Gaphor Documentation

Arjan Molenaar

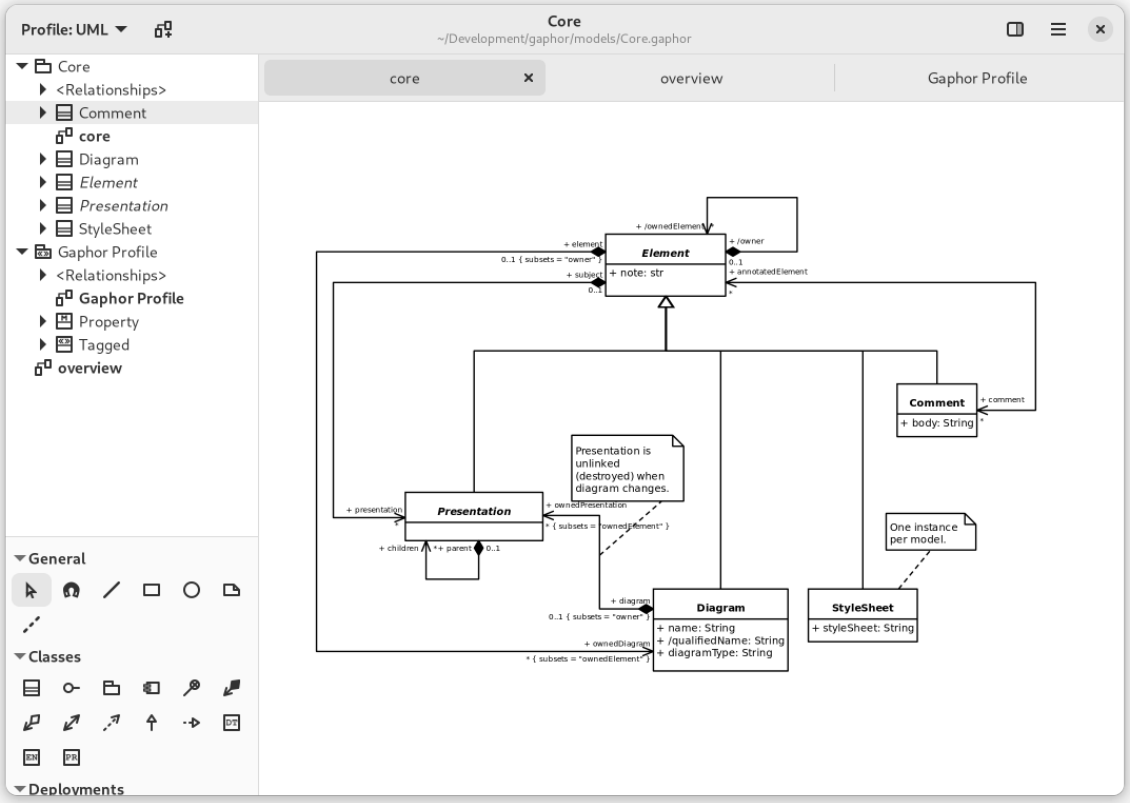
Mar 27, 2024

GETTING STARTED

Note: The documentation is up to date for Gaphor 2.24.0

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, available under a friendly Apache 2 license. 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.

Did you know Gaphor has excellent integration with *Sphinx* and *Jupyter notebooks*?

CHAPTER

ONE

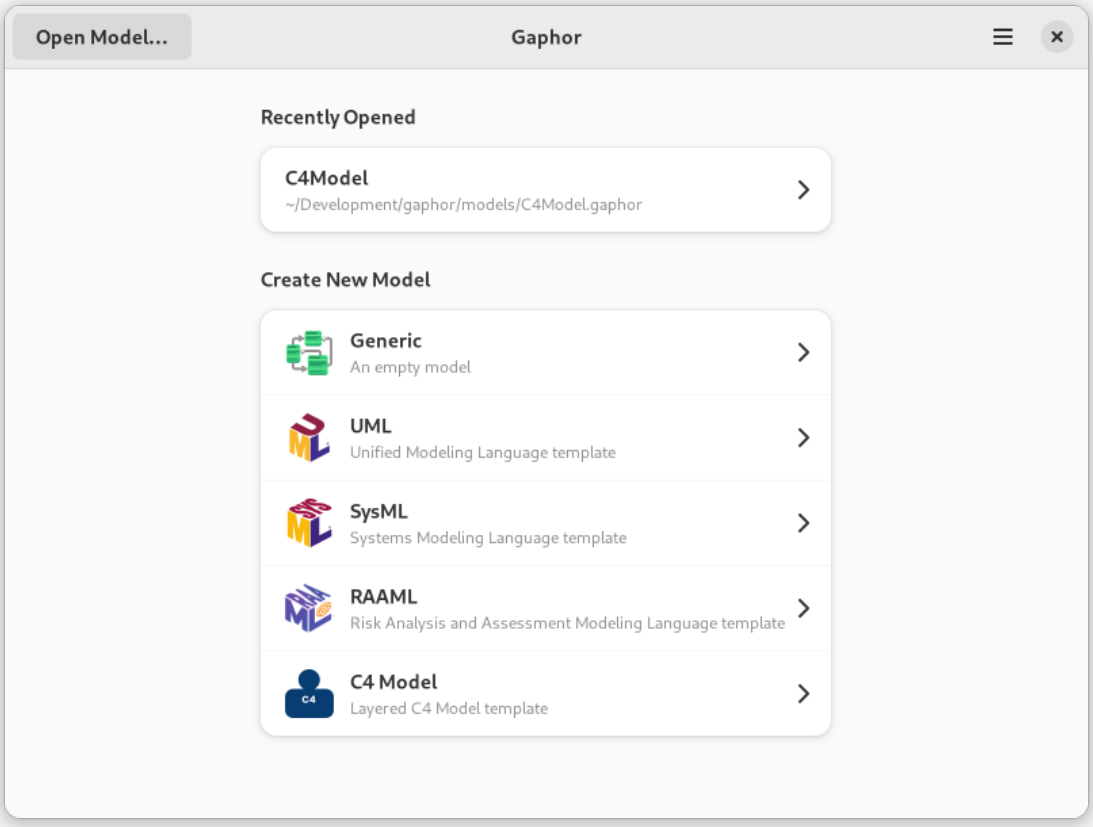
GET STARTED WITH GAPHOR

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), Windows, and macOS.

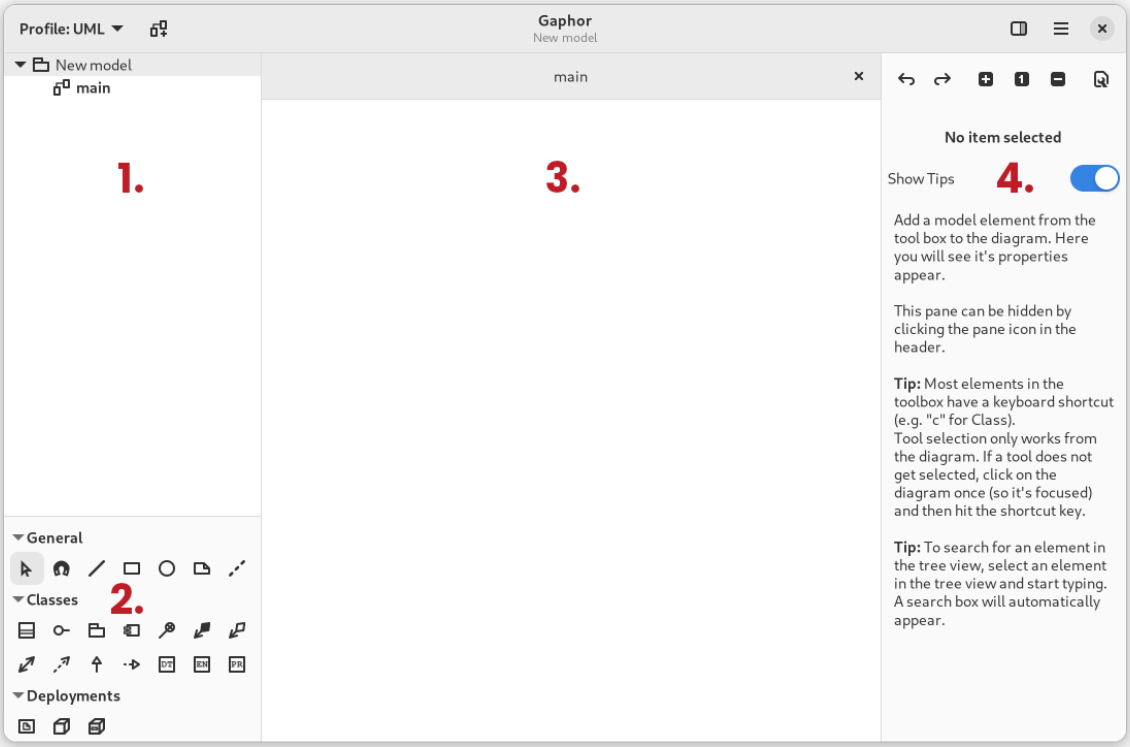
Once Gaphor is launched, it provides you a welcome screen. It shows you previously opened models and model templates.



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 is for lean modeling of software architecture

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



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 items 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 Copy and Paste

As stated before, Gaphor is a modeling environent. This means that every *item* in a diagram is backed by a *model element* found in the model browser. This means that you can show the same *model element* in different diagrams.

- Ctrl+v is used to paste *only* the presentation element.
- Ctrl+Shift+v is used to paste a new presentation with a new model element.

Important: Ctrl+v does a "shallow" paste. Ctrl+Shift+v does a "deep" paste.

1.3.4 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.

1.5.1 Reset Tool Automatically

By default the pointer tool is selected after an element is placed from the toolbox. If this option is turned off, the same type of element will be placed by clicking in the diagram until another element is selected in the toolbox.

1.5.2 Remove Unused Elements

By default elements that are not part of any diagram in the model will be removed. If this option is turned off, elements remain in the model and may be found in the model browser.

1.5.3 Diagram Language

The diagram language modifier is only applicable to the loaded model and how it is shown in the diagram. The diagram language setting is saved as part of the model and defaults to English.

The UI language of Gaphor is controlled by the operating system.

Note: Gaphor considers the `LANG` environment variable on Linux, Windows and macOS.

On Windows and macOS it can be set independently of the operating system's language settings to a different language.

1.5.4 Style Sheet

The *style sheet* allows to change the visual appearance of diagrams and model elements.

CHAPTER

TWO

YOUR FIRST MODEL

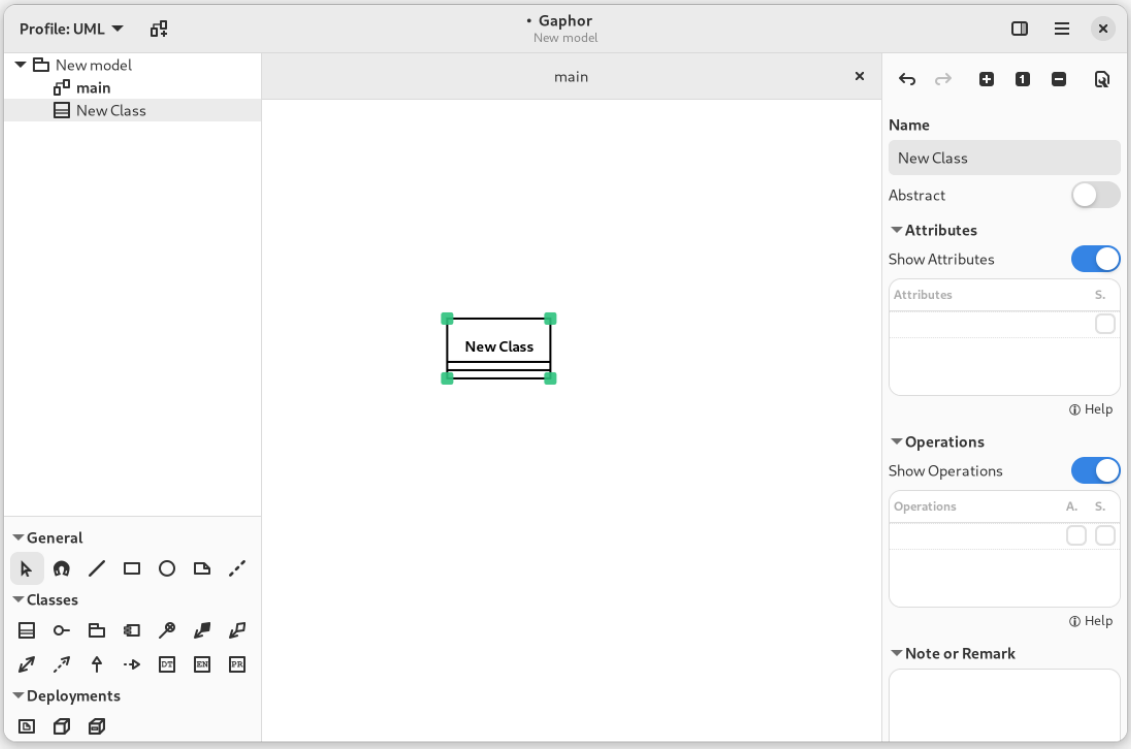
Note: In this tutorial we refer to the different parts of the gaphor interface: *Model Browser*, *Toolbox*, *Property Editor*.

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

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). The Note field can contain any text you wish to associate with the element, (this will not show on a diagram).



It's simple to add elements to a diagram.

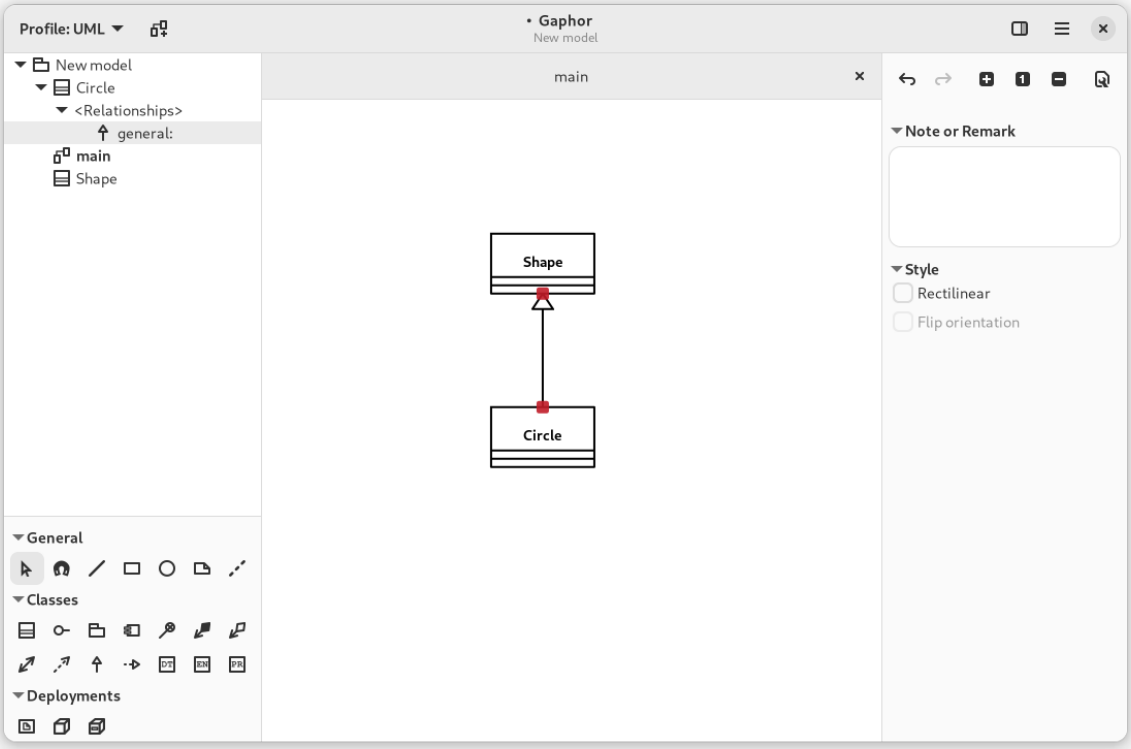
Gaphor does not make any assumptions about which elements should be placed on a diagram. A diagram is a diagram. UML defines all different kinds of diagrams, such as Class diagrams, Component diagrams, Action diagrams, Sequence diagrams. But Gaphor does not place any restrictions.

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 underlying 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

TUTORIAL: COFFEE MACHINE

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.

3.1 Introduction

In the bustling town of Antville, a colony of ants had formed a Systems Engineering consulting company called AntSource. They value collaboration, transparency, and community-driven engineering, and seeks to empower their employees and customers through open communication and participation in the systems engineering process.

The engineers at AntSource all have nicknames that reflect the key principles and concepts of their craft: Qual-ant, Reli-ant, Safe-ant, Usa-ant, and Sust-ant. They were experts in designing and optimizing complex systems, and they took pride in their work.

One day, a new client approached AntSource with an unusual request. Cappuccino, a cat who owned a popular coffee shop called Milk & Whiskers Café, needed a custom espresso machine designed specifically for felines. Cats just love their coffee strong, with a creamy and smooth body and topped with the perfect foamy layer of crema. The ants were intrigued by the challenge and immediately set to work.

Qual-ant was responsible for ensuring that the machine met all quality standards and specifications, while Reli-ant was tasked with making sure that the machine was dependable and would work correctly every time it was used. Safe-ant designed the machine with safety in mind, ensuring that it wouldn't cause harm to anyone who used it. Usa-ant designed the machine to be easy and intuitive to use, while Sust-ant ensured that the machine was environmentally friendly and sustainable. In this tutorial we follow the adventures of AntSource to create the perfect kittie espresso machine.

The first thing the ants did was to open Gaphor to the Greeter window and start a new model with the *SysML* template.

3.2 Abstraction Levels

Abstraction is a way of simplifying complex systems by focusing on only the most important details, while ignoring the rest. It's a process of reducing complexity by removing unnecessary details and highlighting the key aspects of a system in order to focus on the problem to be solved. It is the key to rigorous analysis of a system.

To understand abstraction, think about a painting. When you look at a painting, you see a representation of something - perhaps a person, a landscape, or an object. The artist has simplified the real world into a set of lines, shapes, and colors that represent the most important details of the subject. In the same way, systems engineers, like our friends the ants, use abstraction to represent complex systems by breaking them down into their essential components and highlighting the most important aspects.

Abstraction levels refer to the different levels of detail at which a system can be represented. These levels are used to break down complex systems into smaller, more manageable parts that can be analyzed and optimized. Said another way, abstraction levels group portions of a design where similar types of questions are answered.

There are typically three levels of abstraction in systems engineering and these are the three levels used in the SysML template:

- Concept Level: Sometimes also called Conceptual Level. Defines the problem being solved. This is the highest level of abstraction, where the system is described in terms of its overall purpose, goals, and functions. At this

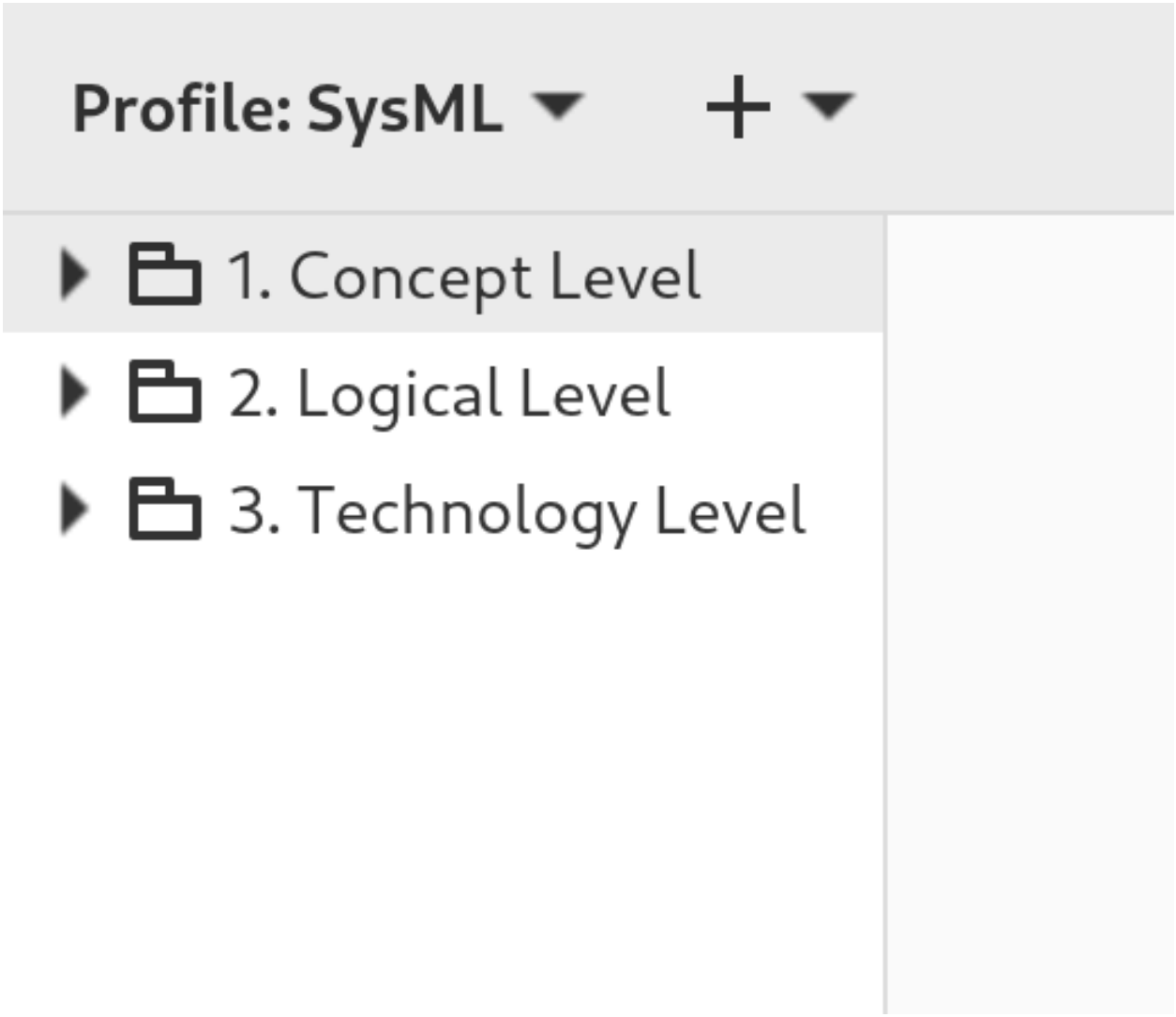
level, the focus is on understanding the system's requirements and how it will interact with other systems.

- Logical Level: Defines a technology-agnostic solution. This is the middle level of abstraction, where the system is described in terms of its structure and behavior. At this level, the focus is on how the system components are organized and how they interact with each other.
- Technology Level: Sometimes also called Physical level. Defines the detailed technical solution. This is the lowest level of abstraction, where the system is described in terms of its components and their properties. At this level, the focus is on the details of the system's implementation.

Each level of abstraction provides a different perspective on the system, and each level is important for different aspects of system design and analysis. For example, the conceptual level is important for understanding the overall goals and requirements of the system, while the physical level is important for understanding how the system will be built and how it will interact with the environment.

There is a fourth abstraction level called the Implementation Level that isn't modeled, which is the concrete built system.

In the upper left hand corner of Gaphor, the Model Browser shows the three top level packages, dividing up the model in to these three abstraction levels.



3.3 Pillars

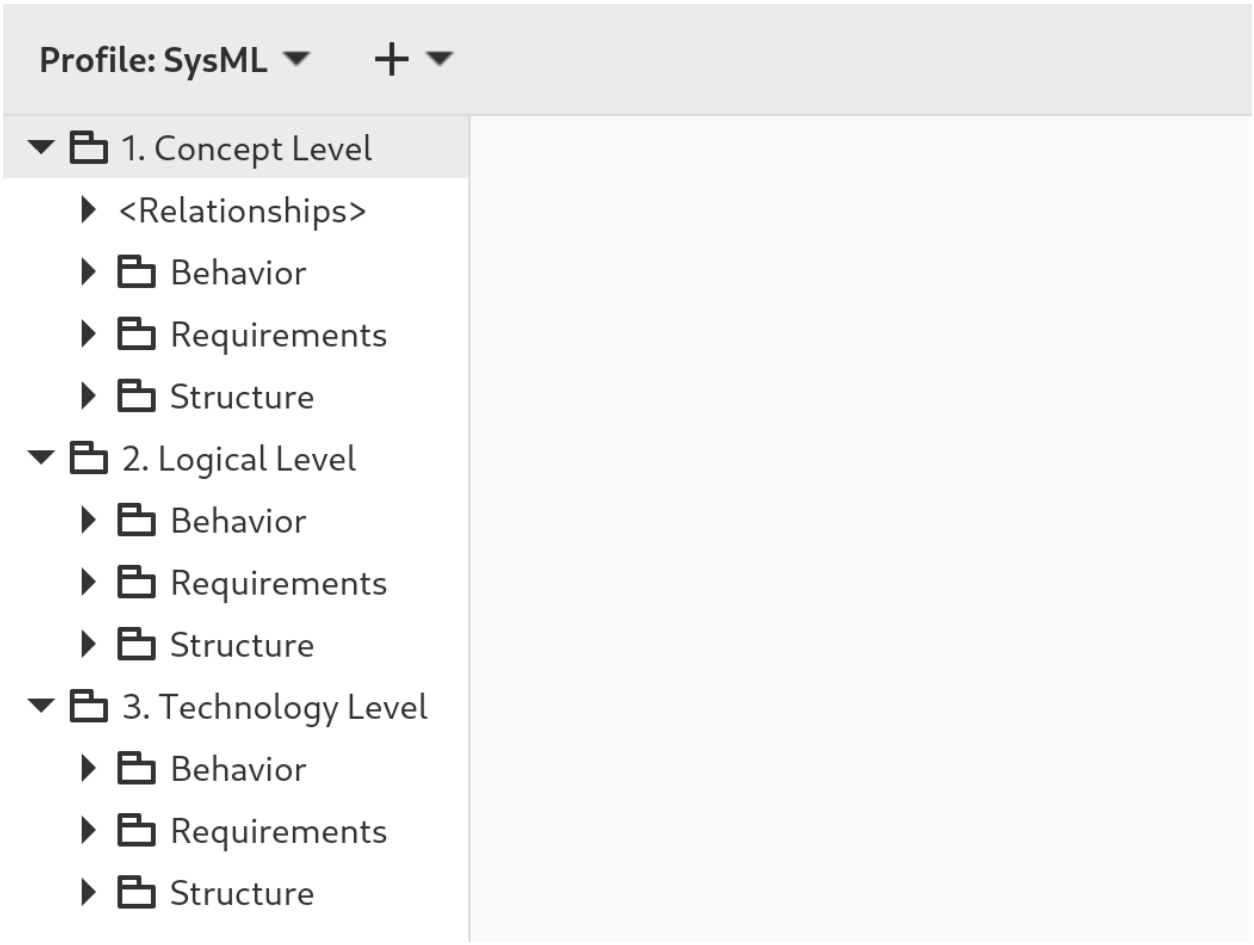
There are four pillars of SysML which help classify the types of diagrams based on what they represent:

- Behavior: The functionality of a system
- Structure: How a system is formed using parts and connections
- Requirements: Written statements that constrain the system
- Parametrics: Enforces mathematical rules across values in the system

If you want to learn more about these four pillars, there is a 30-minute video by Rick Steiner called The Four Pillars of SysML.

Since Parametrics Diagrams are one of the least used diagram types in SysML, we are going to only focus on the first three. The power of SysML comes in being able to make relationships between these three pillars. For example, by allocating behavior like an activity to an element of the structure like a block.

If you expand the top-level Abstraction Level packages in the Model Browser, each one contains three more packages, one for each pillar. It is in these packages that we will start to build up the design for the espresso machine.



3.4 Table of Contents

3.4.1 Coffee Machine: Concept Level

Introduction

The concept level defines the problem we are trying to solve. For the espresso machine, we are going to use diagrams at this abstraction level to answer questions like:

- Who will use the machine and what are their goals while using it?
- What sequence of events will a person take while operating the machine?
- What are the key features and capabilities required for the machine to perform its intended function?
- What are the design constraints and requirements that must be considered when designing the machine?
- What are the key performance metrics that the machine must meet in order to be considered successful?
- How will the machine fit into the larger context of the café, and how will it interact with other systems and components within the café?
- What are the needs of others like those marketing, selling, manufacturing, or buying the machine?

At this level, the focus is on understanding the big picture of the espresso machine and its role within the café system. The answers to these questions will help guide the design and development of the machine at the logical and technology levels of abstraction.

Use Case Diagram

First the ants work on the behavior of the system. Expand the Behavior package in the Model Browser and double-click on the diagram named Use Cases.

A use case diagram is a type of visual representation used in systems engineering to describe the functional requirements of a system, such as an espresso machine. In the context of the espresso machine, a use case diagram would be used to identify and define the different ways in which the machine will be used by its users, such as the café staff and customers.

The diagram would typically include different actors or users, such as the barista, the customer, and possibly a manager or maintenance technician. It would also include different "use cases" or scenarios, which describe the different actions that the users can take with the machine, such as placing an order, making an espresso, or cleaning the machine.

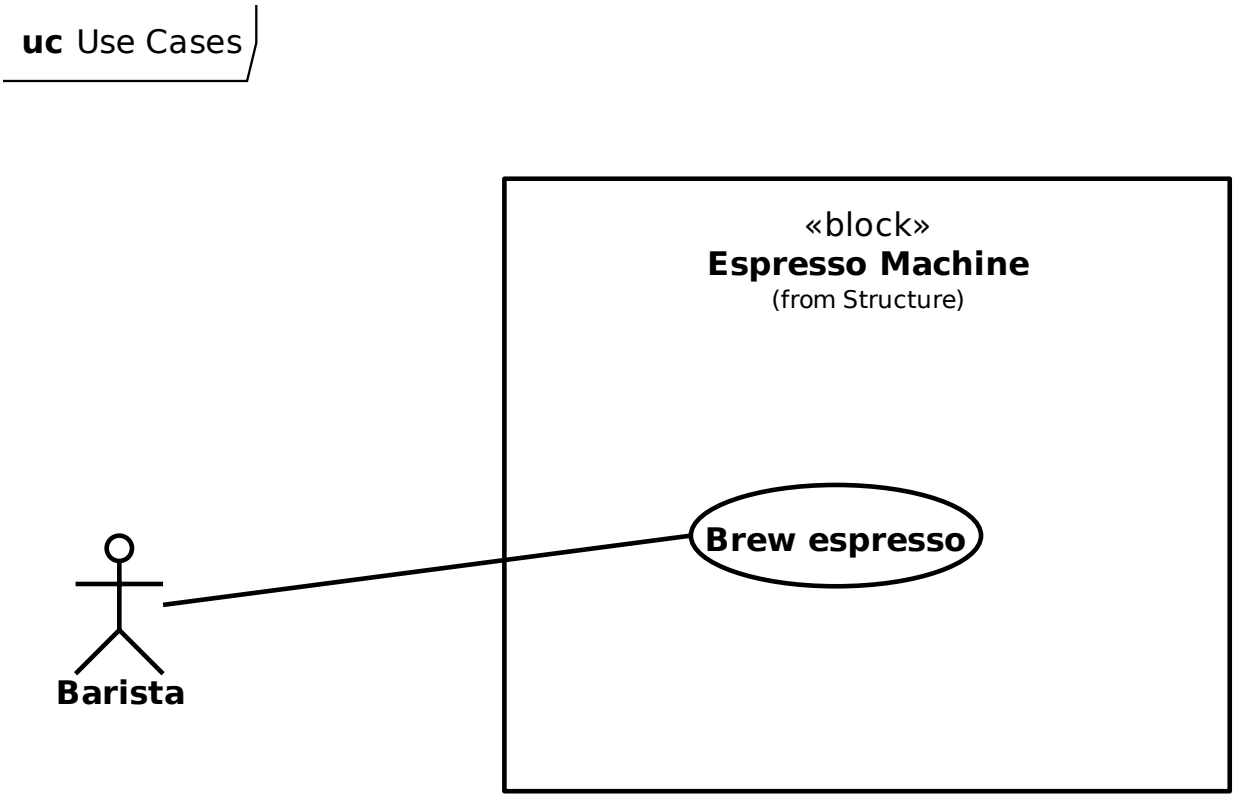
The use case diagram helps to ensure that all the necessary functional requirements of the espresso machine are identified and accounted for, and that the system is designed to meet the needs of its users. It can also be used as a communication tool between the different stakeholders involved in the development of the machine, such as the ants and Cappuccino the cat.

The ants need your help updating the diagrams, so let's get started:

1. Double-click on the actor to pop up the rename dialog, and replace User with Barista.
2. Update the name of the oval Use Case from Use Case #1 to Brew espresso.
3. Update the name of the rectangle Block from Feature to Espresso Machine

A barista interacts with the espresso machine. The barista is provided a simple interface with a few push buttons.

In this particular use case diagram, we have one actor named Barista and one use case called Brew espresso, which is allocated to a block called Espresso Machine. The actor, in this case, is a cat barista who interacts with the system (an espresso machine) to accomplish a particular task, which is brewing espresso.



The use case Brew espresso represents a specific functionality or action that the system (the Espresso Machine block) can perform. It describes the steps or interactions necessary to complete the task of brewing espresso, such as selecting the appropriate settings, starting the brewing process, and stopping the process once it's complete.

The use case diagram shows the relationship between the actor and the use case. It is represented by an oval shape with the use case name inside and an association with the actor. The association represents the interaction from the actor to the use case.

Domain Diagram

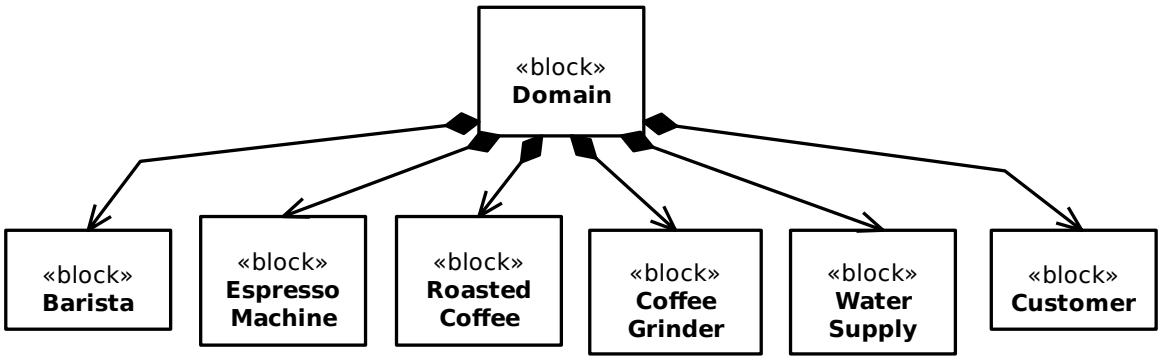
A domain diagram is a graphical representation of the concepts, terms, and relationships within a specific domain. In the case of a coffee shop, a domain diagram could represent the key elements and relationships within the coffee shop domain.

The following is a domain diagram that builds upon the context diagram with additional blocks:

- Barista
- Coffee Machine
- Roasted Coffee
- Coffee Grinder
- Water Supply
- Customer

Each block represents a key concept within the coffee shop domain, and the containment relationship is used between the domain and the blocks to show that they are part of the domain.

bdd Espresso Domain



The Barista block is responsible for preparing and serving the coffee to the customers. The Roasted Coffee block contains the types of coffee available for the barista to use. The Coffee Grinder block grinds the roasted coffee beans to the desired consistency before brewing. The Water Supply block contains the water source for the coffee machine, and finally the Customer block represents the person who orders and receives the coffee.

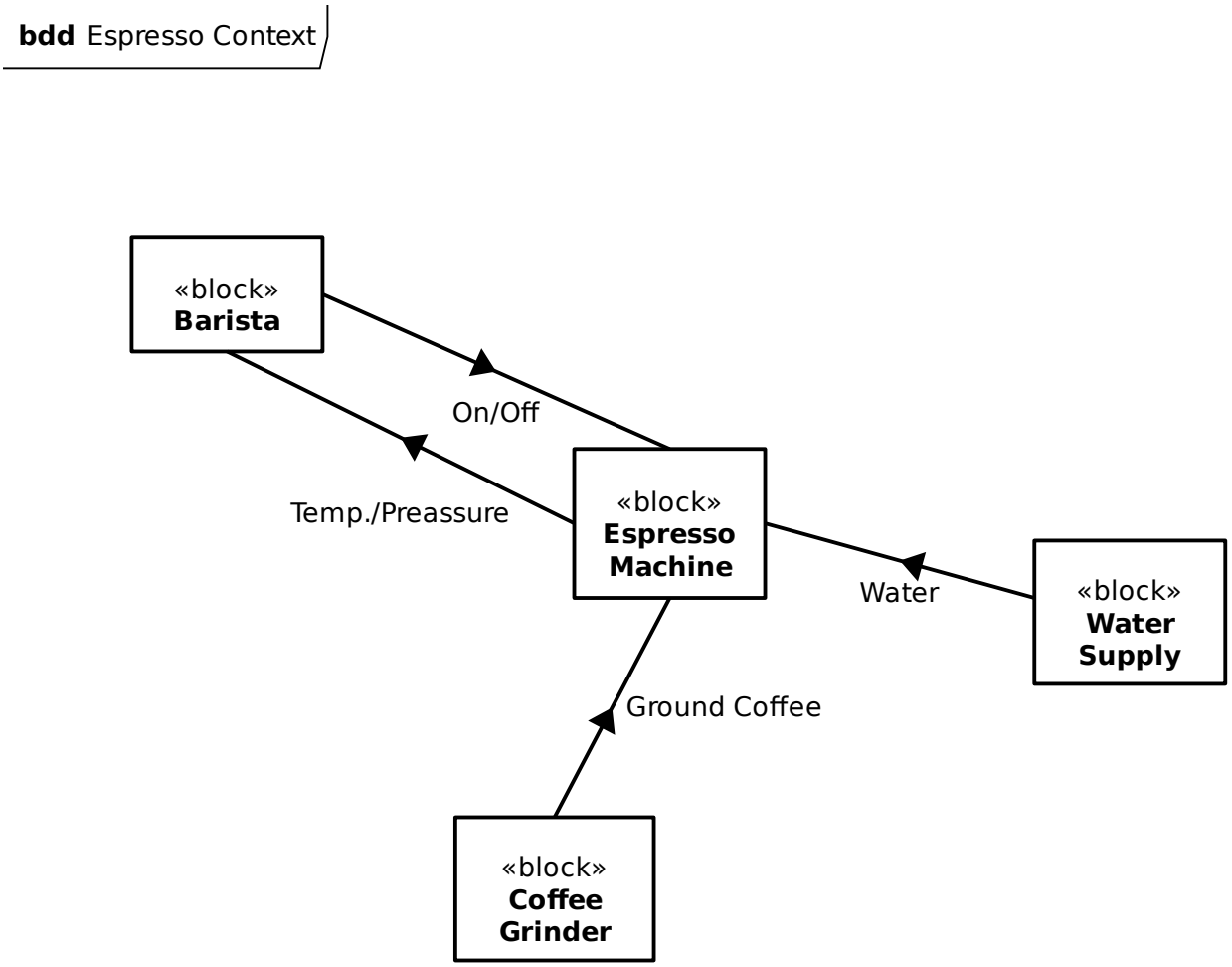
The ants need more of your help to rename the Feature Domain diagram and update it so that it matches the one above.

The domain diagram provides a high-level view of the coffee shop domain and the key concepts and relationships involved in it. It can be a useful tool for understanding the relationships between different elements of the domain and for communicating these relationships to others.

Context Diagram

The context diagram is a high-level view of the system, and it shows its interaction with external entities. In the case of a coffee machine, a context diagram provides a clear and concise representation of the system and its interactions with the external environment.

The context diagram for a coffee machine shows the coffee machine as the system at the center, with all its external entities surrounding it. The external entities include the barista, the power source, the coffee grinder, and the water source.



The ants need more of your help to rename the Feature Context diagram and update it so that it matches the one above.

Overall, the context diagram for a coffee machine provides a high-level view of the system and its interactions with external entities. It is a useful tool for understanding the system and its role in the broader environment.

Concept Requirements

Concept requirements are typically collected by analyzing the needs of the stakeholders involved in the development of the coffee machine. This involves identifying and gathering input from various stakeholders, such as the barista, the other engineers working on the product, manufacturing, and service.

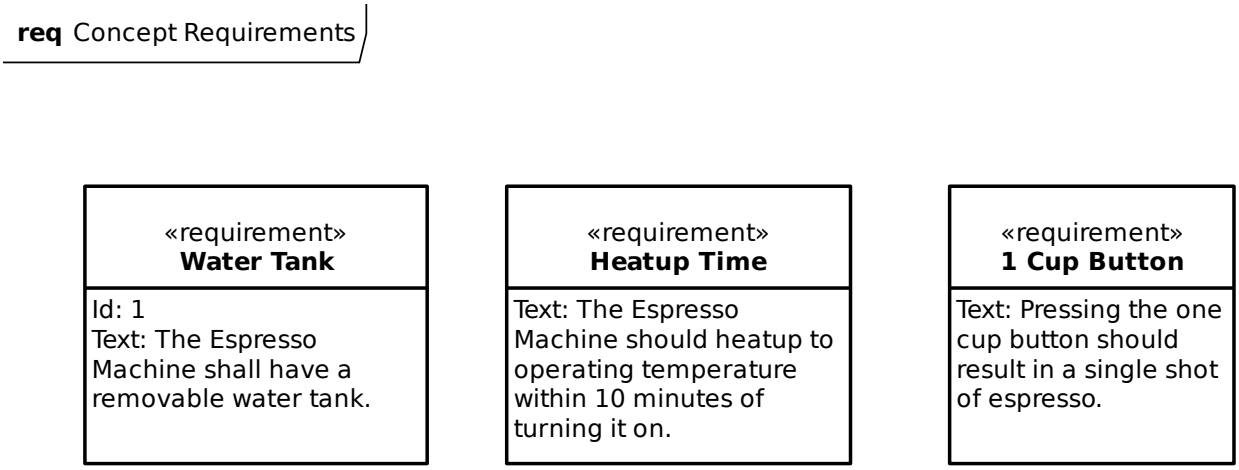
To collect concept requirements, stakeholders may be asked questions about what they want the coffee machine to do, what features it should have, and what problems it should solve. They may also be asked to provide feedback on existing coffee machines to identify areas where improvements could be made.

Once the needs of the stakeholders have been gathered, they can be analyzed to identify common themes and requirements. This information can then be used to develop the concept requirements for the coffee machine, which serve as a starting point for the design process.

The following are some concept requirements for a coffee machine that addresses a water tank, heat-up time, and HMI button:

- Water Tank: The coffee machine should have a water tank of sufficient size to make multiple cups of coffee before needing a refill. The water tank should be easy to access and fill.

- Heat-up Time: The coffee machine should have a heat-up time of no more than 10 minutes from the time the user turns on the machine until it's ready to brew coffee.
- HMI Button: The coffee machine should have an HMI with a 1 cup brew button to make it easy for the user to select the amount of coffee they want to brew. The HMI should be intuitive and easy to use.



Help the ants update the Concept Requirements diagram with these requirements.

Throughout the design process, the concept requirements will be refined and expanded upon as more information becomes available and the needs of the stakeholders become clearer. This iterative process ensures that the final design of the coffee machine meets the needs of all stakeholders and delivers a high-quality product.

3.4.2 Coffee Machine: Logical Level

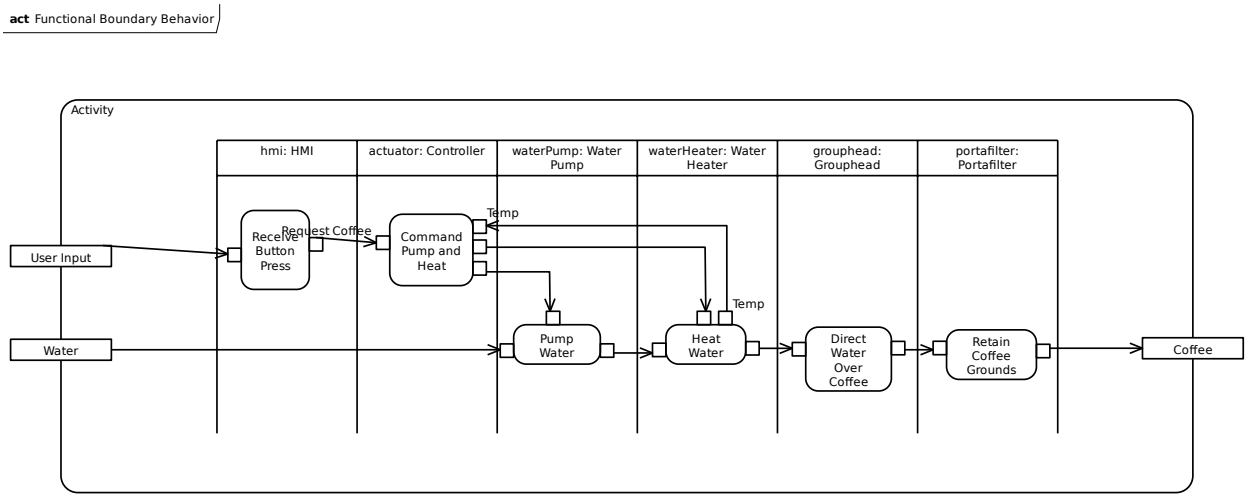
Introduction

At the logical Level, we'll define a technology-agnostic solution. This is the middle level of abstraction, where the system is described in terms of its structure and behavior. At this level, the focus is on how the system components are organized and how they interact with each other.

Functional Boundary Behavior

A Functional Boundary Behavior diagram is a type of SysML Activity diagram used to show the interactions between different logical blocks. The swim lanes divide the diagram into different areas, each representing a different functional block or component.

In this case, the diagram includes swimlanes for the HMI, Controller, Water Pump, Water Heater, Grouphead, and Portafilter. The HMI receives the button press from the barista and then sends a command to the Controller. The Controller then commands the Water Pump and Water Heater to start, and once the water has reached the correct temperature, the Controller commands the Pump and Heater to start. The water would then be pumped through the Grouphead and into the Portafilter, brewing the coffee. The diagram shows the flow of information and actions between the different logical blocks, and help to ensure that the behavior that each block provides is properly connected and integrated into the system.



From the Logical package, expand the Behavior package in the Model Browser and double-click on the diagram named Functional Boundary Behavior. Additional swimlanes can be added by clicking on the swimlanes and add additional partitions in the Property Editor.

In the Structure package, right-click on the Blocks with the B symbol and rename them from the context menu so that the names of the Logical Blocks in each swimlane are correct. The name of the partition before the colon can also be changed in the Property Editor.

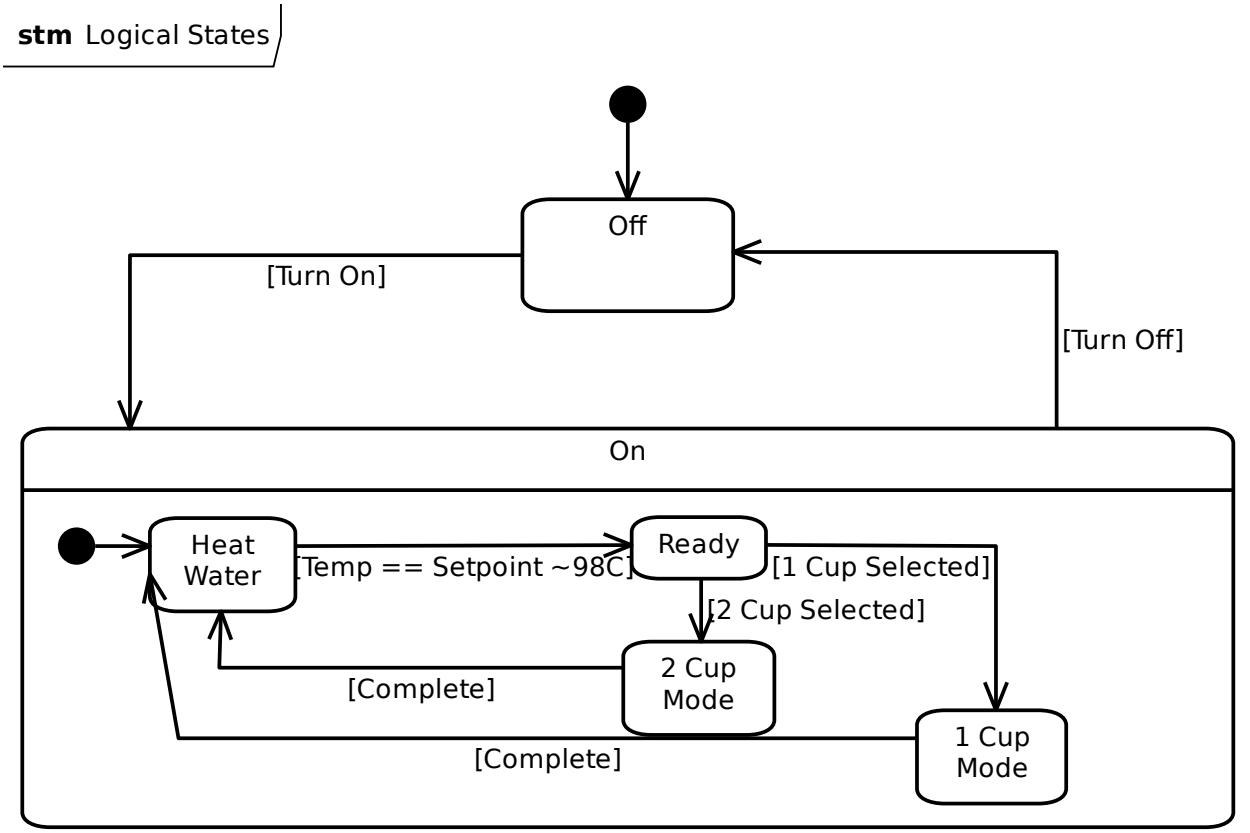
Additional Object Flows, pins, and actions can be created using the Toolbox. The Parameter Nodes which are attached to the Activity on the very left and right of the diagram are renamed and created by clicking on the Activity and modifying them in the Property Editor.

Logical State Machine

The logical state machine for the coffee machine is a diagram that shows the different states and transitions that the machine goes through to make coffee. In this case, there are two main states: On and Off.

When the coffee machine is turned on, it enters the On state. Inside the On state, there are some substates, starting with the heat water state. The machine will transition from the heat water state to the ready state when the temperature reaches the set point.

Once the machine is in the ready state, the user can select one or two cup mode. Depending on the mode selected, the machine will transition to either the one cup mode or two cup mode.



Open the Logical States diagram and use the Toolbox to add the additional substates and transition. Guards for the transitions, shown surrounded by brackets, are added by selecting the transition and adding the guard in the Property Editor.

The logical state machine diagram for the coffee machine shows these states, and the different conditions that trigger the transitions. This helps the ants designing the machine to understand how the coffee machine works and ensure that it functions properly.

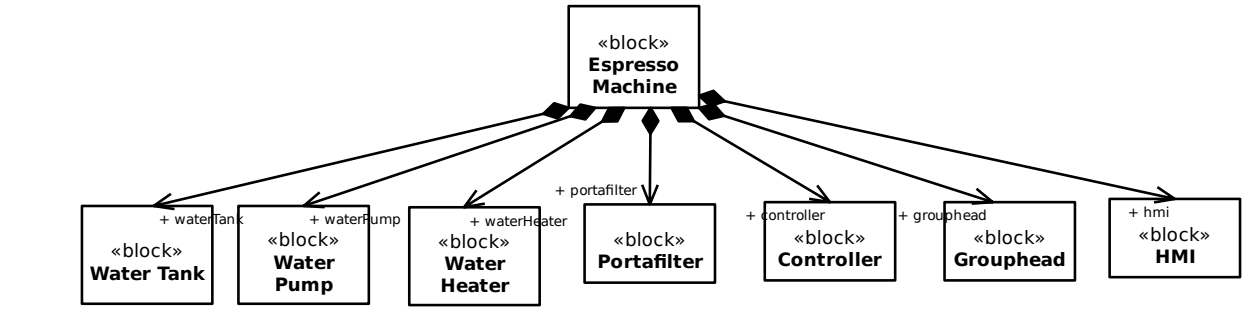
Logical Structure

The logical structure shows which logical blocks the espresso machine is made up of. Since we are at the logical level, these blocks should be agnostic to technical choices.

The following logical blocks are part of the espresso machine:

- Water tank
- Water pump
- Water heater
- Portafilter
- Controller
- Grouphead
- HMI

Each block represents a key portion of the espresso machine, and the containment relationship is used between the espresso machine and its logical parts.



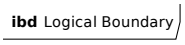
- Water tank: The water tank is a container that stores the water used in the espresso machine. It typically has a specific capacity and is designed for easy filling and cleaning. The water tank supplies water to the water pump when needed.
- Water pump: The water pump is responsible for drawing water from the water tank and creating the necessary pressure to force the water through the coffee grounds in the portafilter. It plays a crucial role in the espresso extraction process by ensuring a consistent flow of water.
- Water heater: The water heater, also known as the boiler or heating element, is responsible for heating the water to the optimal temperature for brewing espresso. It maintains the water at the desired temperature to ensure proper extraction and flavor.
- Portafilter: The portafilter is a detachable handle-like device that holds the coffee grounds. It is attached to the espresso machine and acts as a filter holder. The water from the pump is forced through the coffee grounds in the portafilter to extract the flavors and create the espresso.
- Controller: The controller, often a microcontroller or a dedicated circuit board, is the brain of the espresso machine. It manages and coordinates the operation of various components, such as the water pump, water heater, and HMI, to ensure the correct brewing process. It monitors and controls temperature, pressure, and other parameters to maintain consistency and deliver the desired results.
- Grouphead: The grouphead is a part of the espresso machine where the portafilter attaches. It provides a secure connection between the portafilter and the machine, allowing the brewed espresso to flow out of the portafilter and into the cup. The grouphead also helps to maintain proper temperature and pressure during the brewing process.
- HMI (Human-Machine Interface): The HMI is the user interface of the espresso machine. It provides a means for the user to interact with the machine, usually through buttons, switches, or a touchscreen. The HMI allows the user to select different brewing options, adjust settings, and monitor the status of the machine. It provides feedback and displays information related to the brewing process, such as brewing time, temperature, and cup size selection.

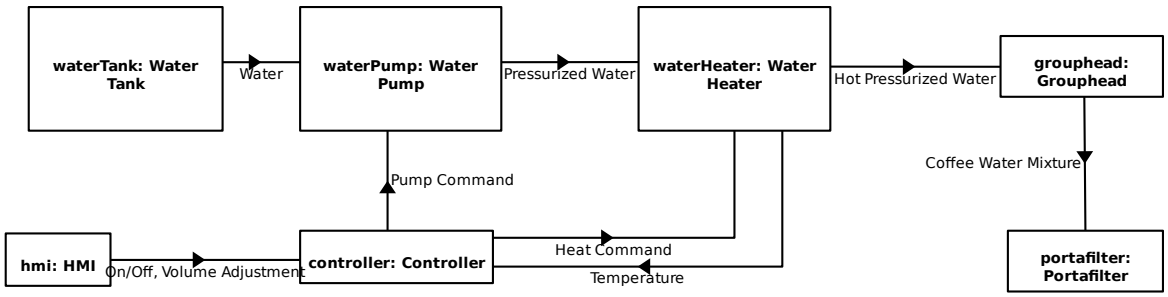
We didn't make any technical choices at this time, for example we didn't specify which type of controller, the pump capacity, or the model of the grouphead. These details will be defined once we get to the Technology level.

The ants need more of your help to update the Logical Structure diagram so that it matches the one above.

Logical Boundary

The Logical Boundary is a type of Internal Block Diagram that represents the internal structure of a system, illustrating the relationships between its internal components or blocks. It helps to visualize how these blocks interact and exchange information within the system. The term boundary used here means a clear box view inside the espresso machine at the logical boundary. It uses part properties of the blocks that were in the Logical Structure diagram above.





The interactions between the part properties inside the espresso machine are shown as ItemFlows on the Connectors.

- Water: Represents the flow of water from the water tank to the water pump.
- On/Off: Represents the command or signal to turn the espresso machine on or off.
- Volume Adjustment: Represents the user-selected volume adjustment for the coffee output.
- Pressurized Water: Represents the water flow under pressure for extracting coffee.
- Heat Command: Represents the command or signal to activate the water heater and initiate the heating process.
- Temperature: Represents the feedback signal indicating the current temperature of the water.
- Hot Pressurized Water: Represents the pressurized hot water for brewing coffee.
- Coffee Water Mixture: Represents the mixture of hot water and coffee grounds during the brewing process.

Attention: Notice that we aren't actually showing anything entering or leaving the boundary of the espresso machine, like the input from the barista or the resulting coffee. Gaphor doesn't current support adding ports to the boundary of an internal block diagram, but hopefully we'll be able to add support soon!

These item flows capture the essential interactions and exchanges within the espresso machine. They represent the flow of water, control signals, temperature feedback, and the resulting coffee water mixture. The item flows illustrate the sequence and connections between the various components, allowing for a better understanding of how the machine functions as a whole.

Once again, help the ants by updating the Logical Boundary diagram so that it matches the one above.

Logical Requirements

Logical requirements refer to the high-level specifications and functionalities that describe what a system or product should accomplish without specifying how it will be implemented. These requirements focus on the desired outcomes and behavior of the system rather than the specific technical details.

