>> from gaphor.core import event_handler
>>> @event_handler(transaction.TransactionBegin)
... def transaction_begin_handler(event):
...     print('tx begin')
>>> event_manager.subscribe(transaction_begin_handler)
```

Same goes for commit and rollback events:

```
>>> @event_handler(transaction.TransactionCommit)
... def transaction_commit_handler(event):
...     print('tx commit')
>>> event_manager.subscribe(transaction_commit_handler)
>>> @event_handler(transaction.TransactionRollback)
... def transaction_rollback_handler(event):
...     print('tx rollback')
>>> event_manager.subscribe(transaction_rollback_handler)
```

A Transaction is started by initiating a Transaction instance:

```
>>> tx = transaction.Transaction(event_manager)
tx begin
```

On success, a transaction can be committed:

```
>>> tx.commit()
tx commit
```

After a commit, a rollback is no longer allowed (the transaction is closed):

```
>>> tx.rollback()
... # doctest: +ELLIPSIS
Traceback (most recent call last):
...
gaphor.transaction.TransactionError: No Transaction on stack.
```

Transactions may be nested:

```
>>> tx = transaction.Transaction(event_manager)
tx begin
>>> tx2 = transaction.Transaction(event_manager)
>>> tx2.commit()
>>> tx.commit()
tx commit
```

Transactions should be closed in the right order (subtransactions first):

```
>>> tx = transaction.Transaction(event_manager)
tx begin
>>> tx2 = transaction.Transaction(event_manager)
>>> tx.commit()
... # doctest: +ELLIPSIS
Traceback (most recent call last):
...
gaphor.transaction.TransactionError: Transaction on stack is not the transaction being␣
↪closed.
>>> tx2.commit()
>>> tx.commit()
tx commit
```

The transactional decorator can be used to mark functions as transactional:

```
>>> @transaction.transactional
... def a():
...     print('do something')
>>> a()
tx begin
do something
tx commit
```

If an exception is raised from within the decorated function a rollback is performed:

```
>>> @transaction.transactional
... def a():
...     raise IndexError('bla')
>>> a() # doctest; +ELLIPSIS
Traceback (most recent call last):
...
IndexError: bla
```

(continues on next page)

(continued from previous page)

```
>>> transaction.Transaction._stack
[]
```

Cleanup:

```
>>> transaction.subscribers.discard(event_manager.handle)
```

INDEX