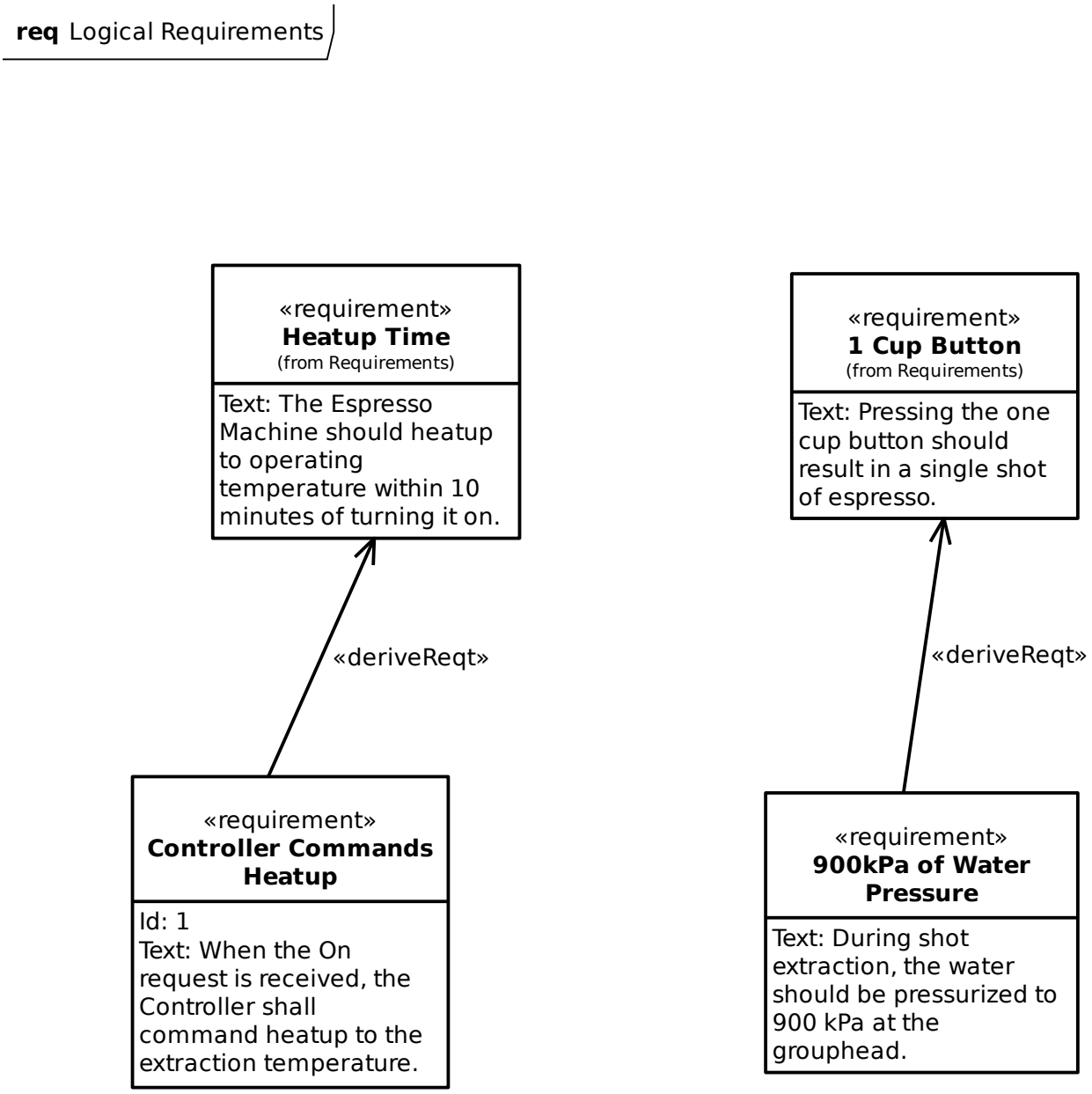
We have also already defined the behavior and the structure of the espresso machine at the logical level, so the main task now is to translate that information in to words as requirement statements.

Tip: If you need help writing good requirements, the INCOSE Guide to Needs and Requirements and the Easy Approach to Requirements Syntax are recommended resources.

We use the Derive Requirement relation from the Logical Requirement to the Concept Requirements that we previously created. The direction of this relationship is in the derived from direction, which might be opposite to what you are used to where the higher level requirement points to the lower level requirement.

Here we derive two requirements:

- Controller commands heatup
- 900kPa of water pressure



Update the Logical Requirements diagram with these requirements. If you want, you can also develop additional requirements for all the logical behavior and structure that we specified in the other diagrams.

3.4.3 Coffee Machine: Summary

The Technology Level design uses a very similar approach as the Logical Level. Work on the behavior, structure, and then the requirements. At this level, you will now specify all the design details for how this specific espresso machine will work. We'll leave this exercise up to you to do, and we would be glad to have contributions of this design back in to this tutorial if you are interested in getting involved in Gaphor.

As they worked, the ants encountered numerous challenges. They had to ensure that the machine was safe, efficient, and easy to use, all while meeting the unique needs of their feline client. But with their deep understanding of systems engineering and their commitment to key principles and concepts, they were able to overcome these challenges and design an exceptional espresso machine.

In the end, Cappuccino was thrilled with the machine, which worked flawlessly and was a big hit with his customers. He was so impressed with the ants' work that he offered them a long-term contract to design all of his café's systems. The ants were proud of their success, knowing that it was all thanks to their expertise and deep understanding of systems engineering principles. They had proven that, with the right tools and approach, anything is possible.

CHAPTER

FOUR

CHANGE LOG

This is a curated list of the changes per version.

Note: The latest version may not have been released yet.

4.1 2.24.0

- Fine grained CSS styling for model elements
- Gaphor is now REUSE compliant
- Connected elements are no longer automatcally removed from a diagram
- Fix header bars for various windows
- Various smaller UX improvements: greeter, diagram background, empty parameters
- Install schemas by running `gaphor install-schemas`
- Mitigations for false virus warnings on Windows
- Fix issue when undoing fails

4.2 2.23.2

- Fix models failing to load for collection not hashable TypeError
- Fix file filters for export image dialog
- Add CSS for “from: “, stereotypes, and compartments
- Update Hungarian translations

4.3 2.23.1

- Fix Activity Swimlanes aren't visible
- Fix CSS attribute rules
- Remove duplicate stereotypes at root of model browser
- Update Croatian translations

4.4 2.23.0

- Support types for parameters
- Restore windows in maximized and full-screen state
- Fine grained CSS: elements in a presentation item can be changed from CSS (experimental)
- Very long element names are now wrapped
- Format files accessed from Flatpak via portals
- Update Property Editor to use Gtk.ColumnView and ListView
- Replace deprecated Gtk.FileChooser by FileDialog
- Replace in-app notifications by Adwaita Toasts
- macOS: Update app icon
- macOS: Fix icon in Property Editor isn't displayed
- macOS and Windows: Apply custom window shadow
- Fix Gaphor should work if GSettings schema is not available
- Fix values showing under part compartment in SysML Block elements
- Fix merge-node icon
- Fix connecting lines when model is loaded and documentation updates
- Fix macOS build and documentation generation
- Update Russian, Chinese, Polish, Hungarian, Turkish and German translations

4.5 2.22.1

- Flatpak: Fix app preferences aren't saved
- Fix TypeError when changing to dark mode and refactor settings
- Fix grouping: allow to group to "root"
- Add support for dropping SysML diagrams on diagrams
- Catch errors when a clipboard is empty
- Fix DnD file opening on macOS
- Fix libadwaita 1.4.0 missing for hypothesis tests
- Make ActivityParameterNode droppable

4.6 2.22.0

- Proxy port improvements
- Add preferences for overriding dark mode and language to English
- Add allocations toolbox with allocate relationship item
- Add members in model browser
- Make line selection easier by increasing tolerance
- Make model loading more lenient
- Remove duplicated elements in Component.provided property
- Replace Black, check toml, isort, and refurb with Ruff
- macOS: fix About dialog links
- macOS: upgrade notarization from altool to notarytool
- Upgrade to libadwaita 1.4.0
- Update Sphinx to version 6.0
- Update PyInstaller to version 6.1
- Clean up transactional event handling
- Use defusedxml to avoid loading potentially dangerous xml
- Update Portuguese (Brazilian), Italian, Tamil,

4.7 2.21.0

- Add picture as core element
- Move diagram elements with the arrow keys
- Add interface block to element creation menu
- Support state entry, exit, and do behavior selection via dropdown
- Add feature to align diagram elements
- Display type of element in the properties panel
- Removed unnecessary operations and attributes for requirements
- Enable macOS keybindings again
- Fix missing derive relationship icon in the model browser
- Fix block not showing parts and references
- Fixed Profile is created instead of Stereotype in model browser
- Add ownership rules to DirectedRelationshipPropertyPath.targetContext
- Add tests and fix `Component.required`
- Present the Greeter, instead of only making it visible
- Fix derive-reqt model browser icon
- Improve coverage reporting

- Add system style sheet to the documentation

4.8 2.20.0

- Add ValueSpecificationAction
- New element creation through model browser
- Interface block support on diagram
- Constrain SysML diagram creation in the model explorer to conform with SysML 1.6 specification
- Add type selection for Lifelines
- Support SysMLDiagram type and diagram type specific header formatting
- Pin type multiplicity
- Deep copy for packages and diagrams
- Add Direct association to toolbar menu
- Add CallBehaviorAction to Activity profile
- Add operations compartment to Blocks
- Toggle visibility of ProxyPort type
- Provide value for 'Show value' in properties page
- Format pins by their name
- Do not remove unused Packages with children
- Fix: tree view should not collapse when an element is deleted
- Fix: only open model browser elements with a model element
- Upgrade Gvsbuild to 2023.7.1
- Update minimal Python version to 3.10
- Fix: ensure a newly placed item is no longer a dropzone item
- Fix: incorrect filling between shapes
- Fix: weird pin rendering
- Fix: diagram background shouldn't be shared between open models
- Fix: activity parameter node is always stuck to the activity when moving
- Fix: notes should be applied to model elements and will be named "Note"
- Fix: error when inverting association
- Update Croatian translations

4.9 2.19.3

- Windows: Fix missing toolbar icons
- Fix loading of ProxyPorts with informationFlow attached
- Fix to resolve CSS style variables before using the values

4.10 2.19.2

- Add SysML Requirements trace derived unions
- Fix Parameter Node and Execution Specification with Dark mode
- Scale parameter nodes to contain long names
- Lenient derived unions
- Fix segfault by reverting Gtk.ListVew for Parameters
- Fix connect interaction fragments
- chore: clean up deprecated properties from UIComponent
- Add Python 3.12 Support, Update Poetry to version 1.5.1
- Apply security best practices to GitHub Actions
- Create a Security Policy and Run Scorecard Checks
- Only use mature translations for released versions of Gaphor
- Update Spanish, Hungarian, and Finnish translations
- Fix scaling of Activity Parameter nodes

4.11 2.19.1

- Fix: order is preserved when undoing a change
- Actions: ObjectNode now also connects to decision/merge and join/fork nodes.
- Fix: ports should be nested under properties
- Experimental: support for plugins
- Add fullscreen mode with F11
- Remove Tkinter from packaging

4.12 2.19.0

- Dropped support for AppImage
- Add Information Flow support for Associations
- Interactions: fixed DnD for partially connected messages
- Restore CSS auto-complete
- Docs: A coffee machine tutorial has been added
- Make model loading more lenient to model corruption
- CLI: export diagrams and run scripts within Gaphor
- Enable PyPI Trusted Publisher
- Replace deprecated Gtk.TreeView with ListView: Activity Parameter Nodes
- Use consistent naming for element_factory in storage module
- Use new style Dropdowns for selecting items in property editor
- Updates to translations

4.13 2.18.1

- Make operations visible on Blocks
- A quick fix for crashes in the CSS editor, disable autocomplete
- Fix doc translation catalogs not found
- Fix encoding warnings for no encoding argument
- Update AppImage build with GTK 4.10
- Add non-goals to README
- Enable translation of docs, and add Crotian, German, and Dutch
- Updates to most translations and add Tamil

4.14 2.18.0

- Support for manually resolving Git merge conflicts
- Drop support for GTK3, bundle macOS with GTK4
- Enable middle-click mouse scrolling of diagrams
- Support for changing the spoken language in a model
- Add diagrams in diagrams
- Upgrade development build to GNOME 44
- Clean up application architecture for copy service
- Css editor dark mode
- macOS: update notarize, staple, and cert actions

- Just load modules for gaphor.modules entry points
- Finnish, Dutch, Spanish, Polish, Portuguese (BRA) translation update
- Toggle the “no tabs” background based on notebook activity
- Fix drag from model browser
- Fix orthogonal lines during copy/paste
- Update diagram directly when partitions change
- Make main window always available to avoid warnings

4.15 2.17.0

- Add support for diagram metadata
- macOS: Fix freeze creating new diagram
- Properly unsubscribe when property page is removed
- Package GSettings daemon schemas for AppImage
- Consider only default modifiers in toolbox shortcuts
- New status page icon
- Workaround removing skip-changelog labels
- Update gvsbuild to version 2023.2.0
- meta: Add .doap-file
- Update “Keep model in sync” design principle
- Update to using the GNOME Code of Conduct
- Add Polish translation
- Spanish, Dutch, Croatian, Turkish, and German Translation Updates

4.16 2.16.0

- Automatic switching to dark mode in diagrams
- Add Model browser multi select and popup menu
- Refactor and improve model browser
- Use normal + icon for new diagram dropdown
- Add support for CSS variables and named colors
- Apply development mode for dev releases
- Show diagram name in header
- Show something when no diagrams are opened
- Fix the packaged data dirs
- Stabilize macOS/GTK tests
- Add a comments option to our documentation

- Split tips in to multiple labels
- Win and macOS: Fix wrong language selected when region not default
- Fix translation warning never logged with missing mo files
- Spanish, Russian, Hungarian, Czech translation update

4.17 2.15.0

- Add basic git merge conflict support by asking which model to load
- Improvements to CSS autocomplete with function completion
- Insert colons, spaces, and () automatically for CSS autocomplete
- Use native file chooser in Windows
- Fix translations not loading in Windows, macOS, and AppImage
- Fix PyInstaller versionfile parse error with pre-release versions
- Update CI to publish to PyPI after all other jobs have passed
- Replace pytest-mock with monkeypatch for tests
- Fix PEP597 encoding warnings
- Fix regression that caused line handles to not snap to elements
- Add Turkish, and update French, Russian, and Swedish translations
- Remove translation Makefile

4.18 2.14.2

- Fix macOS release failed

4.19 2.14.1

- Add autocompletion for CSS properties
- Fix coredumps on Flatpak
- Hide New Package menu unless package selected
- Update Getting Started pages
- Spanish translation update

4.20 2.14.0

- Simplify the greeter and provide more info to new users
- New element handle and toolbox styles
- Use system fonts by default for diagrams
- Add tooltips to the application header icons
- Make sequence diagram messages horizontal by default
- Make keyboard shortcuts more standard especially on macOS
- macOS: cursor shortcuts for text entry widgets
- Load template as part of CI self-test
- Update docs to make it more clear how to edit CSS
- Switch doc style to Furo
- Add custom style sheet language
- Support non-standard Sphinx directory structures
- Continue to make model loading and saving more reliable
- Move Control Flow line style to CSS
- Do not auto-layout sequence diagrams
- Use new actions/cache/(save|restore)
- Remove querymixin from modeling lists
- Improve Windows build reliability by limiting cores to 2
- Croatian, Hungarian, Czech, Swedish, and Finnish translation updates

4.21 2.13.0

- Auto-layout for diagrams
- Relations to actors can connect below actor name
- Export to EPS
- Zoom with Ctrl+scroll wheel works again
- Recent files is disabled if none are present
- Windows and AppImage are upgraded to GTK4
- Update packaging to use Python 3.11
- Many GTK4 improvements: About window, diagram tabs, message dialogs
- Ensure toolbox is always visible
- Add additional tests around architectural rules
- Many translation updates and bug fixes

4.22 2.12.1

- Fix/move connected handle
- Fix error when disconnecting line with multiple segments
- Fail CI build if Windows certificate signing fails
- namespace.py: Actually set properties for rectangle
- Update Shortcuts window

4.23 2.12.0

- GTK4 is now the default for Flatpak; Windows, macOS, and AppImage still use GTK3
- Save folder is remembered across save actions
- State machine functionality has been expanded, including support for regions
- Resize of partition keeps actions in the same swimlane
- Activities (behaviors) can be assigned to classifiers
- Stereotypes can be inherited from other stereotypes
- Many GTK4 fixes: rename, search, instant editors
- Many translation updates

4.24 2.11.0

- Add Copy/Paste for GTK4
- Make dialogs work with GTK4
- Fix instant editors for GTK4
- Update list view for GTK4
- Make SysML Enumerations also ValueTypes
- Add union types
- Let Gaphor check for its own health
- Add error reports window
- Add element to diagram by double click
- Ensure all models are saved with UTF-8 encoding
- Fix states can't transition to themselves
- Fix unlinking elements from the model
- Fix issue with fully pasting a diagram
- Fix scroll speed for touch screens
- Fix codeql warnings and error
- Improve text placement for Associations

- Enable additional pre-commit hooks
- Add example in docs of color for comments using CSS
- Hungarian translation updates

4.25 2.10.0

- Pin support for activity diagram
- Add Activity item to diagram
- Allow to drag and drop all elements from tree view to diagram
- Codegen use all defined modeling languages
- Fix diagram dependency cycle
- Add Skip Duplicate Action and Release-Drafter Permissions
- Update permissions for CodeQL GitHub Action
- Include all diagram items in test model
- Fix GTK4 property page layouts
- Use official RAAML logo in greeter
- Relation metadata to allow better reuse
- Rename relationship connector base classes
- Add design principles to docs
- French, Finnish, Croatian translation update

4.26 2.9.2

- Fix Windows build

4.27 2.9.1

- Fix bad release of version 2.9.0
- Cleanup try except blocks and add more f-strings

4.28 2.9.0

- Separate Control and Object Flow
- Automatically select dark mode for macOS and Windows
- Automatically Enable Rename Prompt for Newly Created Diagrams
- New group function for element grouping
- Simulate user behavior with Hypothesis and fix uncovered bugs

- Proxyport: update ports when proxyport is moved
- Fix AppImage Crashes on Save Command
- Improve reconnect for relationships
- Update connection behavior for Association
- Enable preferences shortcut
- Rename Component Toolbox to Deployment
- Update Finnish, Spanish, Croation, and German translatio

4.29 2.8.2

- Fix splitting of lines
- Update README to reflect new functionality
- Add additional strings to translations
- Update Hungarian, Spanish, and Finnish translations

4.30 2.8.1

- Fix Gaphor fails to load when launched in German
- Simplify the greeter dialogs
- Update Hungarian, Finnish, and Chinese (Simplified) translations

4.31 2.8.0

- Add diagram type support
- Improve the welcoming experience with a greeter window and starting templates
- Add a Magnet-tool
- Support SysML Item flow
- Stereotypes for ItemFlow properties
- Full Copy/Paste of model elements
- Allow for deleting elements in the tree view
- Allocation of structural types to swimlane partitions
- User notification when model elements are automatically removed
- Store toolbox settings per modeling language
- Grow item when an item is dropped on it
- Add “values” compartment to Block item and set a minimal height for compartments
- Support empty square bracket notation in an Operation
- New code generator

- Fix AppImage GLIBC Error on Older Distro Versions
- Fix Sequence diagram loading when message is close to lifeline body
- Support for loading .gaphor files directly from the macOS Finder
- Fix positions of nested items during undo
- Fix ownership of Connector, ProxyPort, and ItemFlow
- Improve GTK4 compatibility
- Improve clarity of syntax for attributes and operations using a popover
- Clean up Toolbox and remove some legacy code
- Invert association creation
- Ensure model consistency on save and fork node loading fixes
- Core as a separate ModelingLanguage
- Use symbolic close icon for notebook tabs
- Update to latest gvsbuild, switch to wingtk repo
- Spanish, Hungarian, Finnish, Dutch, Portuguese, Croation, Espanian, and Galician translations updates
- Add Chinese (Simplified) translation

4.32 2.7.1

- Fix lines don't disconnect when moved
- No GTK required anymore for generating docs
- Update Python to 3.10.0
- Spanish translation updates

4.33 2.7.0

- Add Reflexive Message item for Interactions
- Allow messages to move freely on Lifeline and ExecutionSpec
- Pop-up element name editor on creation of a new element
- Add option to show underlying DecisionNode type
- Add InformationFlow for Connectors
- Swap relationship direction for Generalization, Dependency, Import, Include, and Extend
- Use Jedi for autocomplete in the Python Console
- Sphinx directive for embedding Gaphor models into docs
- Fix lifeline ordering when not all items are linked in a diagram
- Allow generalizations to be reused
- Allow auto-generated elements (Activity, State Machine, Interaction, Region) to be removed
- Fix Windows build by updating to Python 3.9.9

- Emit events for Diagram.ownedPresentation and Presentation.diagram after element creation
- Show underlying DecisionNode type
- Add documentation dependencies to pyproject.toml
- Move enumeration layout to UML.classes
- Rename packaging to _packaging
- Remove names for initial/final nodes
- Update to latest gvsbuild
- Update to PyInstaller 4.6
- Add gtksourceview to Windows docs
- Fix Python 3.10 warnings
- Fix indentation in Style Sheet docs
- Expand the number of strings translated
- Hungarian, Spanish, Japanese, Finnish, and Croatian translation updates

4.34 2.6.5

- Update style sheet editor to be a code editor
- Update strings to improve ability to translate
- Ensure all relationships are brought to top
- Fix errors in Italian translation which prevented model saving
- Add association end properties to editor pane
- Restore rename right click option to diagrams in tree view
- Add Japanese translation
- Update Hungarian, Croatian, and Spanish translations

4.35 2.6.4

- Fix Flatpak build failure by reverting to previous dependencies

4.36 2.6.3

- Fix about dialog logo
- Add translation of more elements
- Remove importlib_metadata dependency
- Simpler services for about dialog
- Up typing compliance to 3.9, and remove typing_extensions
- Finnish translation updates

4.37 2.6.2

- Fix localization of UI files
- Fix icons in dark mode
- Update Spanish, Finnish, Hugarian, and Portuguese (Br) translations

4.38 2.6.1

- Display guard conditions in square brackets
- Use flat buttons in the header bar
- Fix translation support
- Fix drag and drop of elements does not work on diagrams
- Fix parameter is incorrect error with ";;" in path
- Fix fork/join node incorrectly rotates
- Fix close button on about dialog doesn't work in Windows
- Fix wrong label is displayed when object node ordering is enabled
- Improve inline editor undo/redo behavior
- Fixed closing of about dialog
- Add VSCode debug instructions for Windows
- Rename usage of Partitions to Swimlanes
- Update Dutch and Hungarian Translations
- Croatian translation updates
- Simplify attribute and enumeration lookup

4.39 2.6.0

- Improve zoom and pan for mouse
- Add Finnish, Galician, Hungarian, and Korean, update Spanish translations
- Fix disappearing elements from tree view on Windows
- Convert CI from mingw to gvsbuild
- Upgrade Windows Build Script from Bash to Python
- Refactor GitHub Actions to use composite actions
- Add translations for UI files
- Add information flows to UML model
- Add extra rules to avoid cyclic references
- Fix typo in UML.gaphor
- Refactor class property pages in to multiple modules

- Fix Windows and other developer documentation updates
- Enable pyupgrade
- Update the README for Flatpak string translation
- Fix documentation build errors, update dependencies

4.40 2.5.1

- Fix app release signing in Windows and macOS

4.41 2.5.0

- Add initial support for STPA in RAAML
- Add support for notes in property pages and attributes
- Allow for diagrams to be nested under all elements
- Fix delete and undo of a diagram
- Rename C4ContainerDatabaseItem to C4DatabaseItem
- Cleanup model loading
- Change diagram item management to the element factory
- Organize and simplify element events
- Cleanup toolbox and diagram action code

4.42 2.4.2

- Fix AttributeError when creating composite associations
- Add tooltips for A and S in attribute editor
- Improve drag and drop for TreeView
- Started to add support for GTK4
- Upload Linux assets during release automatically
- Sign only builds on the master branch

4.43 2.4.1

- Fix reordering attributes and operations with drag and drop

4.44 2.4.0

- Add support for DataType, ValueType, Primitive, and Enumeration
- Model state is stored per model, restores where you left off
- Add support for Containment Relationship
- Focus already opened model when opening a model file
- Remove the New From Template option
- Upgrade toolbox to be compatible with GTK 4
- Add regression tests
- Fix build fails when GitHub Actions secrets are not available
- Fix association direction arrow is not updated

4.45 2.3.2

- Fix issue where ornaments were not show on associations after loading a model

4.46 2.3.1

- Fix scrollbars cause the diagram to disappear
- Update Italian translation
- Left align the toolbox header labels

4.47 2.3.0

- Add support for C4 model
- Add support for Fault Tree Analysis with RAAML
- Update the UML data model to align closer to version 2.5.1
- Enable arrow keys to expand and collapse namespace tree
- Allow Gaphor profiles to be copy and pasted between models
- Improve diagram drawing and scrolling speed
- Add Croatian translation
- Remove gray borders around editable text
- Complete converting all tests to pytest
- Fix guides are misaligned when top-left handle is moved
- Update development environment instructions
- Fix undo and redo does not set attributes
- Fix selection lasso is in the wrong place after scrolling

4.48 2.2.2

- Fix undo of deleted elements
- Fix requirements are missing ID and text
- Add CSS styling to dropzone and grayed out elements
- Start to remove use of inline styles

4.49 2.2.1

- Fix drawing of composite association

4.50 2.2.0

- Guide users to create valid relationships
- macOS builds are signed and notarized
- New app icon
- Improvements to copy and paste, and undo robustness
- Fix RuntimeError caused by style sheet creation
- Use EventControllers from GTK 3.24

4.51 2.1.1

- Fix copy and paste in Linux with Wayland

4.52 2.1.0

- Improve swimlane behavior
- Add auto select in tree view
- Add in-app notifications
- Improve file load and save dialogs
- Show more elements and relationships in namespace tree
- Update Italian translation
- Make lifelines and messages owned by interactions

4.53 2.0.1

- Fix Gaphor fails to launch in macOS
- Use certificate to sign Windows binaries
- Fix copy/paste issue that causes association ends to be removed
- Improve editing for inline editors (popovers)
- Fix undo on diagram items corrupts the model
- Fix UML composite and shared association tools

4.54 2.0.0

- Add initial support for SysML
- Add support for styling using CSS
- Translate to Italian
- Improve dmg for macOS
- Improve Copy/Paste for nested items
- Add new modeling language service
- Show the element editor by default
- Create completely new data model code generator
- Add part and shared associations to tool palette
- Remove unused imports, enable flake8 checks
- Update App icons
- Update animation gif in README
- Fix Windows Build Errors caused by Missing ZST Archives
- Fix installation on Windows
- Add extra diagram item tests
- Fix macOS Python version problem
- Place UML model and diagram items closer together
- Refactor Code Generator to New Module and add CLI
- Fix MSYS2 package names and disable system update
- Remove CI workaround for console plugin
- Move core modeling concepts to a separate package
- Convert Some Profile Tests to Pytest
- Speed up text rendering
- Fix tree view text to allow names with angle brackets
- Clear the clipboard when diagram items are copied
- Fix name change for activity partitions

CHAPTER

FIVE

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.

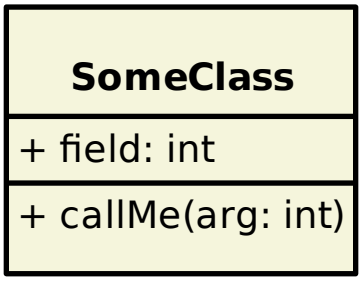
Its application goes well beyond web documents, though. Gaphor uses CSS to provide style elements to items in diagrams. CSS allows us, users of Gaphor, to change the visual appearance of our diagrams. Color and line styles can be changed to make it easier to read the diagrams.

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:

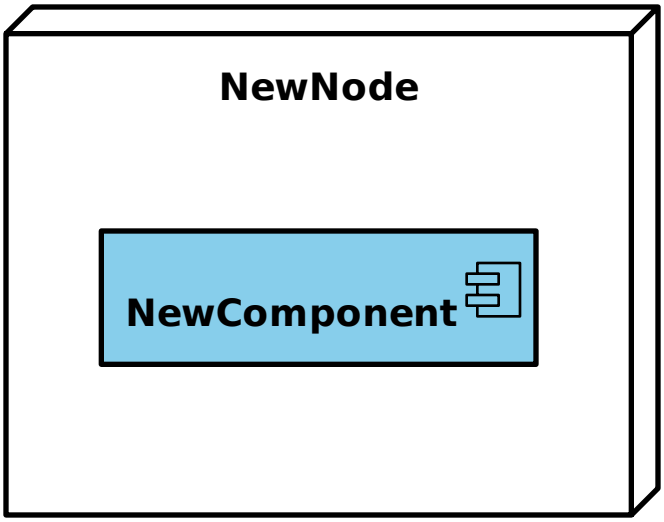
Here is a simple example of how to change the background color of a class:

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



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

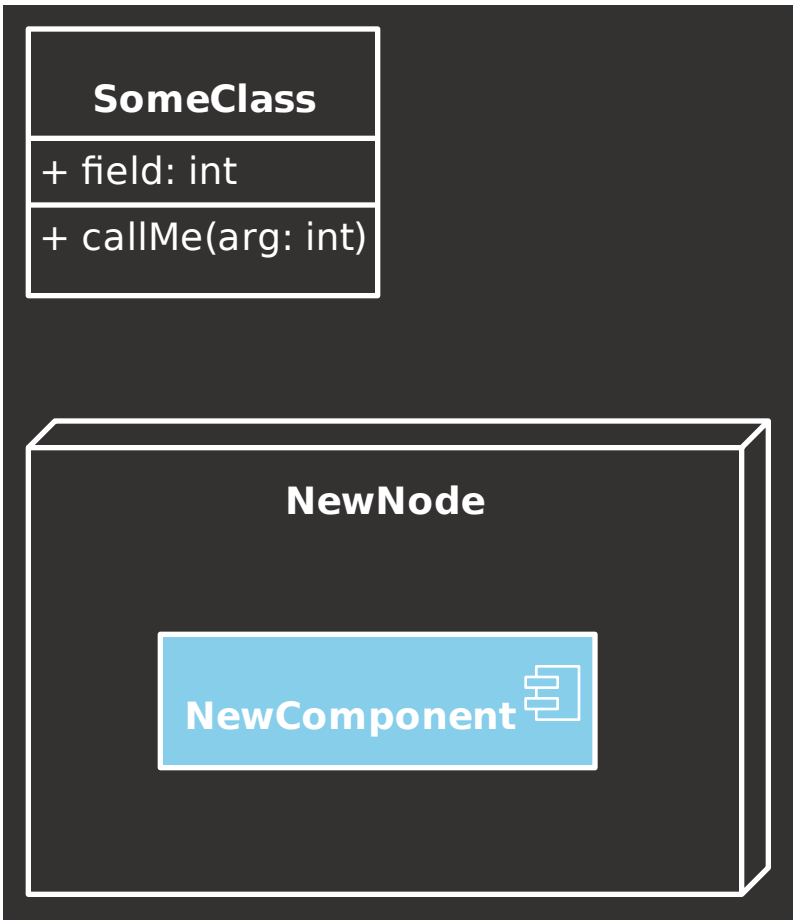
```
node component {
  background-color: skyblue;
}
```



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.

5.1 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.
`generalization[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
`:empty`	A node containing no child nodes in the diagram.
`:root`	Refers to the diagram itself. This is only applicable for the diagram
`:first-child`	A node is the first element among a group of sibling.
`: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".
`::after`	Provide extra content after a text. Only the `content` property is supported.

- 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.

5.2 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.

Some properties are inherited from the parent style. The parent often is a diagram. When you set a `color`` or a font-familyondiagram`, it will propagate down to the items contained in the diagram.

5.2.1 Colors

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

color	Color used for lines. *(inherited)*
text-color	Color for text. *(inherited)* Deprecated since version 2.23.0: Use color if possible.
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.

5.2.2 Text and fonts

font-family	A single font name (e.g. `sans`, `serif`, `courier`). *(inherited)*
font-size	An absolute size (e.g. `14`) or a size value (e.g. `small`). *(inherited)*
font-style	Either `normal` or `italic`. *(inherited)*
font-weight	Either `normal` or `bold`. *(inherited)*
text-align	Either `left`, `center`, `right`. *(inherited)*
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`.
white-space	Change the line wrapping behavior for text. *(inherited)*

- `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.

5.2.3 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`. *(inherited)*
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.

5.2.4 Pseudo elements

Currently, only the `::after` pseudo element is supported.

`content`	Extra content to be shown after a text.

5.2.5 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:

```
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.

5.2.6 Media queries

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`.

```
/* 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;
  }
}
```

(continues on next page)

(continued from previous page)

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

5.2.7 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.

5.3 CSS 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.

5.4 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.

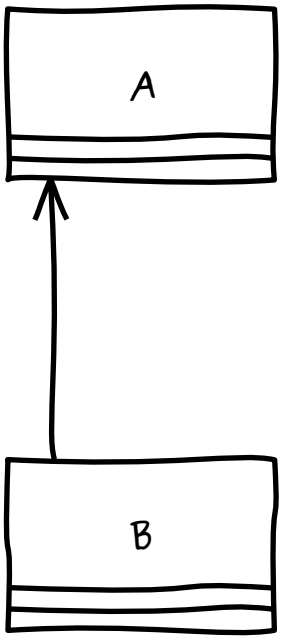
To create your own expression you may want to use the Console (→ Tools → Console). Drop us a line on Gitter and we would be happy to help you.

5.4.1 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 */
}
```



5.4.2 Unconnected relationships

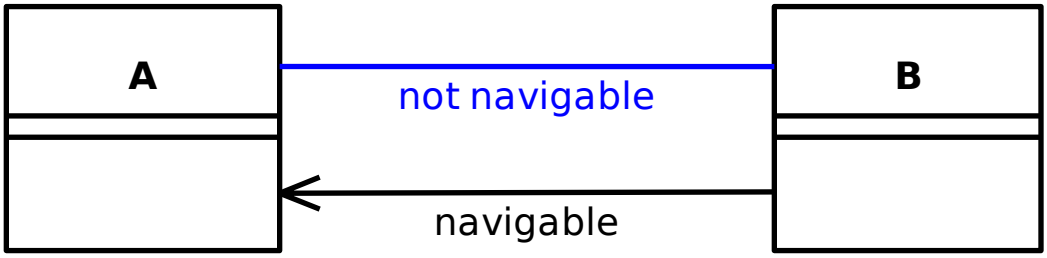
All items on a diagram that are not backed by a model element, should be drawn in a dark red color. This can be used to spot not-so-well connected relationships, such as Generalization, Implementation, and Dependency. These items will only be backed by a model element once you connect both line ends.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

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

5.4.3 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.

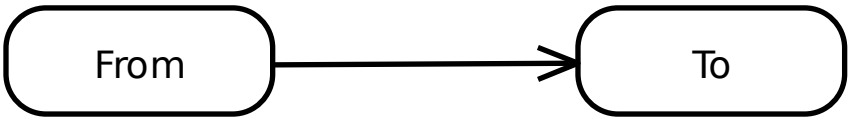
```
association:not([memberEnd.navigability*=true]) {
  color: blue;
}
```



5.4.4 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.

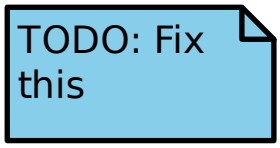
```
controlflow {
  dash-style: 0;
}
```



5.4.5 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.

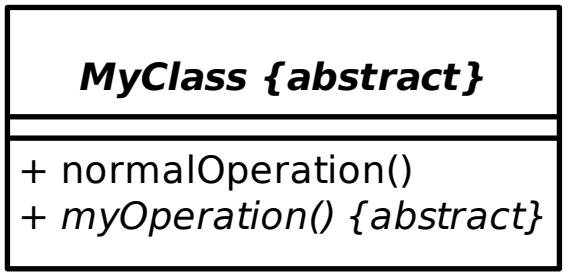
```
comment[body^="TODO"] {
  background-color: skyblue;
}
```



5.4.6 Emphesize abstract classes and operations

It may be that the italic font used is not distinguishable enough to differentiate between concrete and abstract classes or operations. To make this work we check if the `isAbstract` attribute is set on the element:

```
:is(name, operation)[isabstract]::after {
  content: " {abstract}"
}
```



5.5 System Style Sheet

```
/* Gaphor diagram style sheet */

* {
  --opaque-background-color: white;
  background-color: transparent;
}
```

(continues on next page)

(continued from previous page)

```
:drop {
  color: #1a5fb4;
  line-width: 3;
}

:disabled {
  opacity: 0.5;
}

@media light-mode {
  * {
    --opaque-background-color: #fafafa;
  }
}

@media dark-mode {
  * {
    --opaque-background-color: #242424;
    color: white;
  }

  :drop {
    color: #62a0ea;
  }
}

:root {
  color: black;
  font-family: sans;
  font-size: 14 ;
  line-width: 2;
  padding: 0;
}

:root > pentagon {
  line-width: 1;
  background-color: var(--opaque-background-color);
}

:root > pentagon > diagramtype {
  font-weight: bold;
  padding: 4 0 4 4;
}

:root > pentagon > name {
  padding: 4;
}

/* Relationships */

commentline,
```

(continues on next page)

(continued from previous page)

```
dependency,
interfacerealization,
include,
extend,
packageimport,
lifetime {
  dash-style: 7 5;
}

dependency[on_folded_interface = true],
interfacerealization[on_folded_interface = true] {
  dash-style: 0;
}

/* General */

comment {
  text-align: left;
  vertical-align: top;
  padding: 4 16 4 4;
}

comment body {
  padding: 0;
}

diagram > icon {
  padding: 4;
  border-radius: 4;
}

diagram > type {
  font-weight: bold;
}

metadata {
  justify-content: stretch;
  text-align: left;
}

metadata cell {
  padding: 4;
}

metadata heading {
  font-weight: bold;
  font-style: normal;
  font-size: small;
}

pentagon {
  padding: 4;
```

(continues on next page)

(continued from previous page)

```
  justify-content: start;
}

/* UML */

controlflow {
  dash-style: 9 3;
}

objectnode > icon {
  padding: 4 12;
}

decisionnode > type {
  font-size: small;
}

proxyport > icon,
activityparameternode,
executionspecification {
  background-color: var(--opaque-background-color);
}

partition {
  padding: 4 12 4 12;
  justify-content: stretch;
}

package {
  padding: 24 12 4 12;
}

interaction {
  justify-content: start;
}

activity {
  padding: 4 12;
  border-radius: 20;
  justify-content: start;
}

activityparameternode {
  padding: 4 12;
  min-width: 120;
  text-align: center;
}

action,
valuespecificationaction {
  padding: 4 12;
  border-radius: 15;
```

(continues on next page)

(continued from previous page)

```
}

callbehavioraction {
  padding: 4 24 4 12;
  border-radius: 15;
}

sendsignalaction {
  padding: 4 24 4 12;
}

accepteventaction {
  padding: 4 12 4 24;
}

usecase {
  padding: 4;
}

swimlane {
  min-width: 150;
  padding: 4 12 4 12;
  justify-content: start;
  white-space: normal;
}

association > end {
  font-size: x-small;
  padding: 4;
}

/* SysML */

requirement {
  justify-content: start;
}

requirement text {
  white-space: normal;
}

directedrelationshippropertypath {
  dash-style: 7 5;
}

/* Classifiers */

compartment:first-child {
  padding: 12 4;
}

compartment + compartment {
```

(continues on next page)

(continued from previous page)

```
  padding: 4;
  min-height: 8;
  text-align: left;
  justify-content: start;
  white-space: nowrap;
}

artifact compartment:first-child,
component compartment:first-child {
  padding: 12 24 12 4;
}

state compartment:first-child {
  padding: 4;
}

:has(compartment + compartment),
:has(regions),
:not([children=""]):has(compartment),
:not([children=""]) > compartment {
  justify-content: start;
}

regions {
  justify-content: stretch;
}

region {
  padding: 4;
  min-height: 100;
  justify-content: start;
  text-align: left;
}

region + region {
  dash-style: 7 3;
}

and name,
xor name,
intermediateevent name,
dormantevent name,
basicevent name,
houseevent name,
topevent name,
inhibit name,
conditionalevent name,
zeroevent name,
or name,
not name,
transferin name,
transferout name,
```

(continues on next page)

(continued from previous page)

```
undevelopedevent name,
seq name,
majorityvote name,
unsafecontrolaction name,
operationalsituation name,
controlaction name,
interfaceblock name,
block name,
property name,
requirement name,
c4person name,
c4database name,
c4container name,
package name,
enumeration name,
interface name,
class name,
datatype name,
component name,
statemachine name,
usecase name,
actor name,
artifact name,
node name {
  font-weight: bold;
}

name[isabstract] {
  font-style: italic;
}

from {
  font-size: x-small;
}

interaction > pentagon,
activity > :is(name, stereotypes) {
  text-align: left;
}

compartment heading {
  padding: 0 0 4 0;
  font-size: x-small;
  font-style: italic;
  text-align: center;
}

operation[isabstract] {
  font-style: italic;
}

attribute[isstatic],
```

(continues on next page)

(continued from previous page)

```
operation[isstatic] {
  text-decoration: underline;
}

property:not([aggregation="composite"]) {
  dash-style: 7 5;
}

/* Attached */

:has(icon)[connected_side] {
  text-align: right;
  vertical-align: top;
}

:has(icon)[connected_side="left"] {
  text-align: left;
}

:has(icon)[connected_side="bottom"] {
  vertical-align: bottom;
}

/* C4 model */

c4container, c4person {
  padding: 4 4 4 4;
}

c4database {
  padding: 20 4 4 4;
}

:is(c4container, c4database, c4person):not([children=""]) {
  justify-content: end;
}

:is(c4container, c4database, c4person):not([children=""]) > :is(name, technology) {
  text-align: left;
}

:is(c4container, c4database, c4person) technology {
  font-size: x-small;
}

:is(c4container, c4database, c4person) description {
  padding: 4 4 0 4;
}
```

CHAPTER

SIX

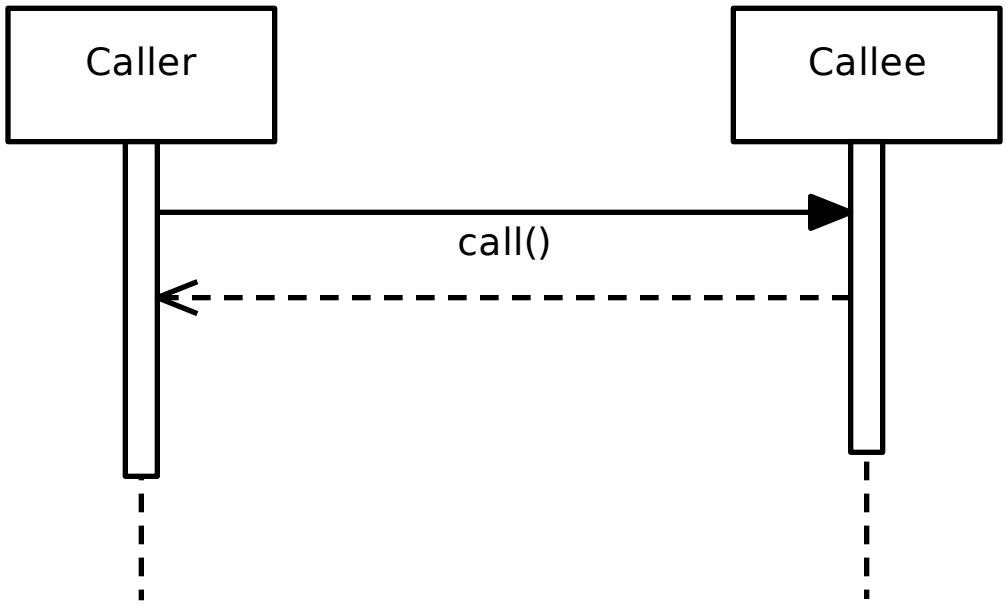
SPHINX EXTENSION

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`).

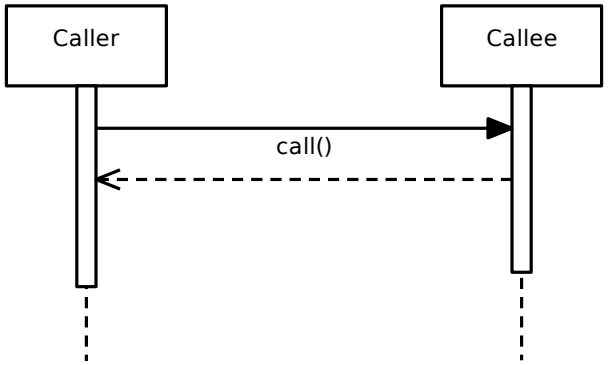
```
.. diagram:: main
   :model: example
```

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
```



6.1 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.

6.1.1 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
  jobs:
    pre_install:
    - pip install --constraint=.github/constraints.txt poetry
    - poetry config virtualenvs.create false
    post_install:
    - VIRTUAL_ENV=$READTHEDOCS_VIRTUALENV_PATH poetry install --with docs
sphinx:
  configuration: docs/conf.py
  fail_on_warning: true
```

- `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.

6.2 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

SEVEN

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.

7.1 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.

7.2 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()
```

```
Settings schema not found and settings won't be saved. Run `gaphor install-schemas`.
```

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

event_manager = EventManager()

modeling_language = ModelingLanguageService(event_manager=event_manager)

with open("../models/Core.gaphor", encoding="utf-8") 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
```

(continues on next page)

(continued from previous page)

```
RefChange
PendingChange
ChangeKind
Picture
```

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

7.3 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>
```

7.4 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.

7.5 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)
```

7.6 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!

7.7 Examples

Expanding on the information above the following snippetts show how to create requirements and interfaces.

7.7.1 Requirements from text fields

```
txts = ['req1', 'req2', 'bob the cat']
my_diagram = element_factory.create(Diagram)
my_diagram.name=' my diagram'
reqPackage = element_factory.create(UML.Package)
reqPackage.name = "Requirements"
drop(reqPackage, my_diagram, x=0, y=0)

for req_id,txt in enumerate(txts):
    my_class = element_factory.create(SysML.sysml.Requirement)
    my_class.name = f"{req_id}-{txt[:3]}"
    my_class.text = f"{txt}"
    my_class.externalId = f"{req_id}"

    drop(my_class, my_diagram, x=0, y=0)

with open(outfile, "w") as out:
    storage.save(out, element_factory)
```

7.7.2 Interfaces from dictionaries

```
# get interface definitions from file into this dictionary format
interfaces = {'Interface1': ['signal1:type1', 'signal2:type1', 'signal3:type1'],
              'Interface2': ['signal4:type2', 'signal5:type2', 'signal6:type2']}


my_diagram = element_factory.create(Diagram)
my_diagram.name=' my diagram'
intPackage = element_factory.create(UML.Package)
intPackage.name = "Interfaces"
drop(intPackage, my_diagram, x=0, y=0)

for interface,signals in interfaces.items():
    my_class = element_factory.create(UML.uml.Interface)
    my_class.name = f"{interface}"
    for s in signals:
        my_attr = element_factory.create(UML.Property)
        name,vtype = s.split(':')
        my_attr.name = name
        my_attr.typeValue = vtype
        my_class.ownedAttribute = my_attr

    drop(my_class, my_diagram, x=0, y=0)
```

(continues on next page)

(continued from previous page)

```
with open(outfile, "w") as out:
    storage.save(out, element_factory)
```

Here is another example:

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
```

```
Settings schema not found and settings won't be saved. Run `gaphor install-schemas`.
```

```
{'align': gaphor.plugins.align.align.AlignService,
 '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,
 'model_browser': gaphor.ui.modelbrowser.ModelBrowser,
 'model_changed': gaphor.ui.modelchanged.ModelChanged,
 'modeling_language': gaphor.services.modelinglanguage.ModelingLanguageService,
 'properties': gaphor.services.properties.Properties,
```

(continues on next page)

(continued from previous page)

```
 '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.

```
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
```

```
{'align': <gaphor.UML.uml.Component element 30622a7c-ec7b-11ee-b289-0242ac110002>,
 'auto_layout': <gaphor.UML.uml.Component element 3062331e-ec7b-11ee-b289-0242ac110002>,
 'component_registry': <gaphor.UML.uml.Component element 306234c2-ec7b-11ee-b289-
↪0242ac110002>,
 'console_window': <gaphor.UML.uml.Component element 3062363e-ec7b-11ee-b289-
↪0242ac110002>,
 'copy': <gaphor.UML.uml.Component element 30623788-ec7b-11ee-b289-0242ac110002>,
 'diagram_export': <gaphor.UML.uml.Component element 306238b4-ec7b-11ee-b289-
↪0242ac110002>,
 'diagrams': <gaphor.UML.uml.Component element 306239d6-ec7b-11ee-b289-0242ac110002>,
 'element_dispatcher': <gaphor.UML.uml.Component element 30623b20-ec7b-11ee-b289-
↪0242ac110002>,
 'element_editor': <gaphor.UML.uml.Component element 30623c9c-ec7b-11ee-b289-
↪0242ac110002>,
 'element_factory': <gaphor.UML.uml.Component element 30623db4-ec7b-11ee-b289-
↪0242ac110002>,
 'event_manager': <gaphor.UML.uml.Component element 30623eb8-ec7b-11ee-b289-0242ac110002>
↪,
 'export_menu': <gaphor.UML.uml.Component element 30623fbc-ec7b-11ee-b289-0242ac110002>,
 'file_manager': <gaphor.UML.uml.Component element 306240b6-ec7b-11ee-b289-0242ac110002>,
 'main_window': <gaphor.UML.uml.Component element 306241b0-ec7b-11ee-b289-0242ac110002>,
 'model_browser': <gaphor.UML.uml.Component element 306242aa-ec7b-11ee-b289-0242ac110002>
↪,
 'model_changed': <gaphor.UML.uml.Component element 3062439a-ec7b-11ee-b289-0242ac110002>
↪,
 'modeling_language': <gaphor.UML.uml.Component element 3062448a-ec7b-11ee-b289-
↪0242ac110002>,
 'properties': <gaphor.UML.uml.Component element 3062457a-ec7b-11ee-b289-0242ac110002>,
 'recent_files': <gaphor.UML.uml.Component element 3062466a-ec7b-11ee-b289-0242ac110002>,
 'sanitizer': <gaphor.UML.uml.Component element 3062475a-ec7b-11ee-b289-0242ac110002>,
```

(continues on next page)

(continued from previous page)

```
 'toolbox': <gaphor.UML.uml.Component element 3062484a-ec7b-11ee-b289-0242ac110002>,
 'tools_menu': <gaphor.UML.uml.Component element 3062499e-ec7b-11ee-b289-0242ac110002>,
 'undo_manager': <gaphor.UML.uml.Component element 30624aac-ec7b-11ee-b289-0242ac110002>,
 'xmi_export': <gaphor.UML.uml.Component element 30624ba6-ec7b-11ee-b289-0242ac110002>}
```

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

EIGHT

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.

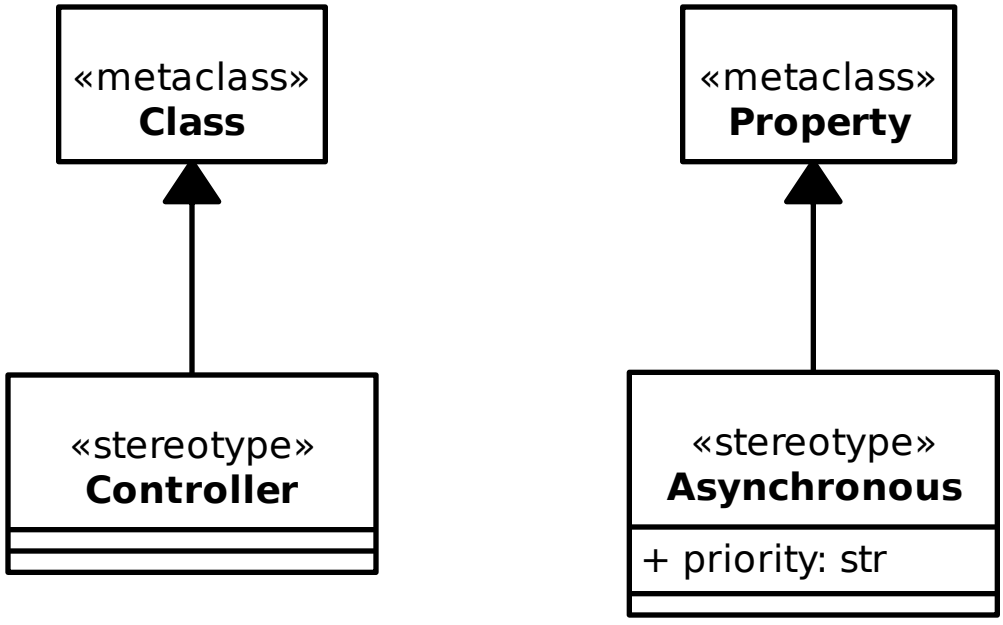
Gaphor supports stereotypes too. They're *the* way for you to adapt your models to your specific needs.

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.

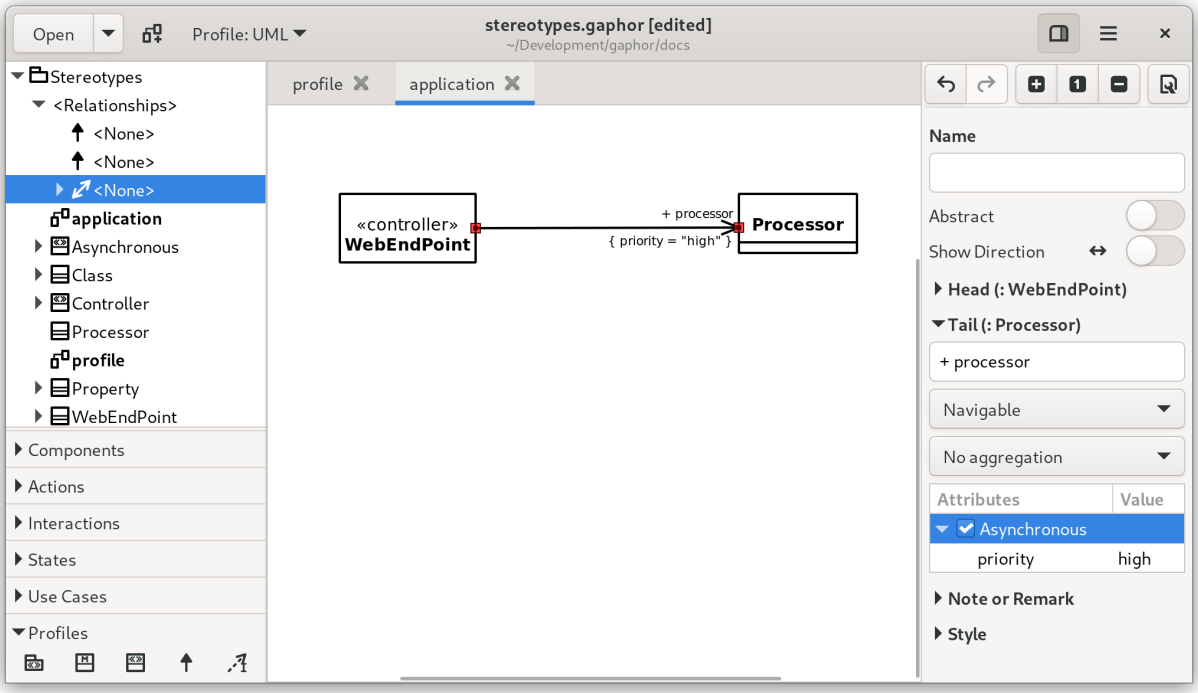
Note: The class names in the metaclass should be a class name from the UML model, such as `Class`, `Interface`, `Property`, `Association`. Or even `Element` if you want to use the stereotype on all elements.

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.



8.1 Creating a profile

In SysML extending the profile using stereotypes is often required to tailor the model to your needs. For example, creating Customer vs System requirements.

8.1.1 To add a profile to your model:

- Create a package called `profile` this can be done by right clicking in the left hand column.
- Switch modelling language to the UML profile (top of left hand menu drop down)
- Within the package create a profile diagram (prf)
- Add a profile element to the diagram
- Add a meta-class element to the diagram, within the profile.
- Add a stereotype element to the diagram, within the profile.
- Connect meta-class and stereotype with an Extension relation. The head should be attached to the class. As soon as the Extension is connected, the class will get a stereotype *metaclass* assigned.

With the meta-class and stereotype placed on the diagram, either:

- Double-click the meta-class and name it after the base element you want to create your stereotype from.
- Select the base element from the drop down menu in the Property Editor on the right hand side. In this case only UML elements can be used as base elements.

8.1.2 Styling Stereotypes

You can apply styling to stereotypes. For example here the base element `requirement` has a stereotype `system requirement`

```
/*Add style to Requirement element*/
requirement{
    background-color: #C5E7E7;
    text-color: #2A2A2A;
}
/*Update Requirement styling for the System stereotype*/
requirement[appliedStereotype.classifier.name=system]{
    background-color: #D5F7E7;
    text-color: #2A2A2A;
}
```

Style Sheets has more detail on how CSS works in Gaphor

CHAPTER

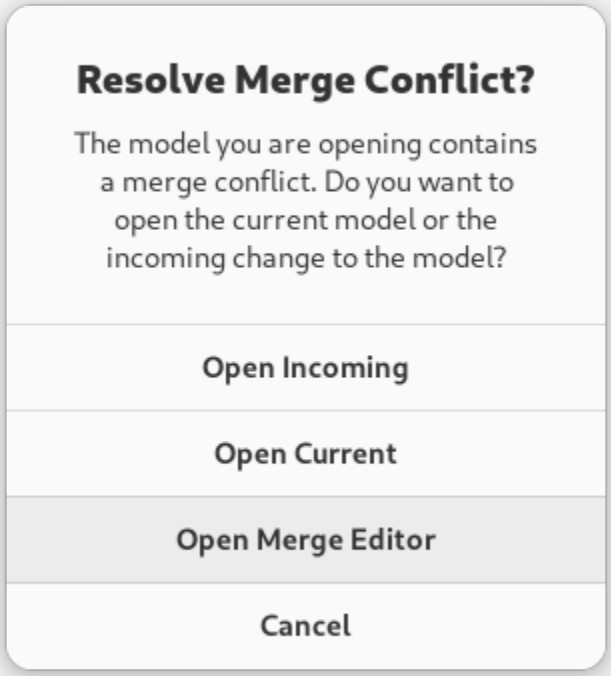
NINE

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.

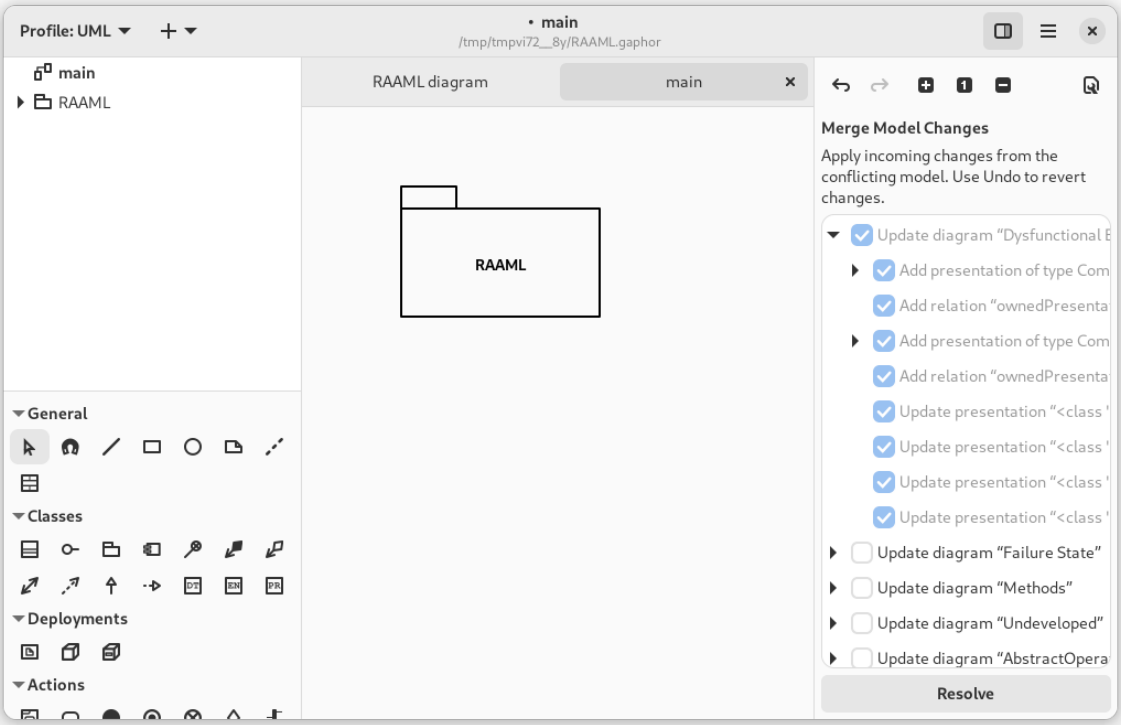
From Gaphor 2.18 onwards Gaphor is also capable of merging models. Once a merge conflict has been detected (i.e., when the model file contains git conflict-resolution markers <<<<<<<, =======, and >>>>>>>), Gaphor will offer the option to open the current model, the incoming model or merge changes manually via the Merge Editor.



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 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

TEN

PLUGINS

Important: Plugins is an experimental feature! The API may change.

We welcome you to try and provide your feedback.

Plugins allow you to extend the functionality of Gaphor beyond the features provided in the standard distributions. In particular, plugins can be helpful if you install the binary distributions available on the download page.

Gaphor can be extended via entry points in several ways:

1. Application (global) services (`gaphor.appservices`)
2. Session specific services (`gaphor.services`)
3. *Modeling languages* (`gaphor.modelinglanguages`)
4. (Sub)command line parsers (`gaphor.argparsers`)
5. Indirectly loaded modules (`gaphor.modules`), mainly for UI components

The default location for plugins is `$HOME/.local/gaphor/plugins-2` (`$USER/.local/gaphor/plugins-2` on Windows). This location can be changed by setting the environment variable `GAPHOR_PLUGIN_PATH` and point to a directory.

10.1 Install a plugin

At this moment Gaphor does not have functionality bundled to install and maintain plugins. To install a plugin, use `pip` from a Python installation on your computer. On macOS and Linux, that should be easy, on Windows you may need to install Python separately from python.org or the Windows Store.

Important:

1. Since plugins are installed with your system Python version, it's important that plugins are pure Python and do not contain compiled C code.
2. If you use Gaphor installed as Flatpak, you need to grant Gaphor access to user files (`filesystem=home`), so Gaphor can find files in your `.local` folder. You can use FlatSeal to change permissions of Flatpaks.

For example: to install the Hello World plugin on Linux and macOS, enter:

```
pip install --target $HOME/.local/gaphor/plugins-2 git+https://github.com/gaphor/gaphor_
↪plugin_helloworld.git
```

Then start Gaphor as you normally would. A new Hello World entry has been added to the tools menu (→ Tools → Hello World).

10.2 Create your own plugin

If you want to write a plugin yourself, you can familiarize yourself with Gaphor's *design principles*, *service oriented architecture* (includes a plugin example), and *event driven framework*.

10.3 Example plugin

You can have a look at the Hello World plugin available on GitHub for an example of how to create your own plugin.

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:

```
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
        ...  # 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

ELEVEN

GAPHOR ON LINUX

Gaphor can be installed as Flatpak 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.

11.1 Development Environment

There are two ways to set up a development environment:

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

11.1.1 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.

11.1.2 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.

11.1.3 Debugging using Visual Studio Code

Before you start debugging you'll need to open Gaphor in vscode (the folder containing `pyproject.toml`). You'll need to have the Python extension installed.

Create a file `.vscode/launch.json` with the following content:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Gaphor UI",
            "type": "python",
            "request": "launch",
            "module": "gaphor",
            "justMyCode": false,
            "env": {
                "GDK_BACKEND": "wayland",
            }
        }
    ]
}
```

`GDK_BACKEND` is added since VSCode by default uses XWayland (the X11 emulator).

11.2 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
```

11.3 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

TWELVE

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.

12.1 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=/opt/homebrew/opt/libffi/lib/pkgconfig  # use /usr/local/ for␣
↪older Homebrew installs
poetry install
```

12.1.1 Debugging using Visual Studio Code

Before you start debugging you'll need to open Gaphor in VSCode (the folder containing `pyproject.toml`). You'll need to have the Python extension installed.

Create a file `.vscode/launch.json` with the following content:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Gaphor UI",
            "type": "python",
            "request": "launch",
            "module": "gaphor",
            "justMyCode": false,
        }
    ]
}
```

12.2 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

THIRTEEN

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.

13.1 Development Environment

13.1.1 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.

13.1.2 Git

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

```
choco install git
```

13.1.3 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
```

13.1.4 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.

13.1.5 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;"
$env:XDG_DATA_HOME = "$HOME\.local\share"
```

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
```

13.1.6 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.

13.2 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

FOURTEEN

GAPHOR IN A CONTAINER

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

14.1 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.

14.2 Remote access to Gaphor graphic window with Codespaces

When using Codespaces, chances are that you also want to interact with the graphical window of Gaphor.

This is facilitated in Gaphor by use of container feature called desktop-lite. This feature is activated by default in the Gaphor's devcontainer.json file.

Notice the webPort/vncPort and password values. These are used in subsequent steps.

```
                "desktop-lite": {
                    "password": "vscode",
                    "webPort": "6080",
                    "vncPort": "5901"
            },
```

There are two options:

14.2.1 Using a local VNC viewer

1. Download and install VNC viewer of your choice (e.g. realvnc)
2. Specify remote hostname as *localhost* and port as *5901* and connect VNC. The port number should be same as specified in attribute *vncPort*
3. Upon debugging/running Gaphor the familiar Graphic window should be displayed in VNC view

14.2.2 Using noVNC viewer on the Browser

1. This is based on noVNC application
2. Open the browser on your local machine and give address as *http://127.0.0.1:6080/*. The port number should be same as specified in attribute *webPort*
3. A noVNC window will open, click on Connect and provide password as *vscode*. The password should be same as specified in attribute *password*
4. Upon debugging/running Gaphor the familiar Graphic window should be displayed in noVNC view on Browser

CHAPTER

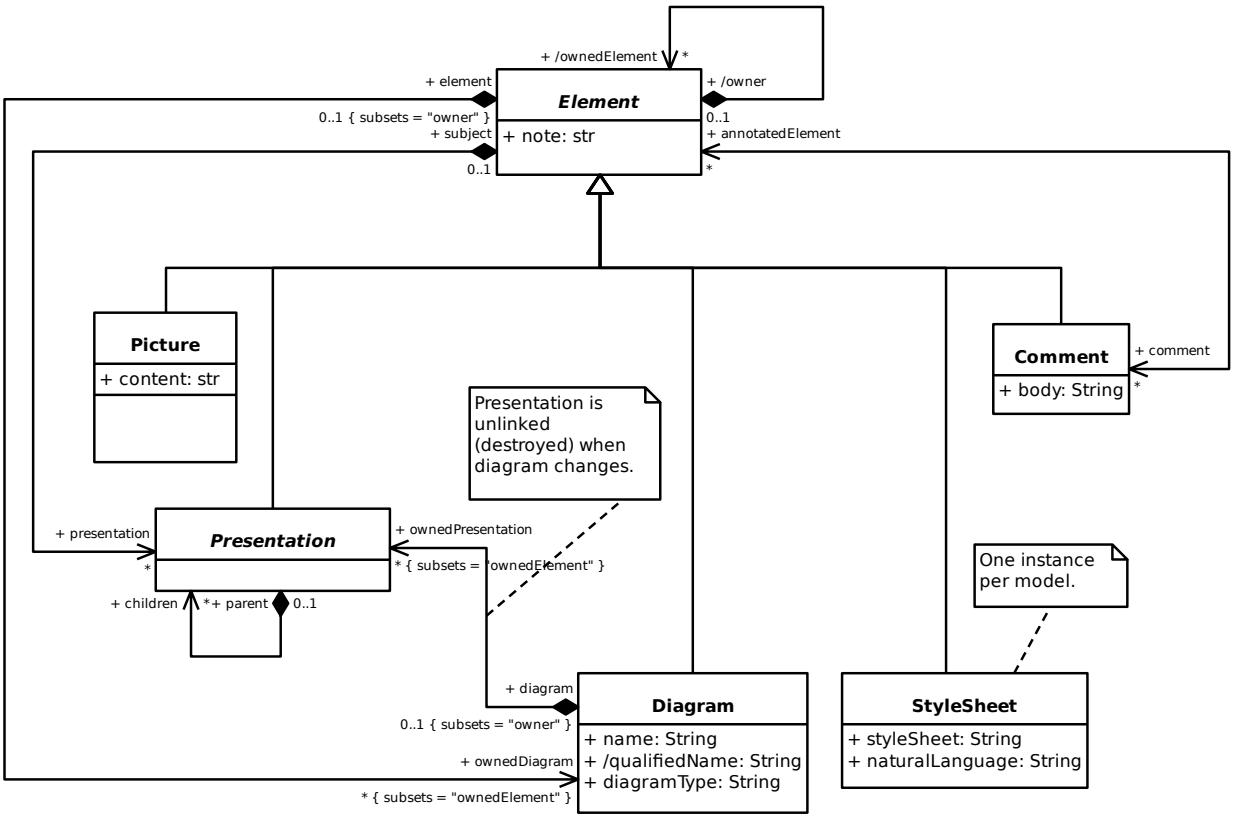
FIFTEEN

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.

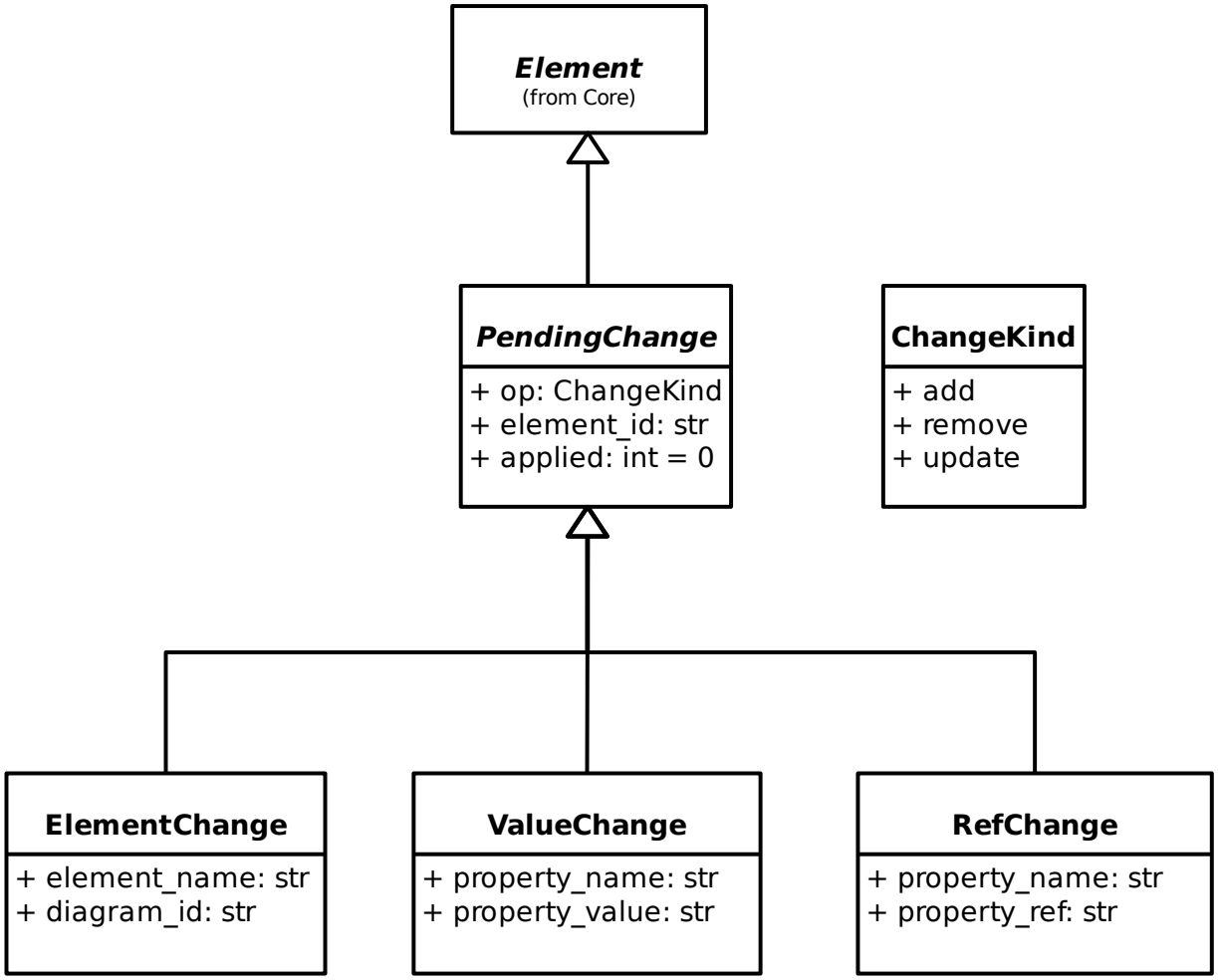


The *Element* base class provides event notification and integrates with the model repository (internally known as `ElementFactory`). Bi-directional relationships are also possible, as well as derived relations.

15.1 Change Sets

The core model has support for change sets, sets of pending changes. Normally you end up with a change set when you *resolve a merge conflict* in your model.

This diagram is provided for completion sake.



15.1.1 Core Modeling Classes

Here you can find a short description of the base classes in a Gaphor model: *Element*, *Presentation*, and *Diagram*.

The *RepositoryProtocol*, and *EventWatcherProtocol* protocols are important to connect the model to the repository and event handling mechanisms.

The Element Class

The class Element is the core of Gaphor's data model.

class gaphor.core.modeling.**Element**(*id: str | None = None*, *model:* RepositoryProtocol | *None = None*)

Base class for all model data classes.

property id: str

An id (read-only), unique within the model.

property model: *RepositoryProtocol*

The owning model, raises TypeError when model is not set.

unlink() → None

Unlink the element.

All the elements references are destroyed. For composite associations, the associated elements are also unlinked.

The unlink lock is acquired while unlinking this element's properties to avoid recursion problems.

Event handling

handle(*event*) → None

Propagate incoming events.

This only works if the element has been created by an ElementFactory

watcher(*default_handler: Callable[[ElementUpdated], None] | None = None*) → *EventWatcherProtocol*

Create a new watcher for this element.

Watchers provide a convenient way to get signalled when a property relative to self has been changed.

To use a watcher, the element should be created by a properly wired up ElementFactory`.

This example is purely illustrative:

```
>>> element = Element()
>>> watcher = element.watcher(default_handler=print)
>>> watcher.watch("note")  # Watch for changed on element.note
```

Loading and saving

load(*name*, *value*) → None

Loads value in name.

Make sure that after all elements are loaded, postload() should be called.

postload() → None

Fix up the odds and ends.

This is run after all elements are loaded.

save(*save_func*) → None

Save the state by calling `save_func(name, value)`.

OCL-style methods

isKindOf(*class_: type[*Element*]*) → bool

Returns `True` if the object is an instance of `class_`.

isTypeOf(*other:* Element) → bool

Returns `True` if the object is of the same type as the `other`.

The `Presentation` class

class gaphor.core.modeling.**Presentation**(*diagram:* Diagram, *id: Id* | *None* = *None*)

A special type of *`Element`* that can be displayed on a *`Diagram`*.

Subtypes of `Presentation` should implement the `gaphas.item.Item` protocol.

request_update() → None

Mark this presentation object for update.

Updates are orchestrated by diagrams.

watch(*path: str*, *handler: Callable[[ElementUpdated], None]* | *None* = *None*) → Self

Watch a certain path of elements starting with `self`.

The handler is optional and will default to a simple *`request_update`*.

Watches should be set in the constructor, so they can be registered and unregistered in one shot.

```
self.watch("subject[NamedElement].name")
```

This interface is fluent: returns `self`.

change_parent(*new_parent:* Presentation | *None*) → None

Change the parent and update the item's matrix so the item visually remains in the same place.

The `Diagram` class

class gaphor.core.modeling.**Diagram**(*id: str | None = None*, *model:* RepositoryProtocol | *None = None*)

Diagrams may contain *`Presentation`* elements and can be owned by any element.

create(*type_: type[P]*, *parent:* Presentation | *None = None*, *subject:* Element | *None = None*) → P

Create a new diagram item on the diagram.

It is created with a unique ID, and it is attached to the diagram's root item. The type parameter is the element class to create. The new element also has an optional parent and subject.

lookup(*id: str*) → *Presentation* | None

Find a presentation item by id.

Returns a presentation in this diagram or return `None`.

select(*expression: Callable[[*Presentation*], bool]*) → Iterator[*Presentation*]

select(*expression: type[P]*) → Iterator[P]

select(*expression: None*) → Iterator[*Presentation*]

Return an iterator of all canvas items that match expression.

request_update(*item: Item*) → None

Schedule an item for updating.

No update is done at this point, it's only added to the set of to-be updated items.

This method is part of the `gaphas.model.Model` protocol.

update(*dirty_items: Collection[*Presentation*] = ()*) → None

Update the diagram.

All items that requested an update via *`request_update()`* are now updates. If an item has an `update(context: UpdateContext)` method, it's invoked. Constraints are solved.

Protocols

class gaphor.core.modeling.element.**RepositoryProtocol**(**args*, ***kwargs*)

create(*type: type[T]*) → T

Create a new element in the repository.

select(*self*, *expression: Callable[[*Element*], bool]*) → Iterator[*Element*]

Select elements from the repository that fulfill `expression`.

select(*self*, *type_: type[T]*) → Iterator[T]

Select all elements from the repository of type `type_`.

select(*self*, *expression: None*) → Iterator[*Element*]

Select all elements from the repository.

lookup(*id: str*) → *Element* | None

Get an element by id from the repository.

Returns `None` if no such element exists.

class gaphor.core.modeling.element.**EventWatcherProtocol**(**args*, ***kwargs*)

watch(*path: str*, *handler: Callable[[ElementUpdated], None] | None = None*) → *EventWatcherProtocol*

Add a watch for a specific path. The path is relative to the element that created the watcher object.

Returns `self`, so watch operations can be chained.

unsubscribe_all() → None

Should be called before the watcher is disposed, to release all watched paths.

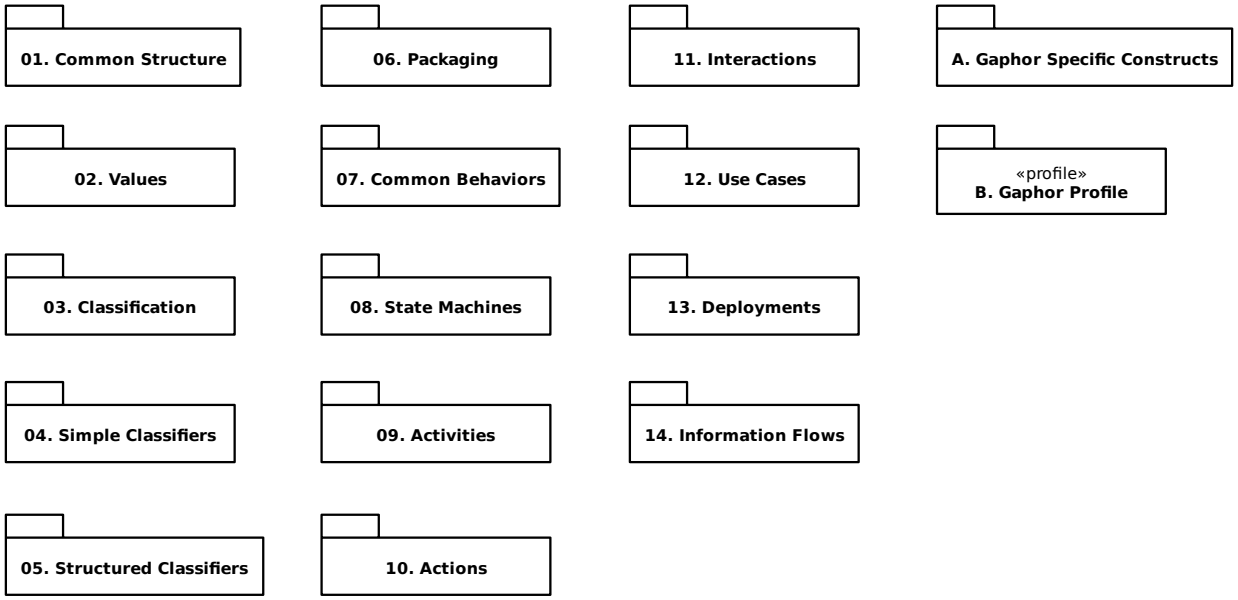
CHAPTER

SIXTEEN

UNIFIED MODELING LANGUAGE

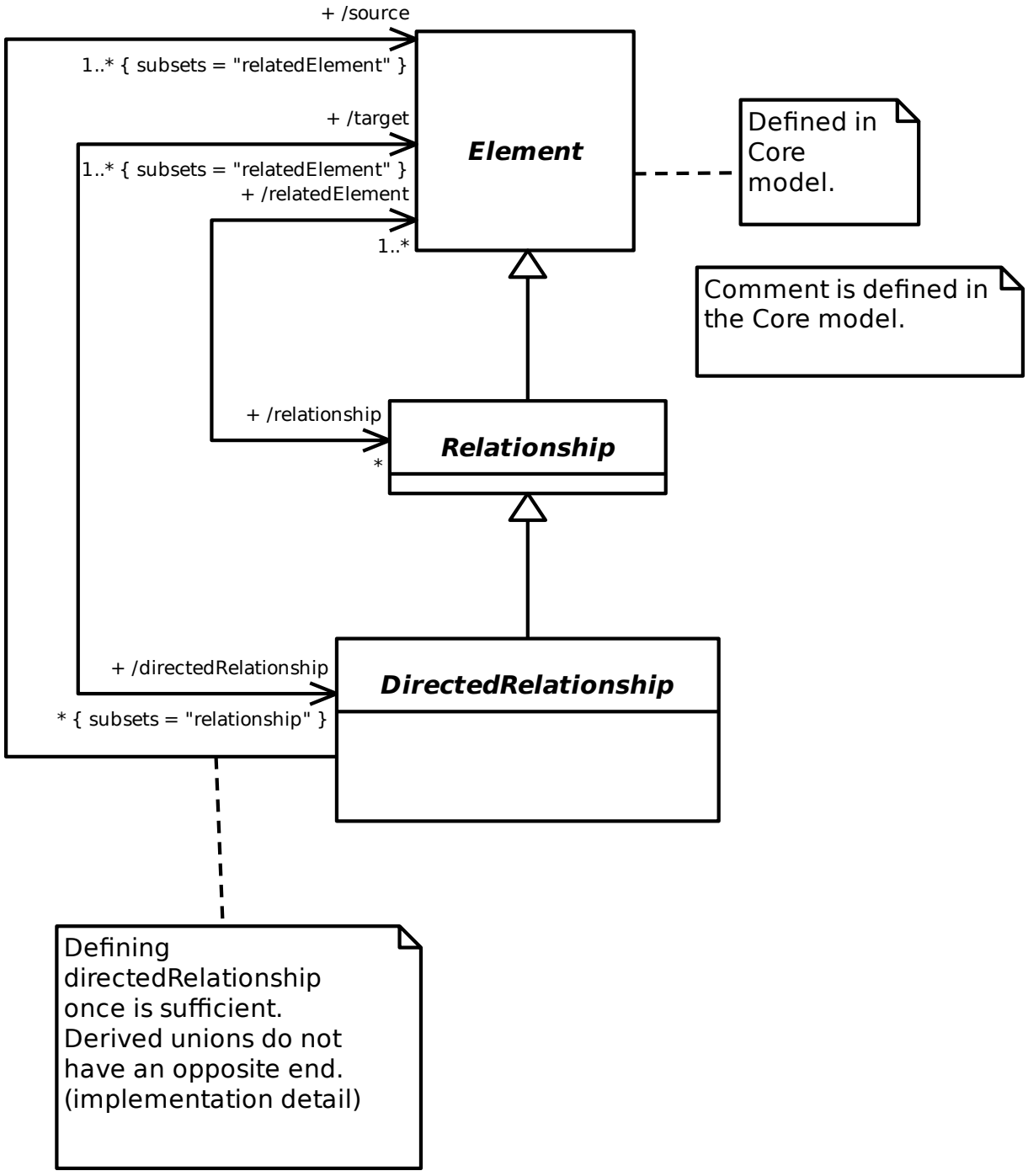
The UML model is the most extensive model in Gaphor. It is used as a base language for *SysML*, *RAAML*, and *C4*.

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.



16.1 01. Common Structure

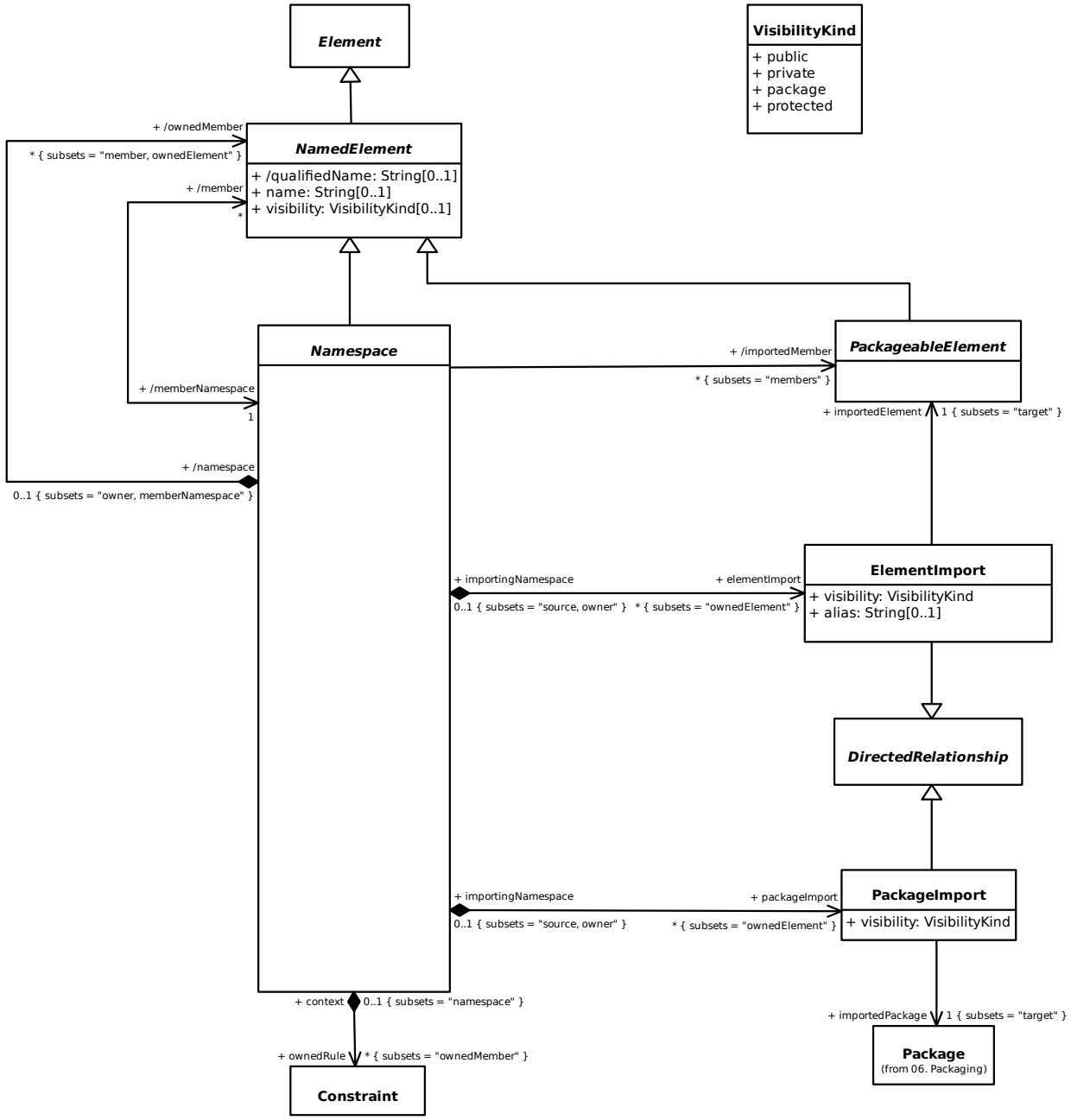
16.1.1 1. Root



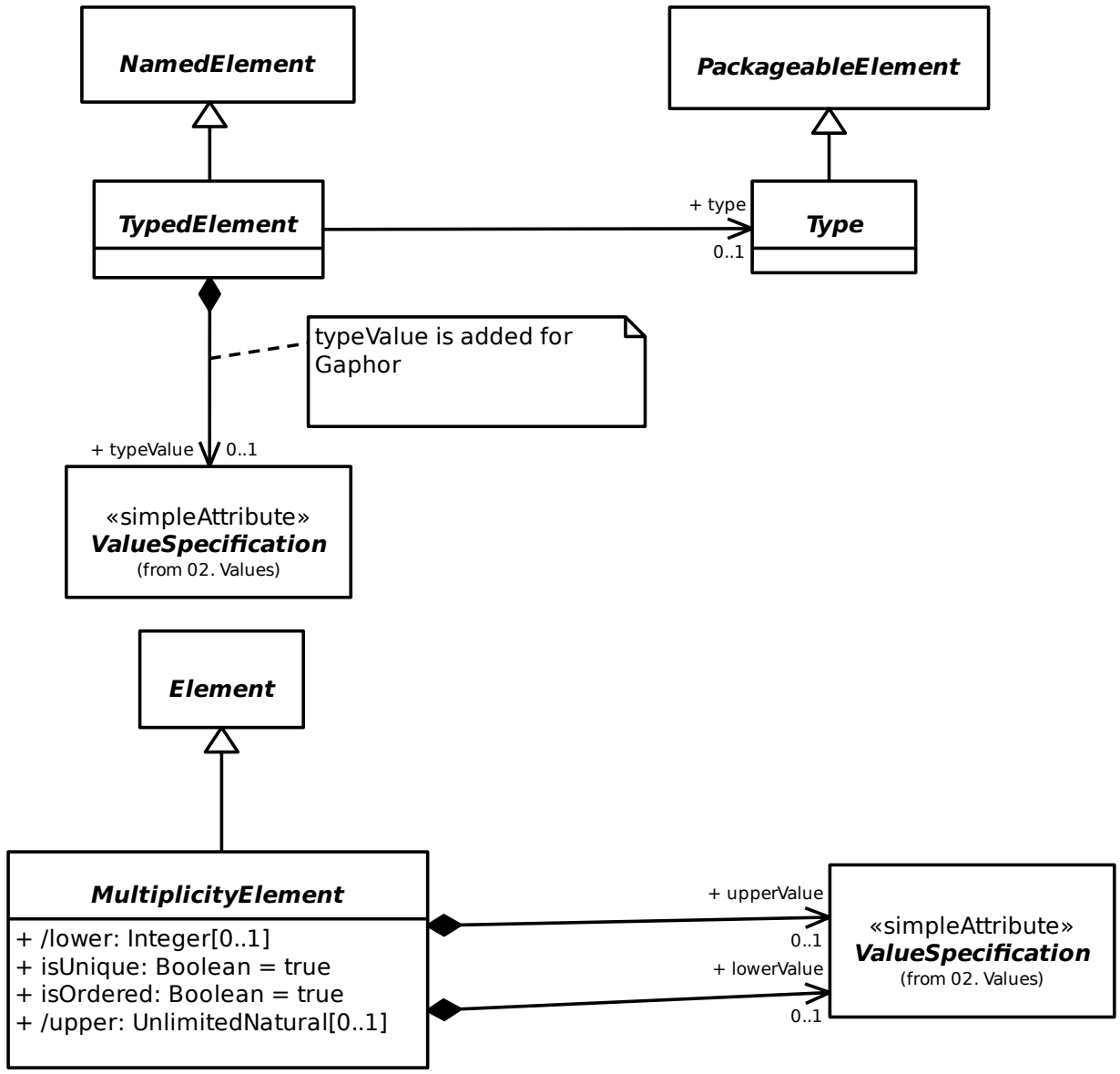
16.1.2 2. Templates

Not implemented.

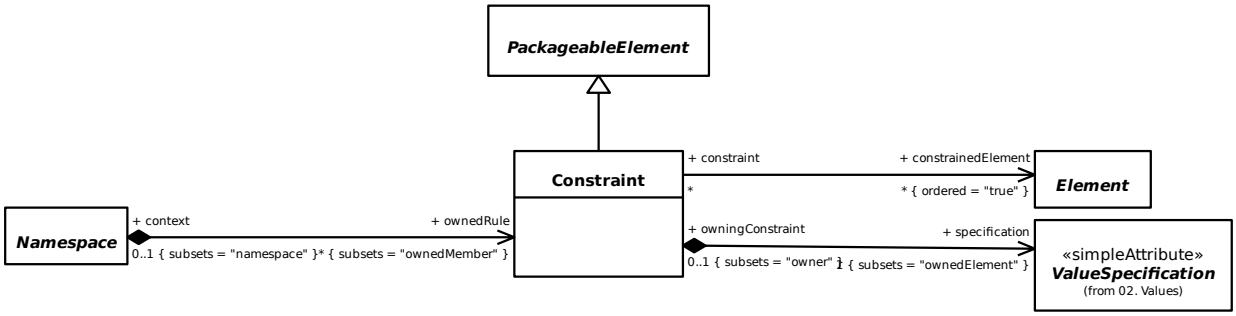
16.1.3 3. Namespaces



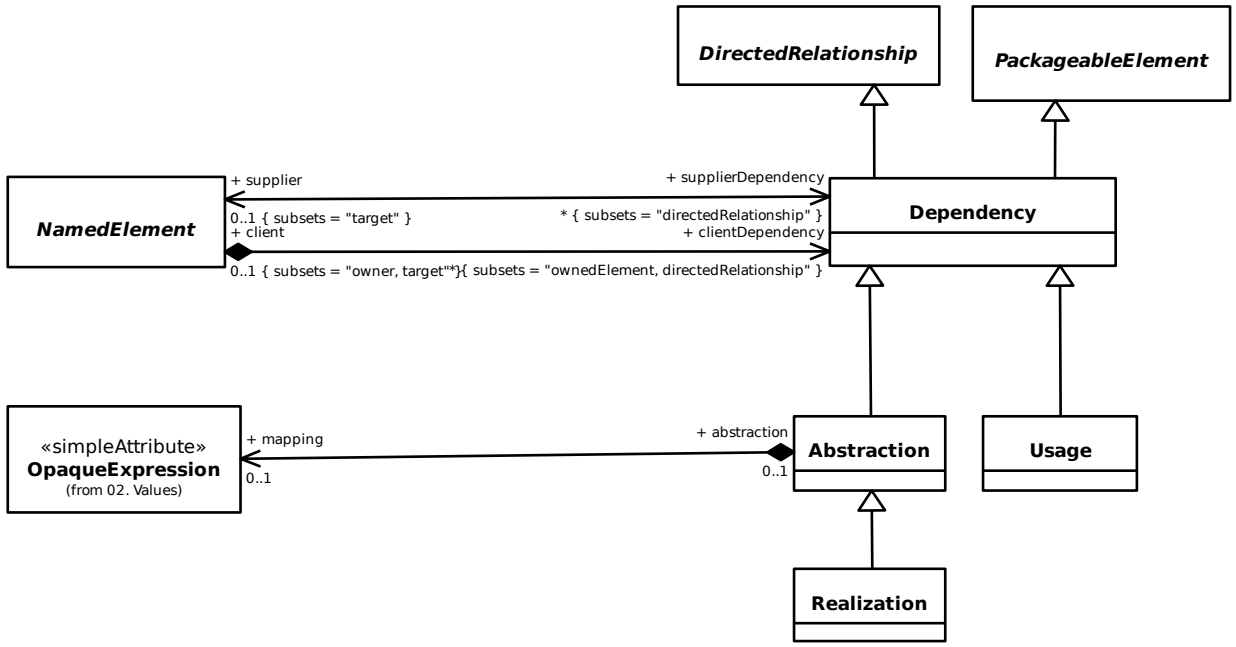
16.1.4 4. Types and Multiplicity



16.1.5 5. Constraints

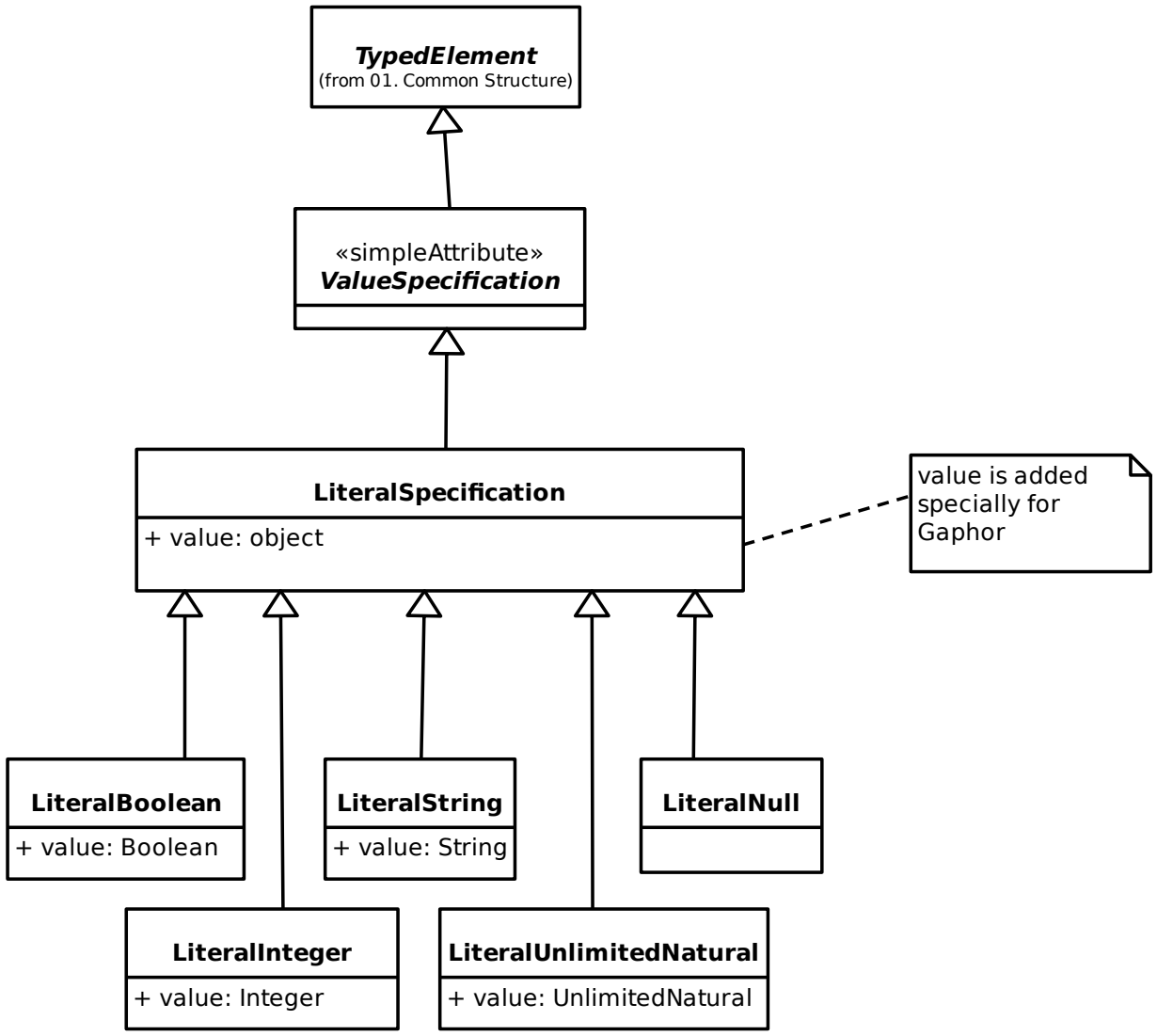


16.1.6 6. Dependencies

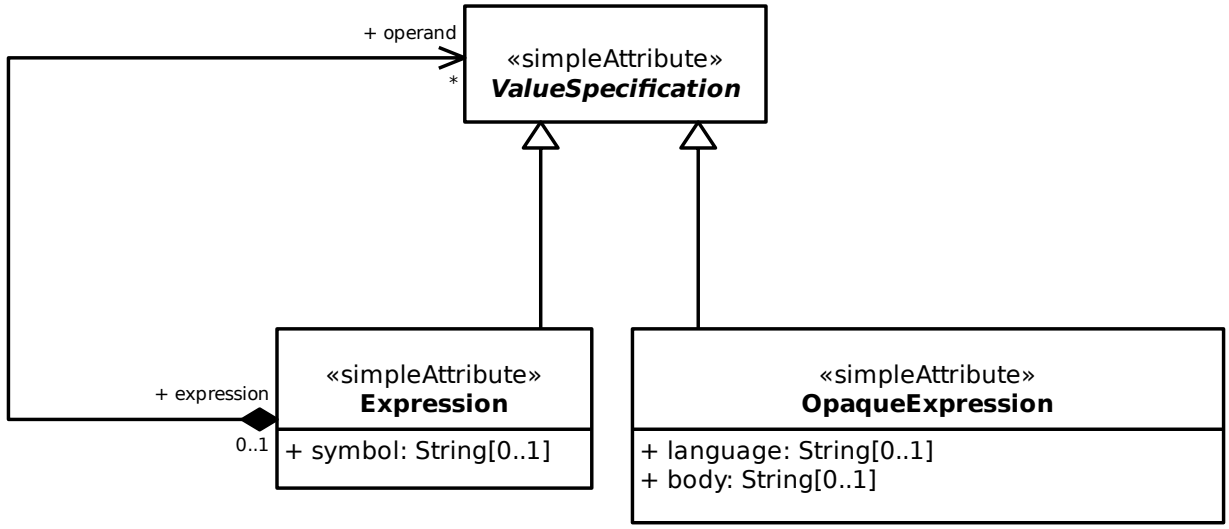


16.2 02. Values

16.2.1 1. Literals

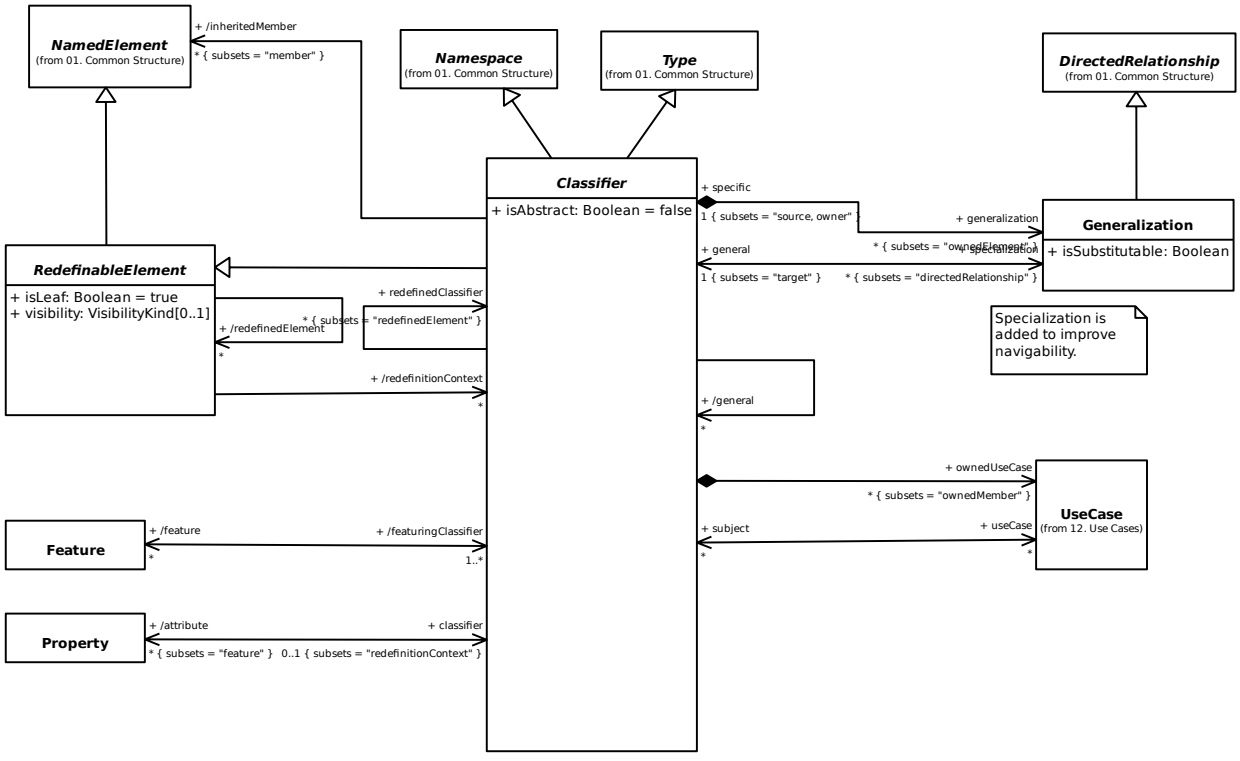


16.2.2 2. Expressions

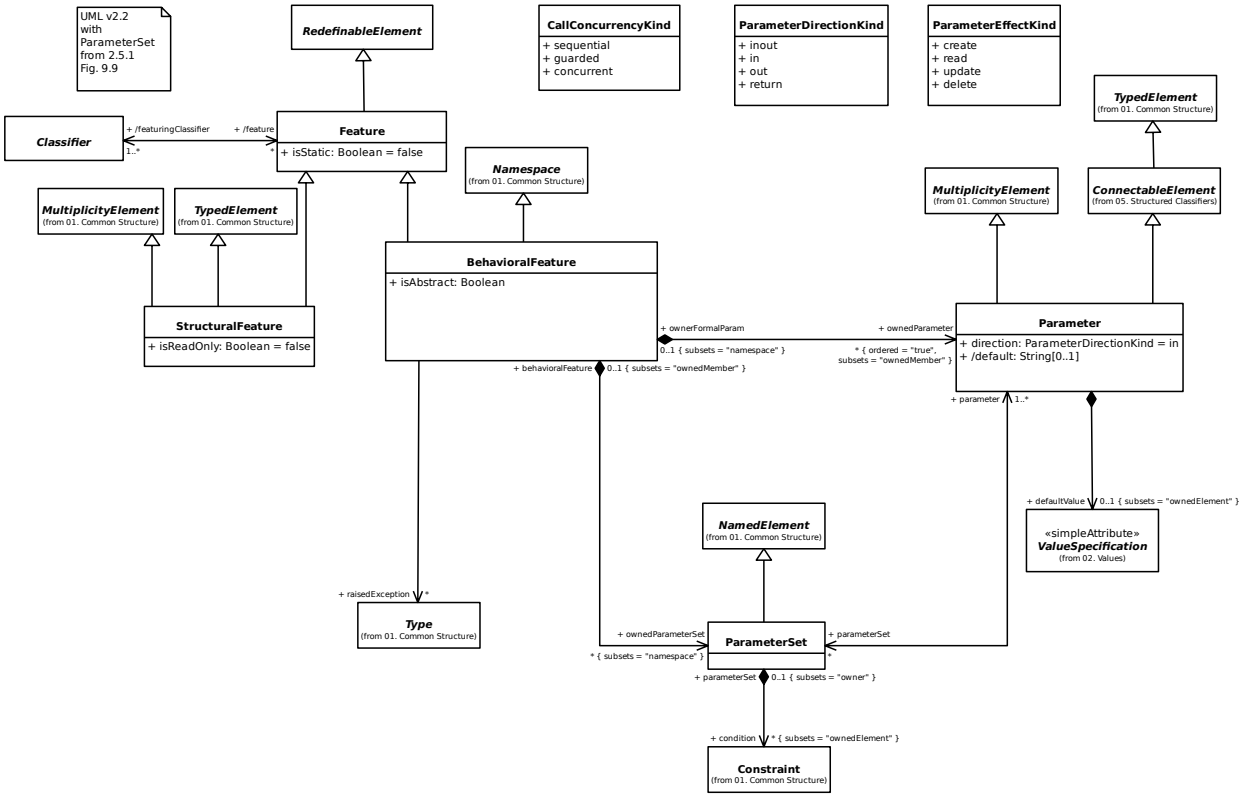


16.3 03. Classification

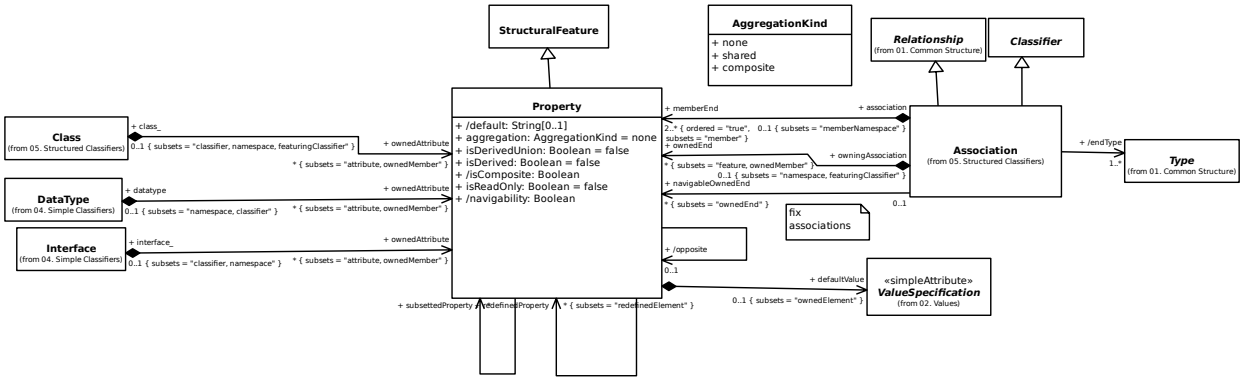
16.3.1 1. Classifiers



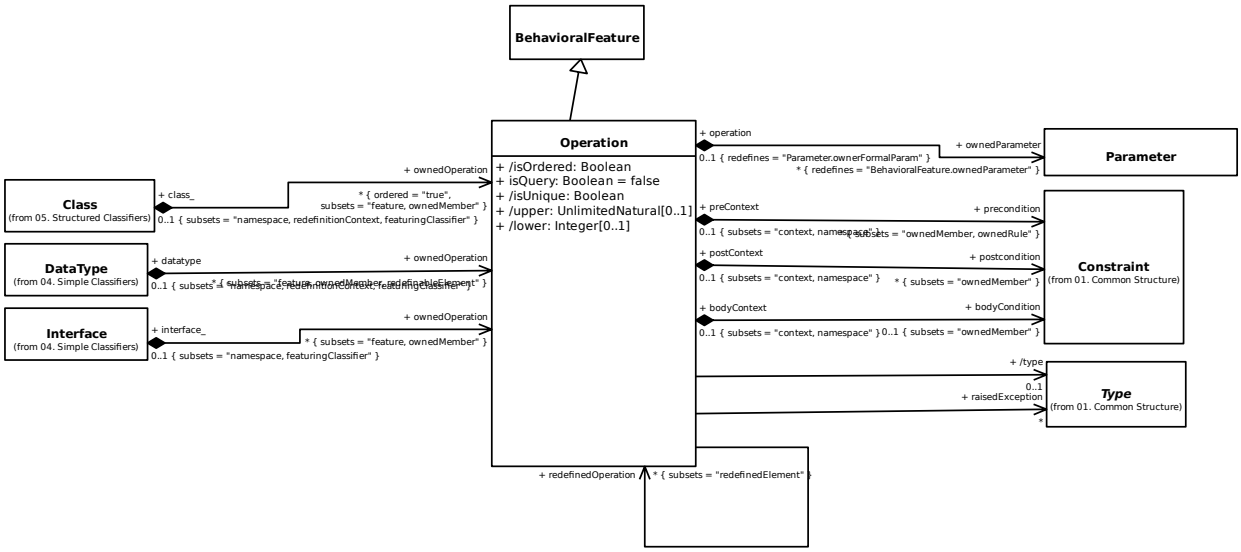
16.3.2 3. Features



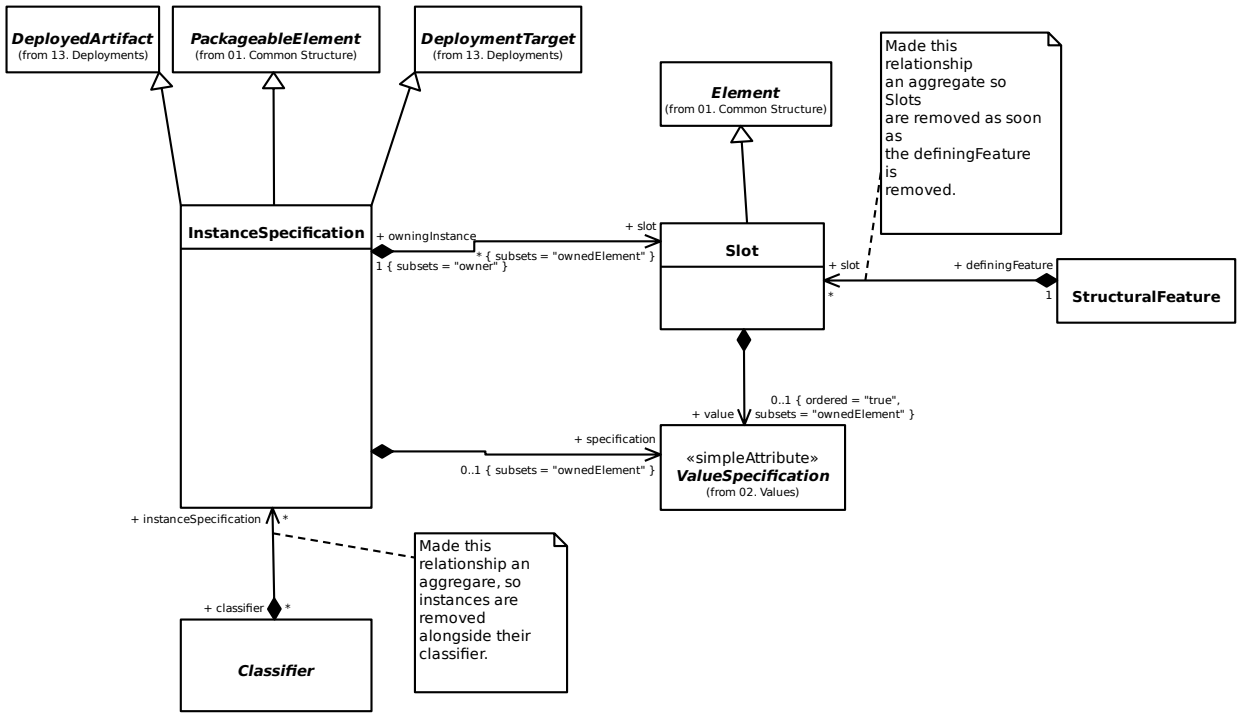
16.3.3 4. Properties



16.3.4 5. Operations

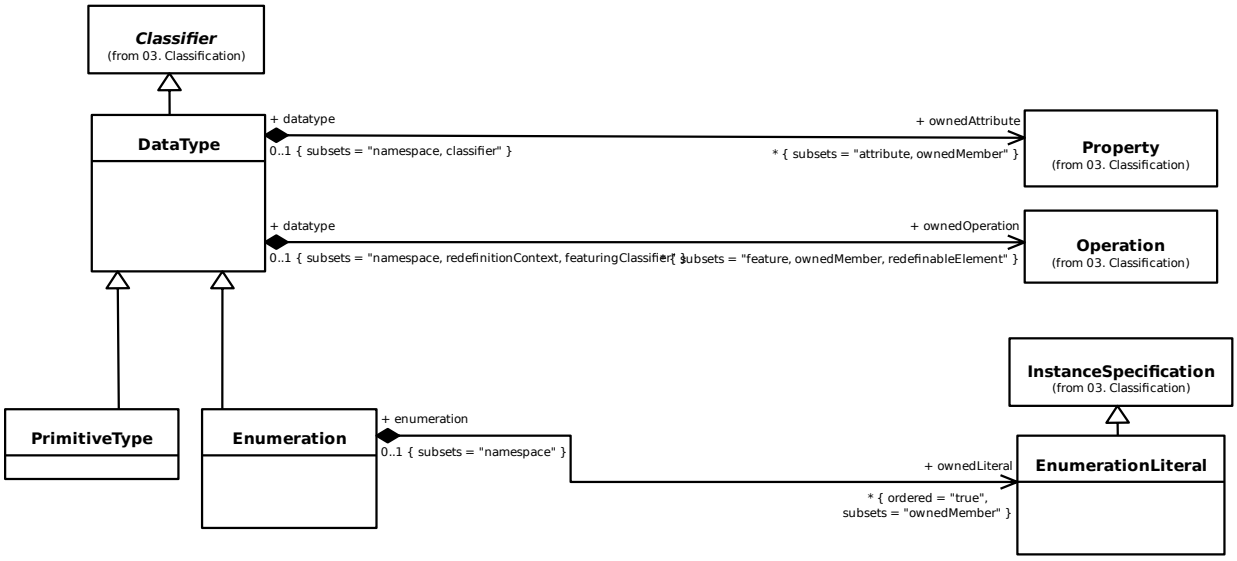


16.3.5 7. Instances

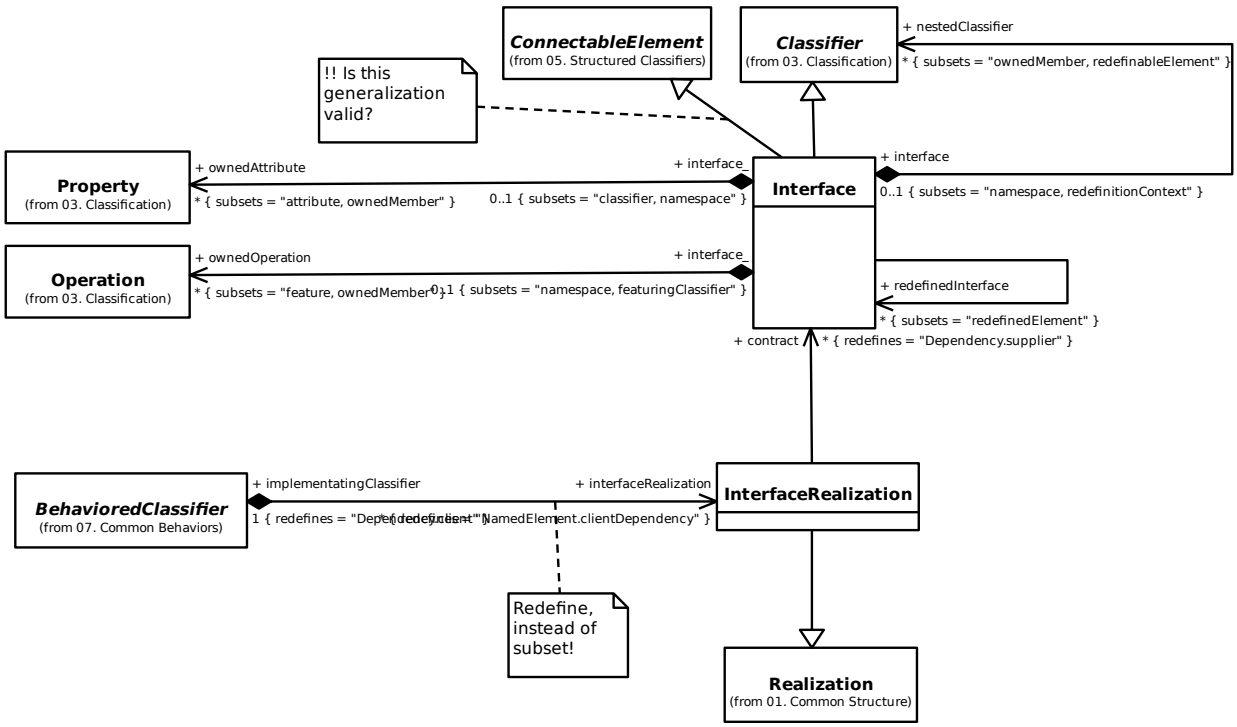


16.4 04. Simple Classifiers

16.4.1 1. Data Types

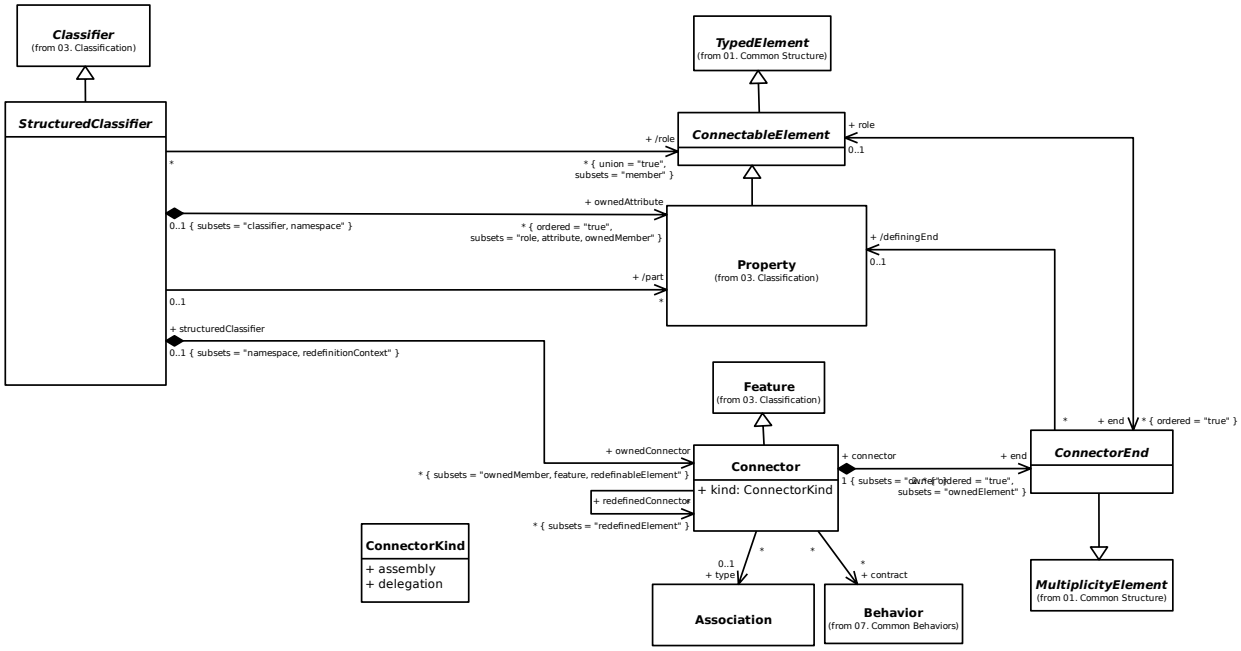


16.4.2 3. Interfaces

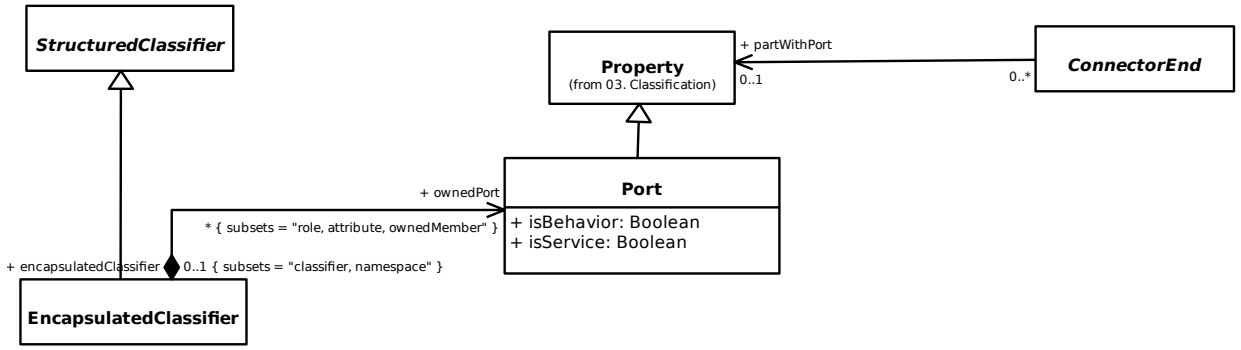


16.5 05. Structured Classifiers

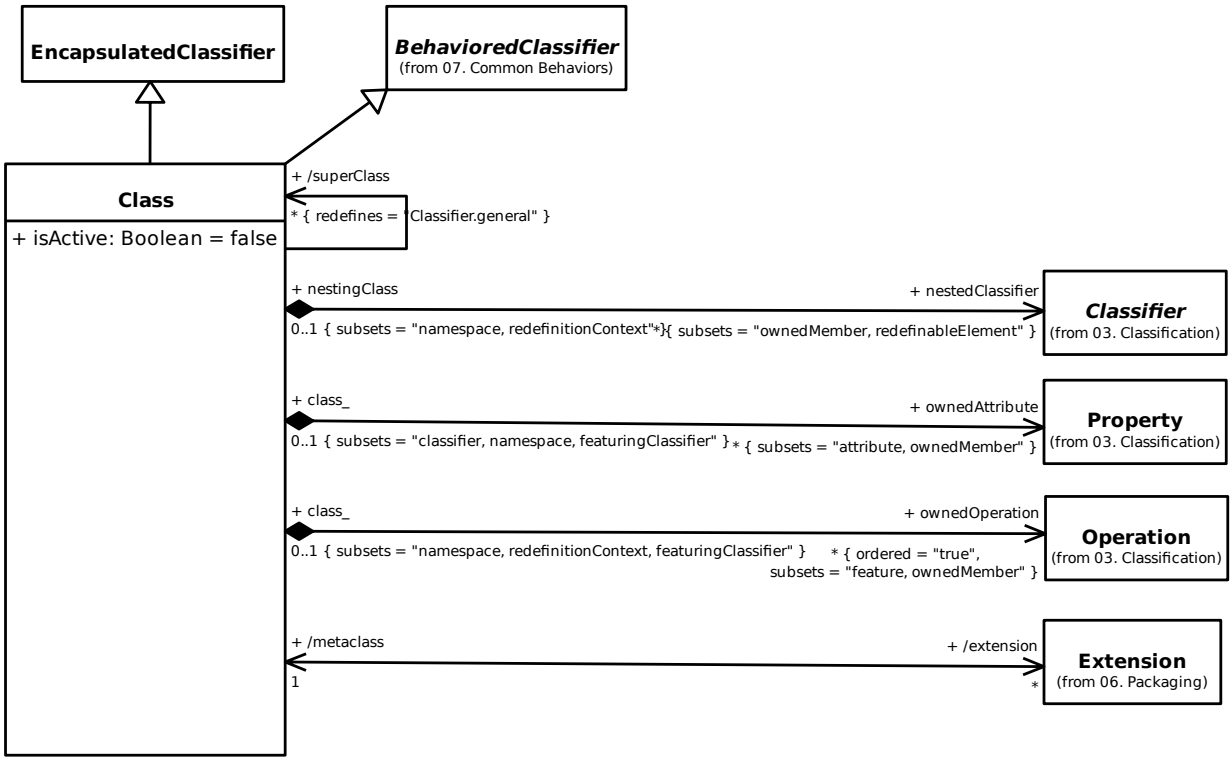
16.5.1 1. Structured Classifiers



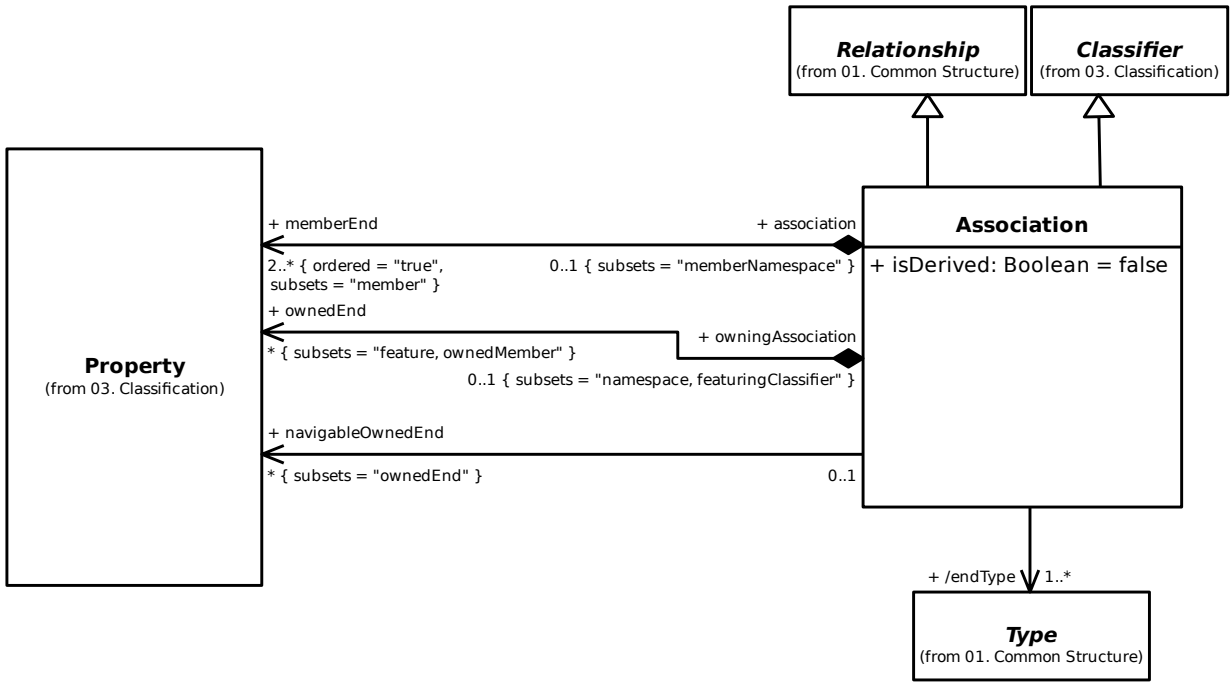
16.5.2 2. Encapsulated Classifiers



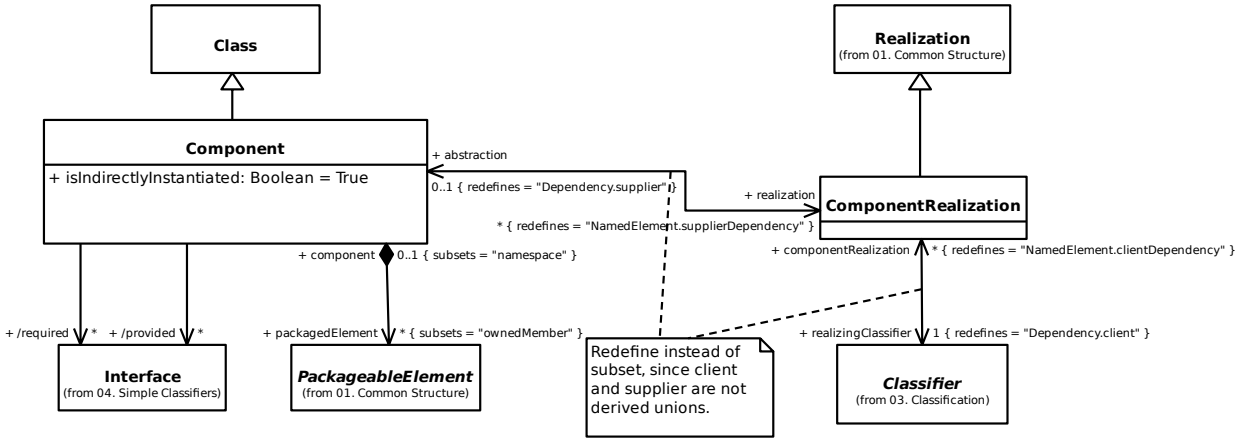
16.5.3 3. Classes



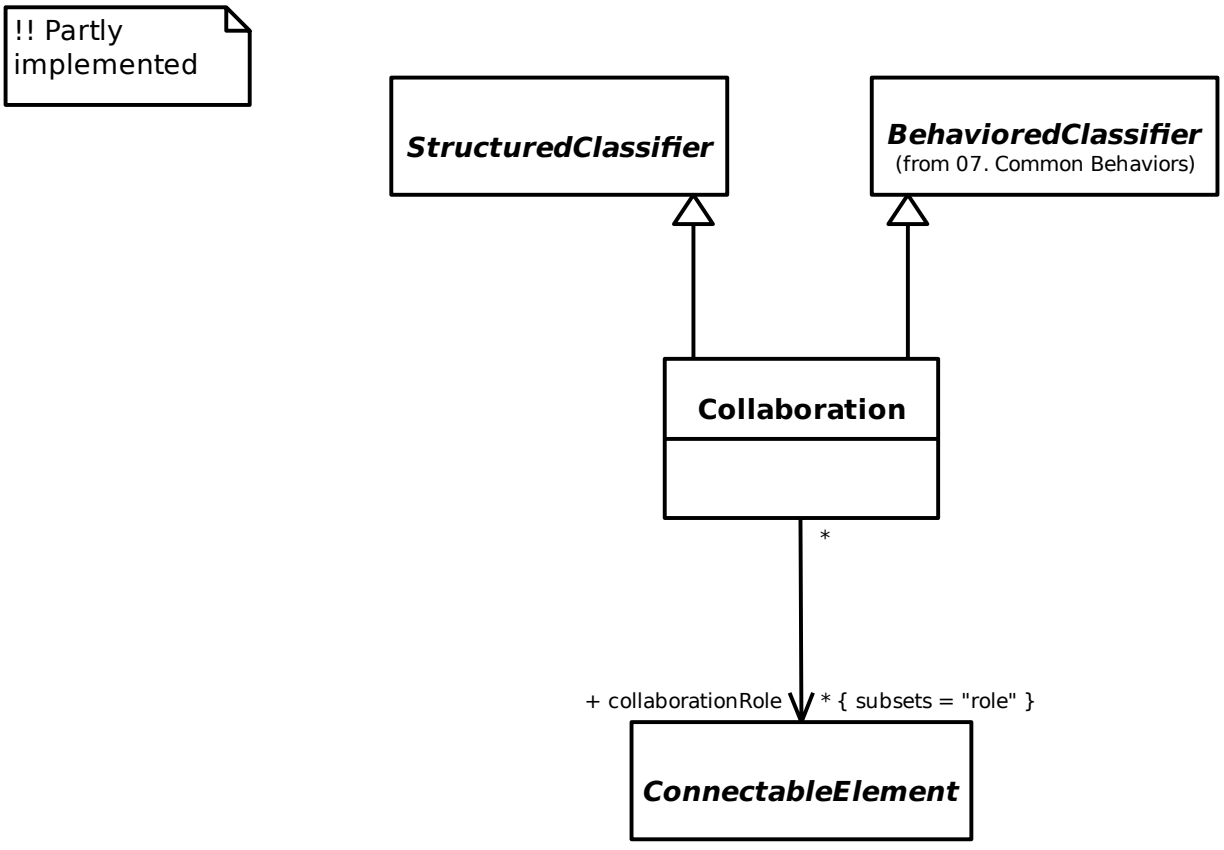
16.5.4 4. Associations



16.5.5 5. Components

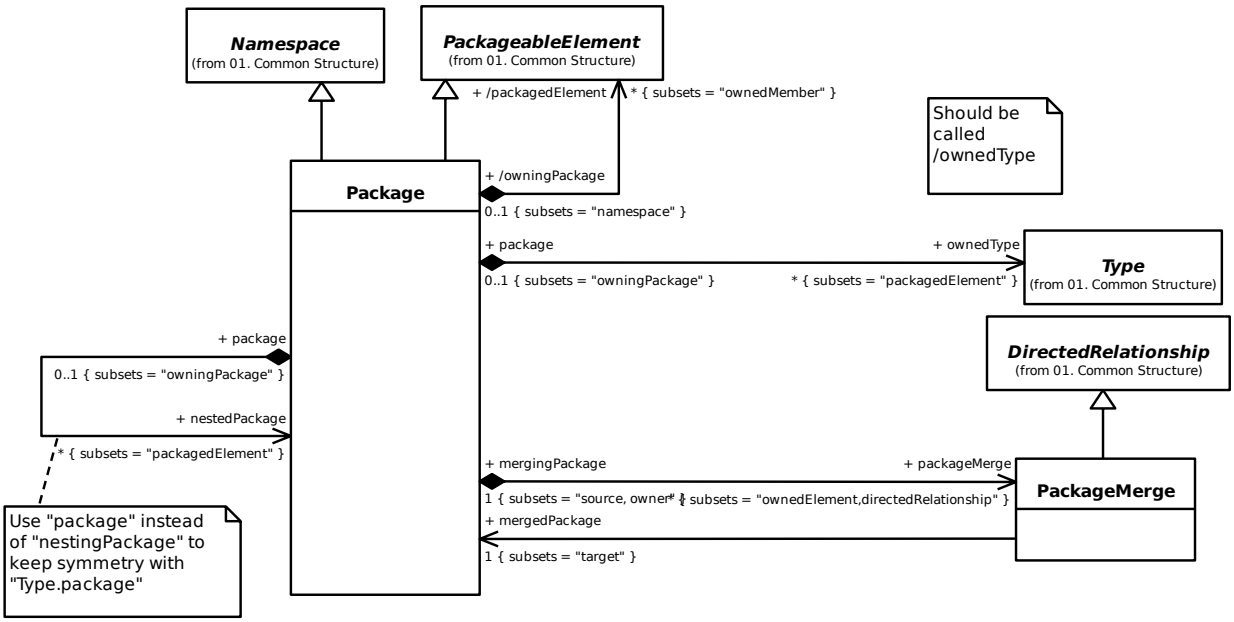


16.5.6 6. Collaborations

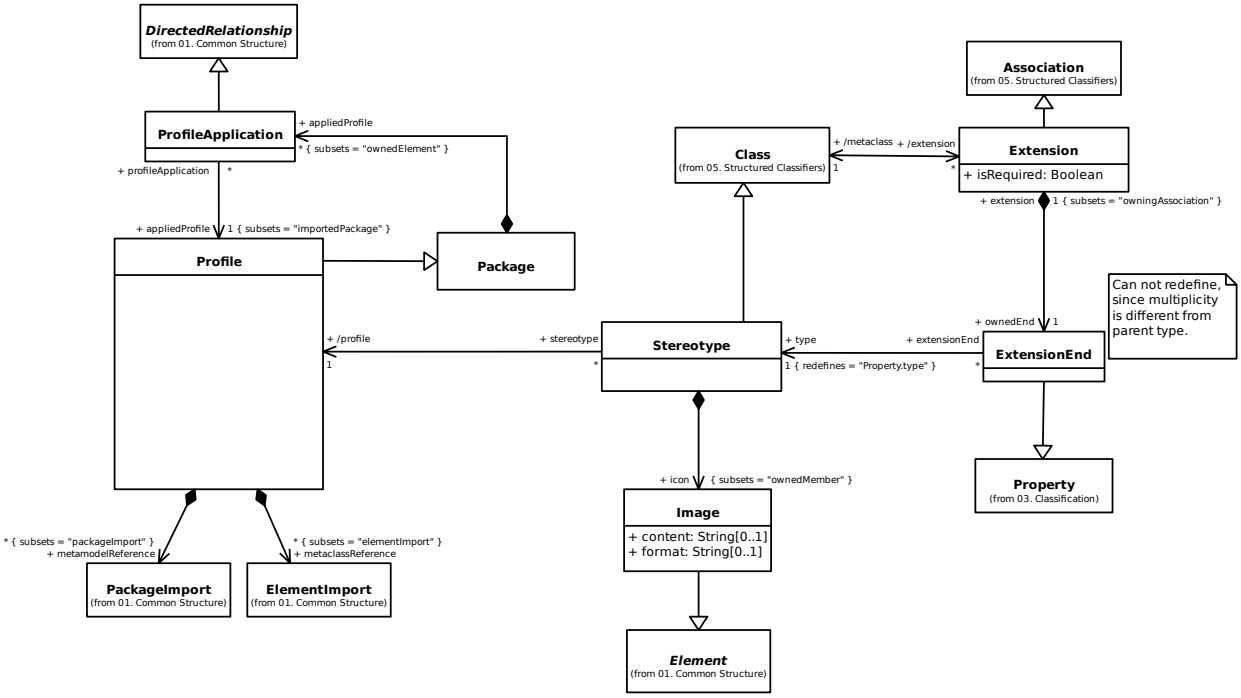


16.6 06. Packaging

16.6.1 1. Packages

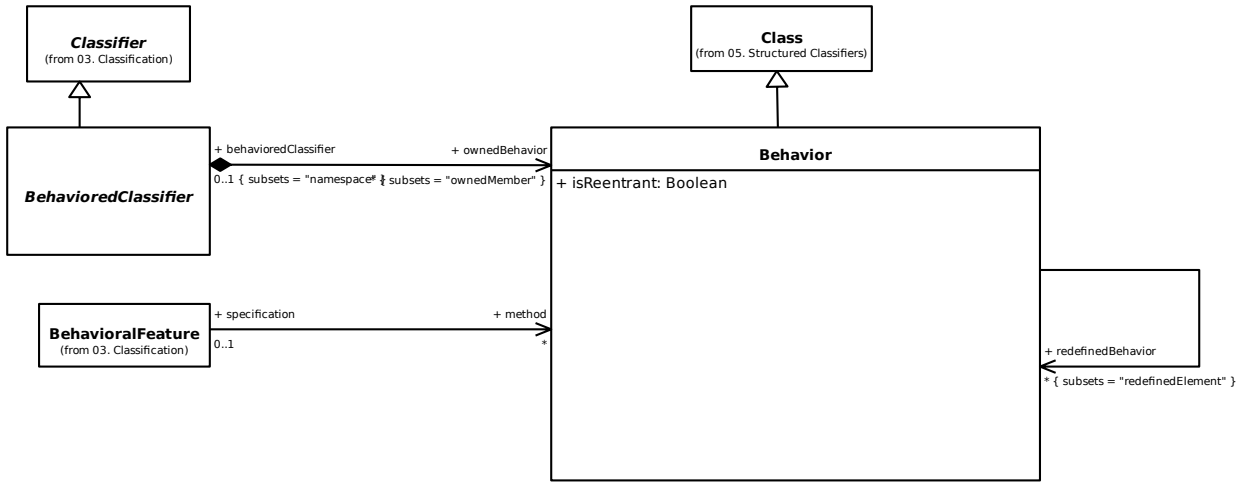


16.6.2 2. Profiles

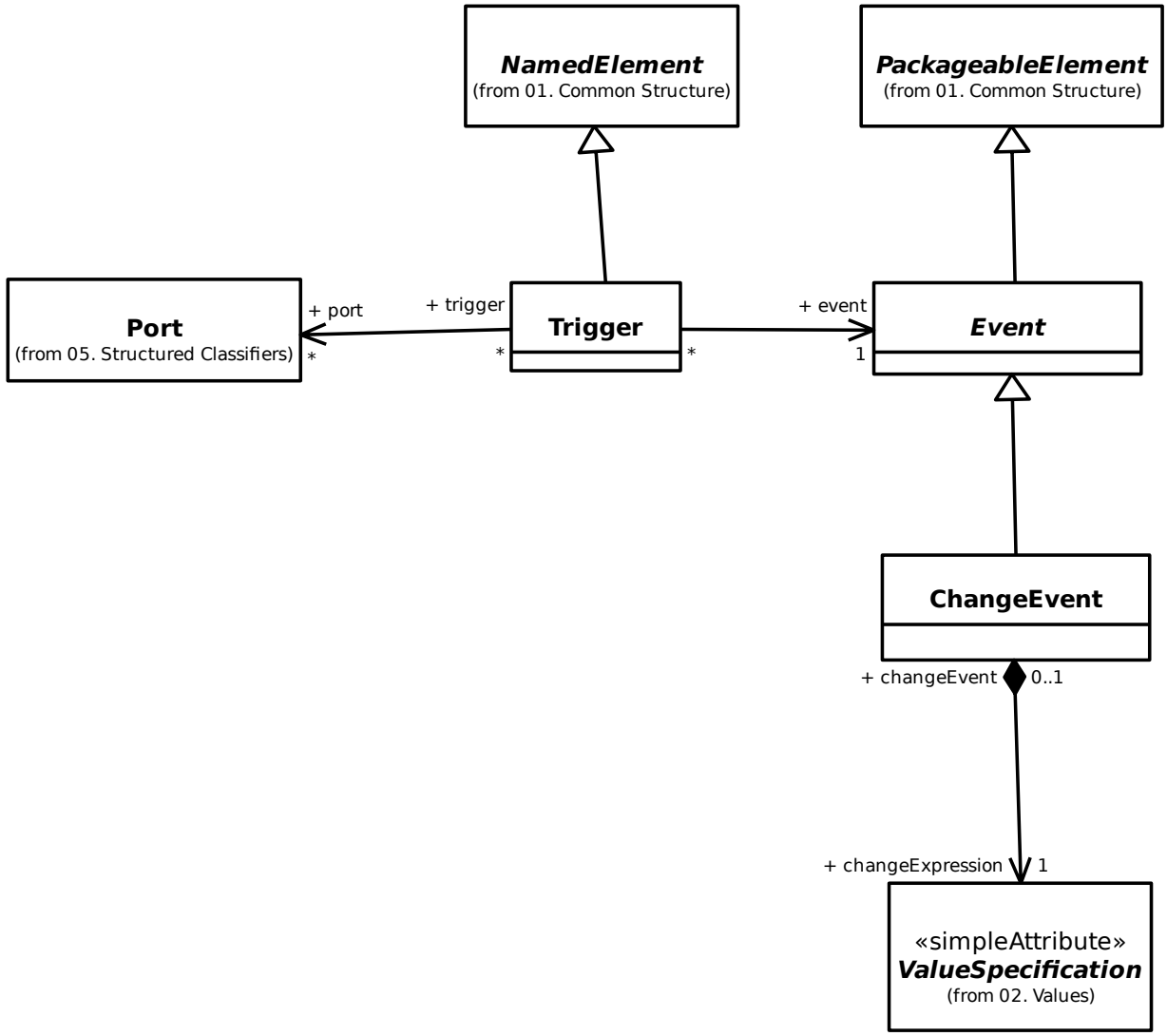


16.7 07. Common Behaviors

16.7.1 1. Behaviors

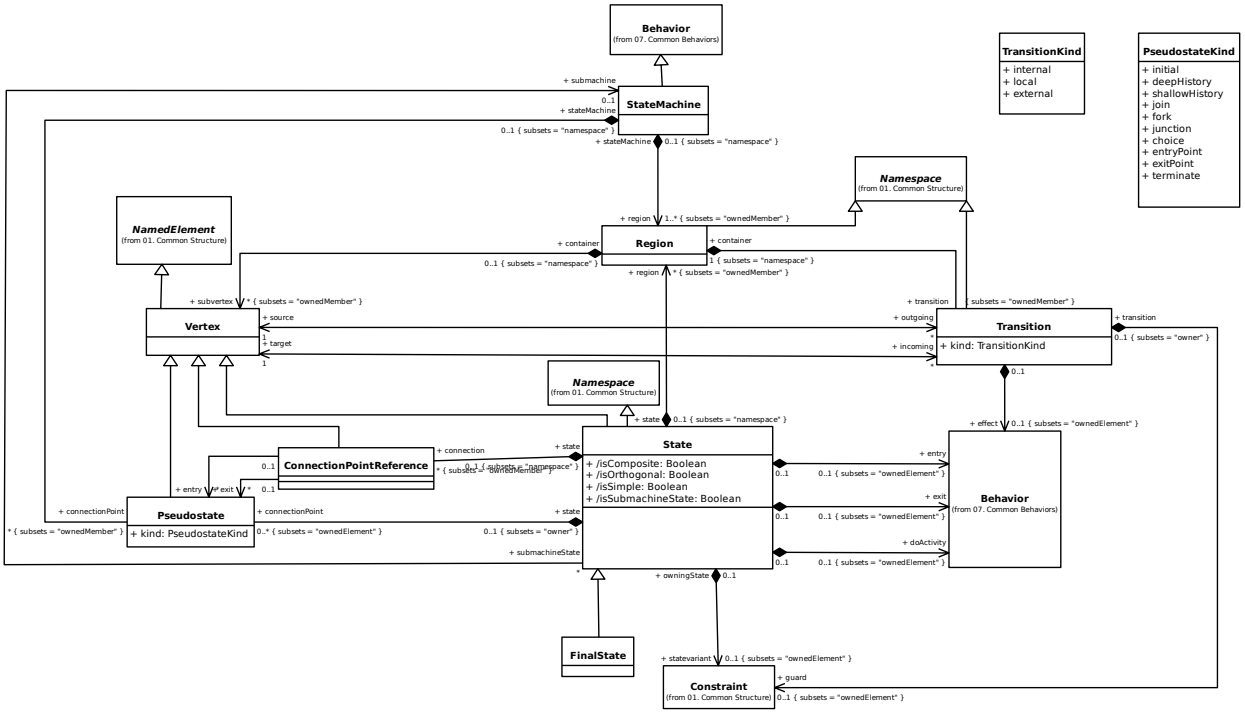


16.7.2 2. Events



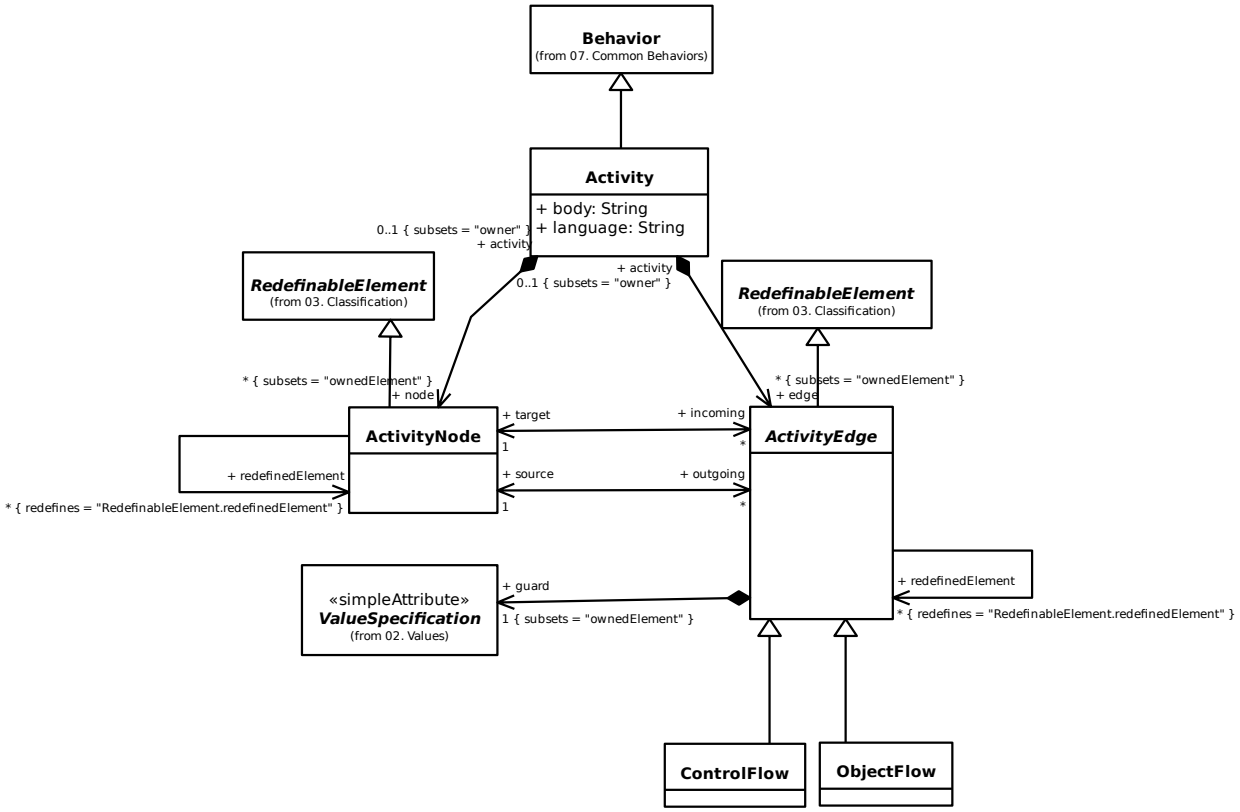
16.8 08. State Machines

16.8.1 1. Behavior State Machines

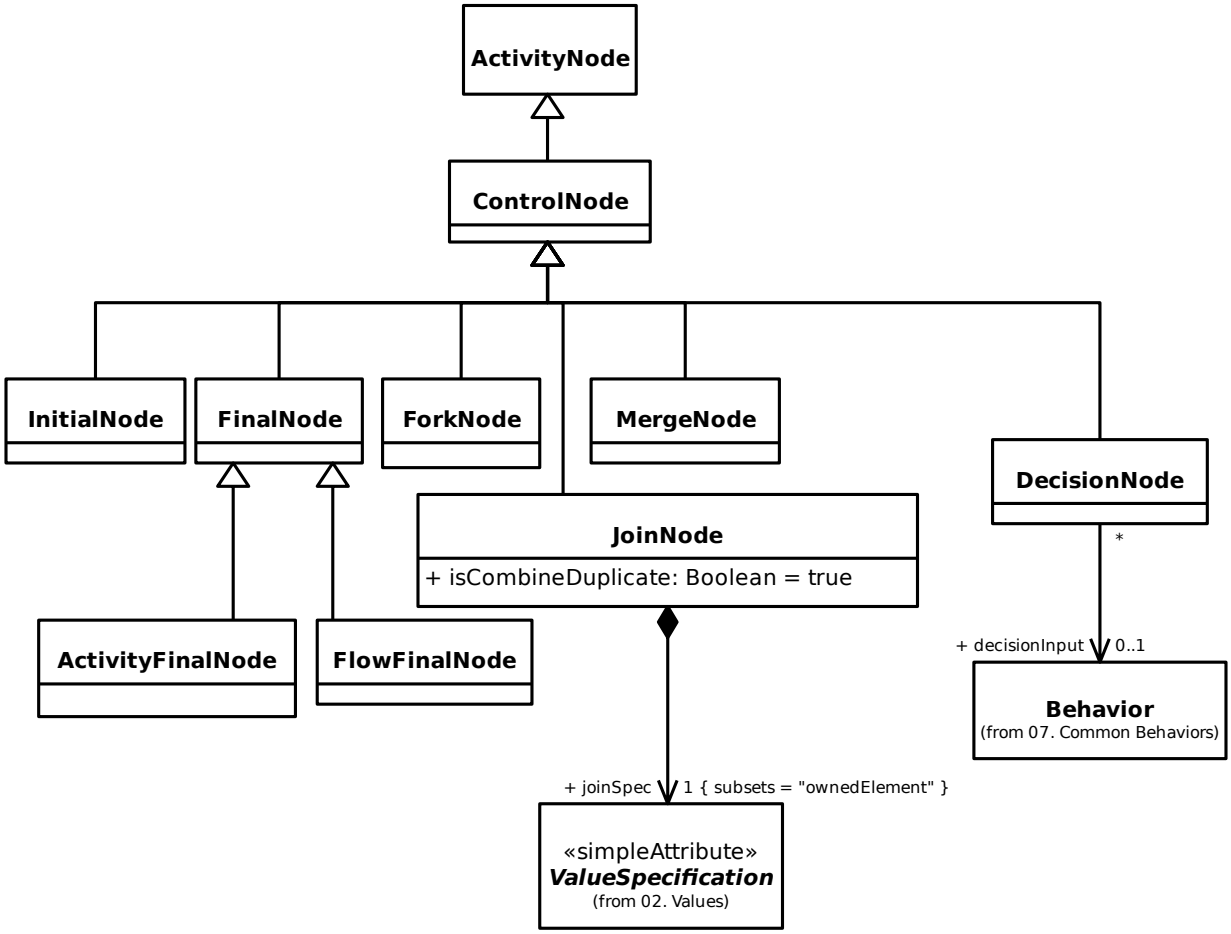


16.9 09. Activities

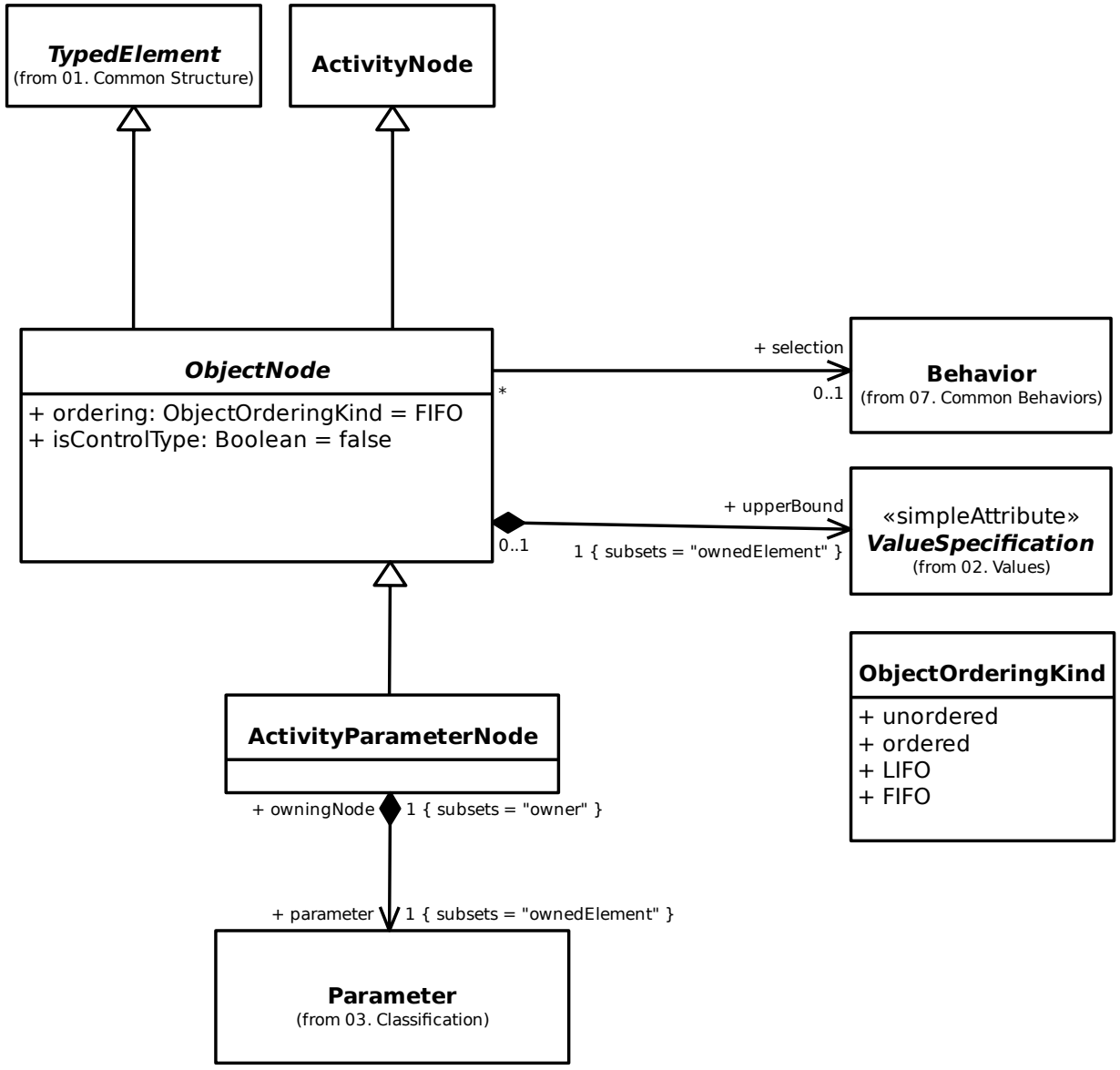
16.9.1 1. Activities



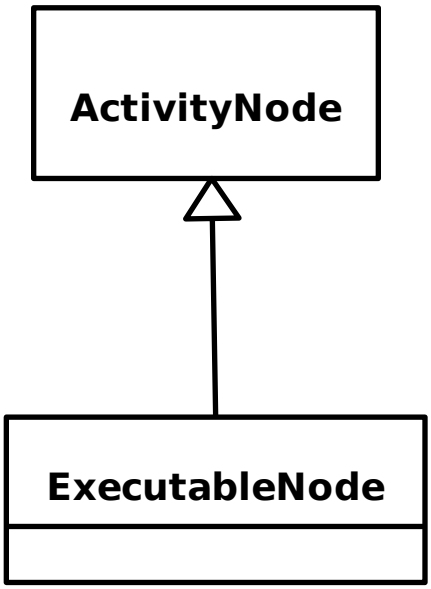
16.9.2 2. Control Nodes



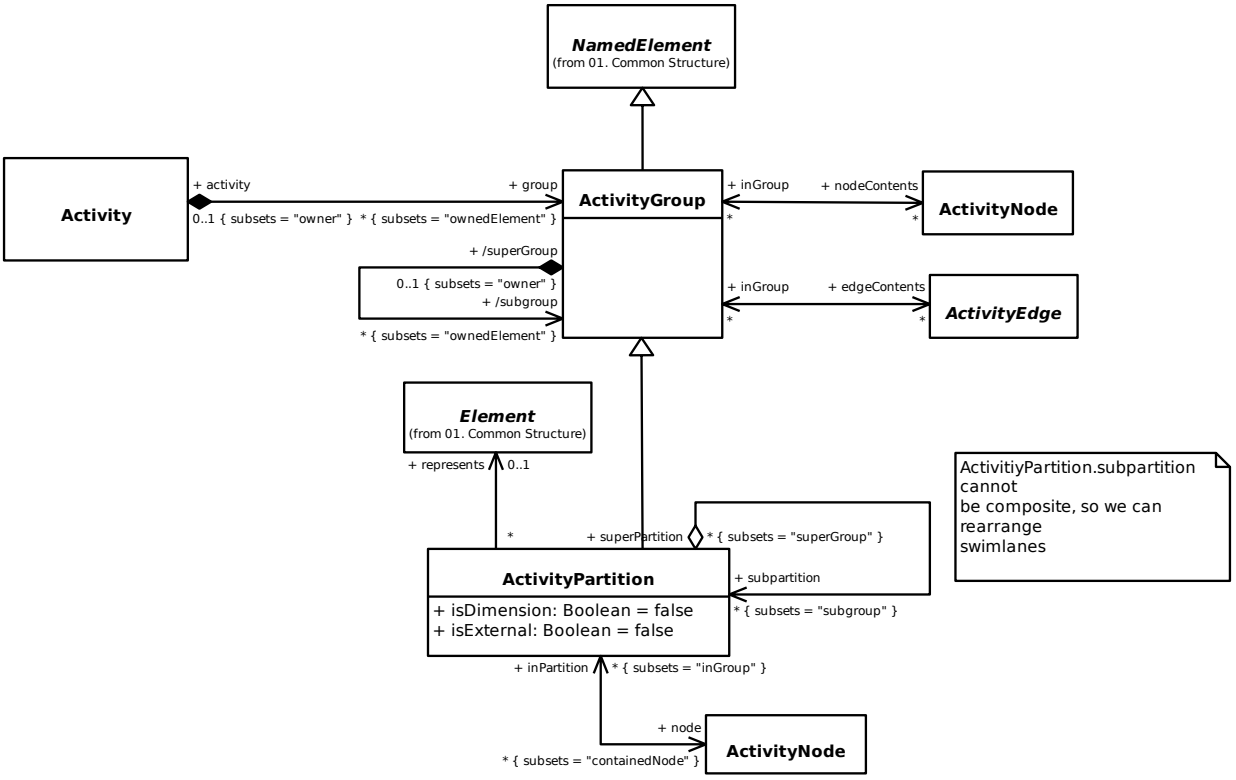
16.9.3 3. Object Nodes



16.9.4 4. Executable Nodes

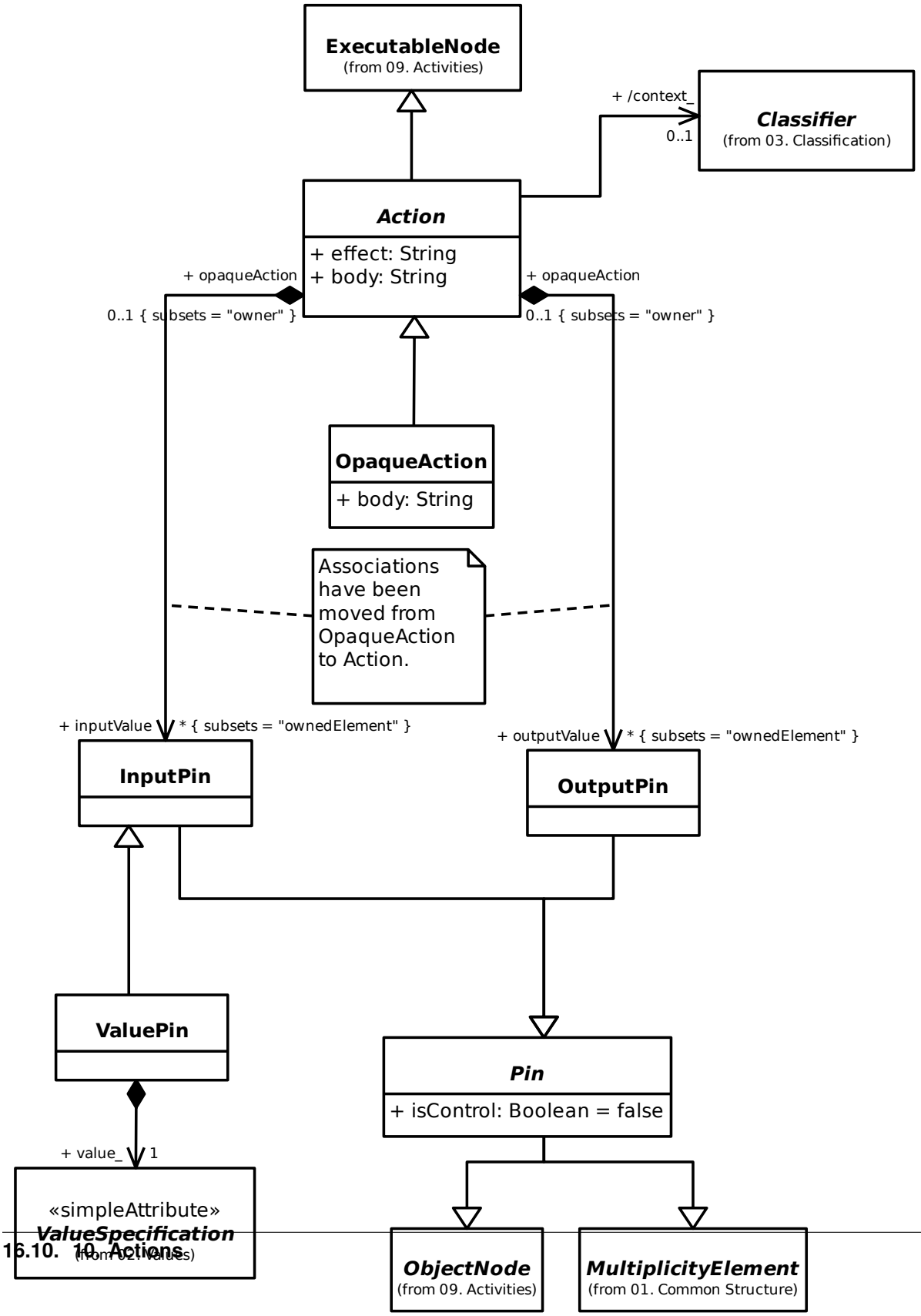


16.9.5 5. Activity Groups

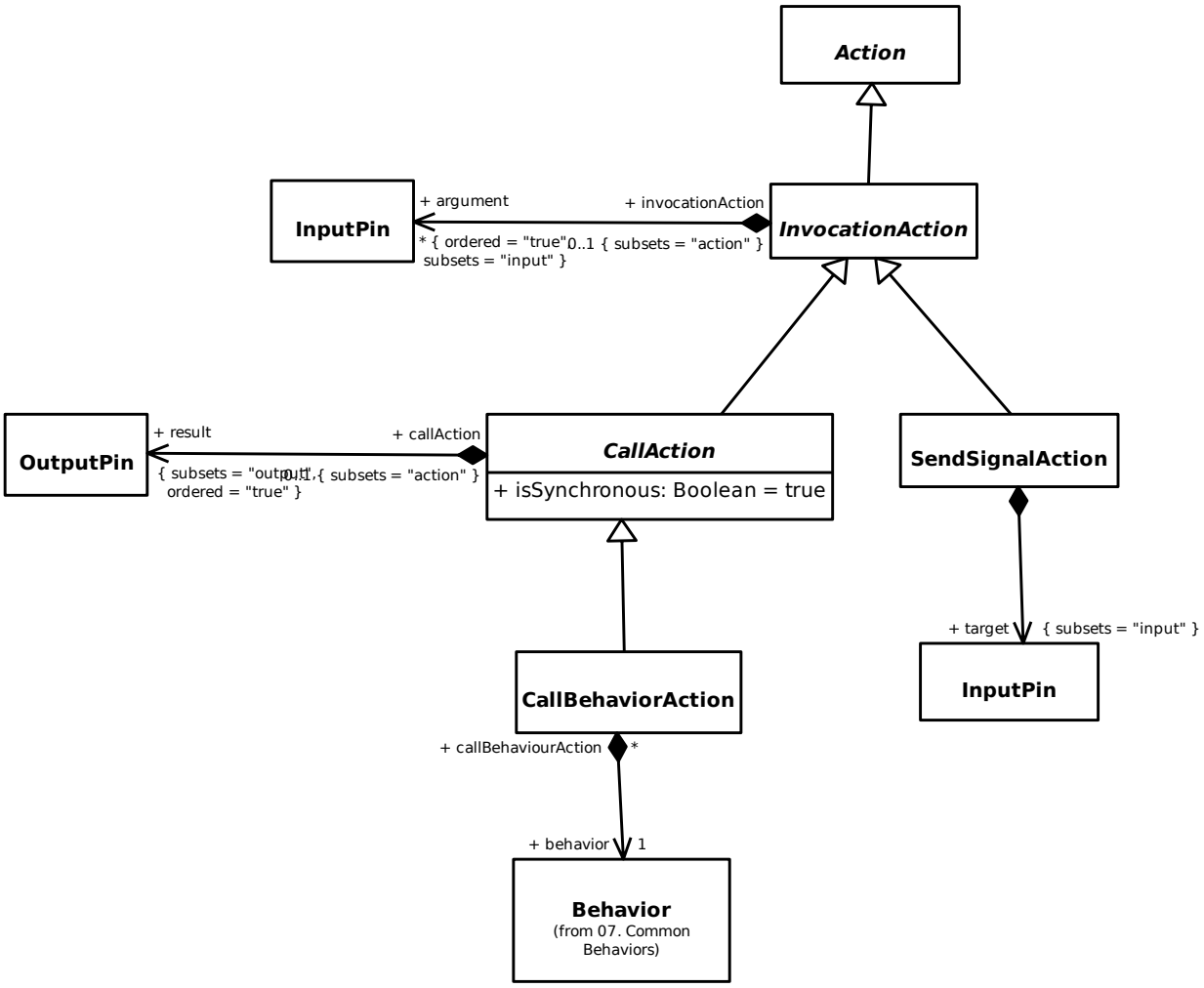


16.10 10. Actions

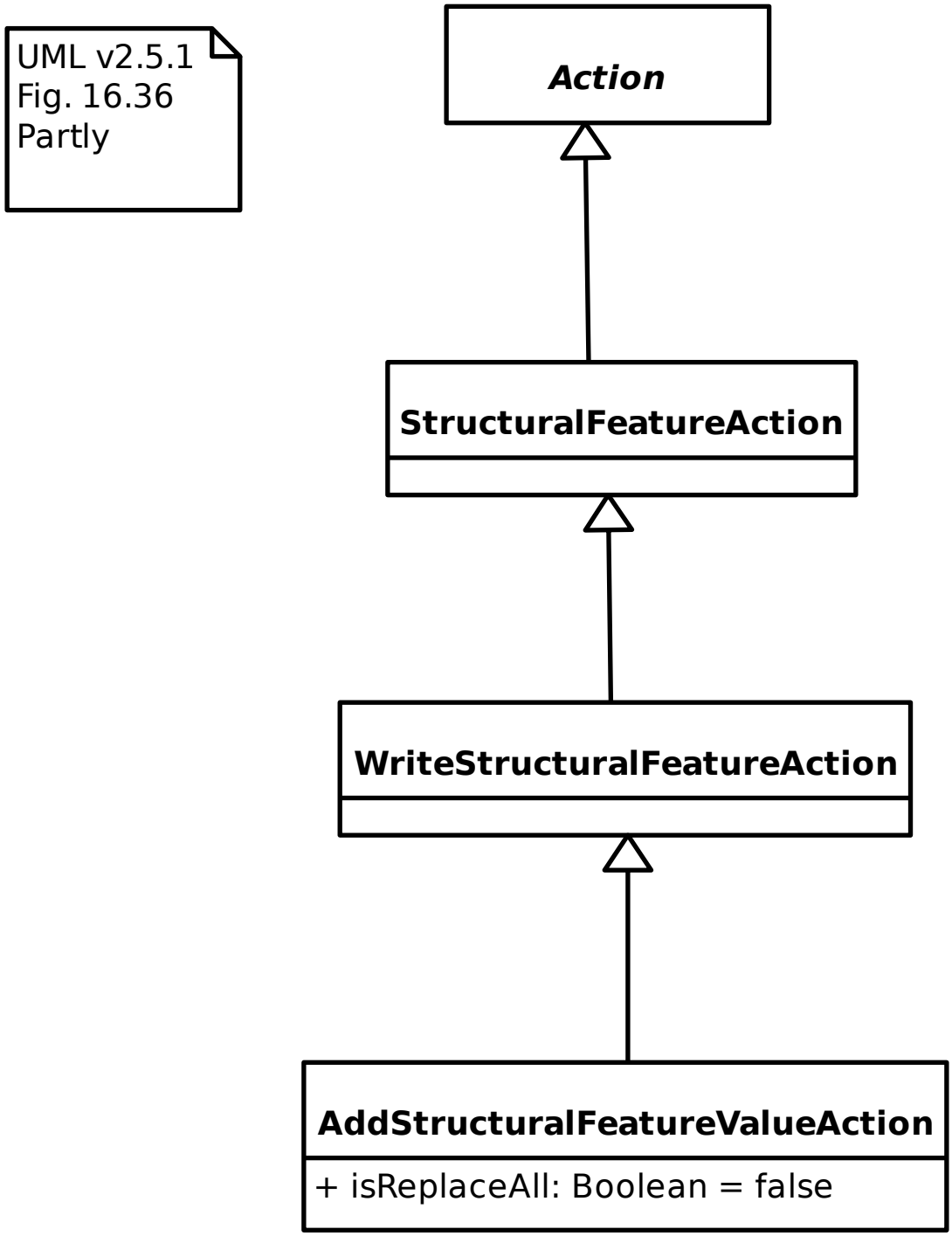
16.10.1 1. Actions



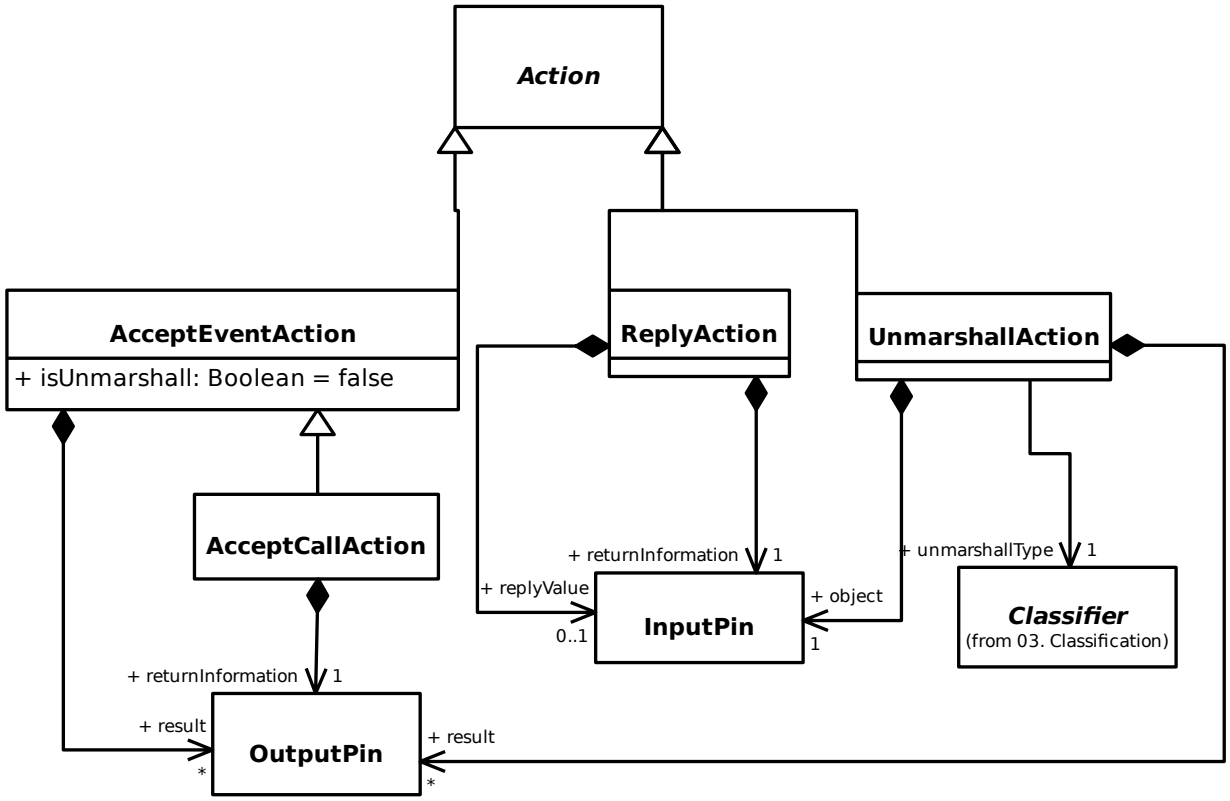
16.10.2 2. Invocation Actions



16.10.3 7. Structural Feature Actions

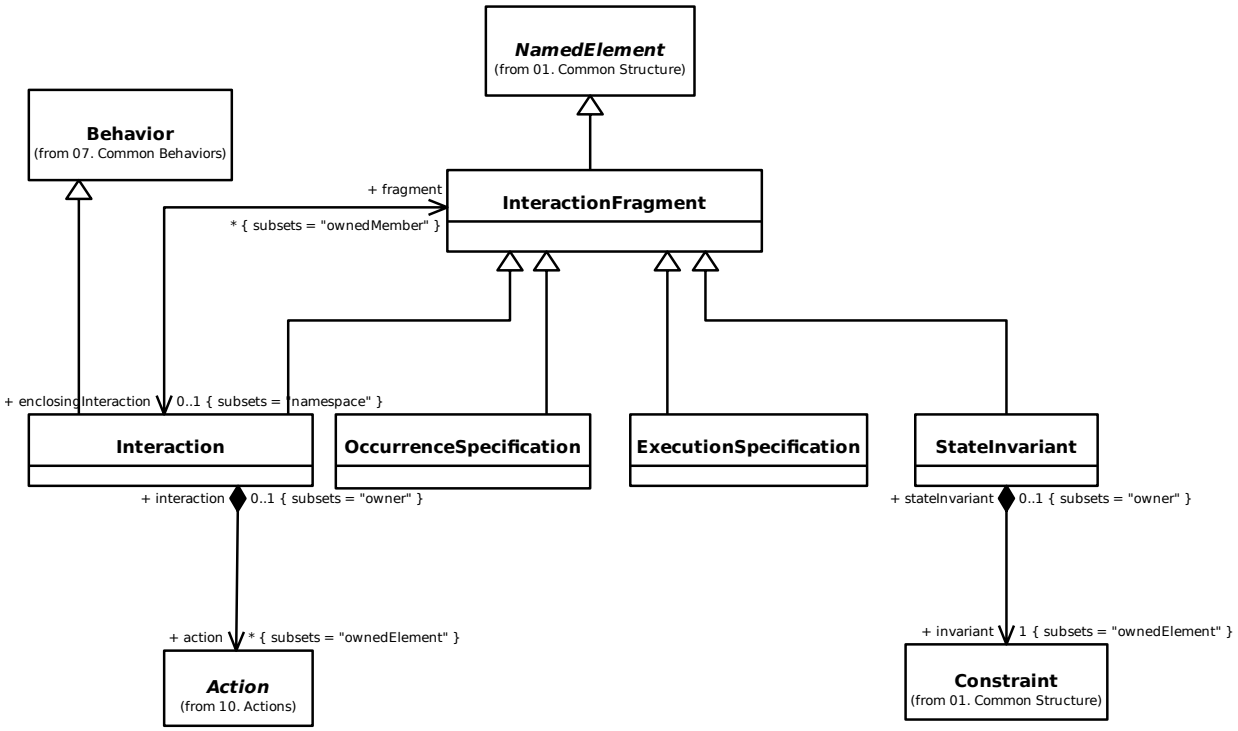


16.10.4 9. Accept Event Actions

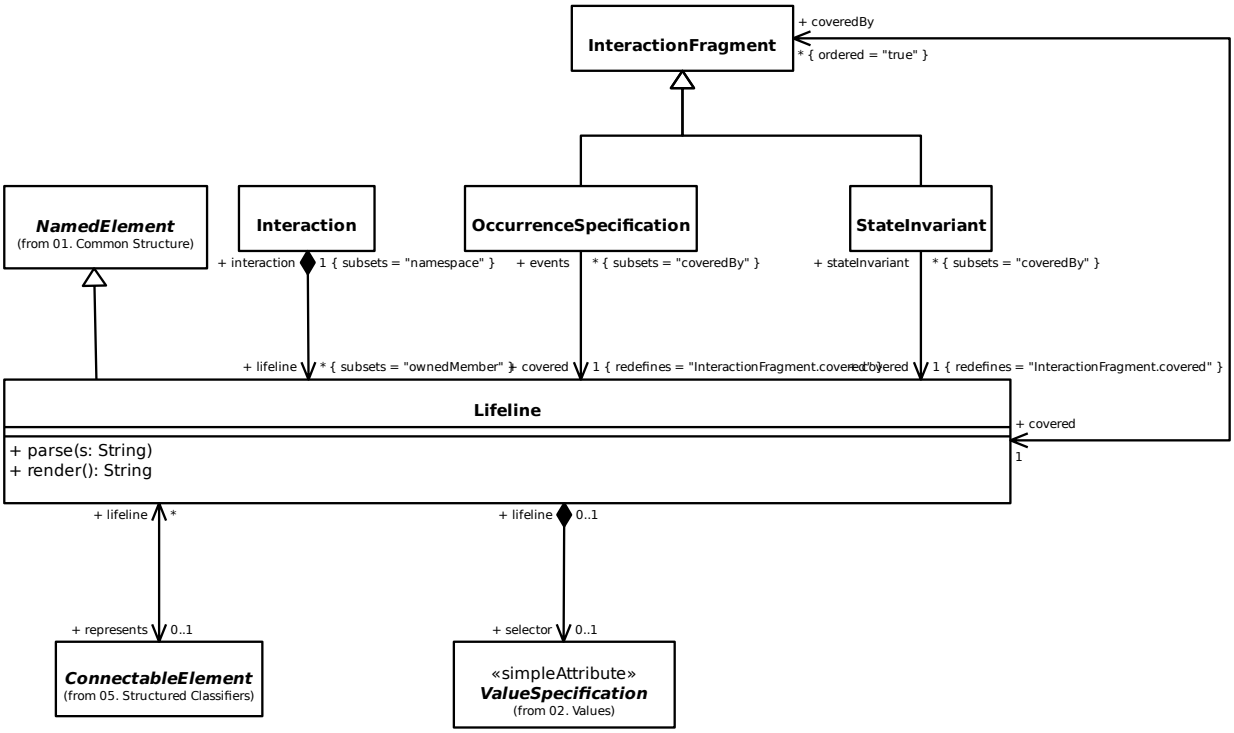


16.11 11. Interactions

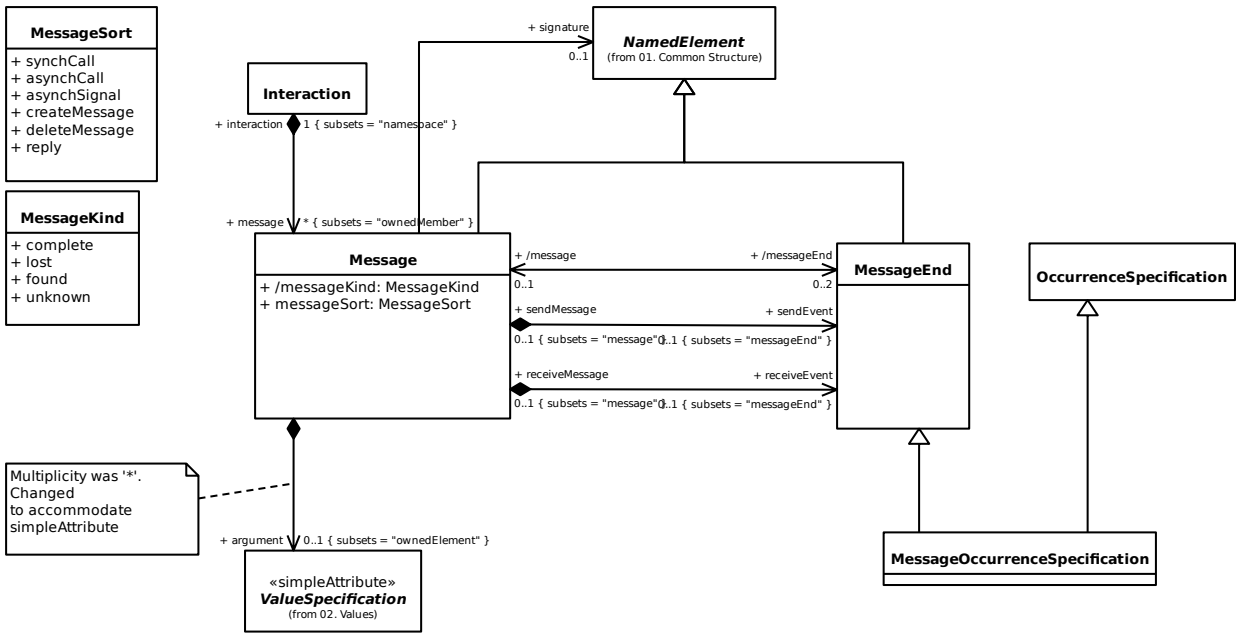
16.11.1 1. Interactions



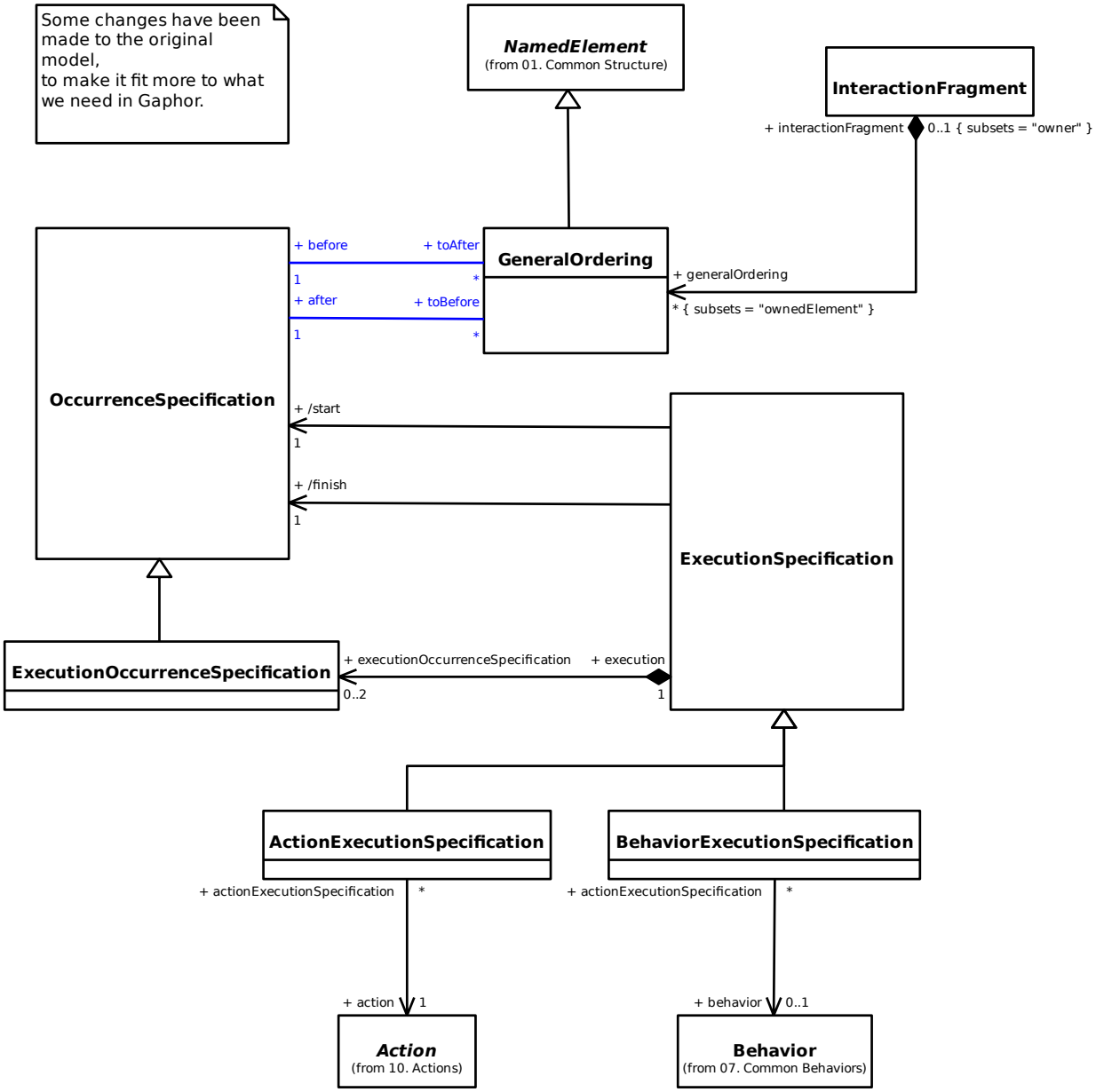
16.11.2 2. Lifelines



16.11.3 3. Messages

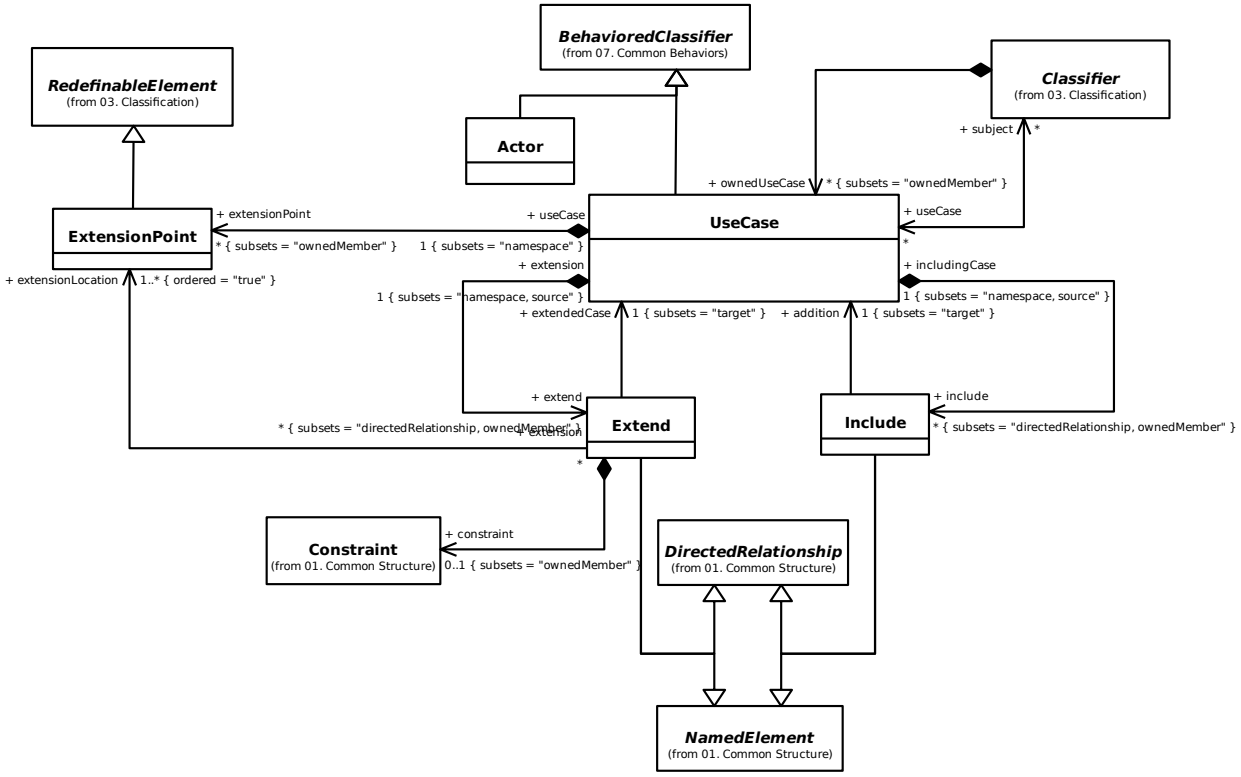


16.11.4 4. Occurrences



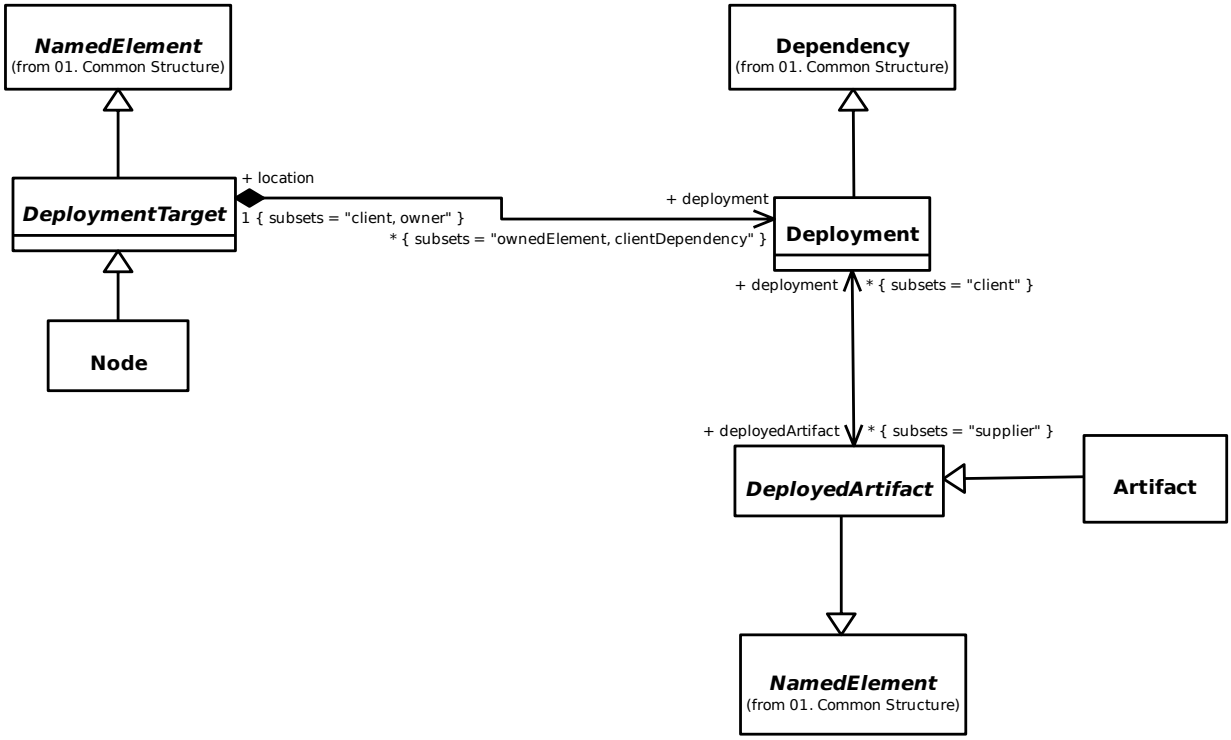
16.12 12. Use Cases

16.12.1 UseCases

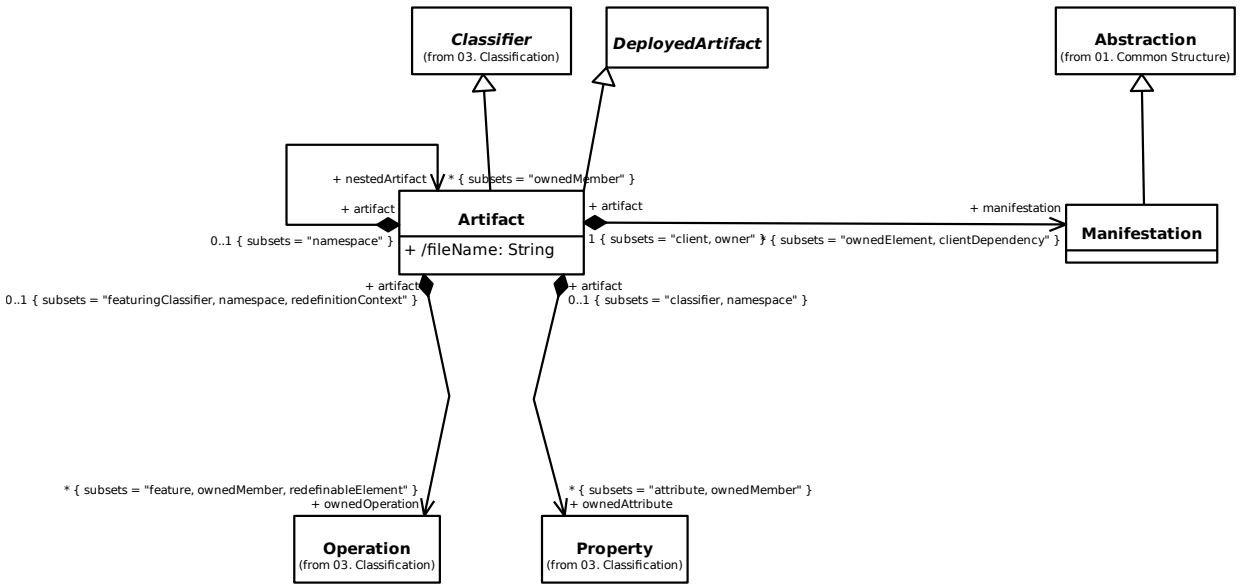


16.13 13. Deployments

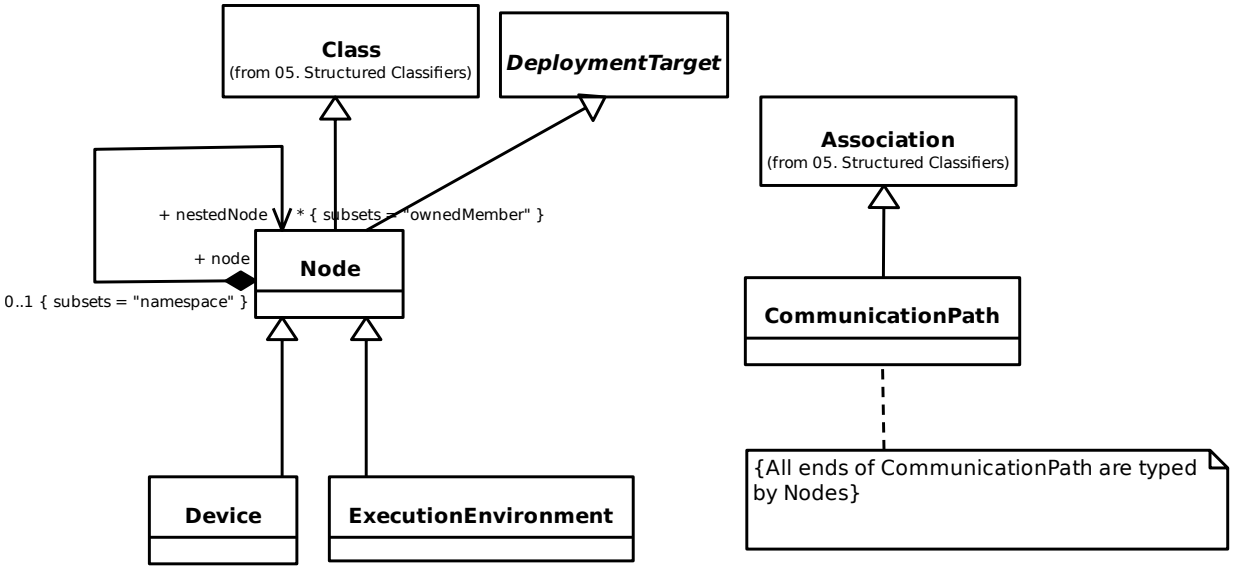
16.13.1 1. Deployments



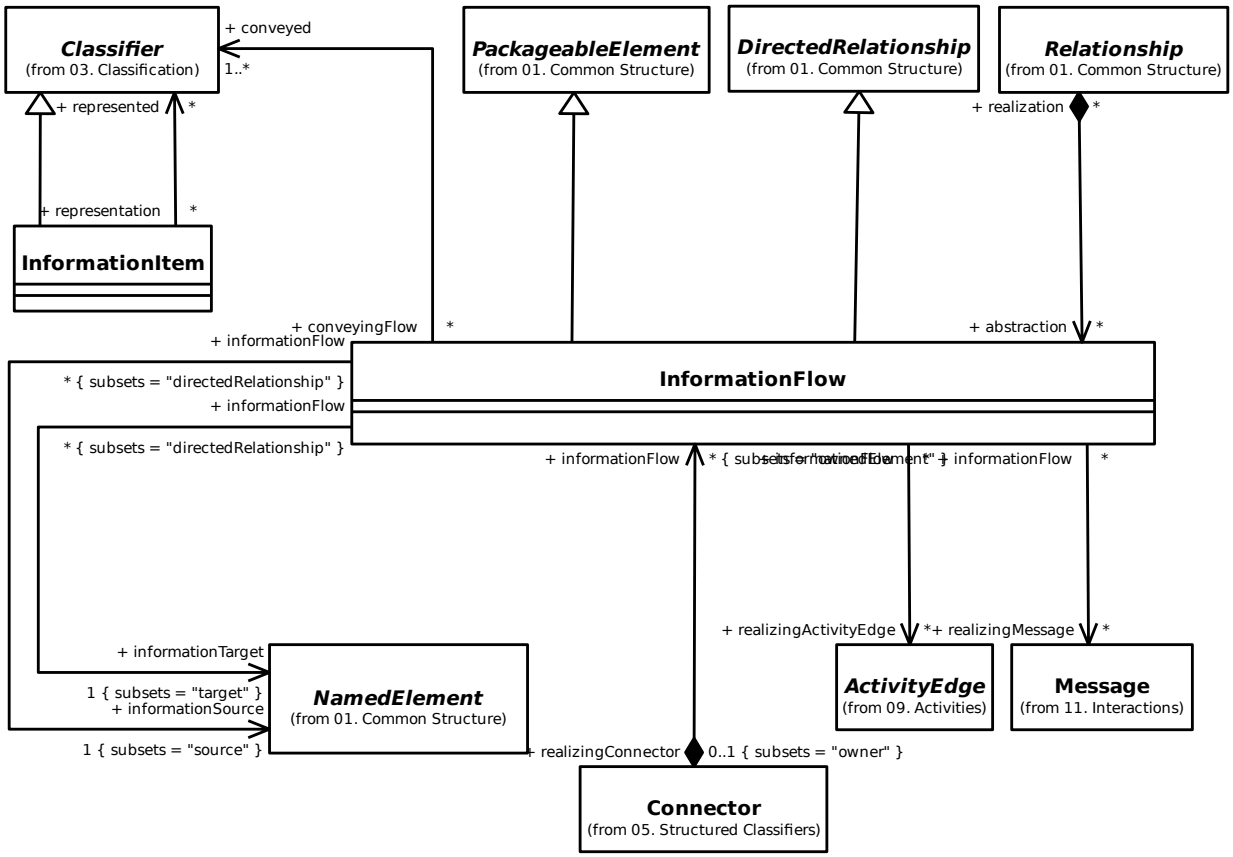
16.13.2 2. Artifacts



16.13.3 3. Nodes

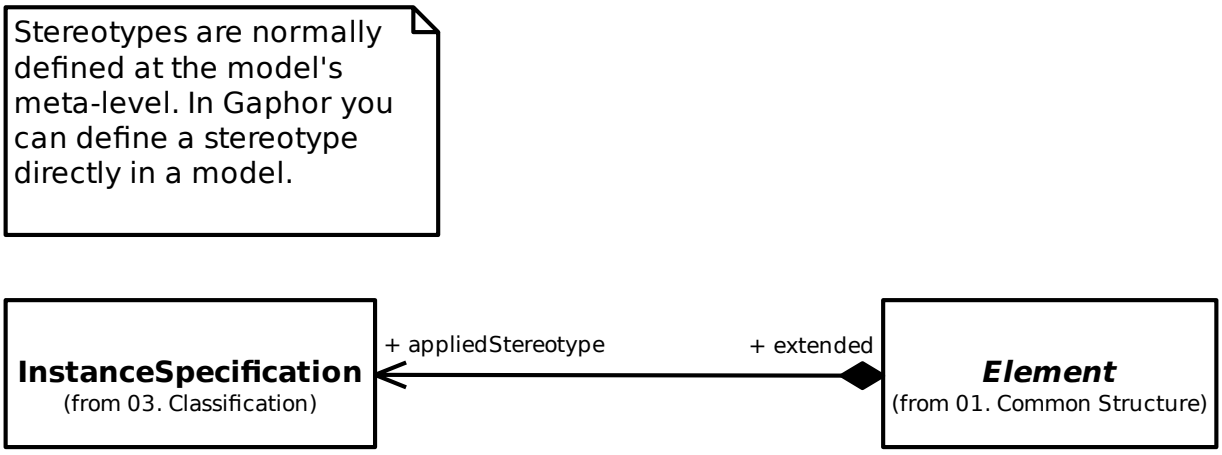


16.14 14. Information Flows



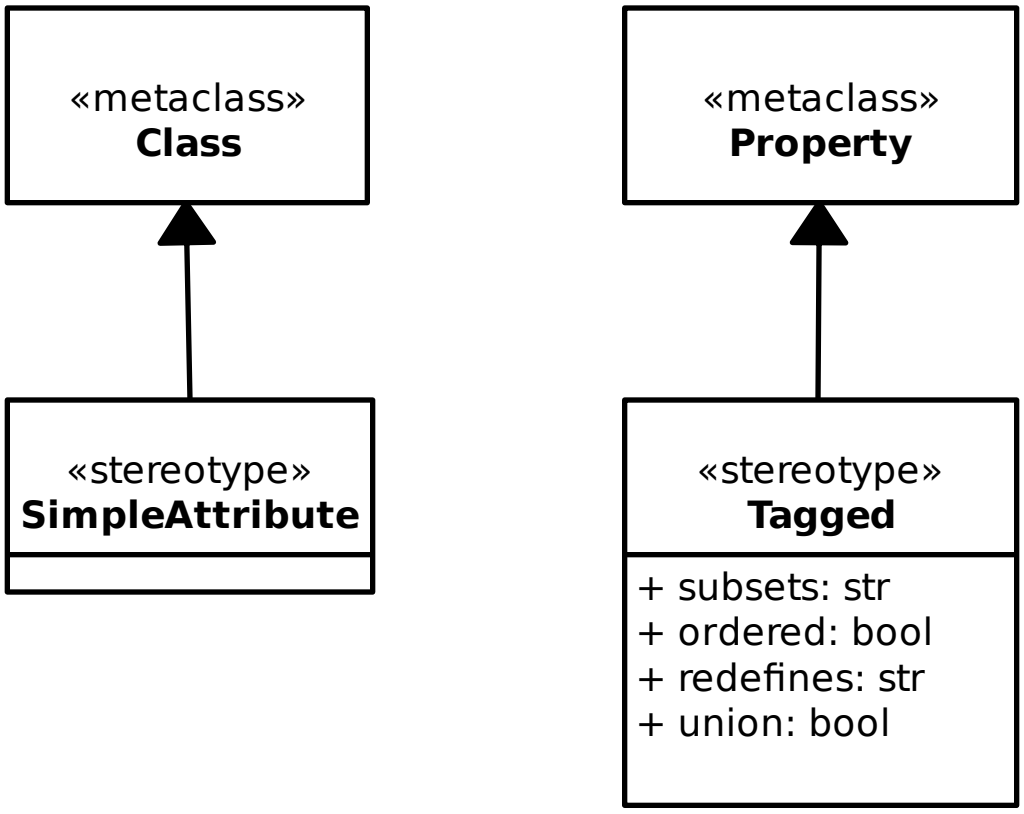
16.15 A. Gaphor Specific Constructs

16.15.1 1. Stereotype Applications



16.16 B. Gaphor Profile

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

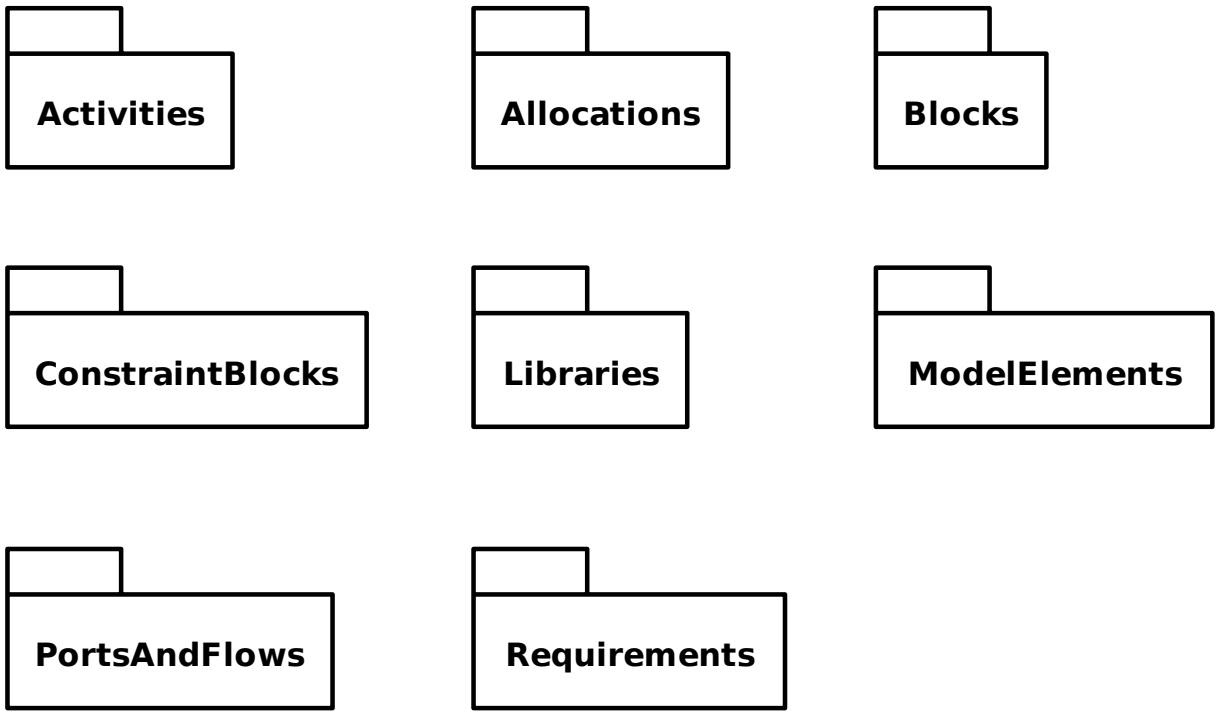


CHAPTER

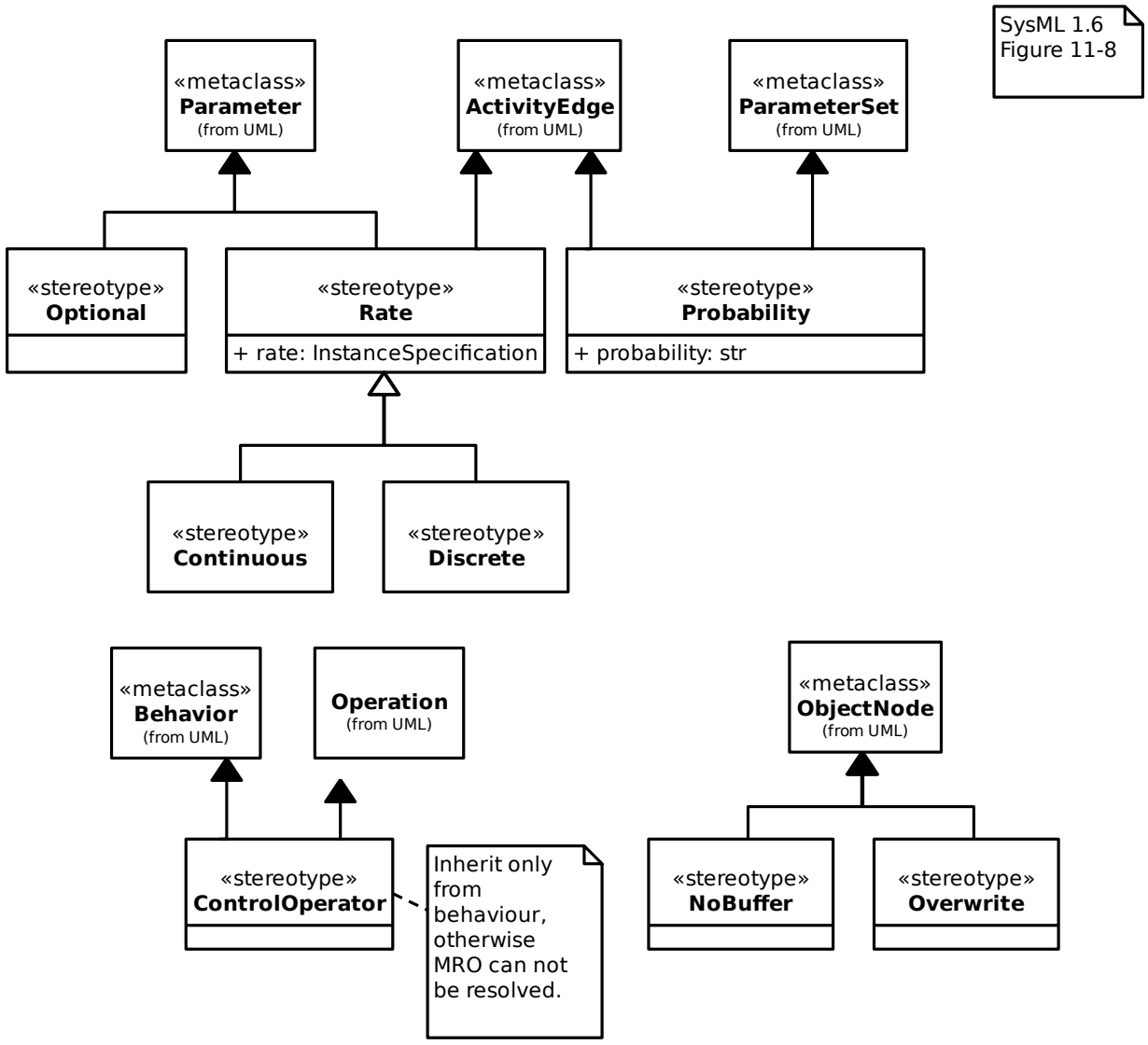
SEVENTEEN

SYSTEMS MODELING LANGUAGE

Gaphor implements part of the SysML 1.6 specification.

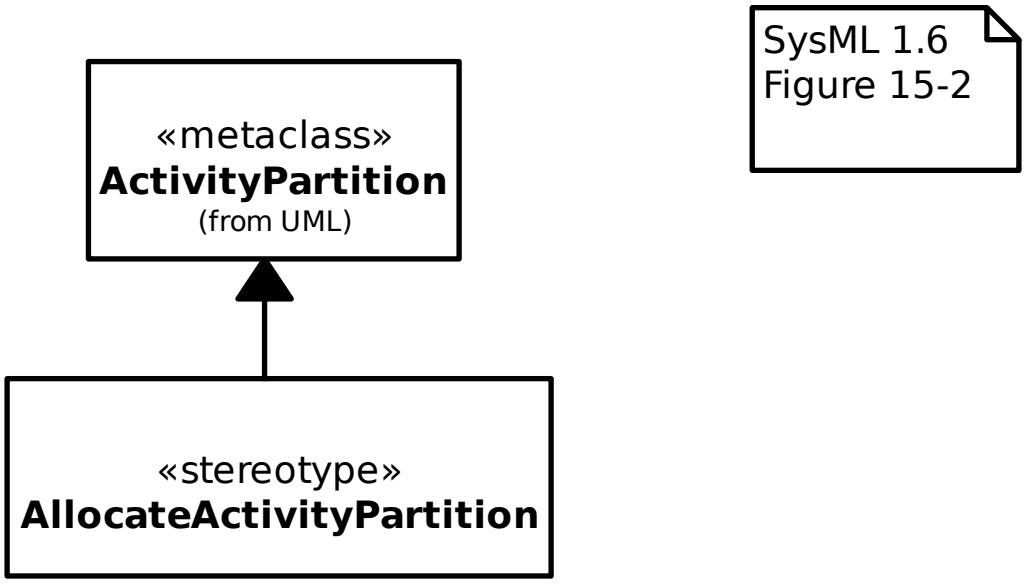


17.1 Activities

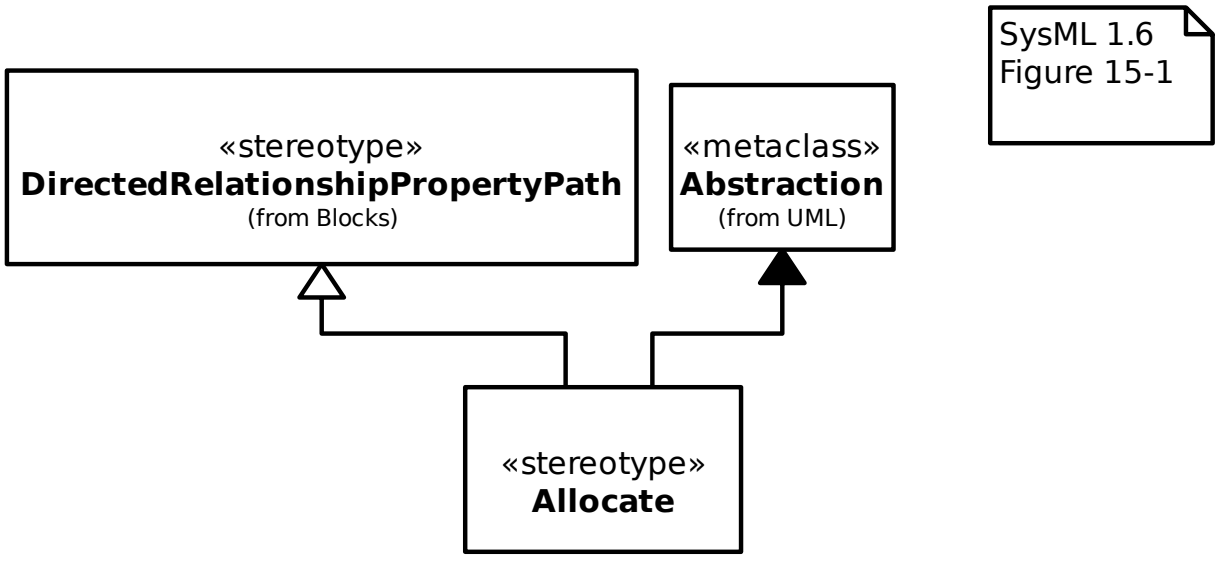


17.2 Allocations

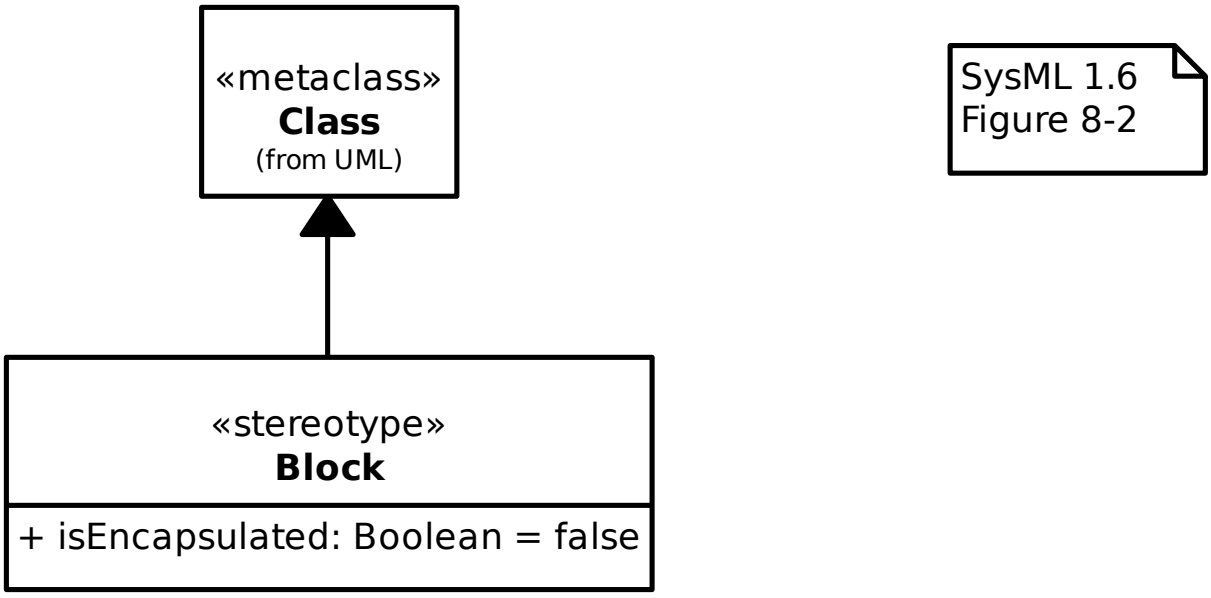
17.2.1 AllocatedActivityPartition



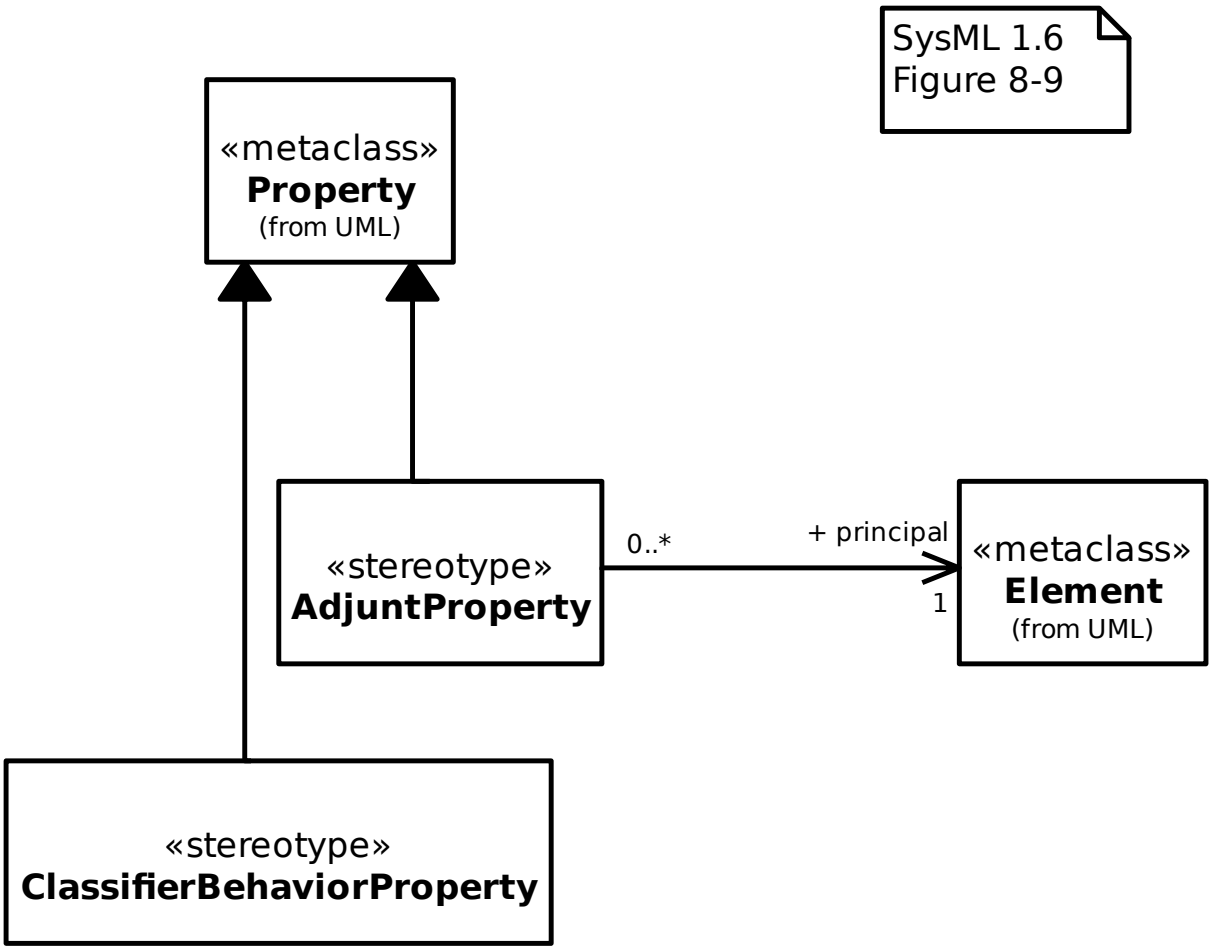
17.2.2 Allocation



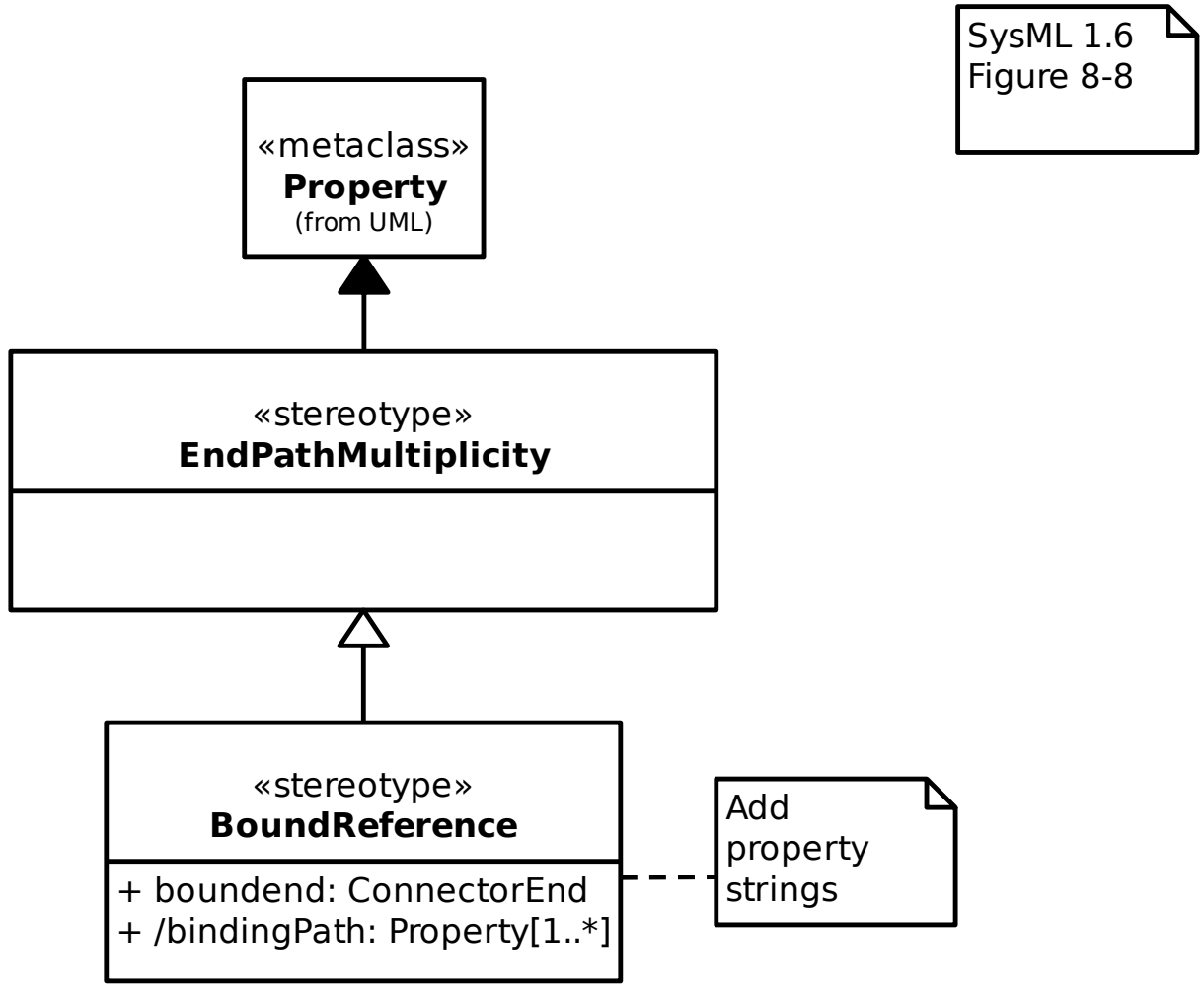
17.3 Blocks


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

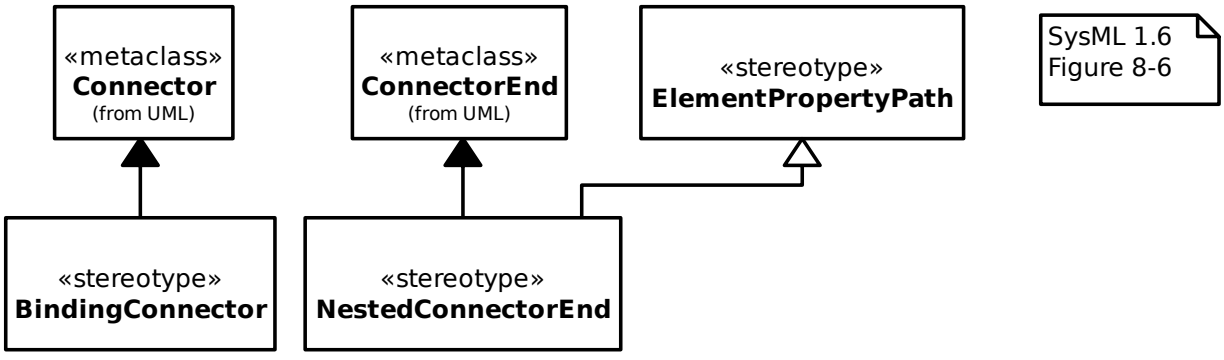
17.3.1 Adjunt and Classifier Behavior Properties



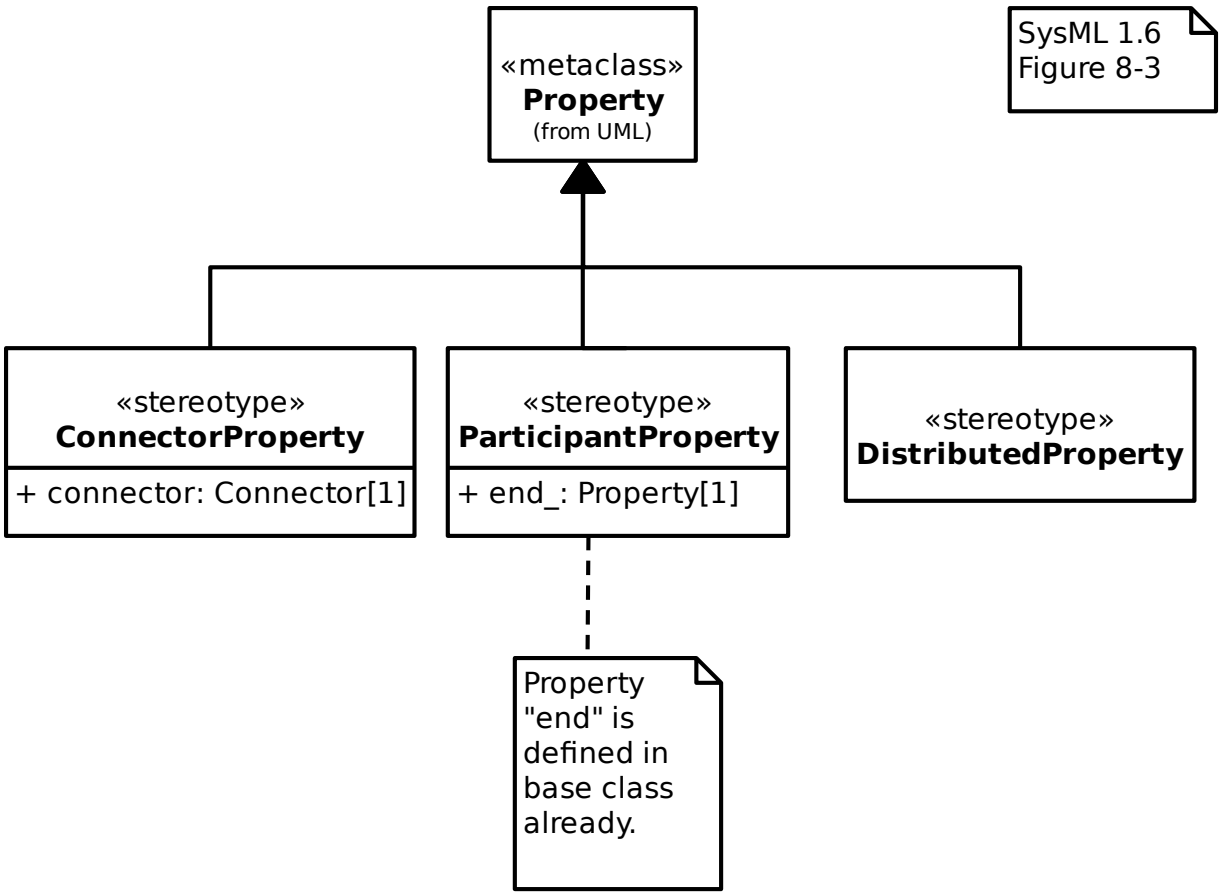
17.3.2 Bound References



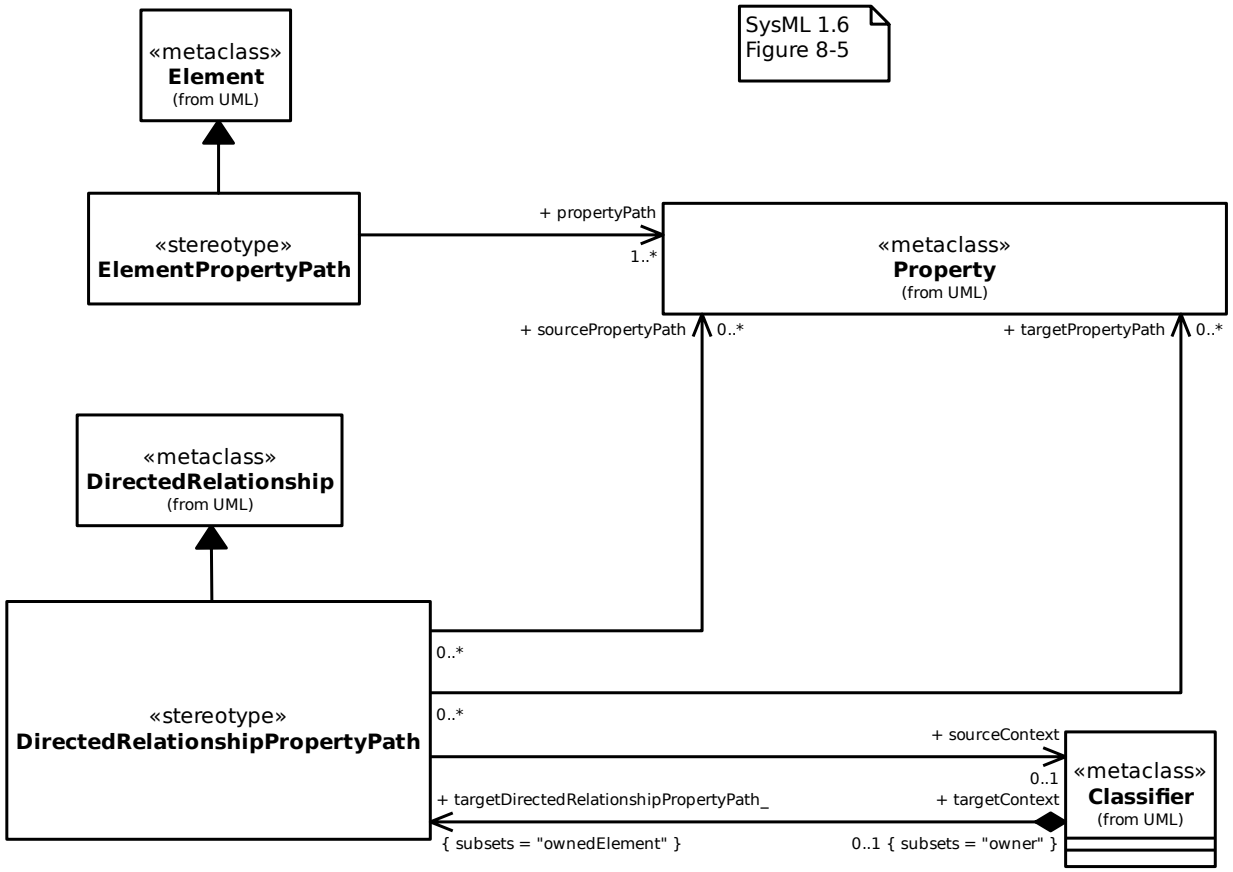
17.3.3 Connector Ends



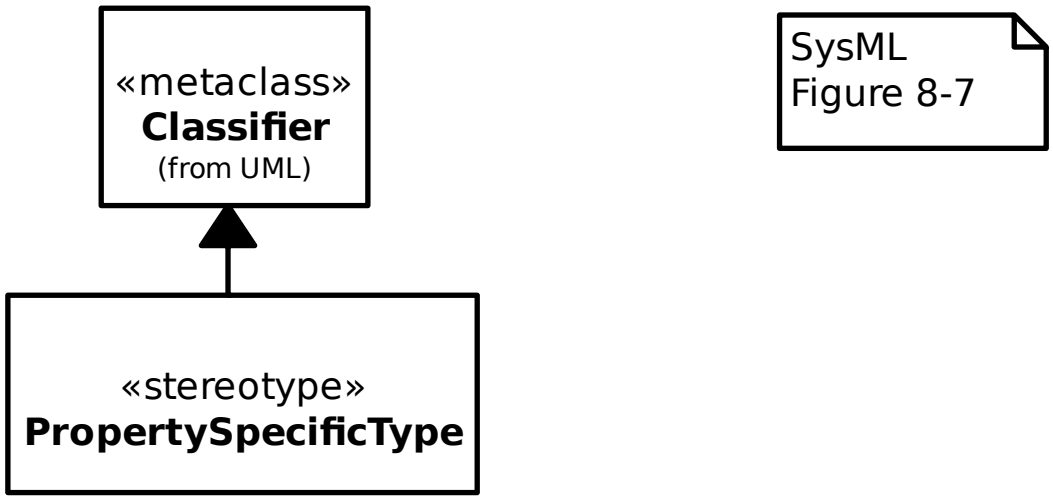
17.3.4 Properties



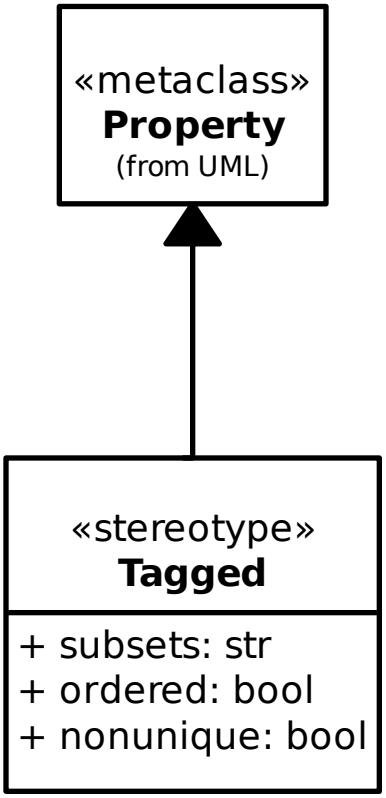
17.3.5 Property Paths



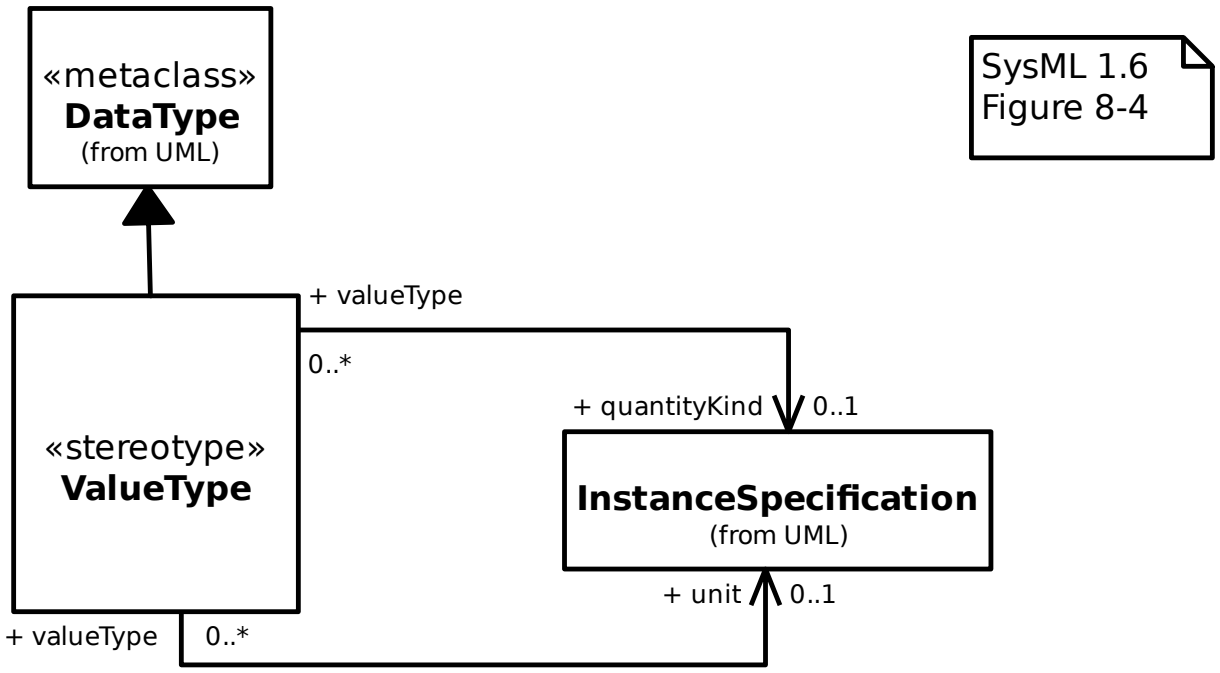
17.3.6 Property-Specific Types



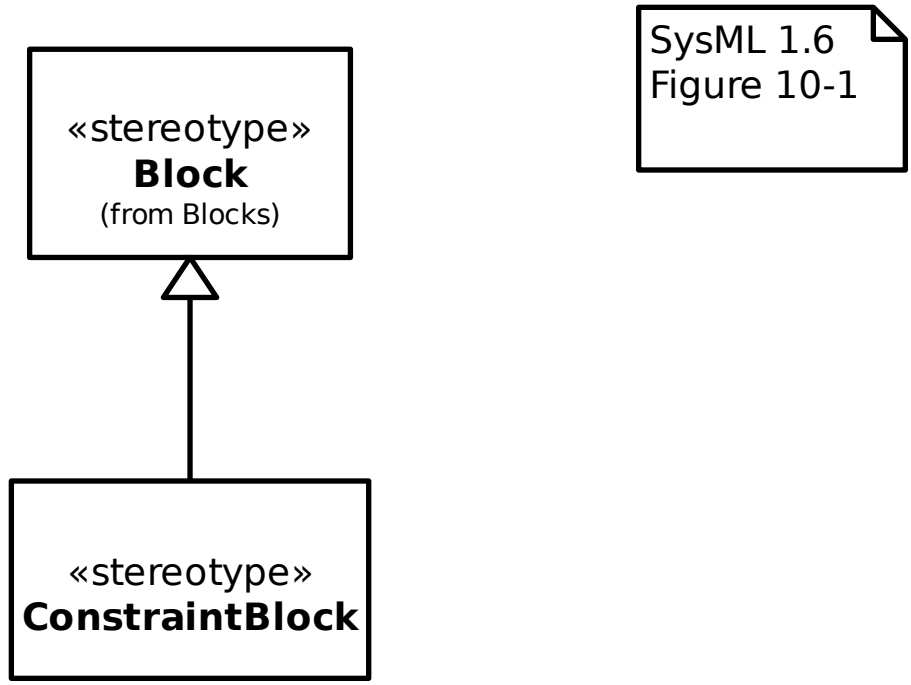
17.3.7 Property Strings



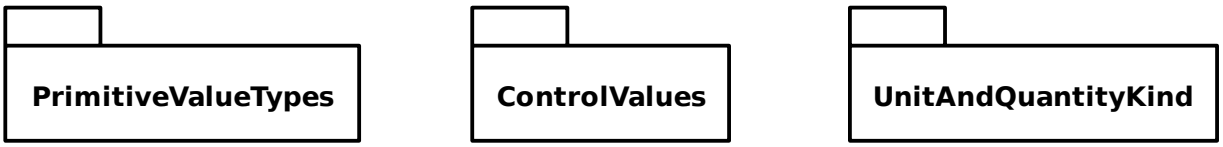
17.3.8 Value Types



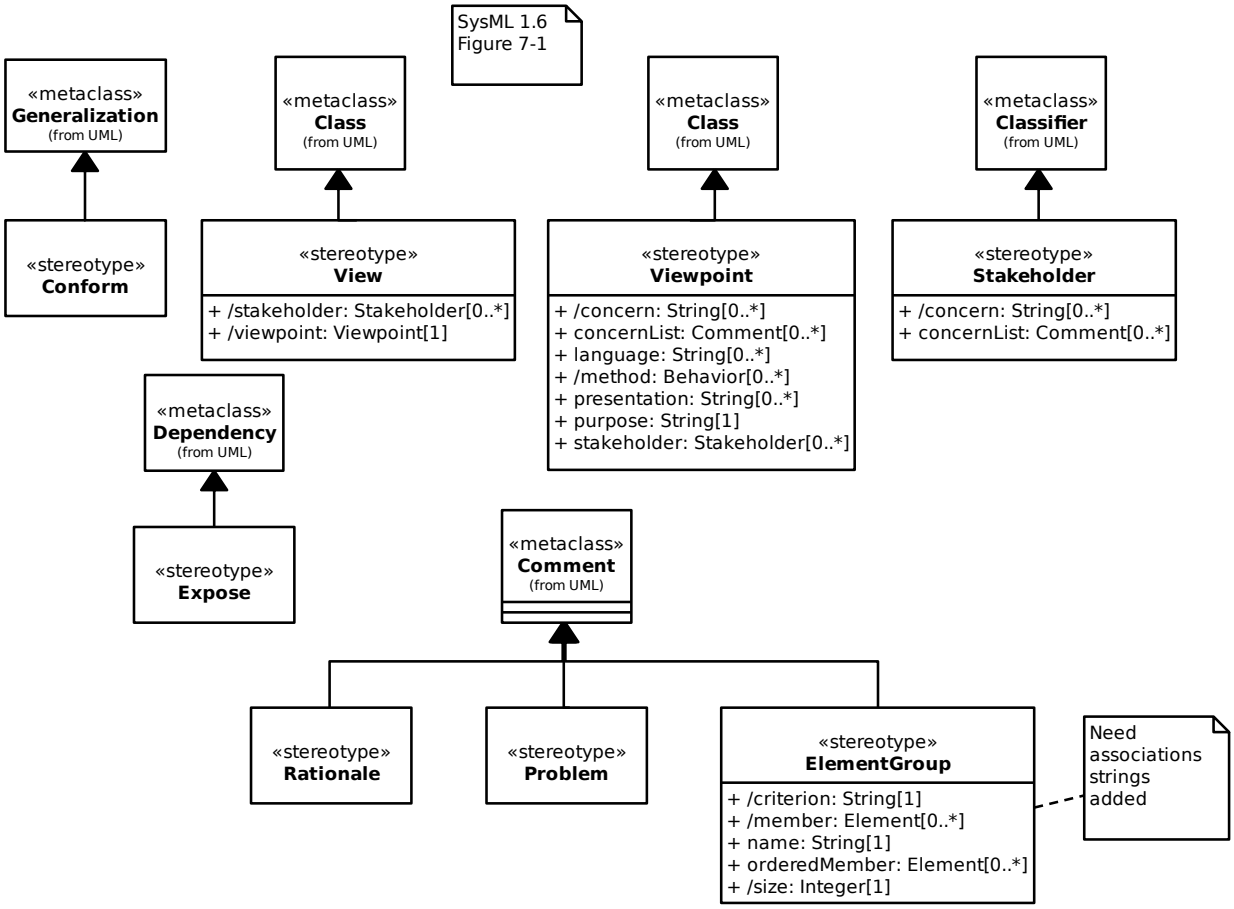
17.4 ConstraintBlocks



17.5 Libraries

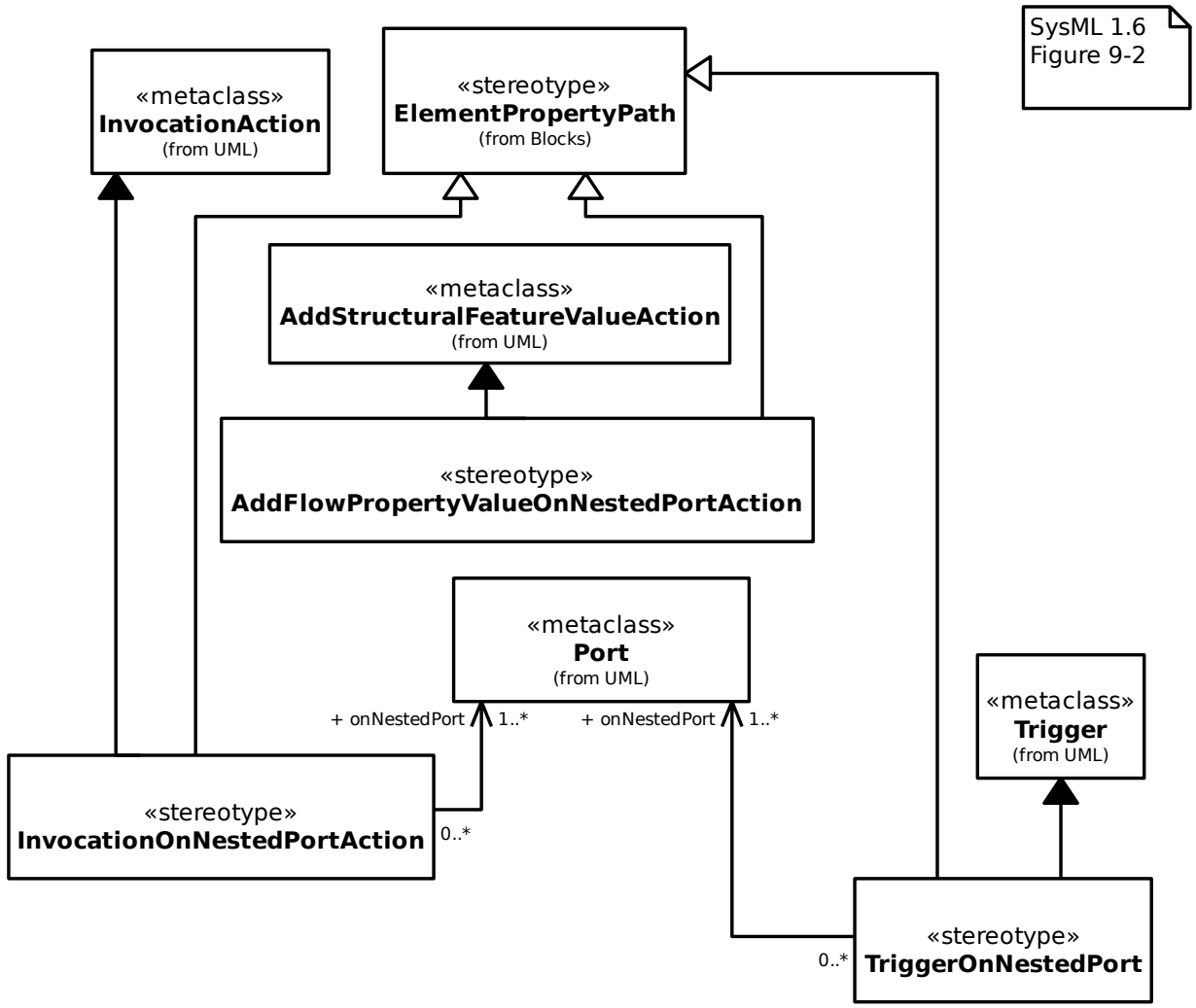


17.6 ModelElements

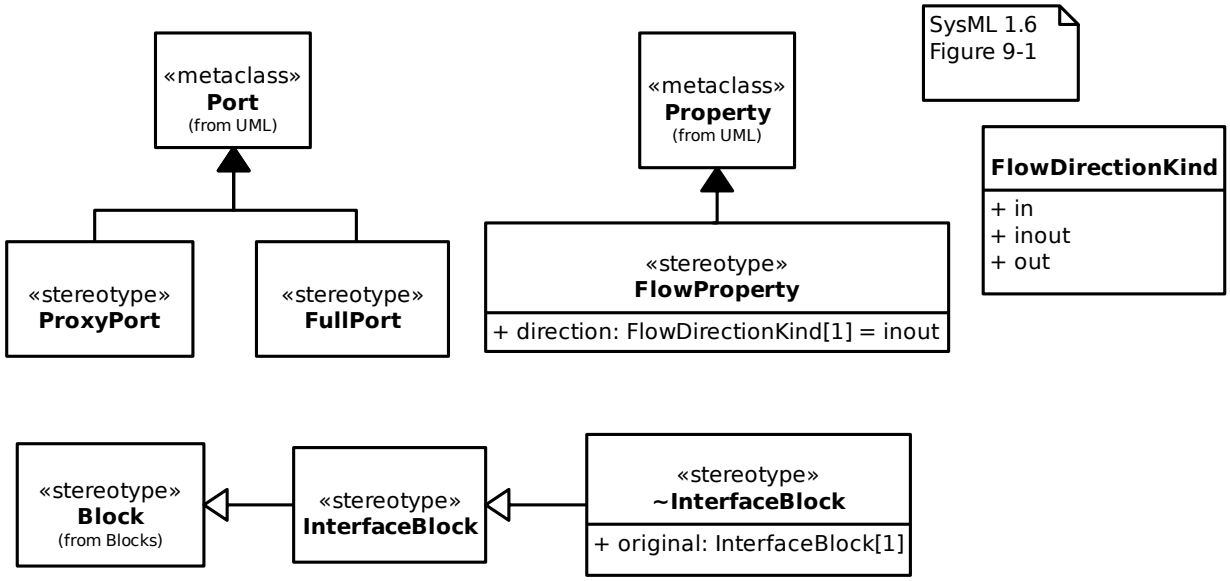


17.7 PortsAndFlows

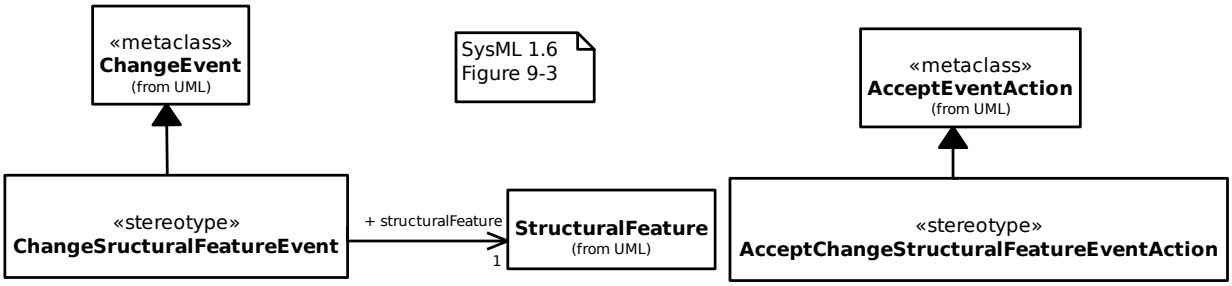
17.7.1 Actions on Nested Ports



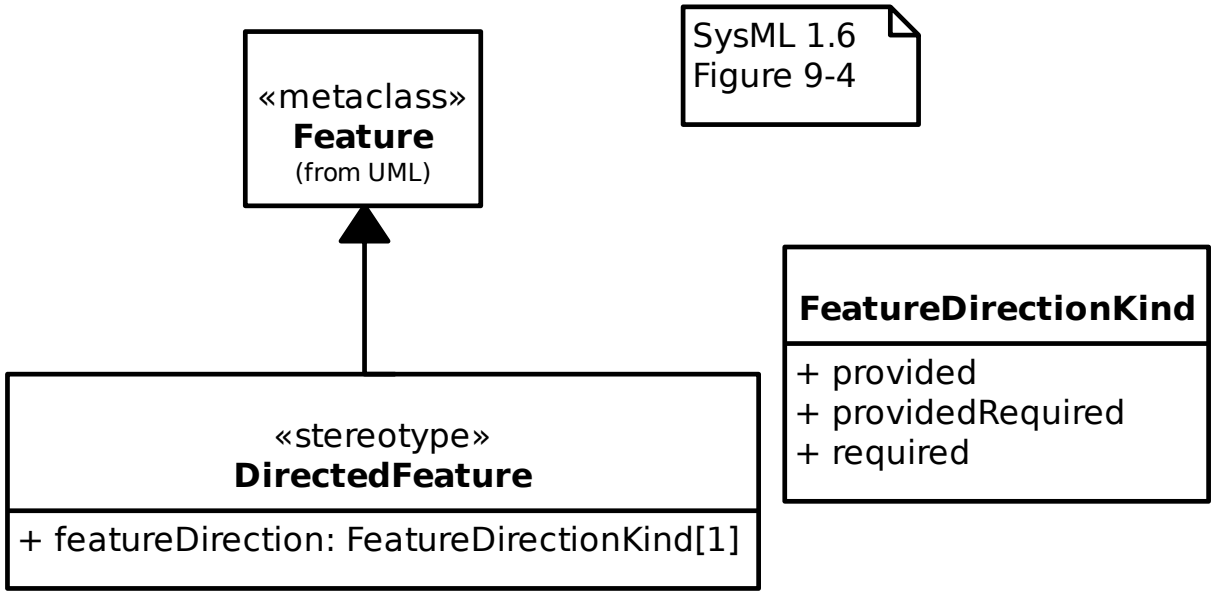
17.7.2 Port Stereotypes



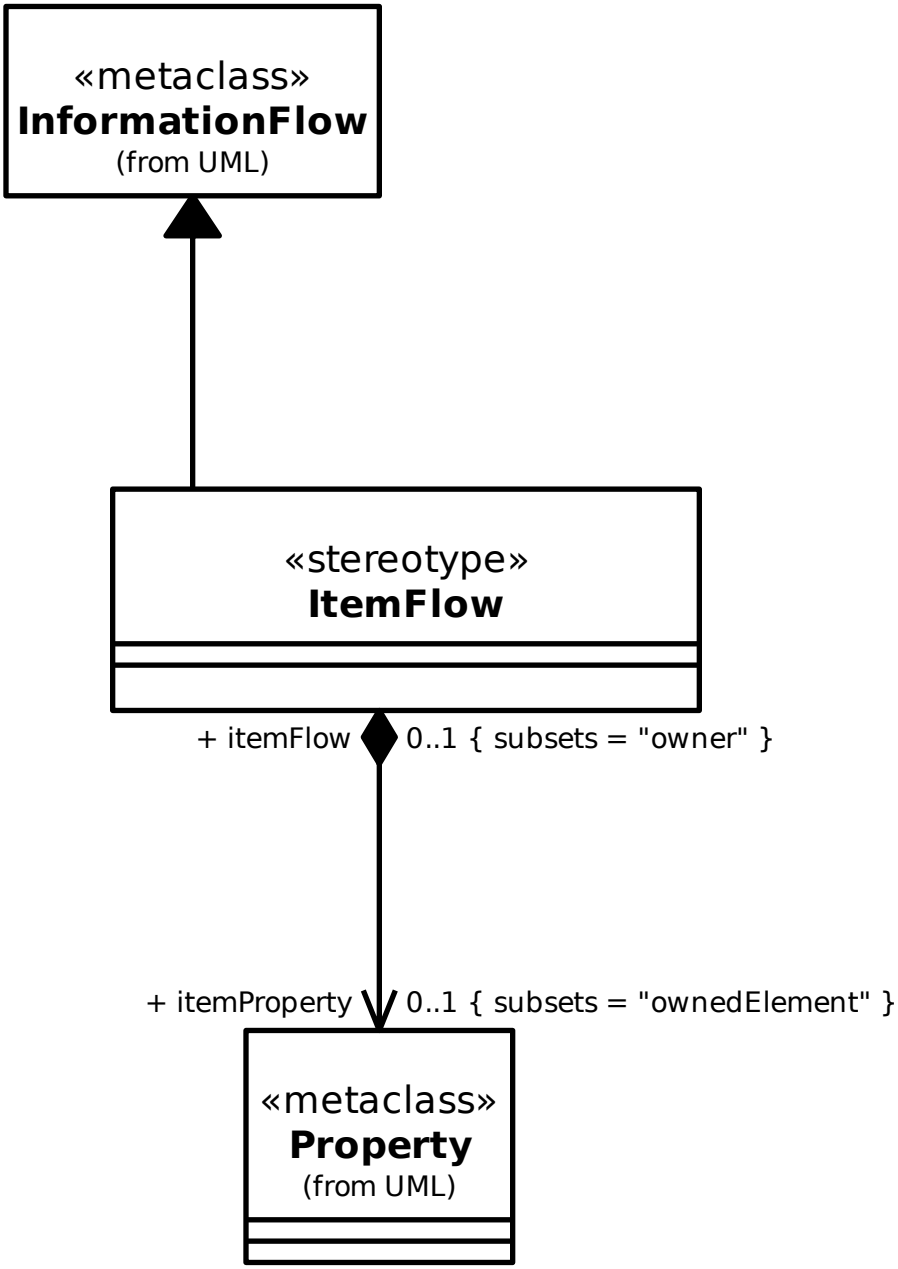
17.7.3 Property Value Change Events



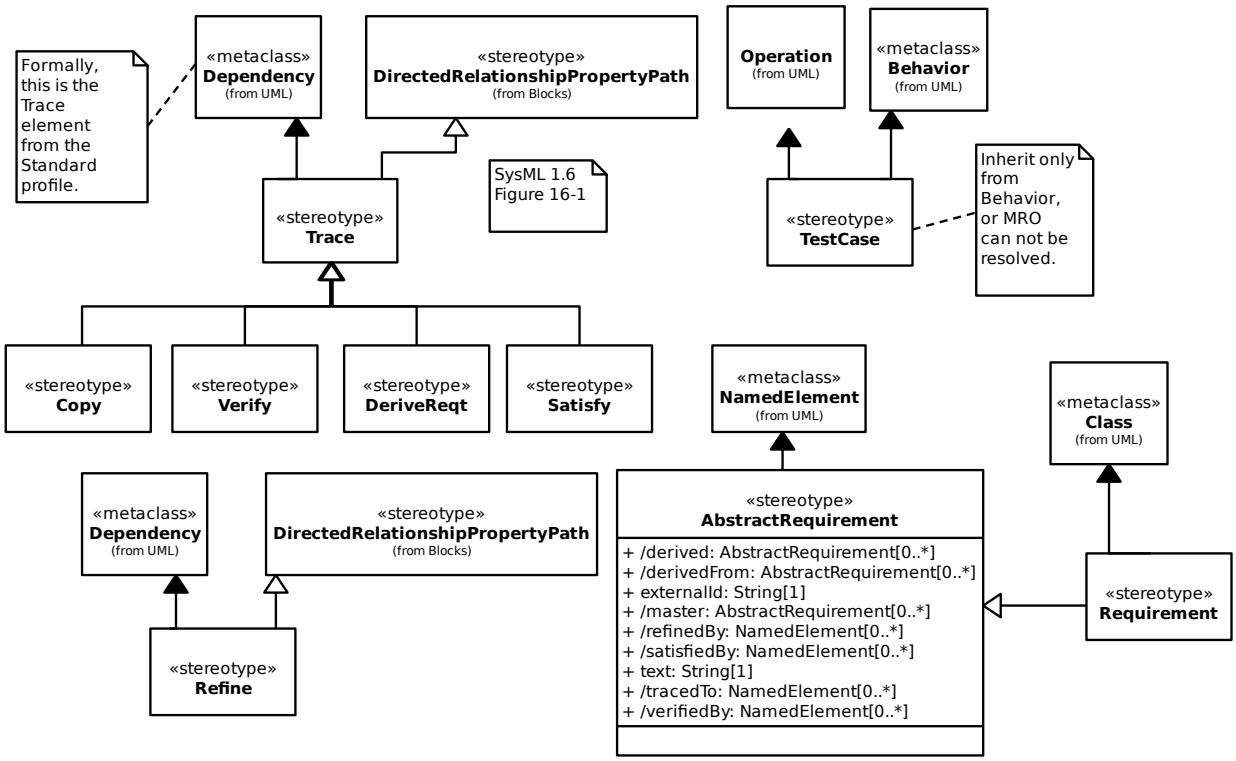
17.7.4 Provided and Required Features



17.7.5 Item Flow



17.8 Requirements

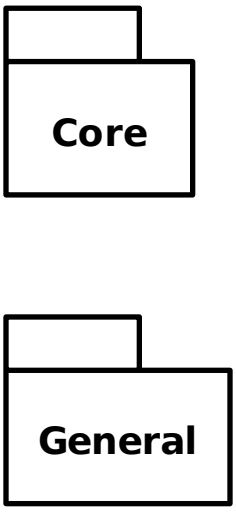


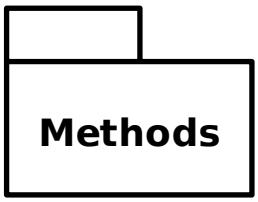
CHAPTER

EIGHTEEN

RISK ANALYSIS AND ASSESSMENT MODELING LANGUAGE

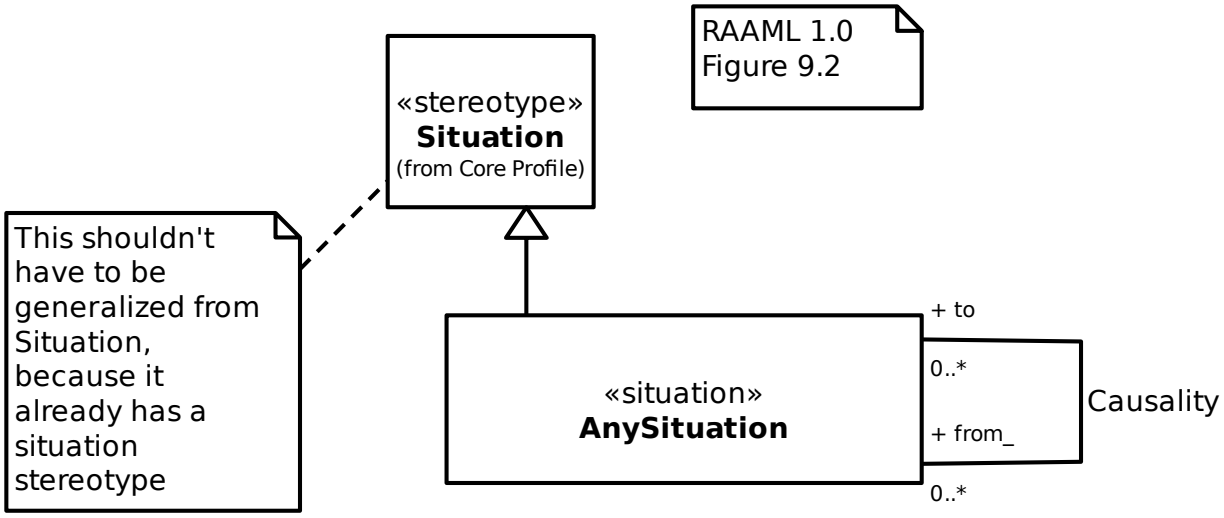
Gaphor implements parts of the RAAML 1.0 specification.



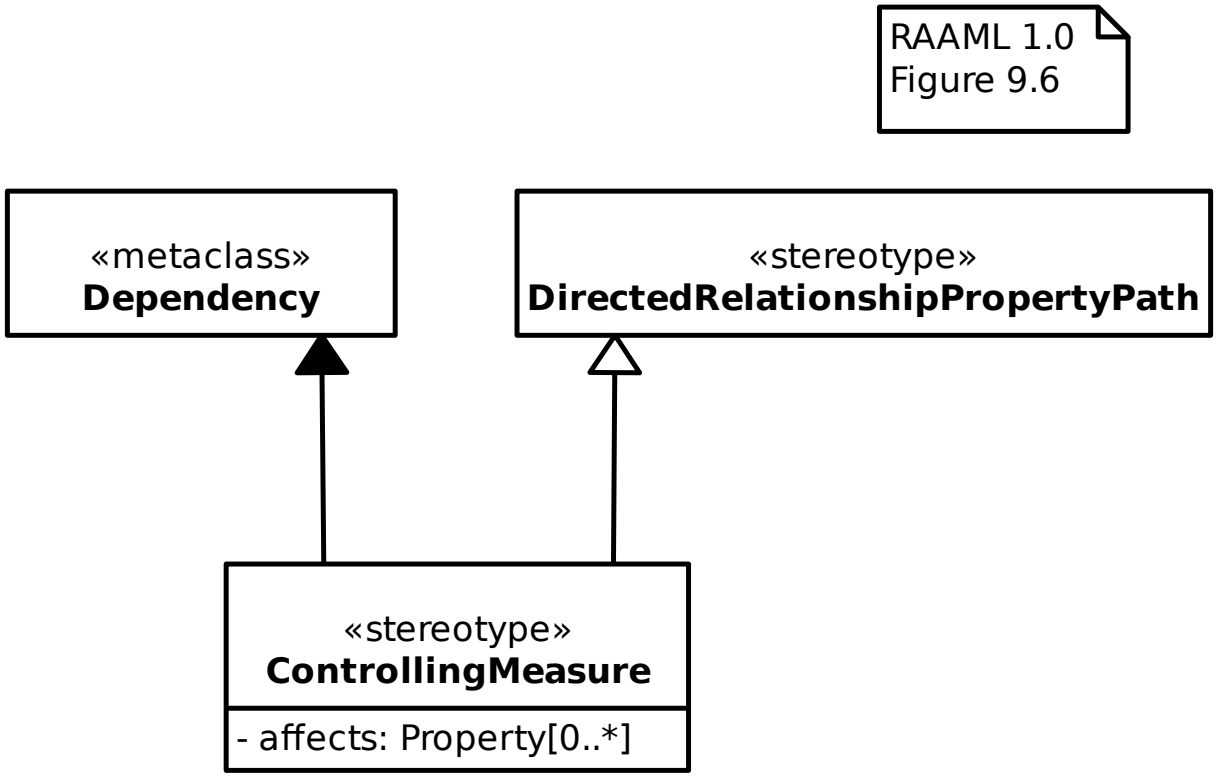


18.1 Core

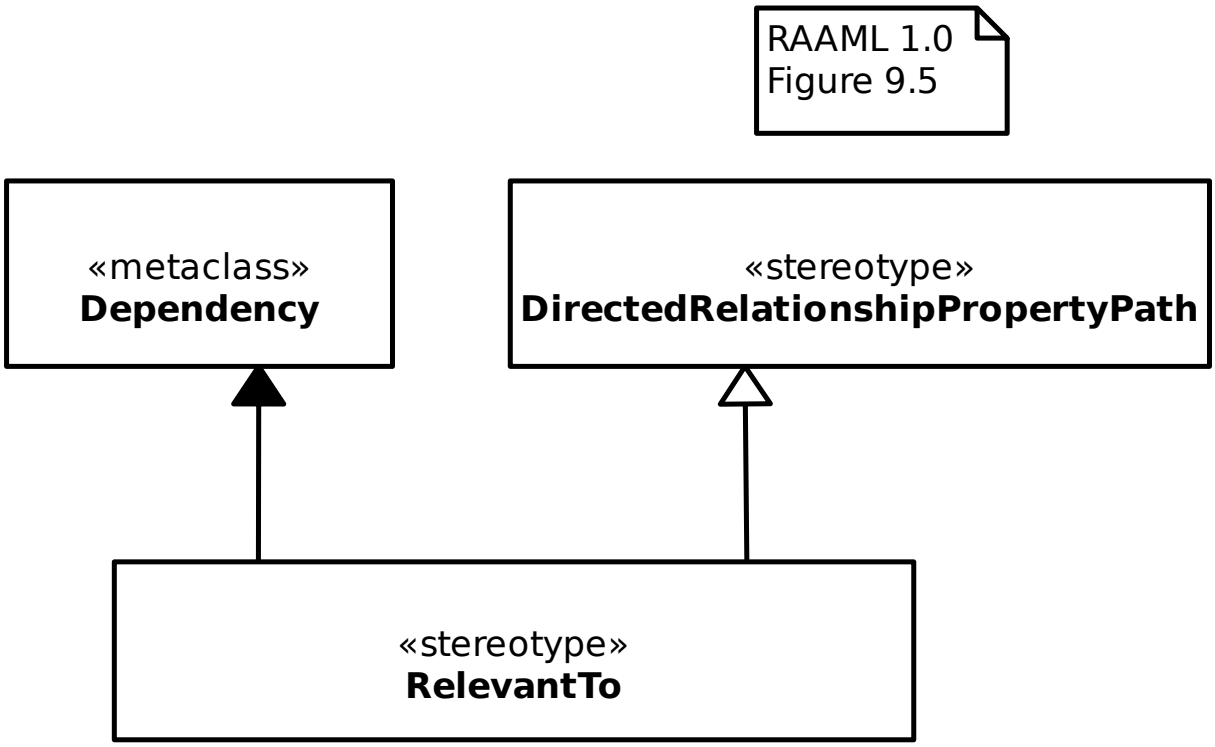
18.1.1 Core Library/Any Situation



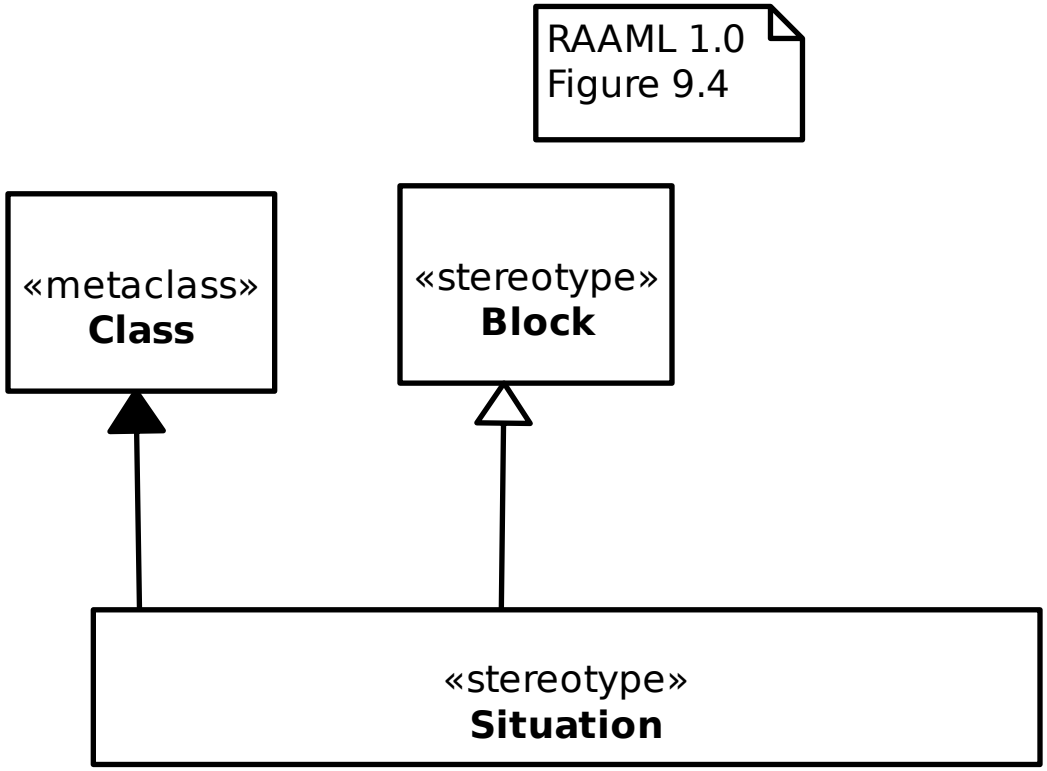
18.1.2 Core Profile/Controlling Measure



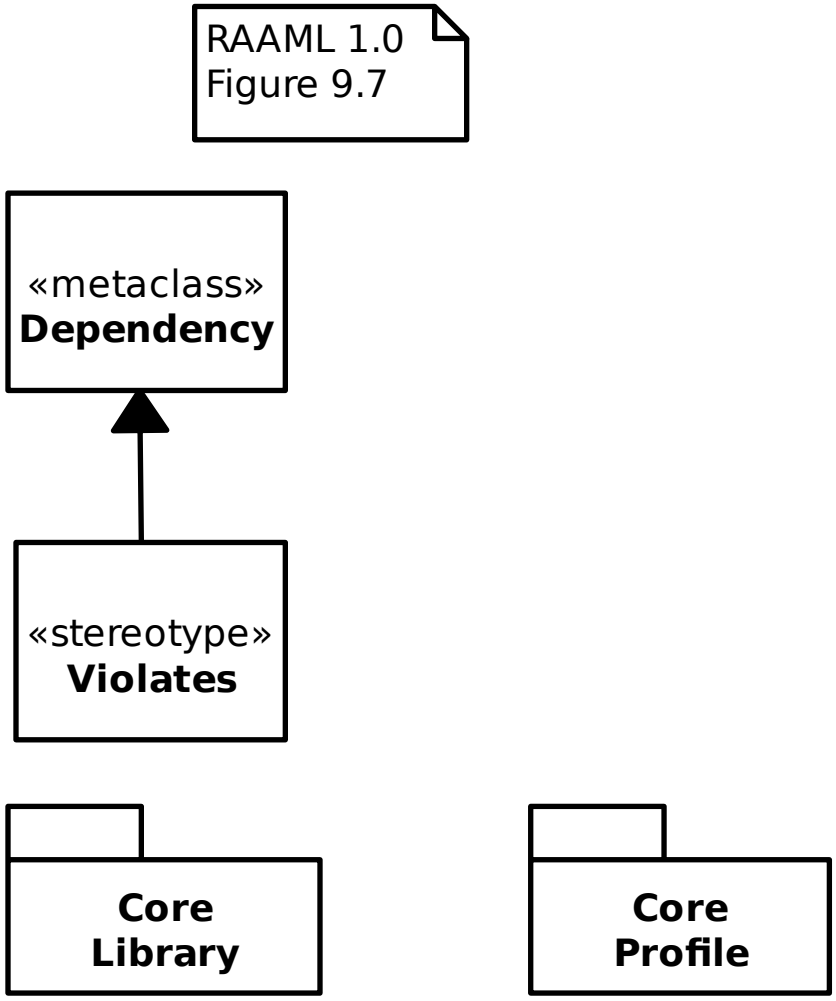
18.1.3 Core Profile/Relevant To



18.1.4 Core Profile/Situation

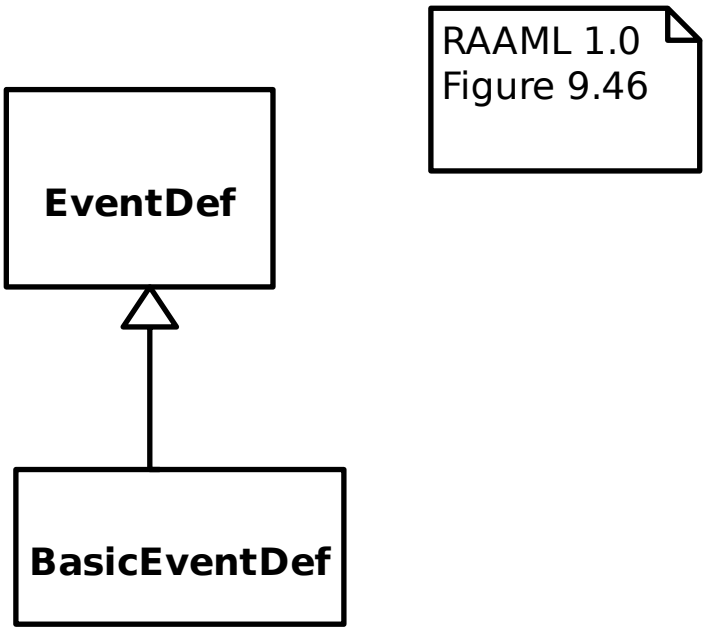


18.1.5 Core Profile/Violates

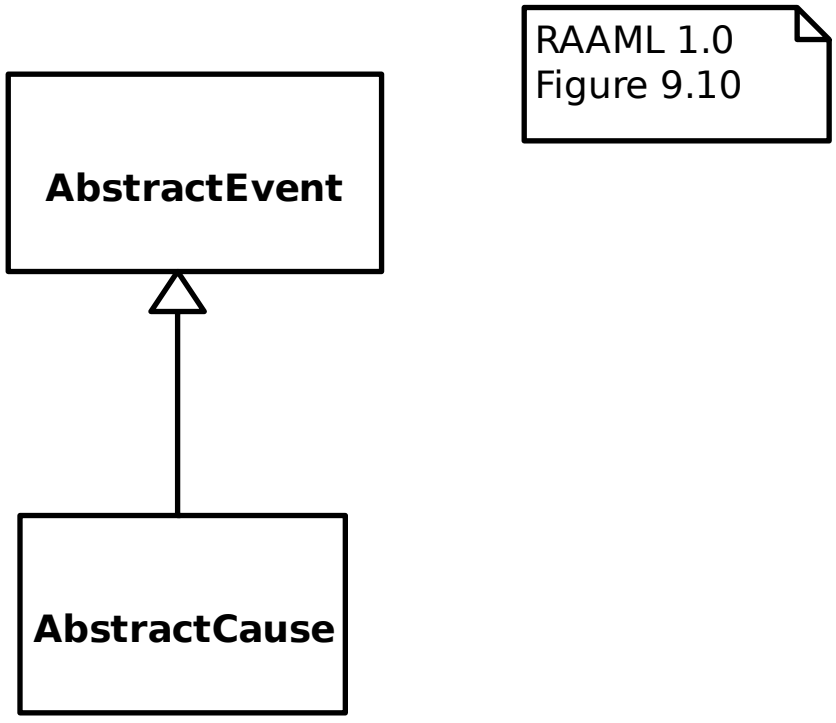


18.2 General

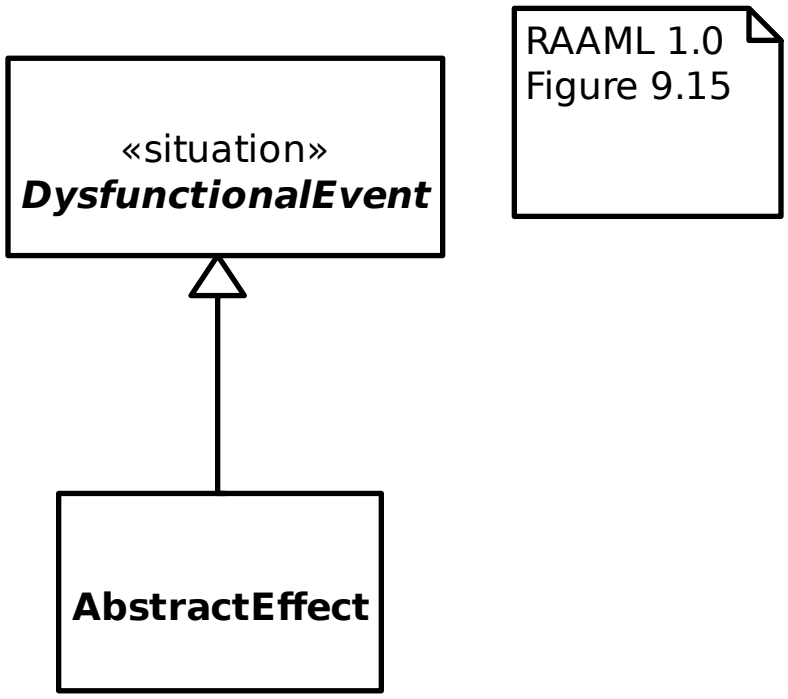
18.2.1 Basic Event



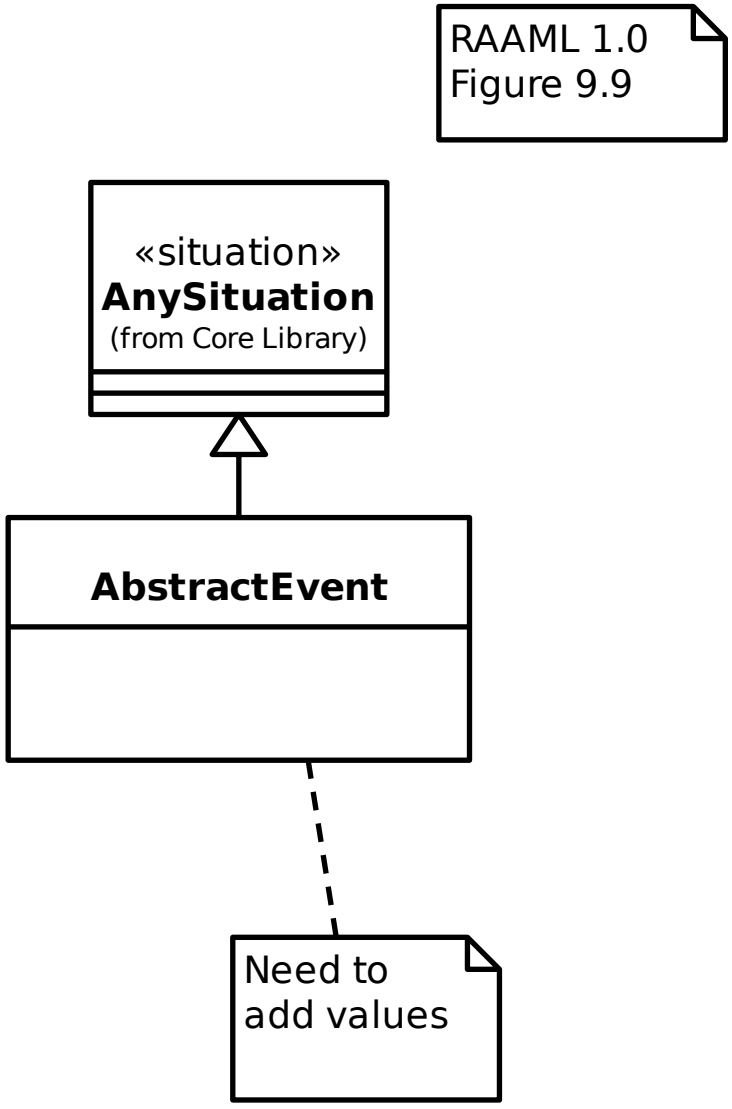
18.2.2 General Concepts Library/Abstract Cause



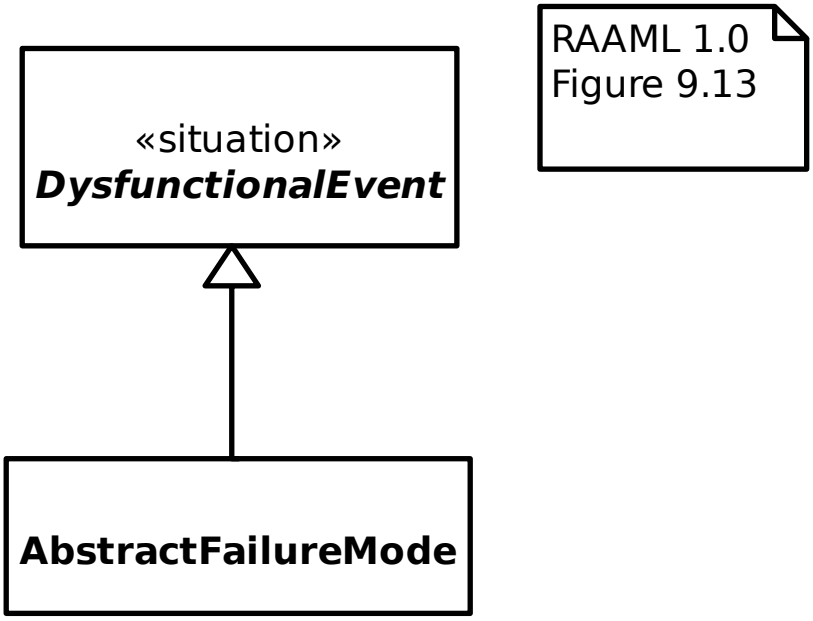
18.2.3 General Concepts Library/Abstract Effect



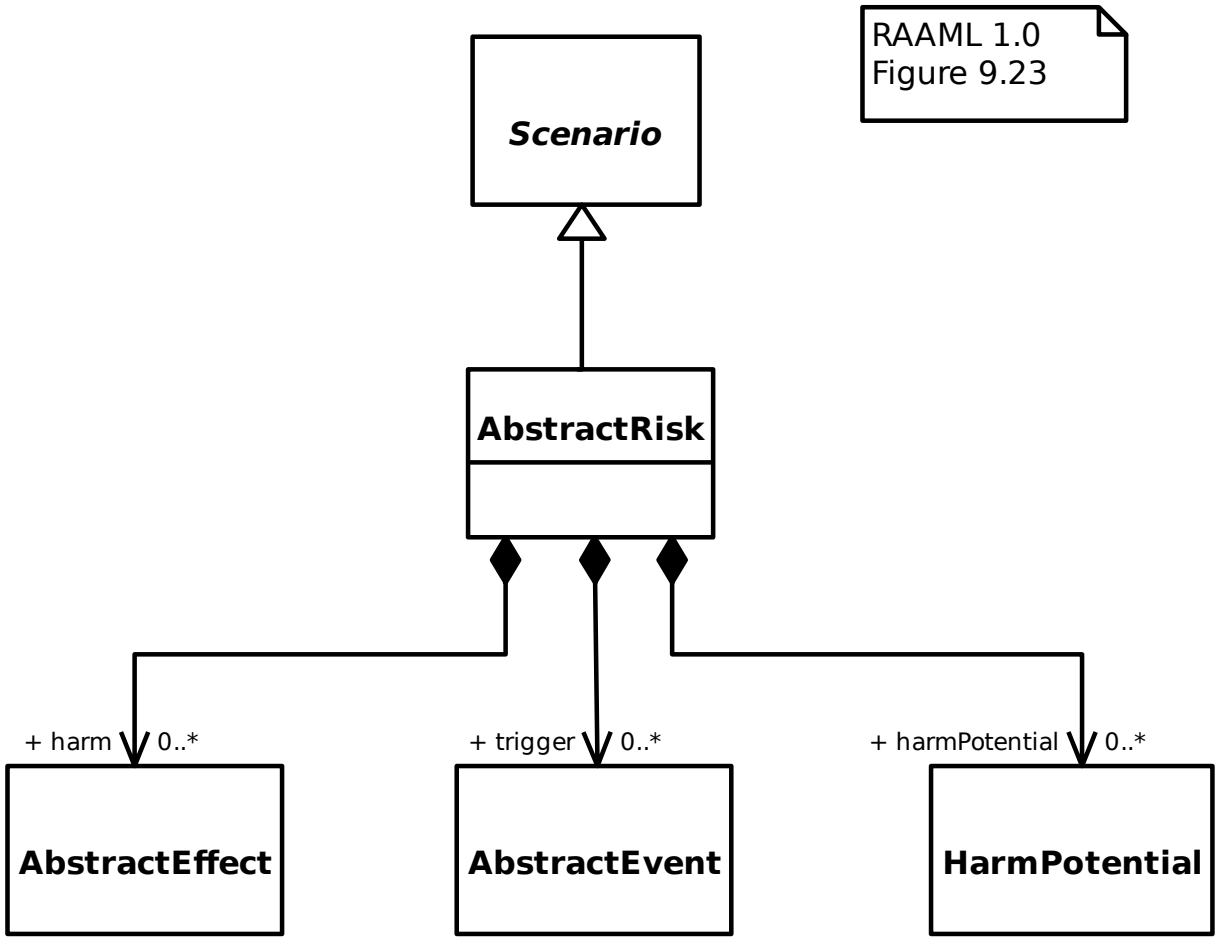
18.2.4 General Concepts Library/Abstract Event



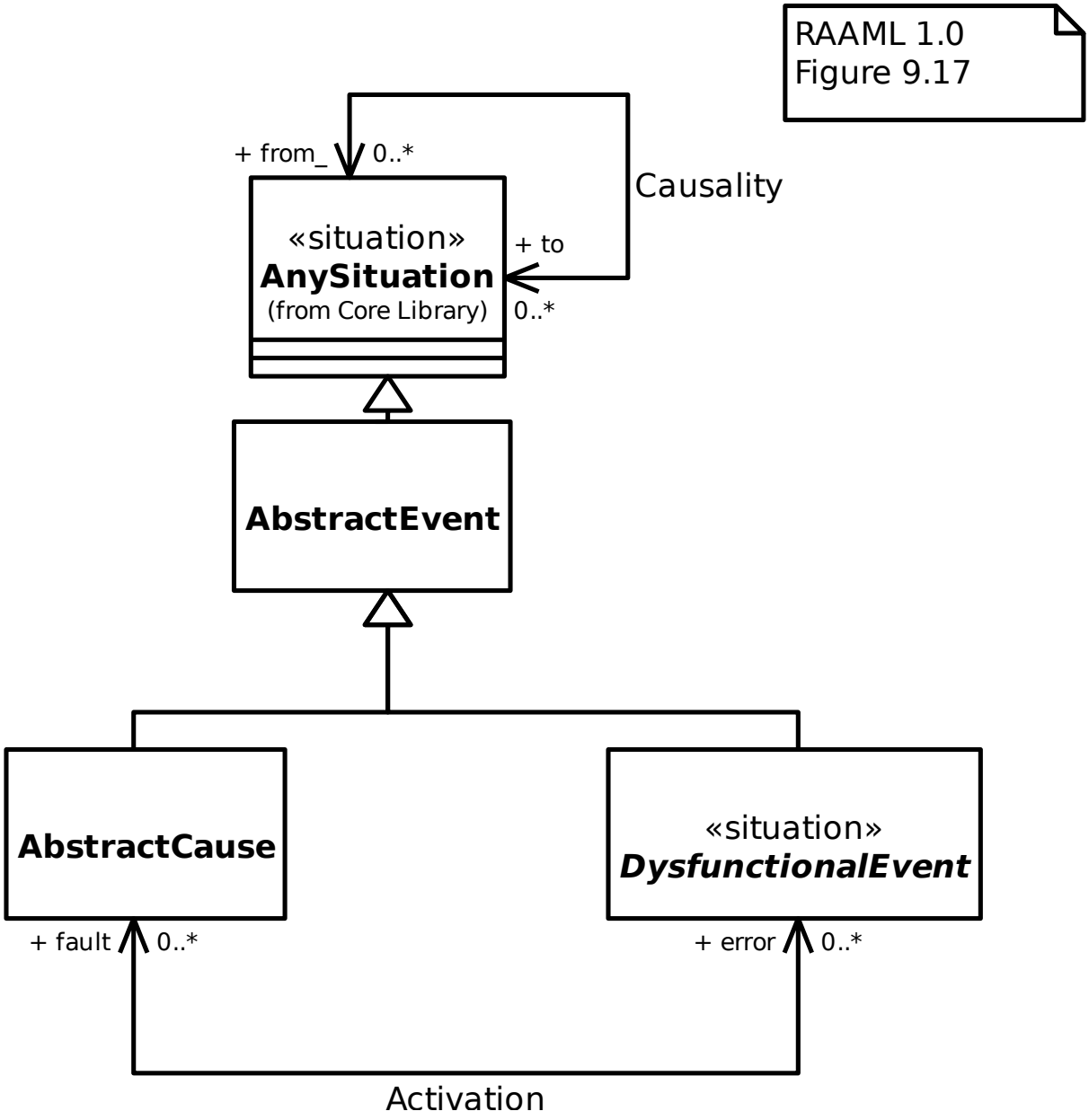
18.2.5 General Concepts Library/Abstract Failure Mode



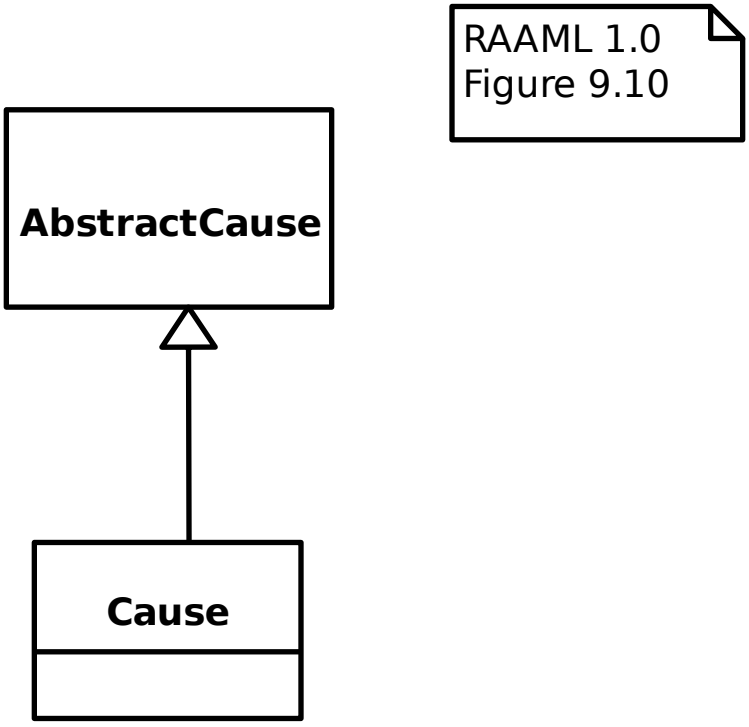
18.2.6 General Concepts Library/Abstract Risk



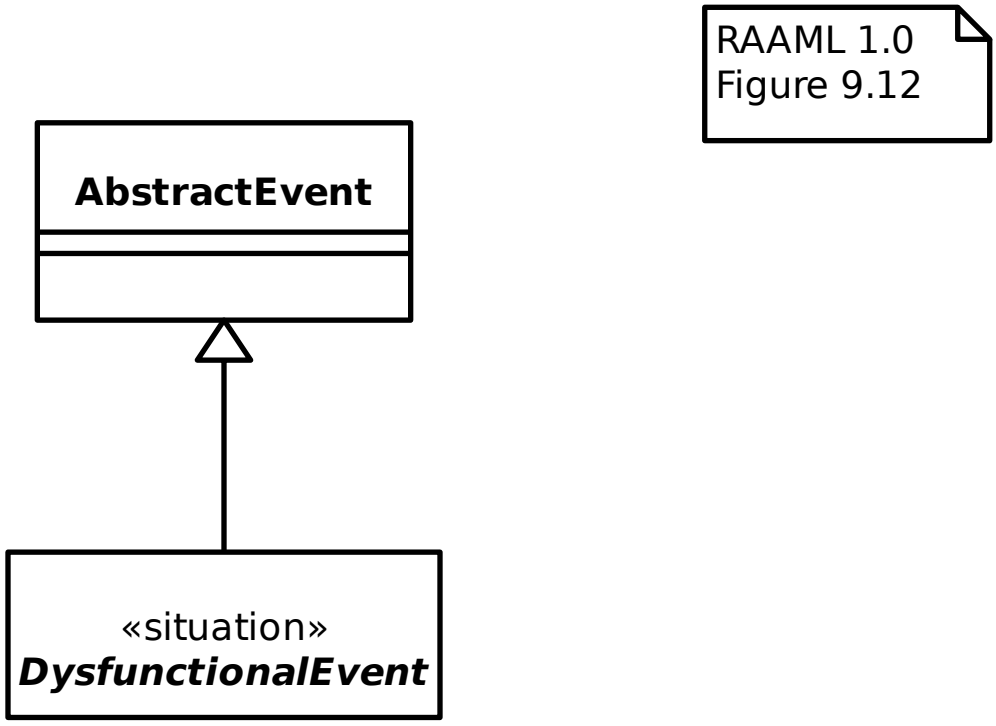
18.2.7 General Concepts Library/Activation



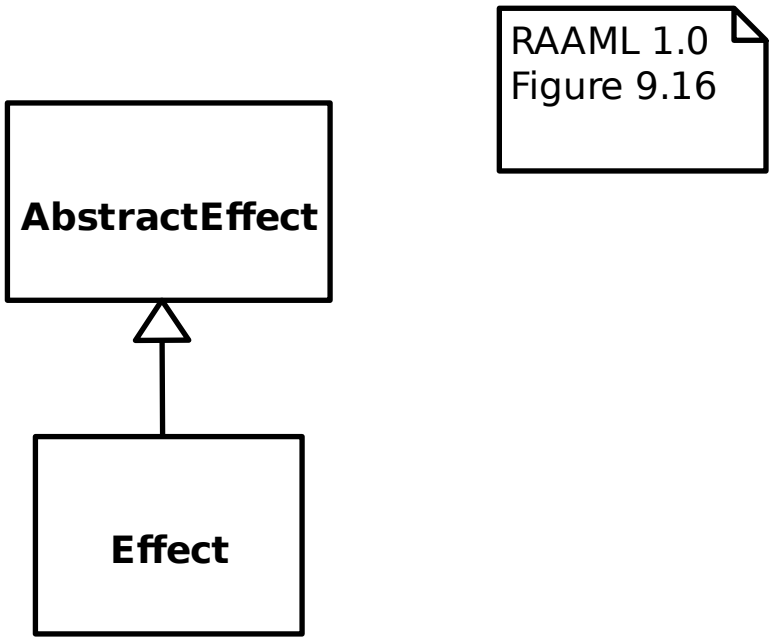
18.2.8 General Concepts Library/Cause



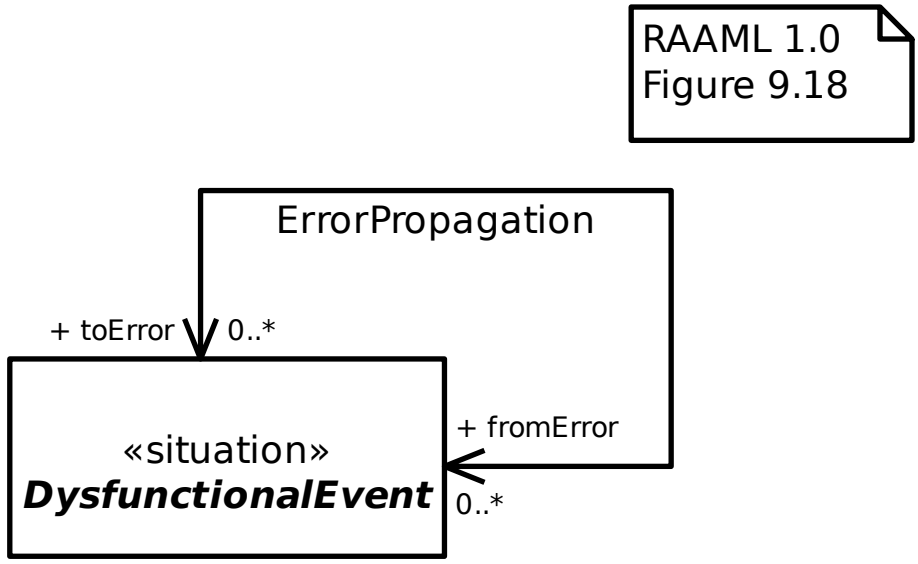
18.2.9 General Concepts Library/Dysfunctional Event



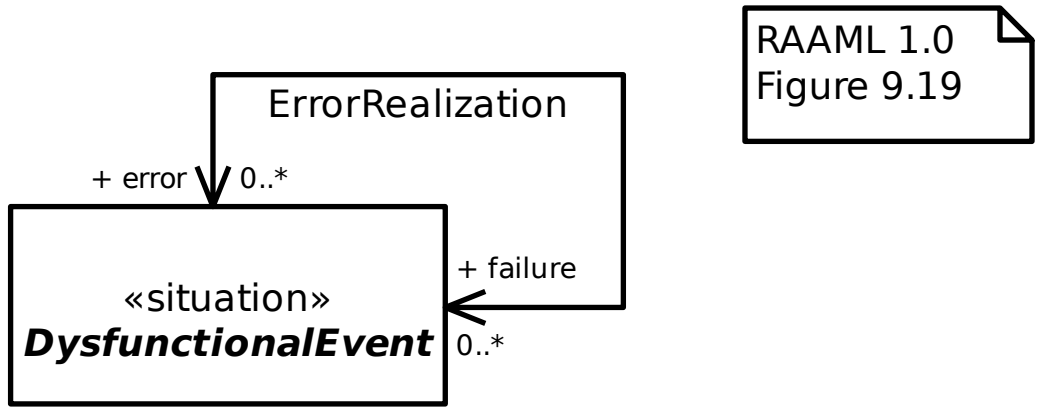
18.2.10 General Concepts Library/Effect



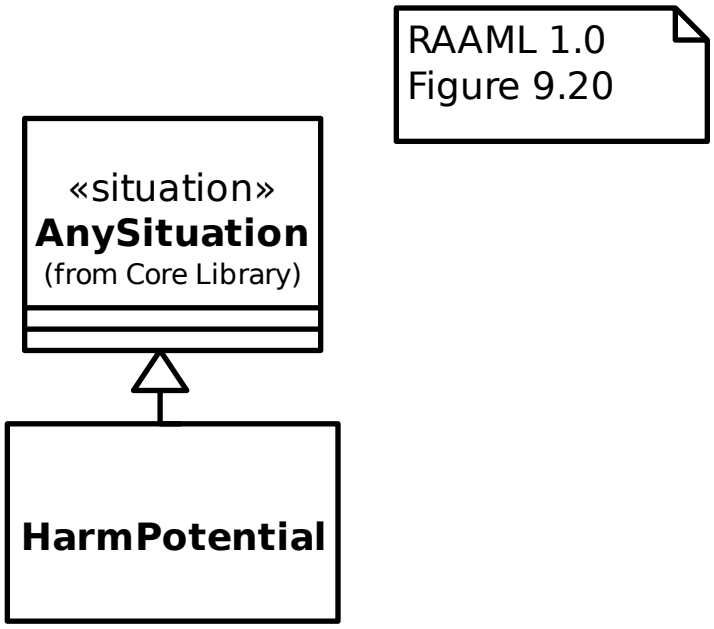
18.2.11 General Concepts Library/Error Propagation



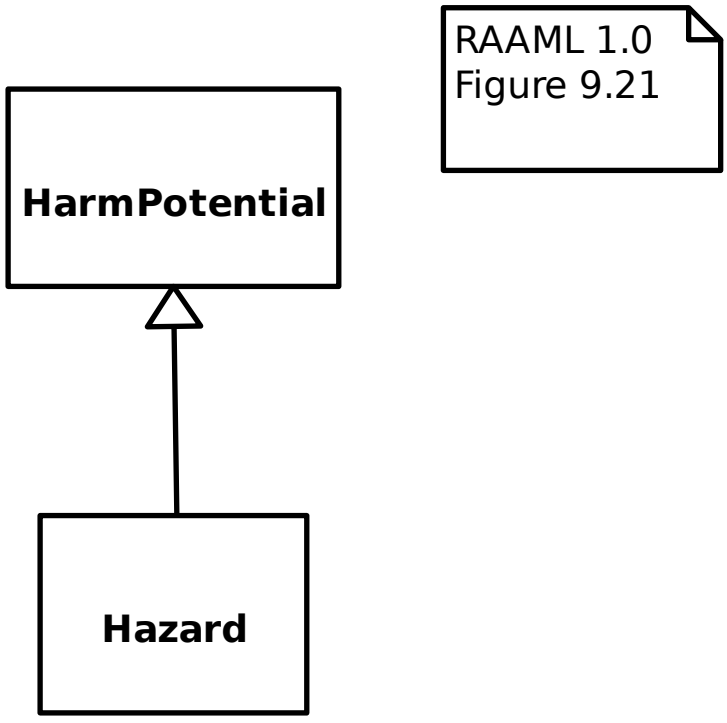
18.2.12 General Concepts Library/Error Realization



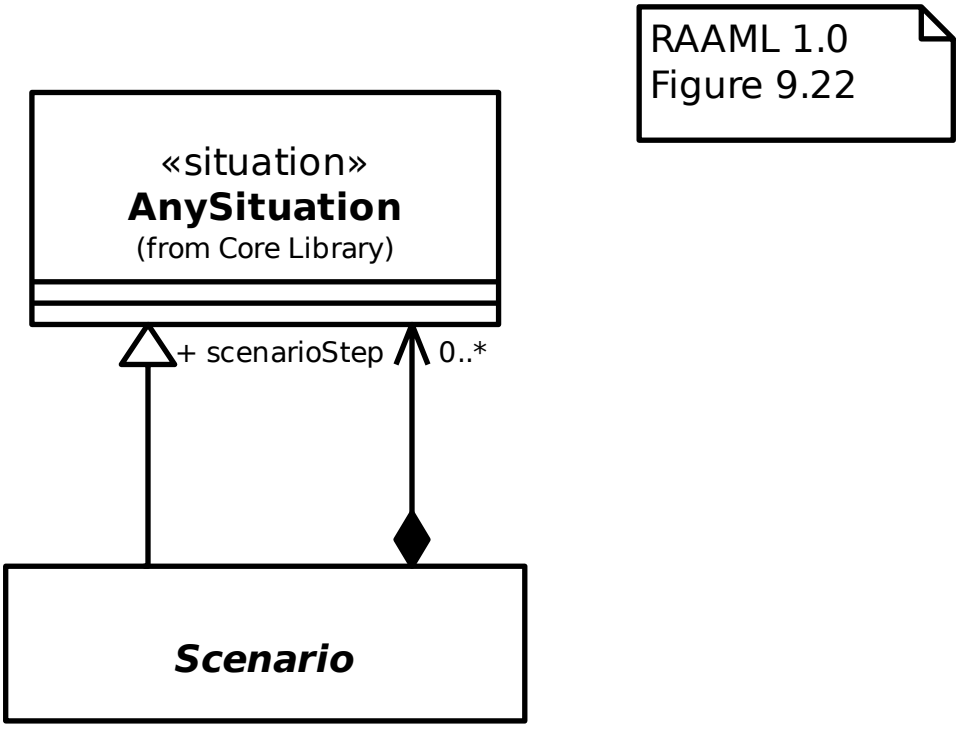
18.2.13 General Concepts Library/Harm Potential



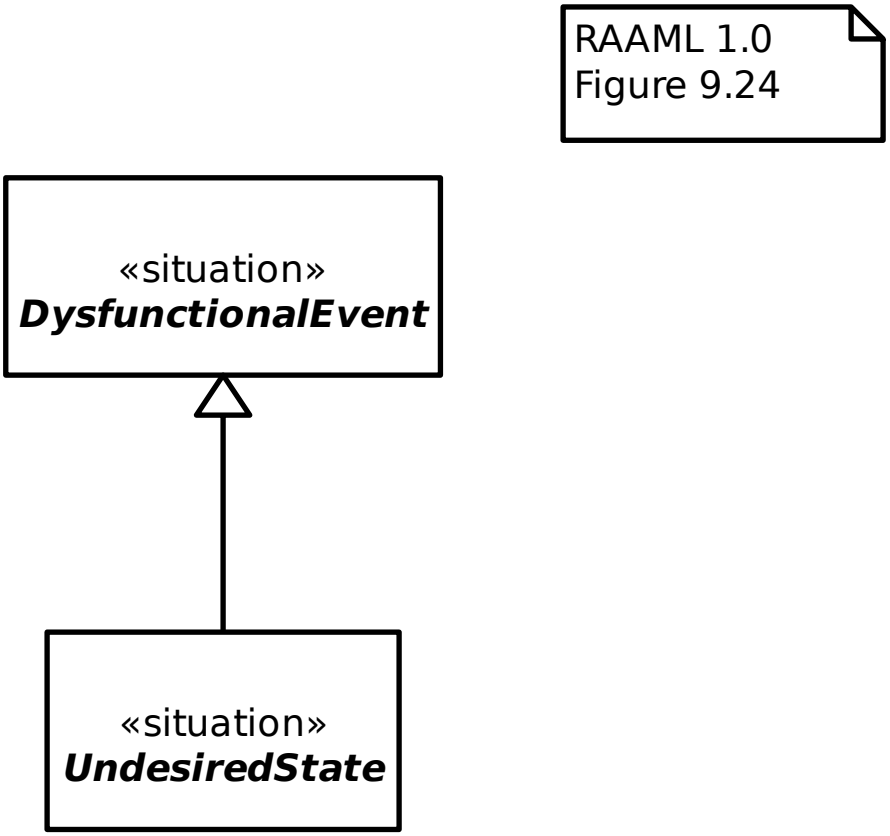
18.2.14 General Concepts Library/Hazard



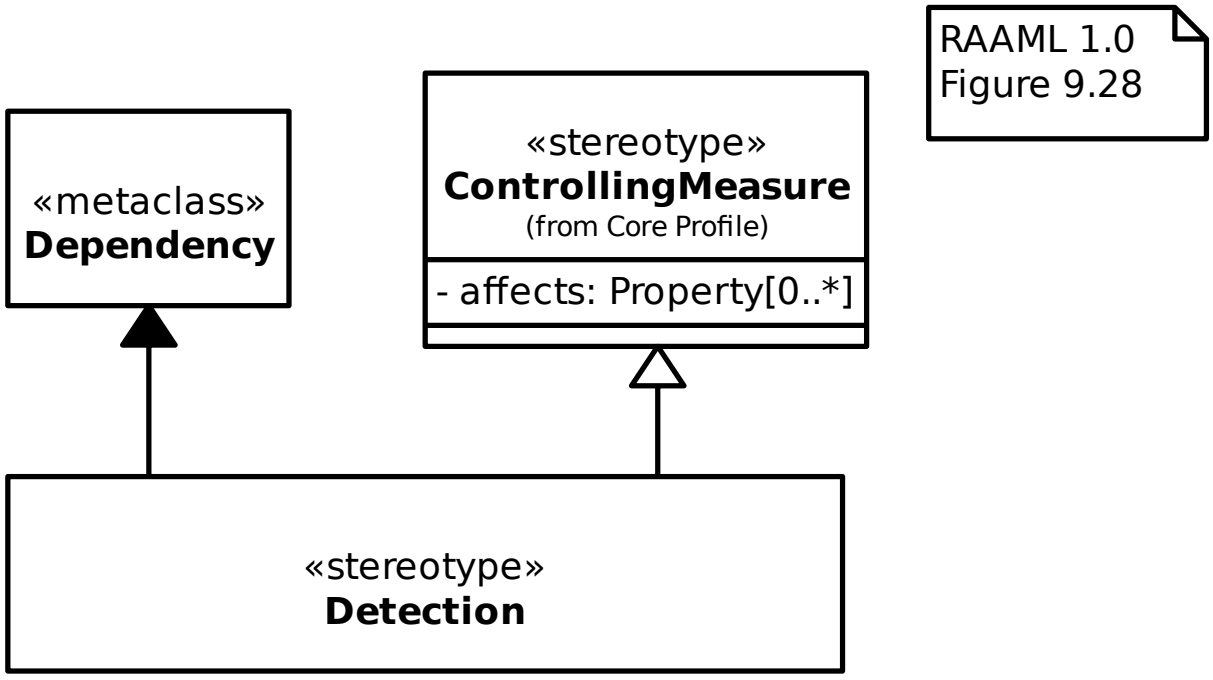
18.2.15 General Concepts Library/Scenario



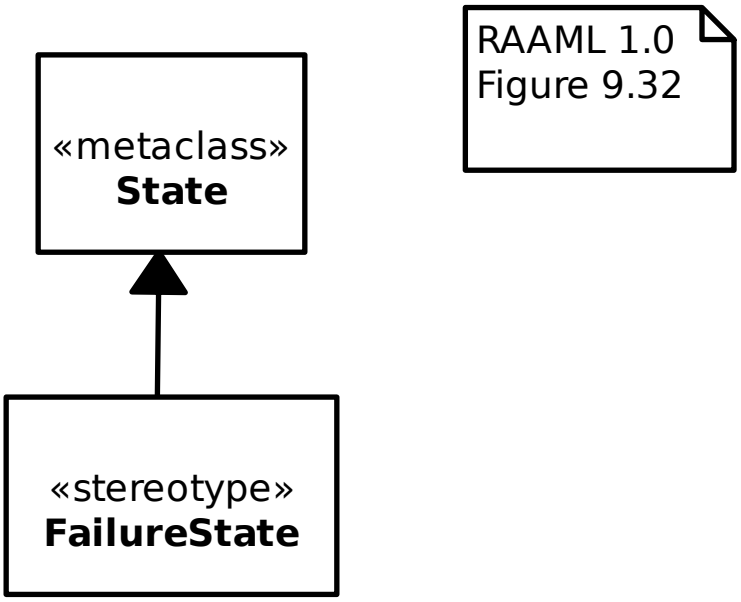
18.2.16 General Concepts Library/Undesired State



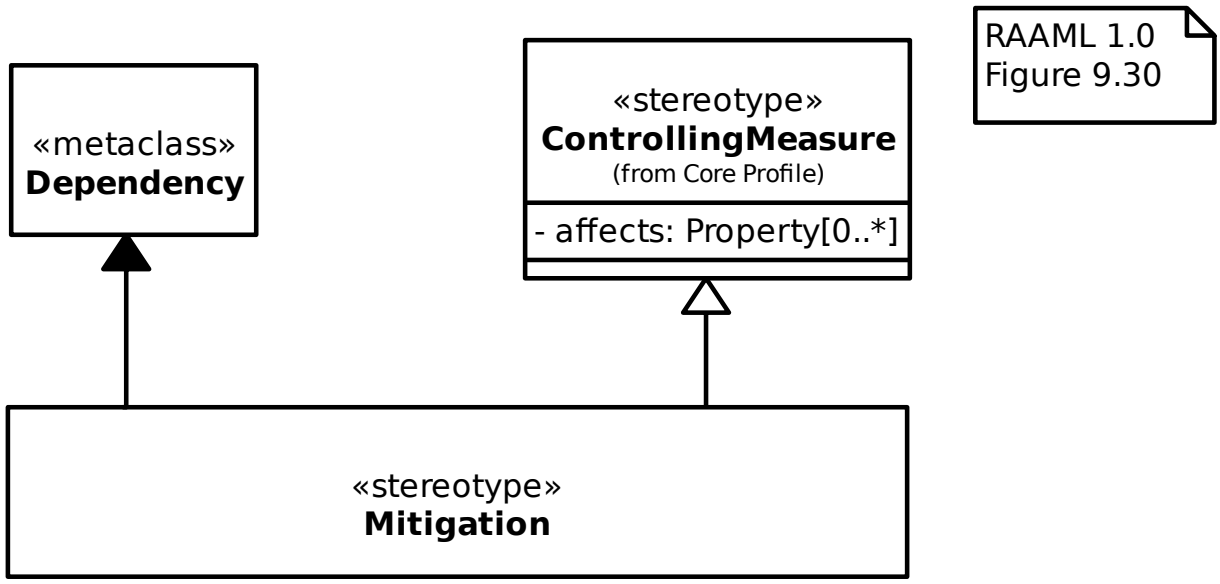
18.2.17 General Concepts Profile/Detection



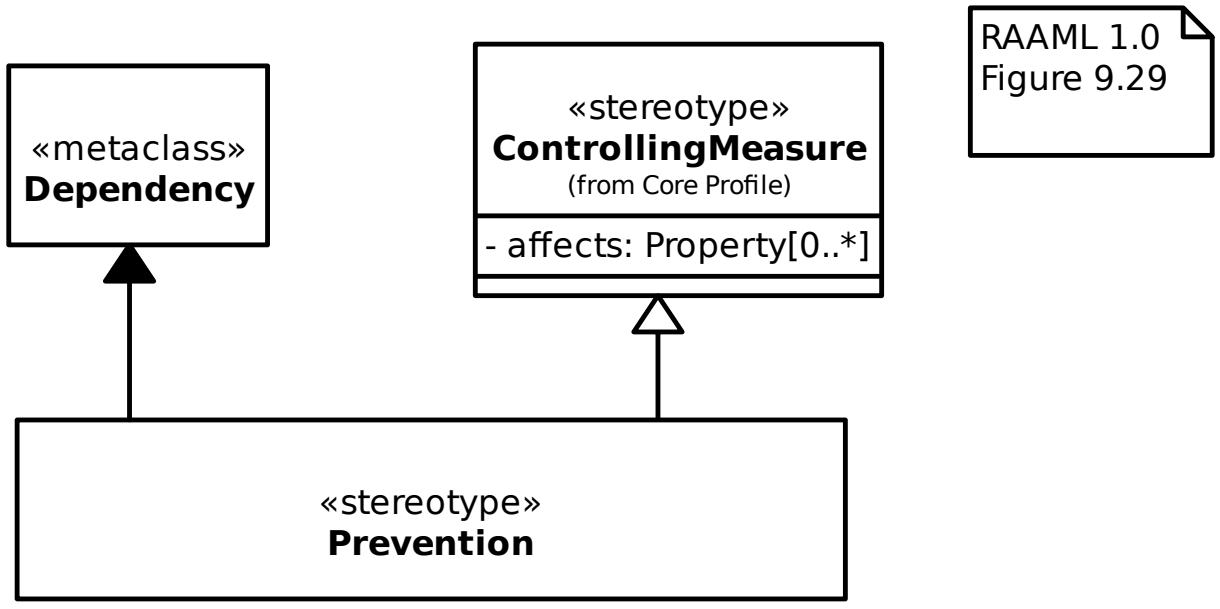
18.2.18 General Concepts Profile/Failure State



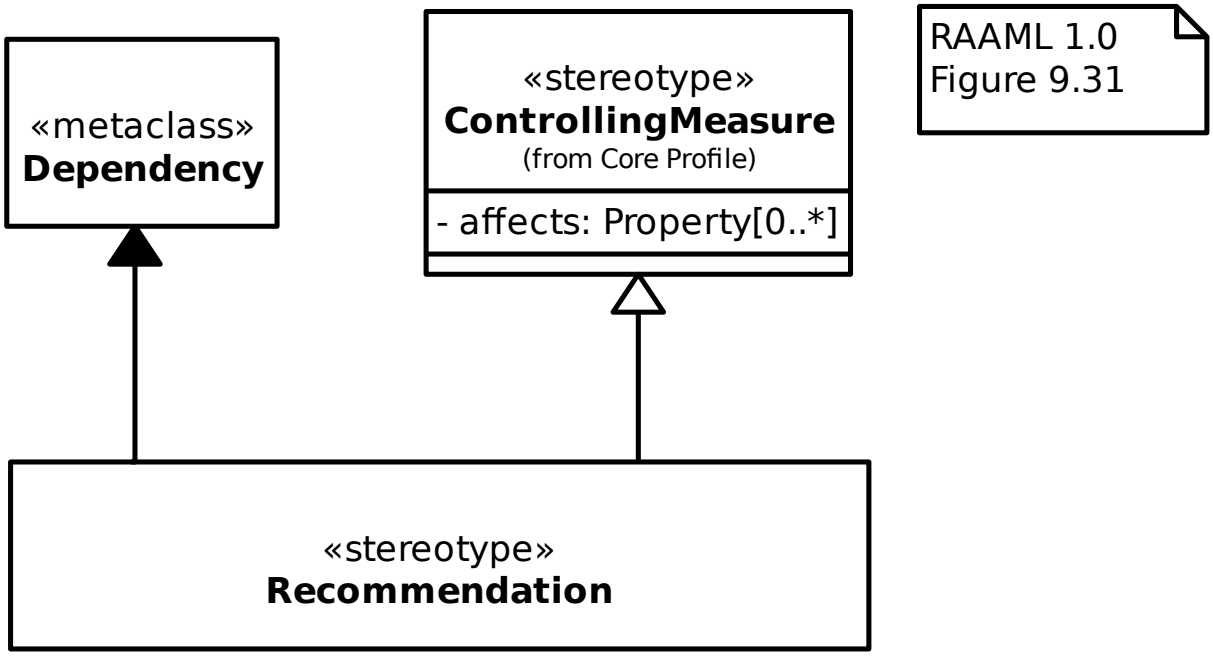
18.2.19 General Concepts Profile/Mitigation



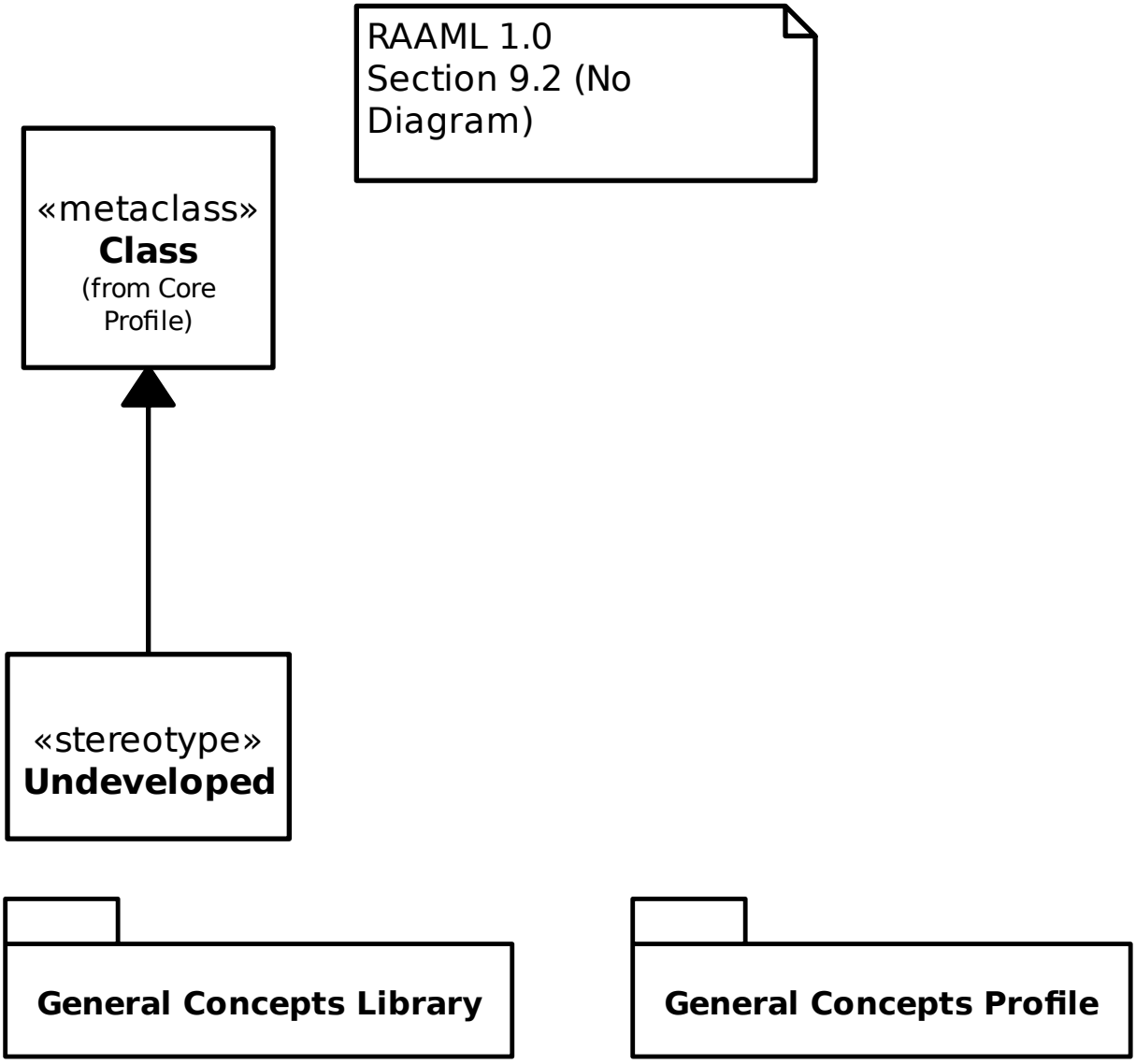
18.2.20 General Concepts Profile/Prevention



18.2.21 General Concepts Profile/Recommendation

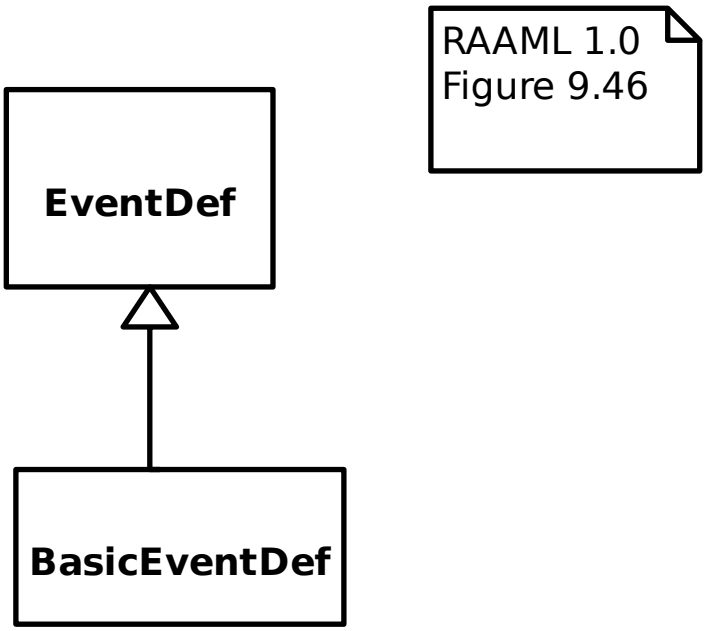


18.2.22 General Concepts Profile/Undeveloped

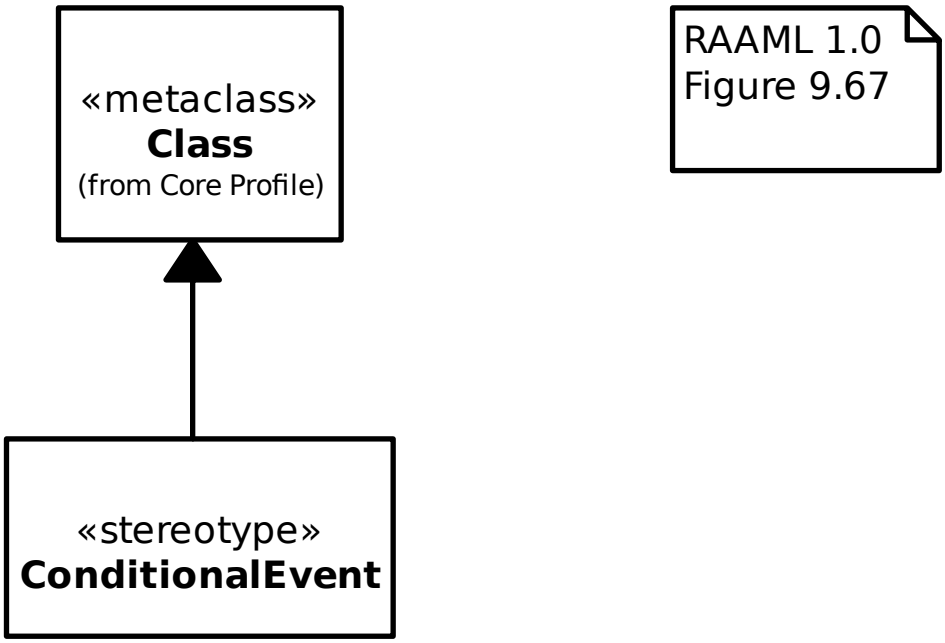


18.3 Methods

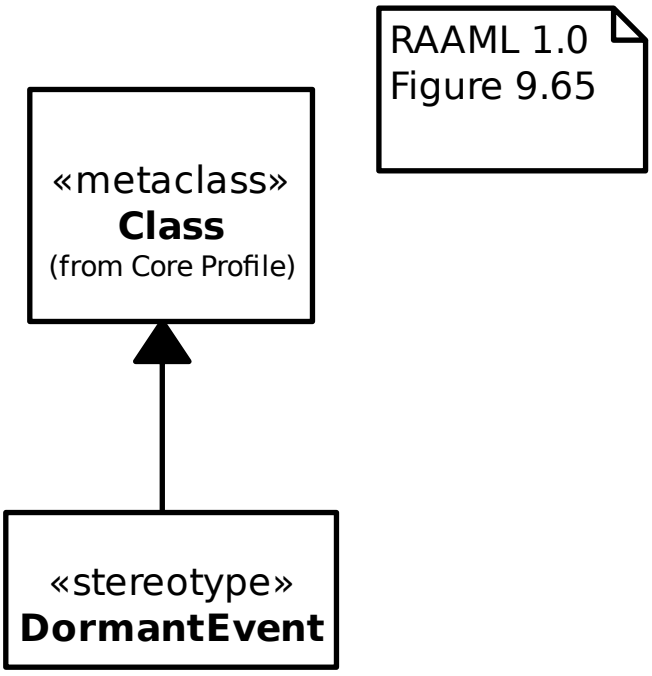
18.3.1 FTA/FTA Library/Events/Basic Event



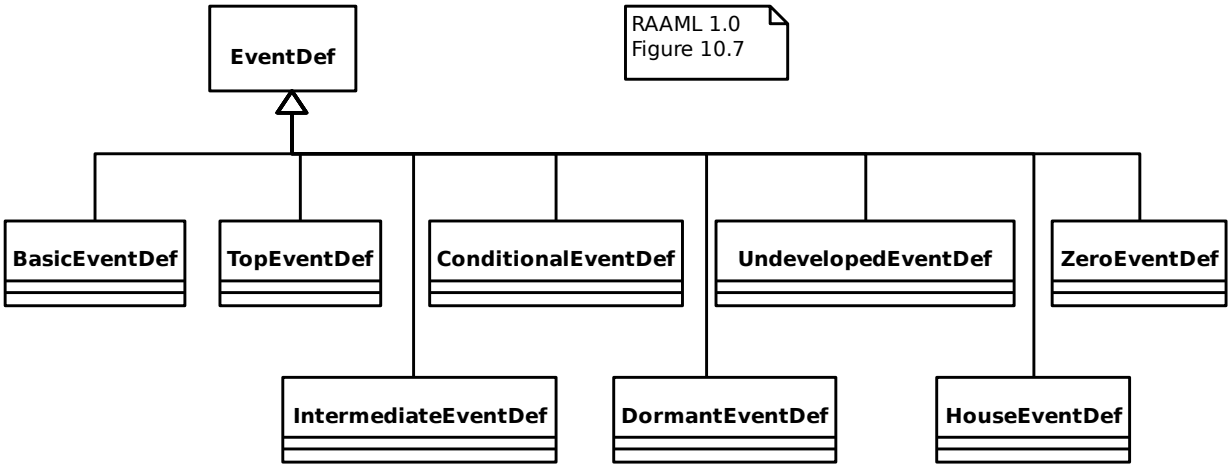
18.3.2 FTA/FTA Library/Events/Conditional Event



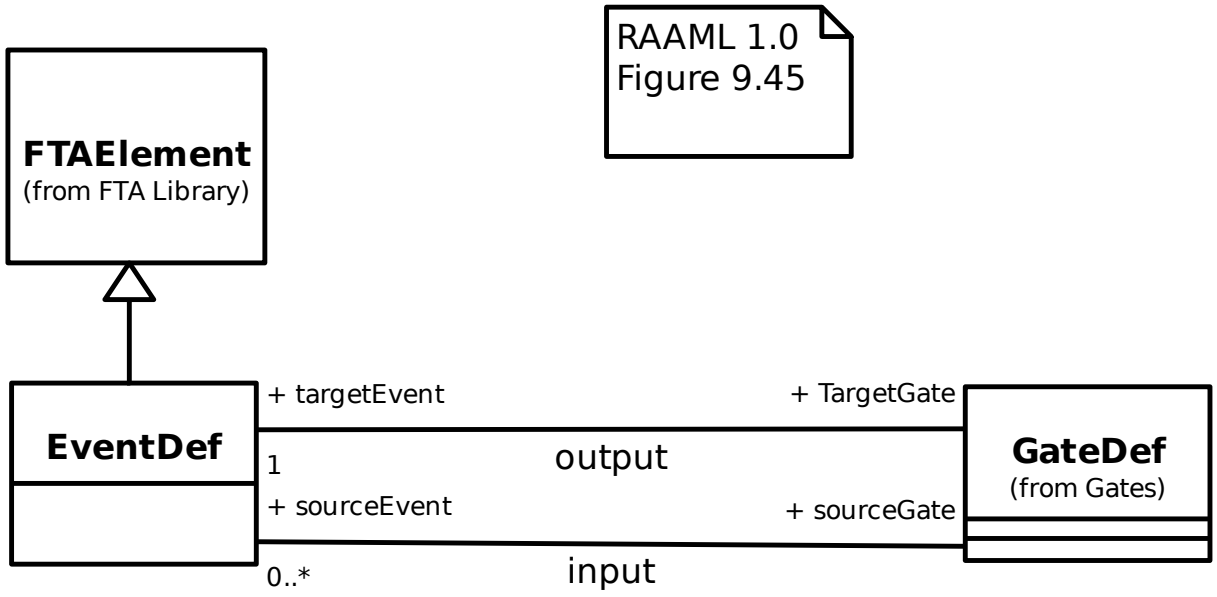
18.3.3 FTA/FTA Library/Events/Dormant Event



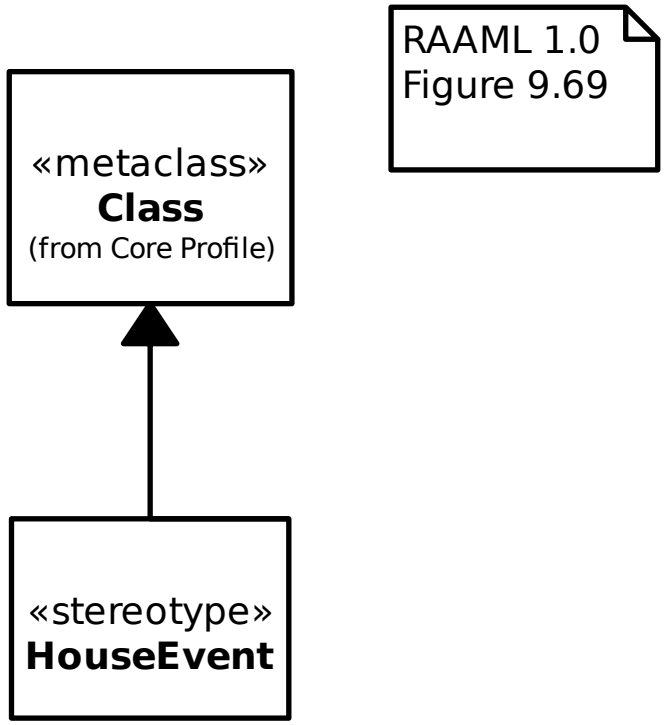
18.3.4 FTA/FTA Library/Events/Events



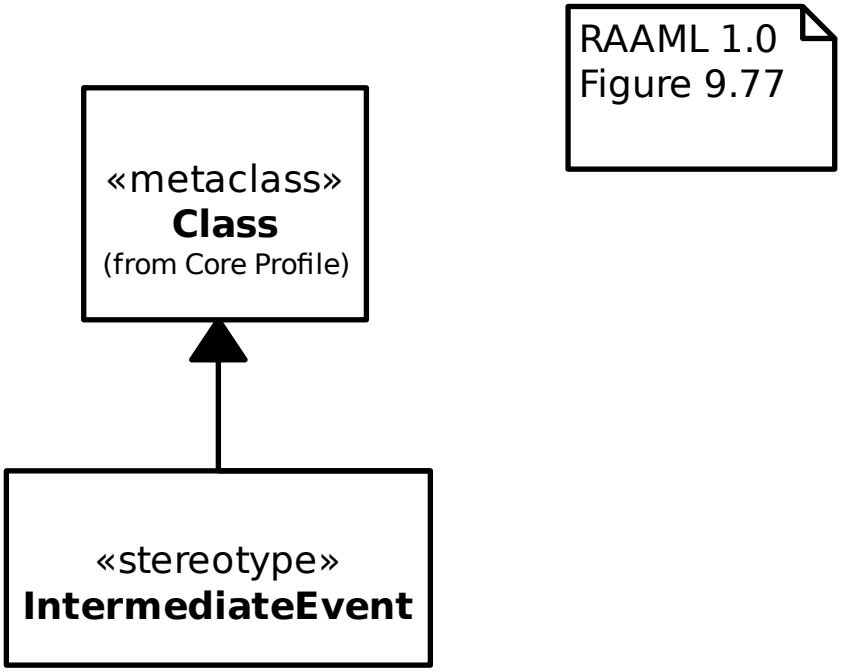
18.3.5 FTA/FTA Library/Events/Event



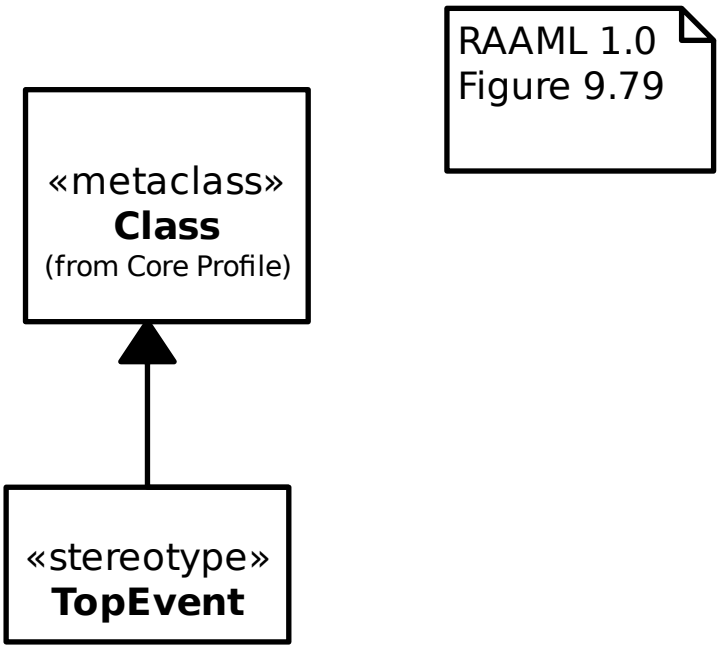
18.3.6 FTA/FTA Library/Events/House Event



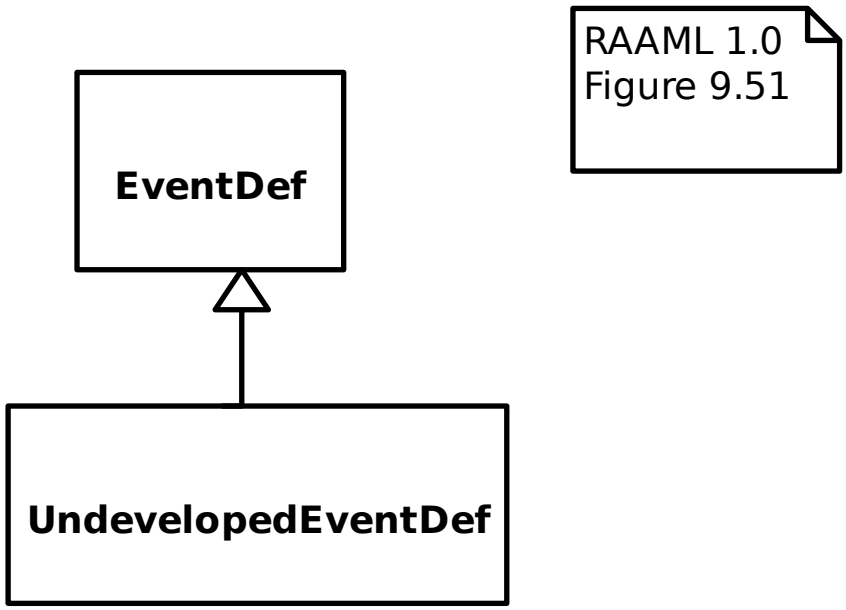
18.3.7 FTA/FTA Library/Events/Intermediate Event



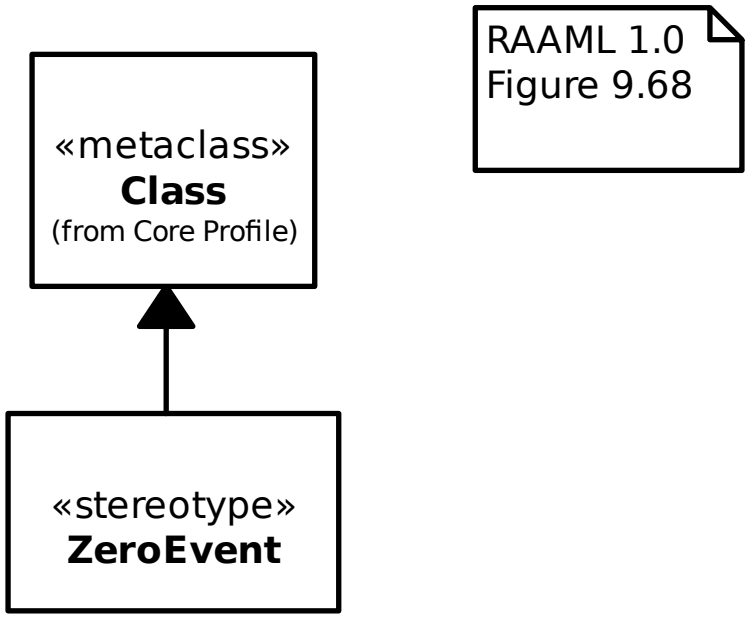
18.3.8 FTA/FTA Library/Events/Top Event



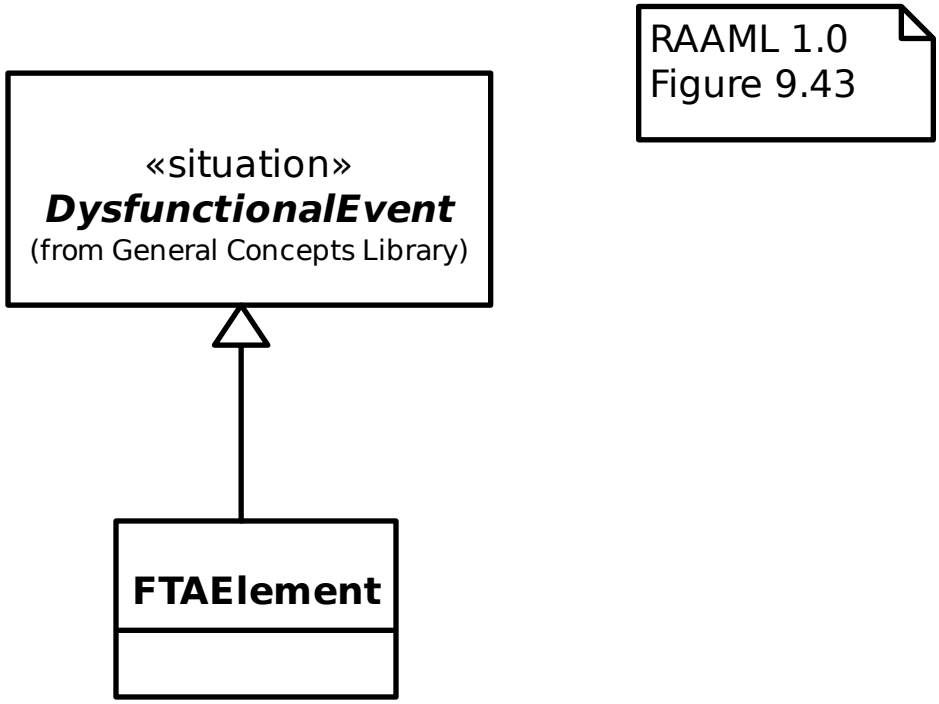
18.3.9 FTA/FTA Library/Events/Undeveloped Event



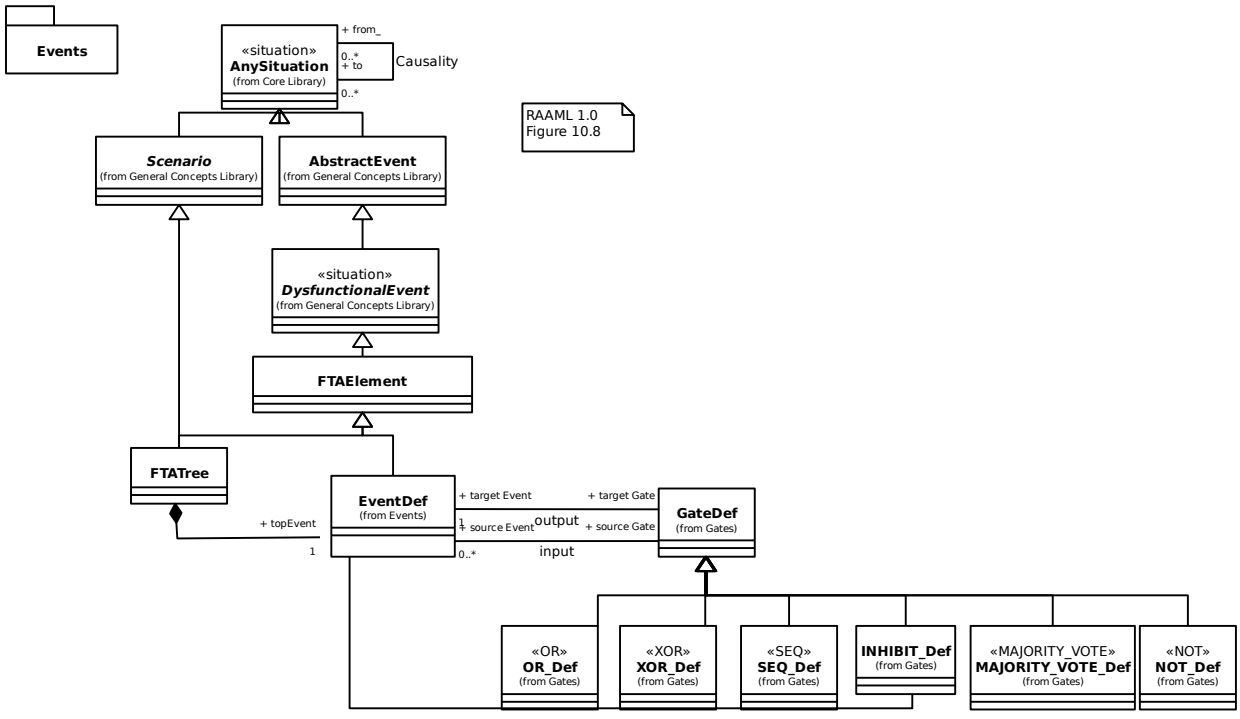
18.3.10 FTA/FTA Library/Events/Zero Event



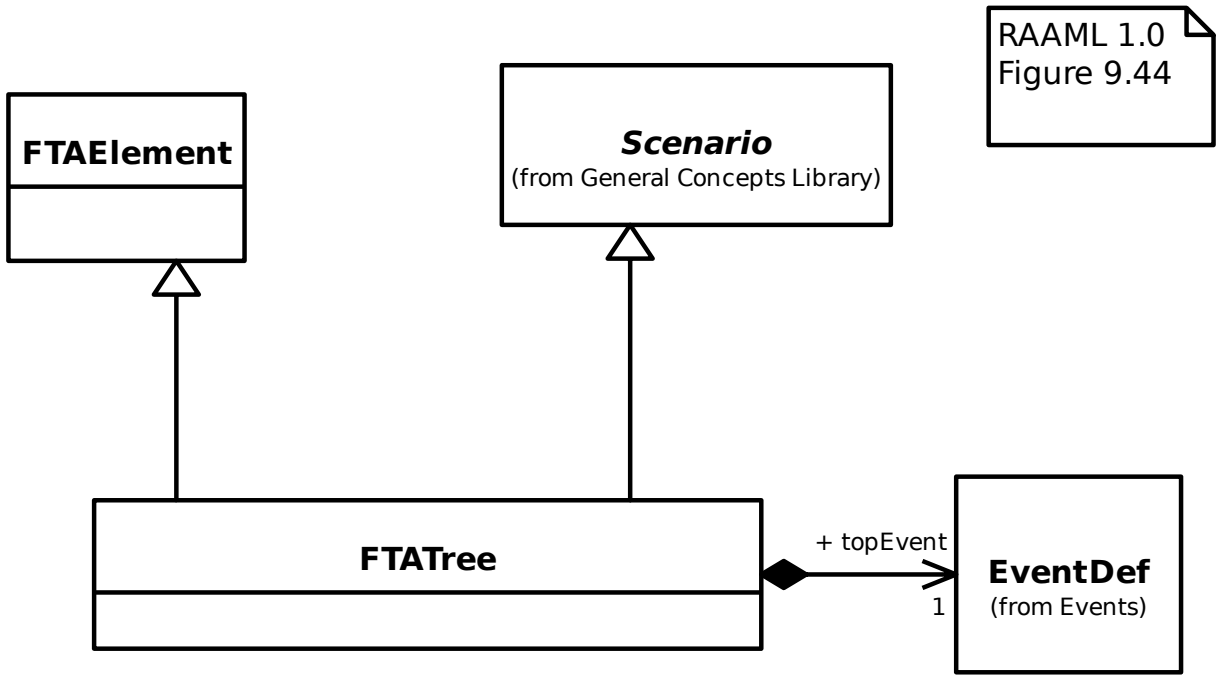
18.3.11 FTA/FTA Library/FTA Element



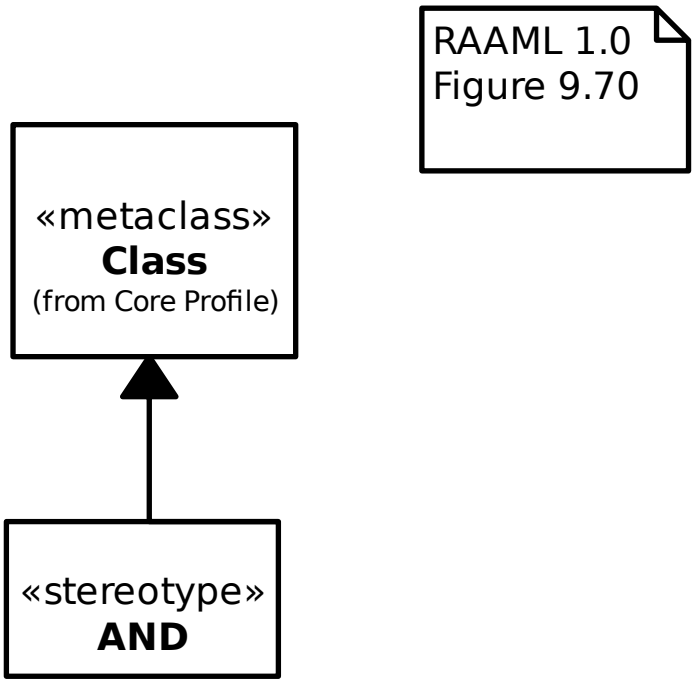
18.3.12 FTA/FTA Library/FTA Library



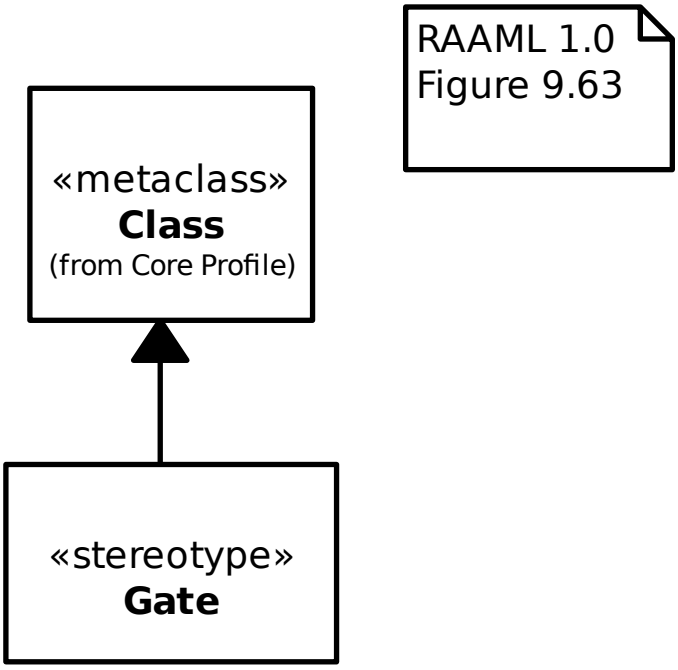
18.3.13 FTA/FTA Library/FTA Tree



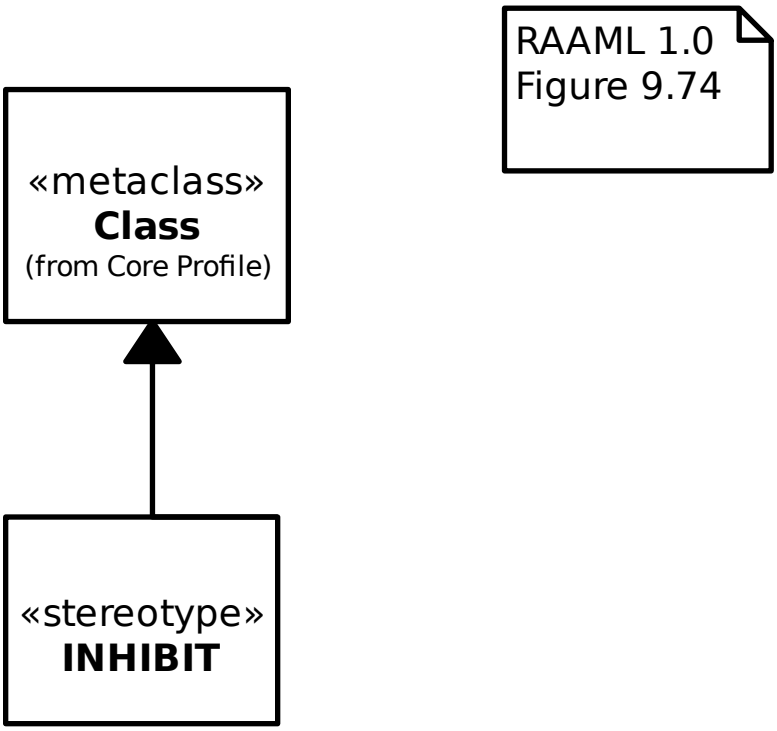
18.3.14 FTA/FTA Library/Gates/AND



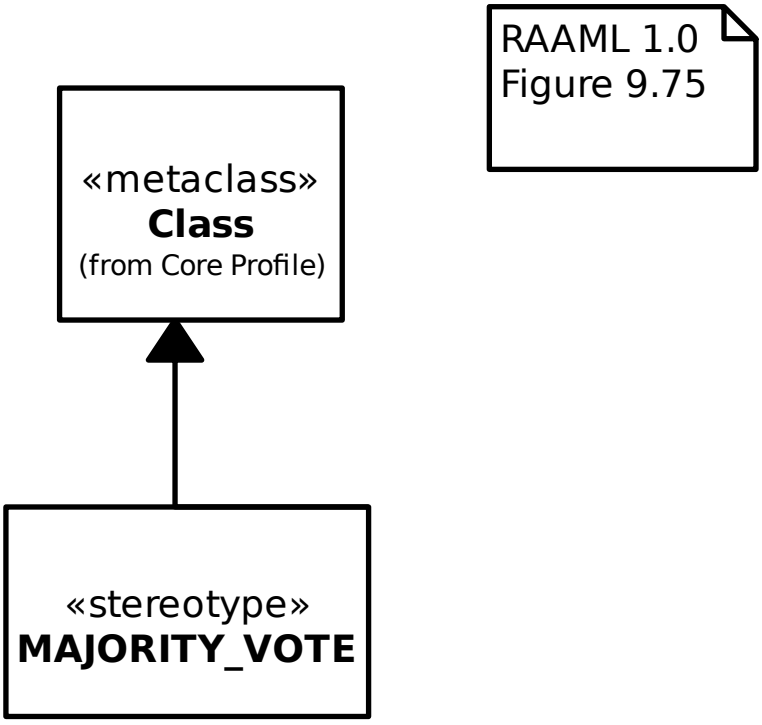
18.3.15 FTA/FTA Library/Gates/Gate



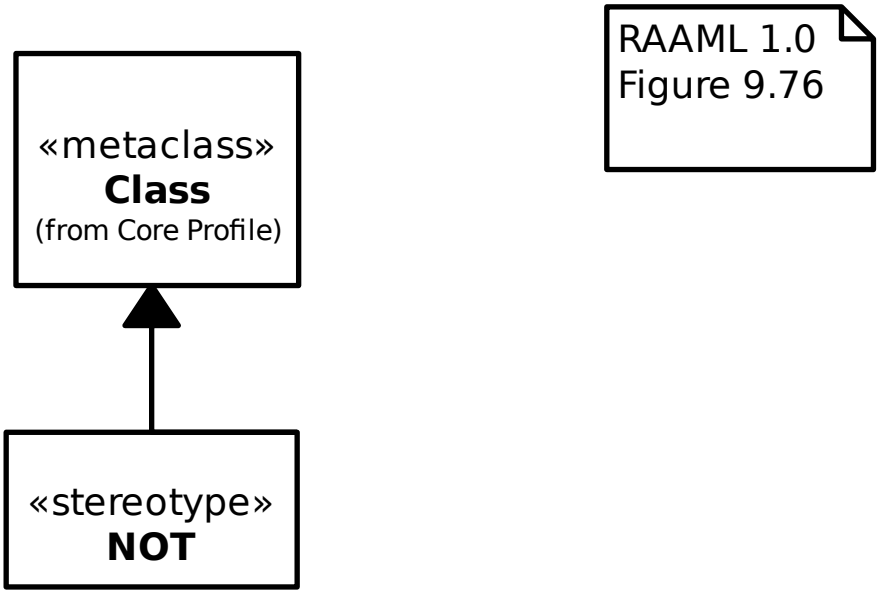
18.3.16 FTA/FTA Library/Gates/INHIBIT



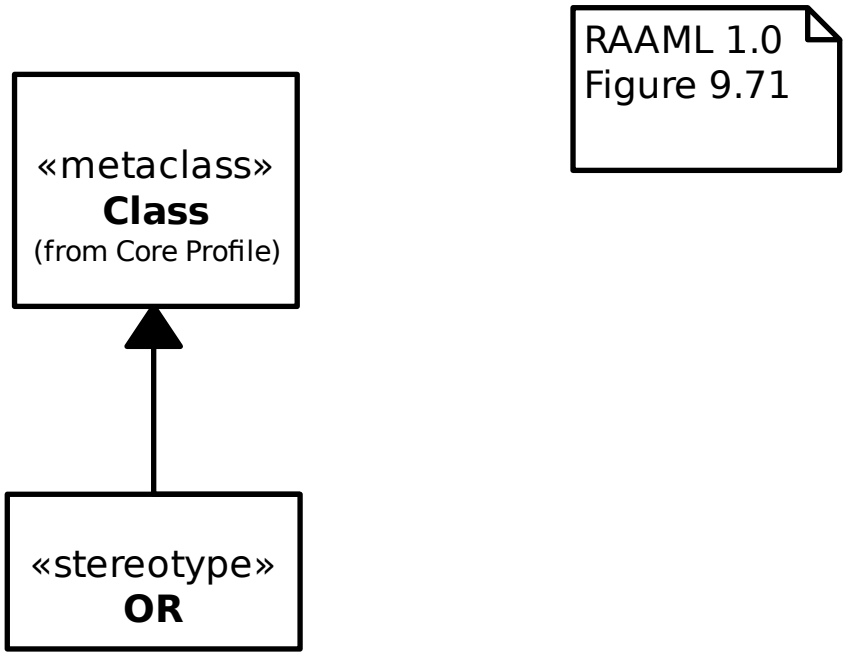
18.3.17 FTA/FTA Library/Gates/MAJORITY_VOTE



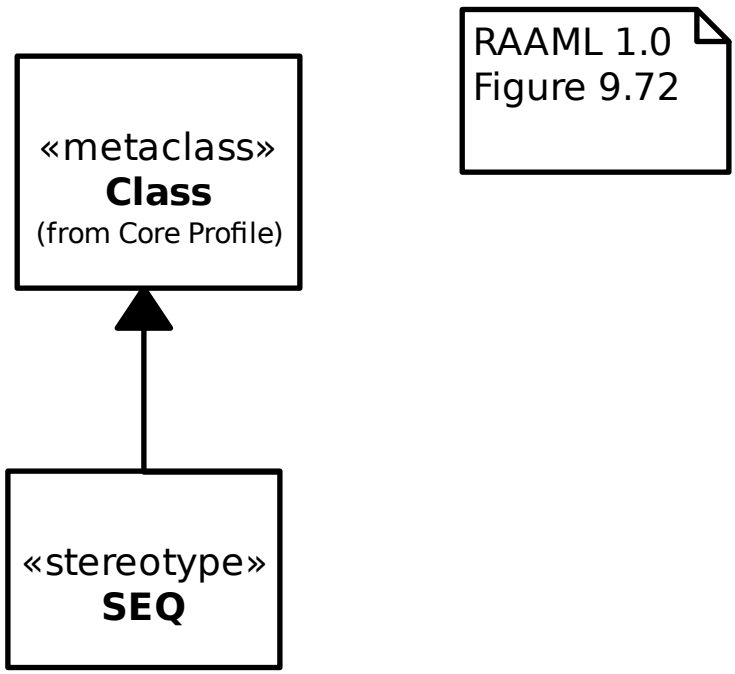
18.3.18 FTA/FTA Library/Gates/NOT



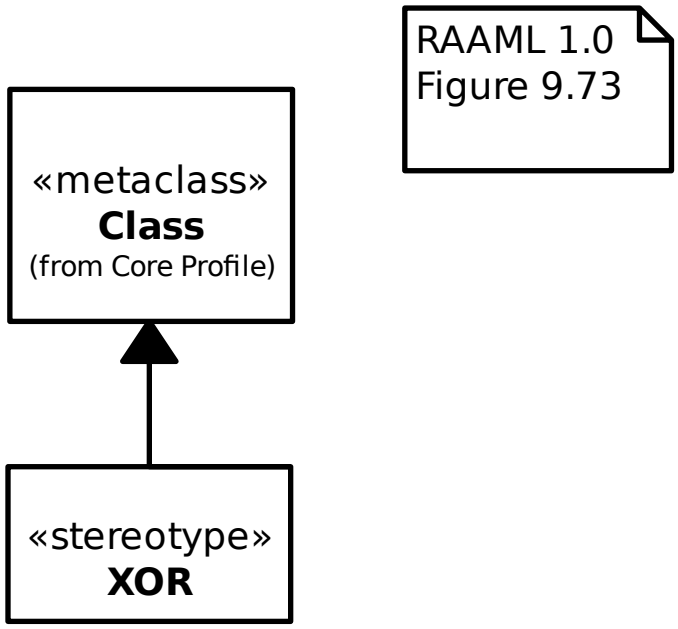
18.3.19 FTA/FTA Library/Gates/OR



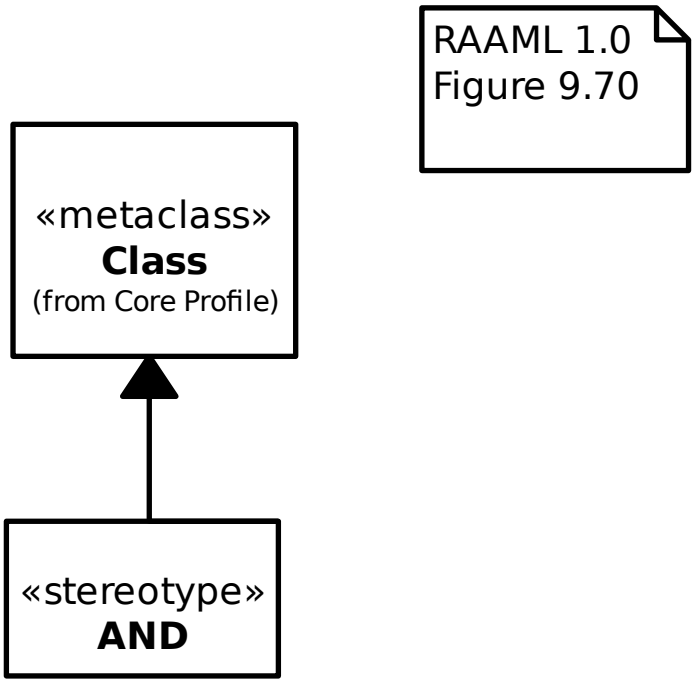
18.3.20 FTA/FTA Library/Gates/SEQ



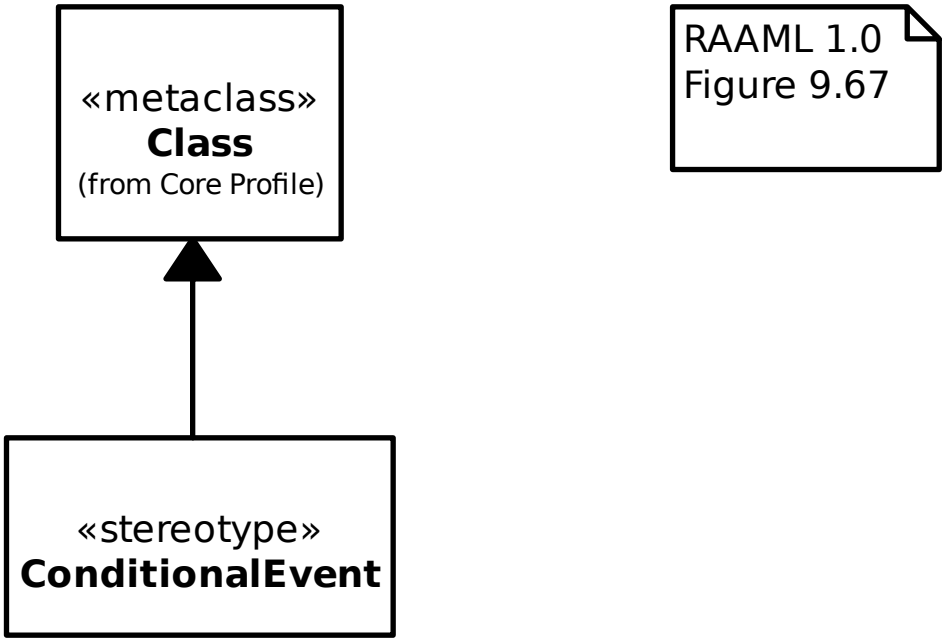
18.3.21 FTA/FTA Library/Gates/XOR



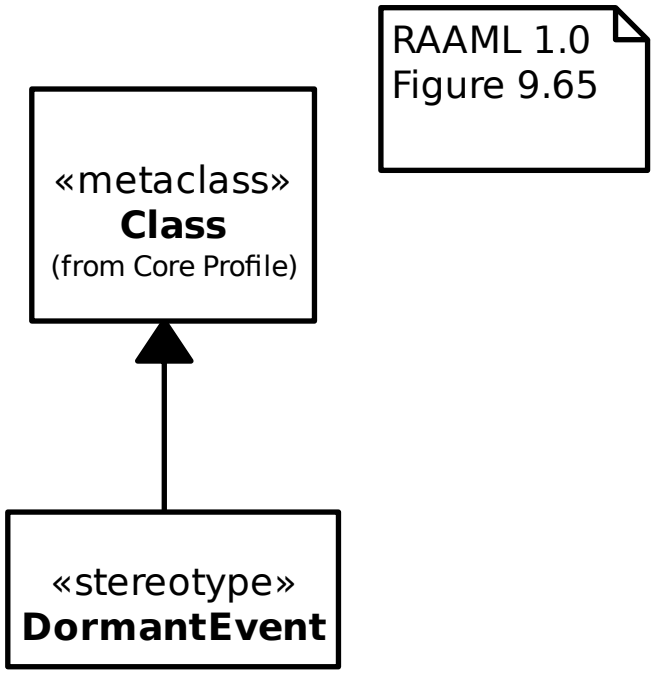
18.3.22 FTA/FTA Profile/AND



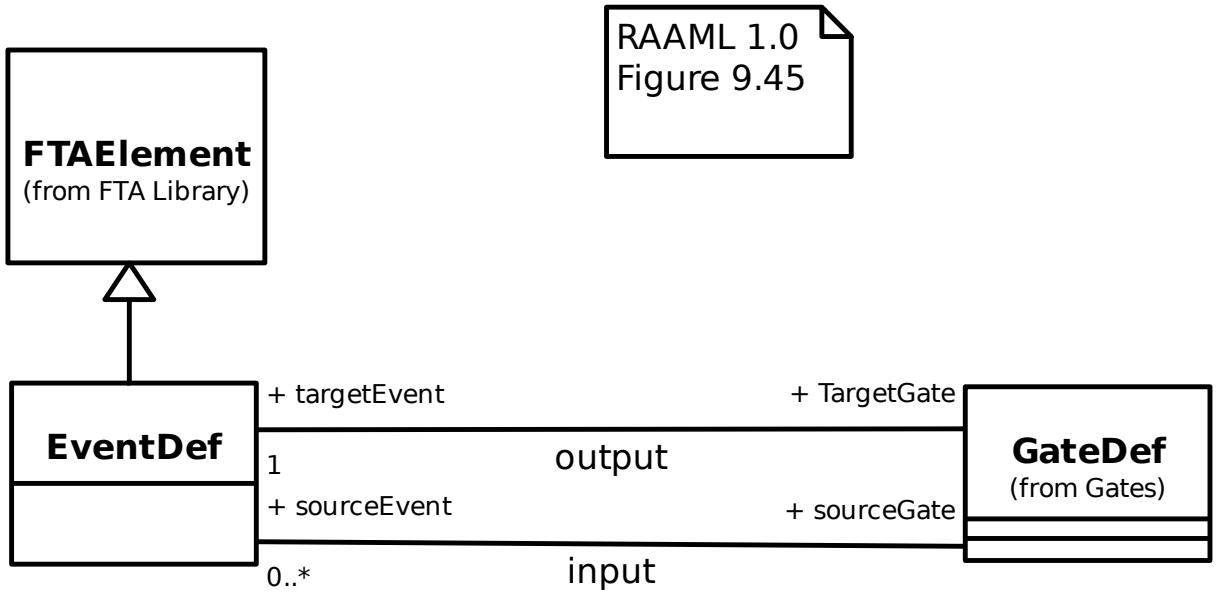
18.3.23 FTA/FTA Profile/Conditional Event



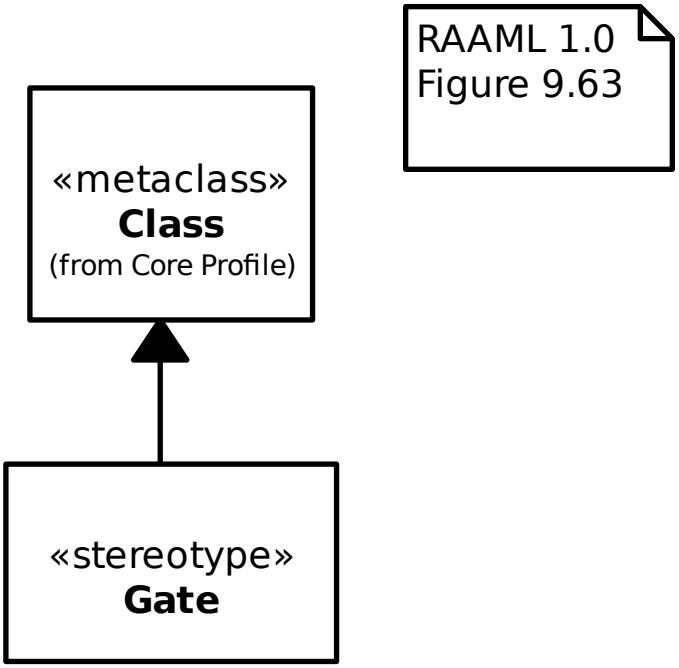
18.3.24 FTA/FTA Profile/Dormant Event



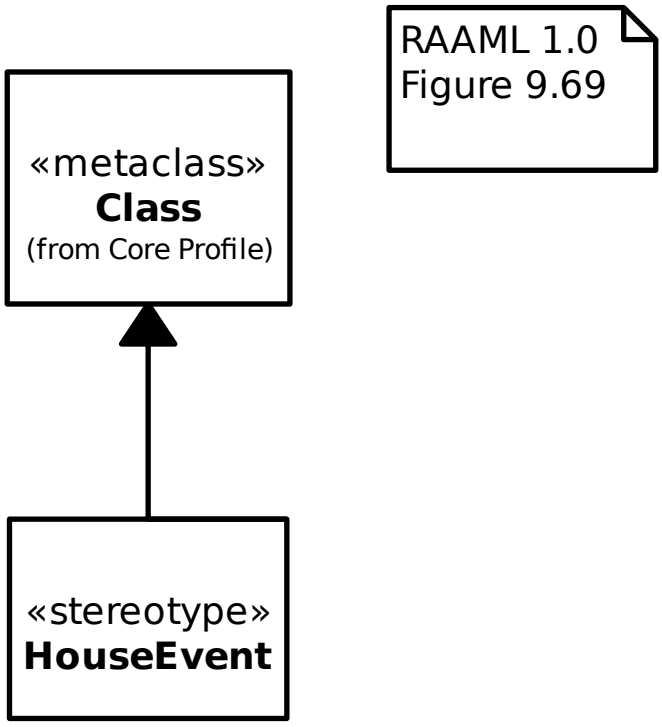
18.3.25 FTA/FTA Profile/Event



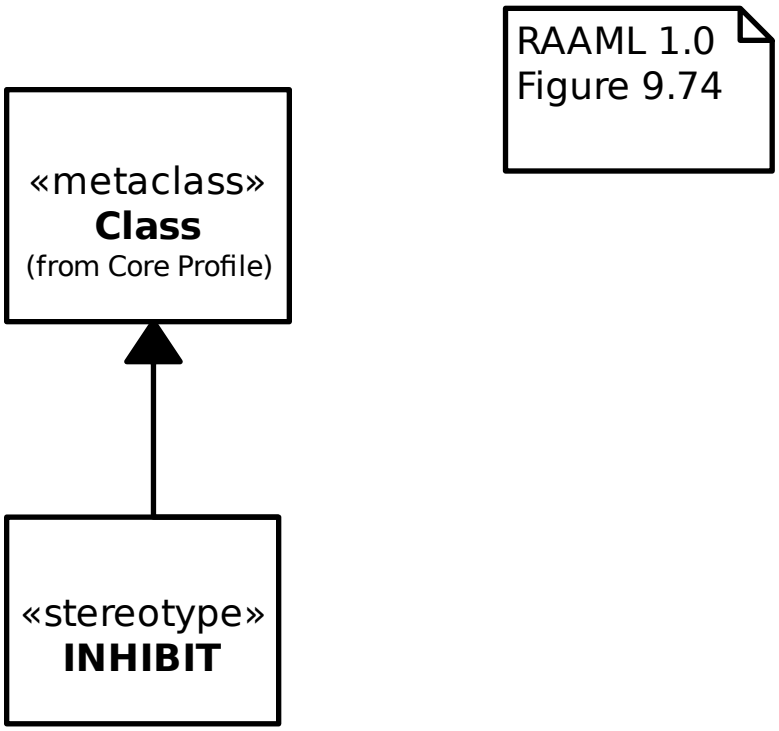
18.3.26 FTA/FTA Profile/Gate



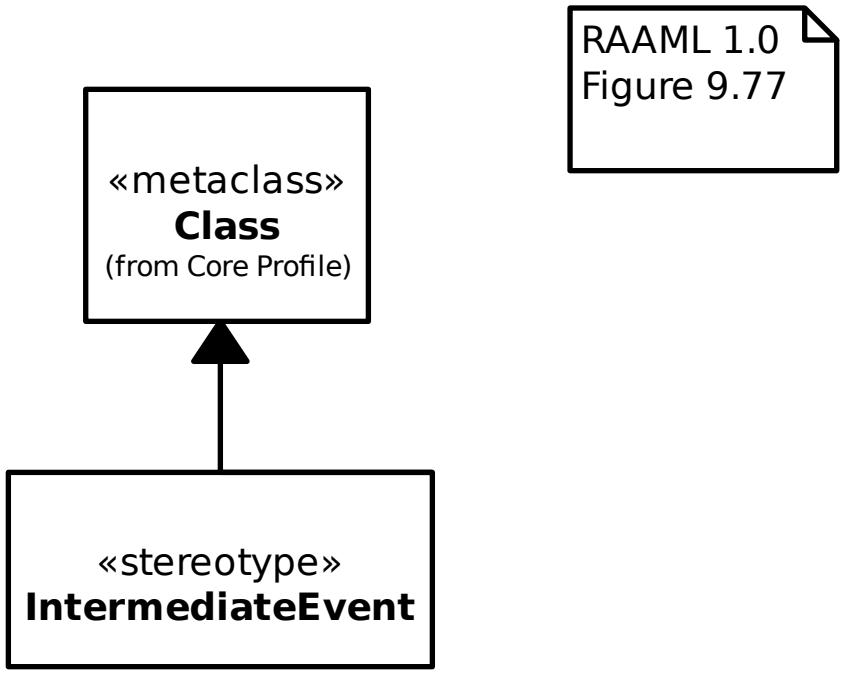
18.3.27 FTA/FTA Profile/House Event



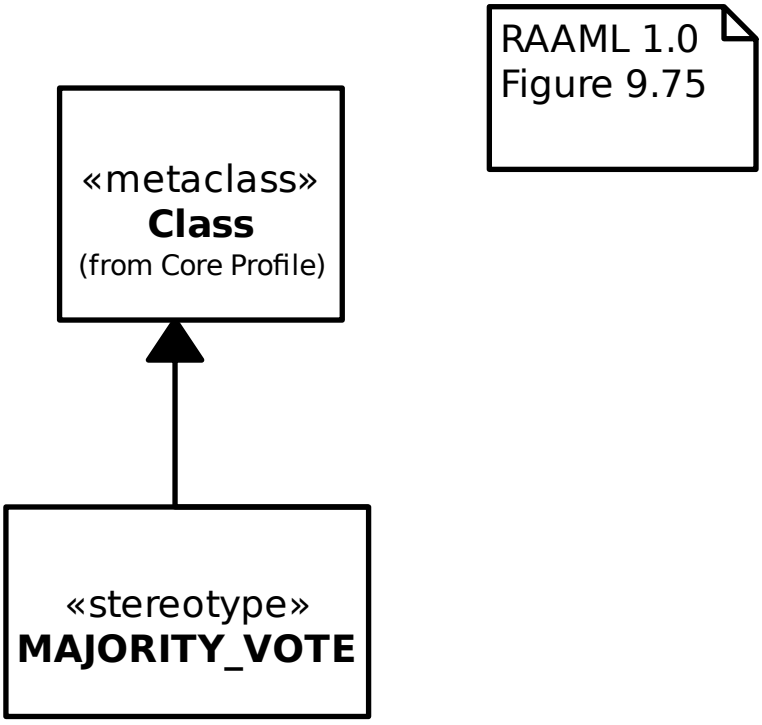
18.3.28 FTA/FTA Profile/INHIBIT



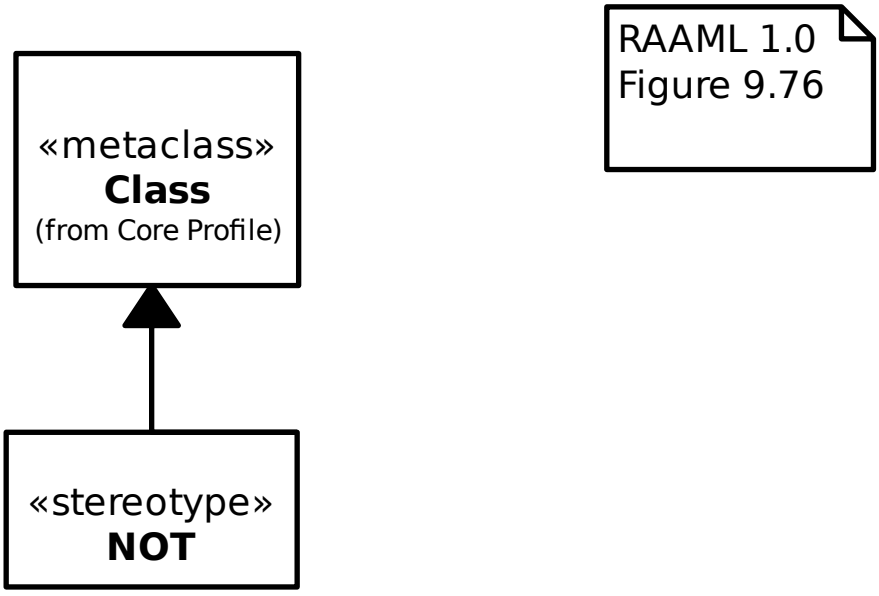
18.3.29 FTA/FTA Profile/Intermediate Event



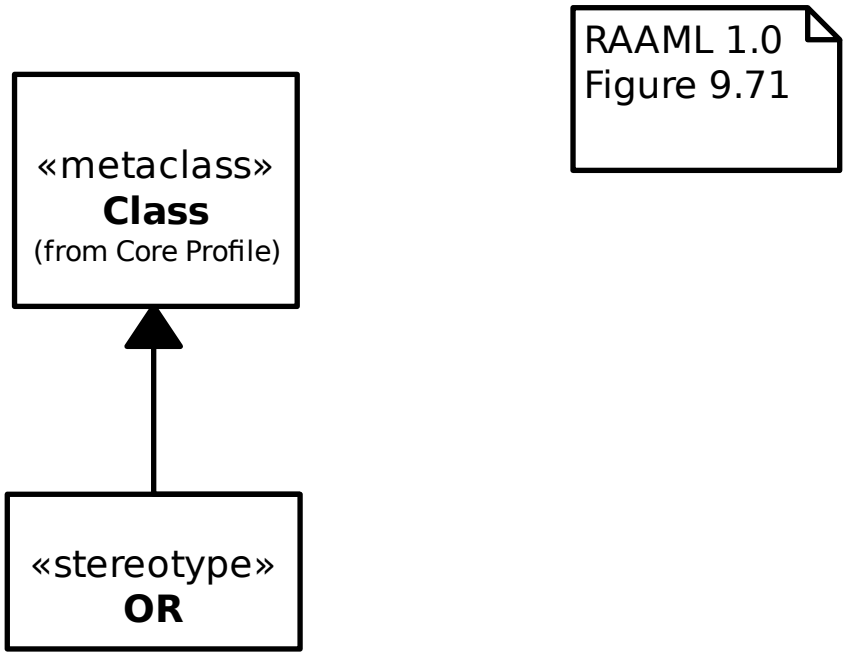
18.3.30 FTA/FTA Profile/MAJORITY_VOTE



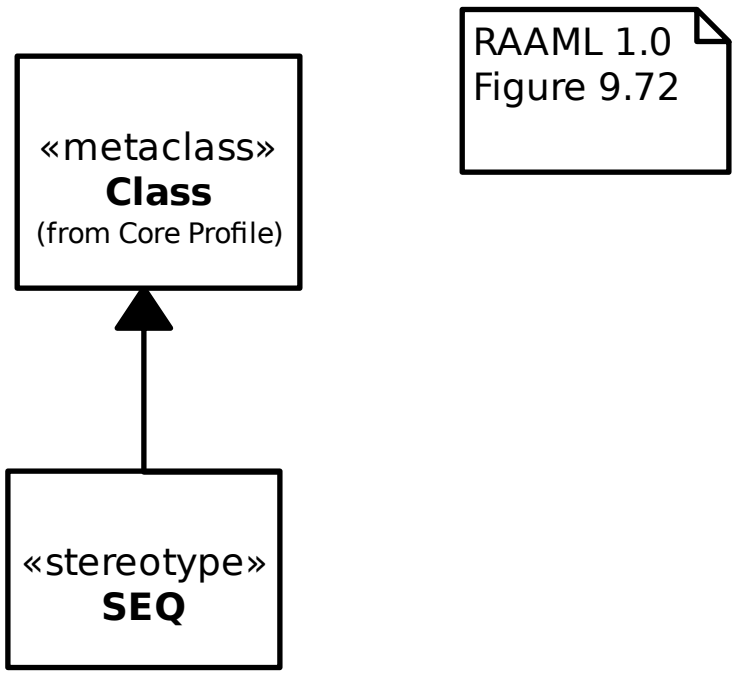
18.3.31 FTA/FTA Profile/NOT



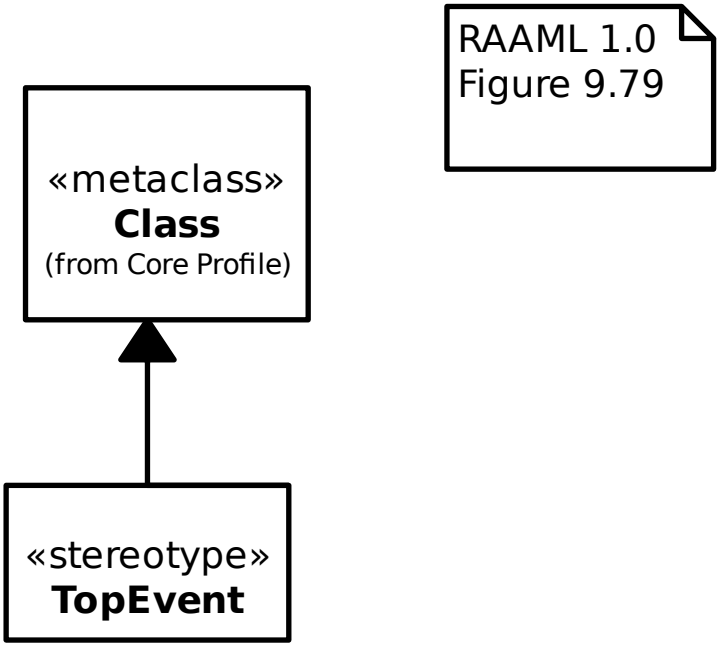
18.3.32 FTA/FTA Profile/OR



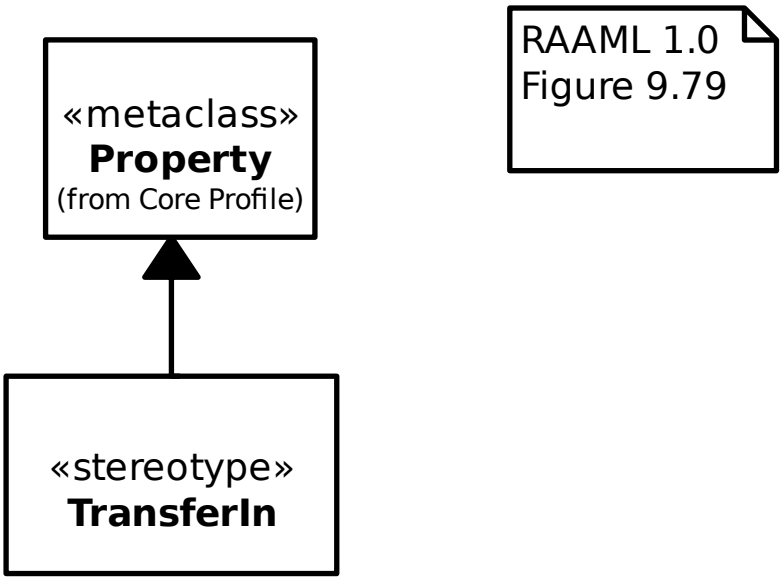
18.3.33 FTA/FTA Profile/SEQ



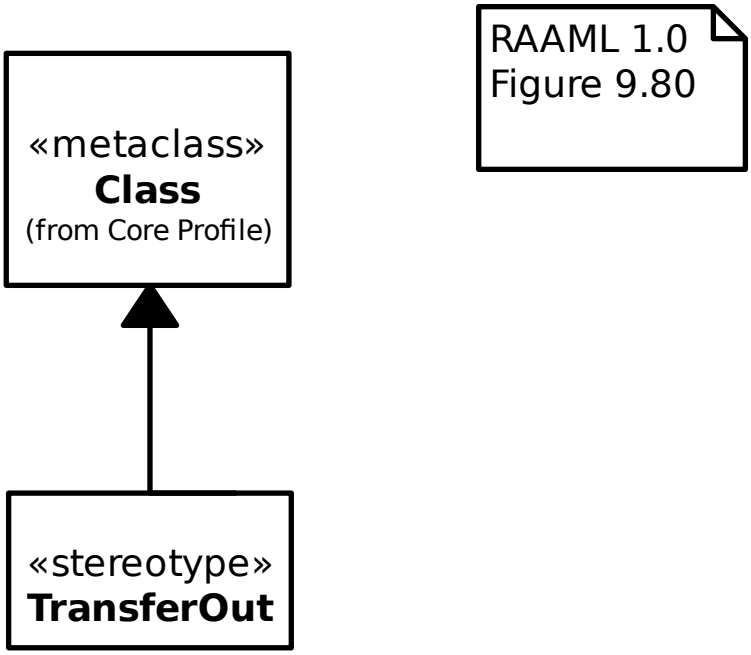
18.3.34 FTA/FTA Profile/Top Event



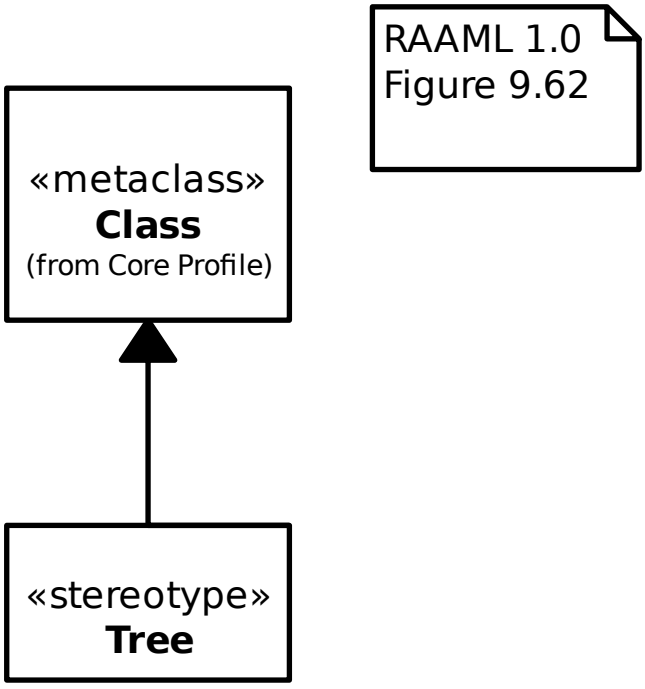
18.3.35 FTA/FTA Profile/Transfer In



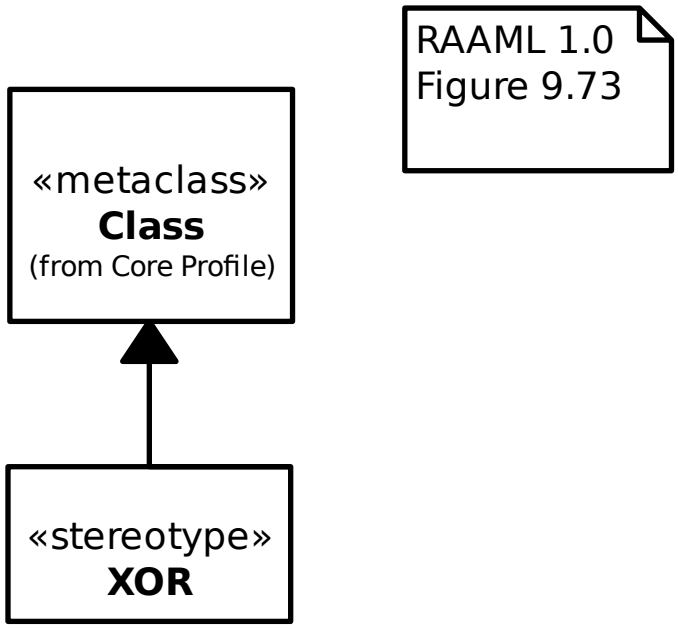
18.3.36 FTA/FTA Profile/Transfer Out



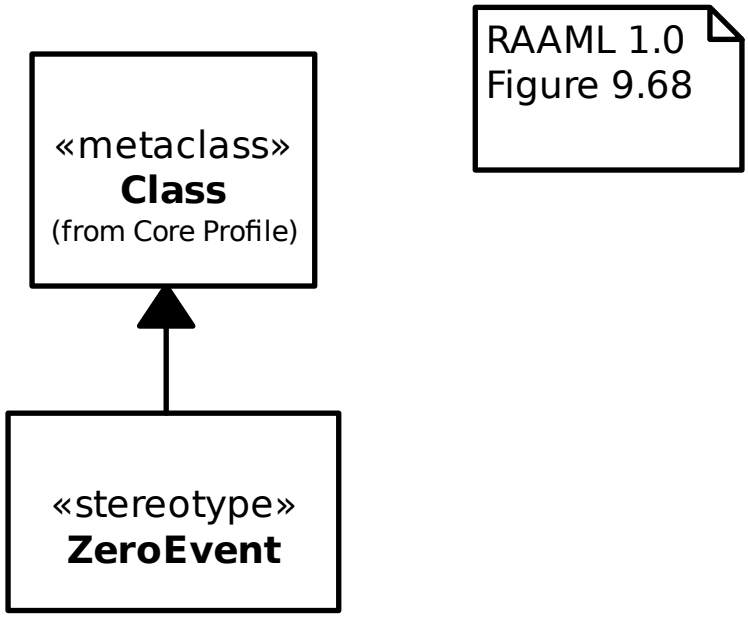
18.3.37 FTA/FTA Profile/Tree



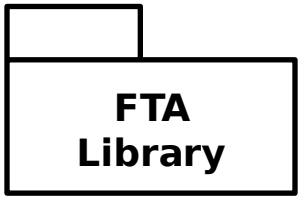
18.3.38 FTA/FTA Profile/XOR

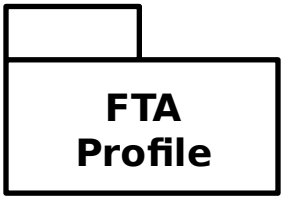


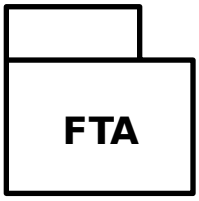
18.3.39 FTA/FTA Profile/Zero Event

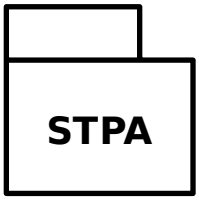


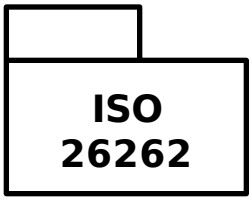
18.3.40 FTA/FTA











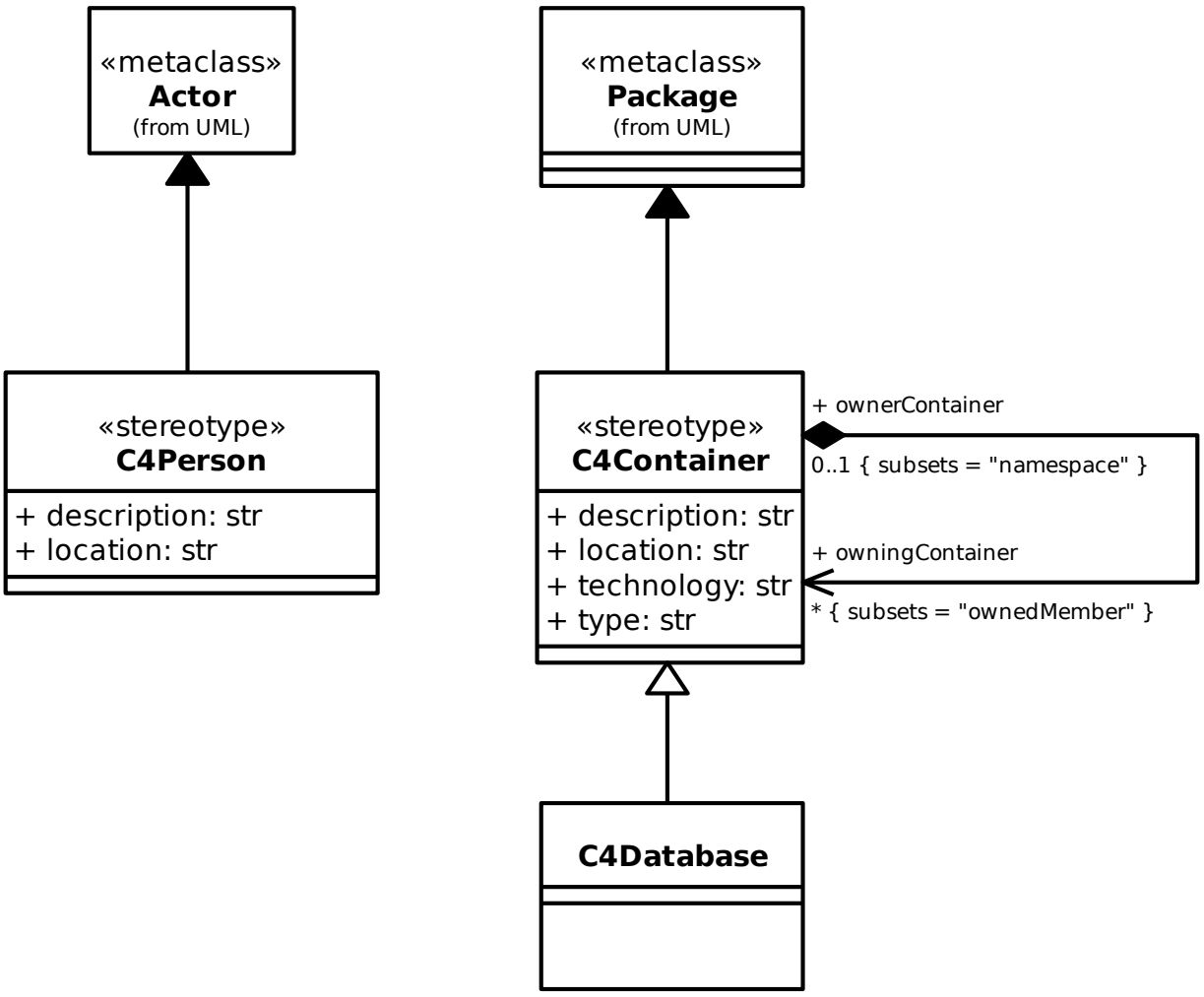
CHAPTER

NINETEEN

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

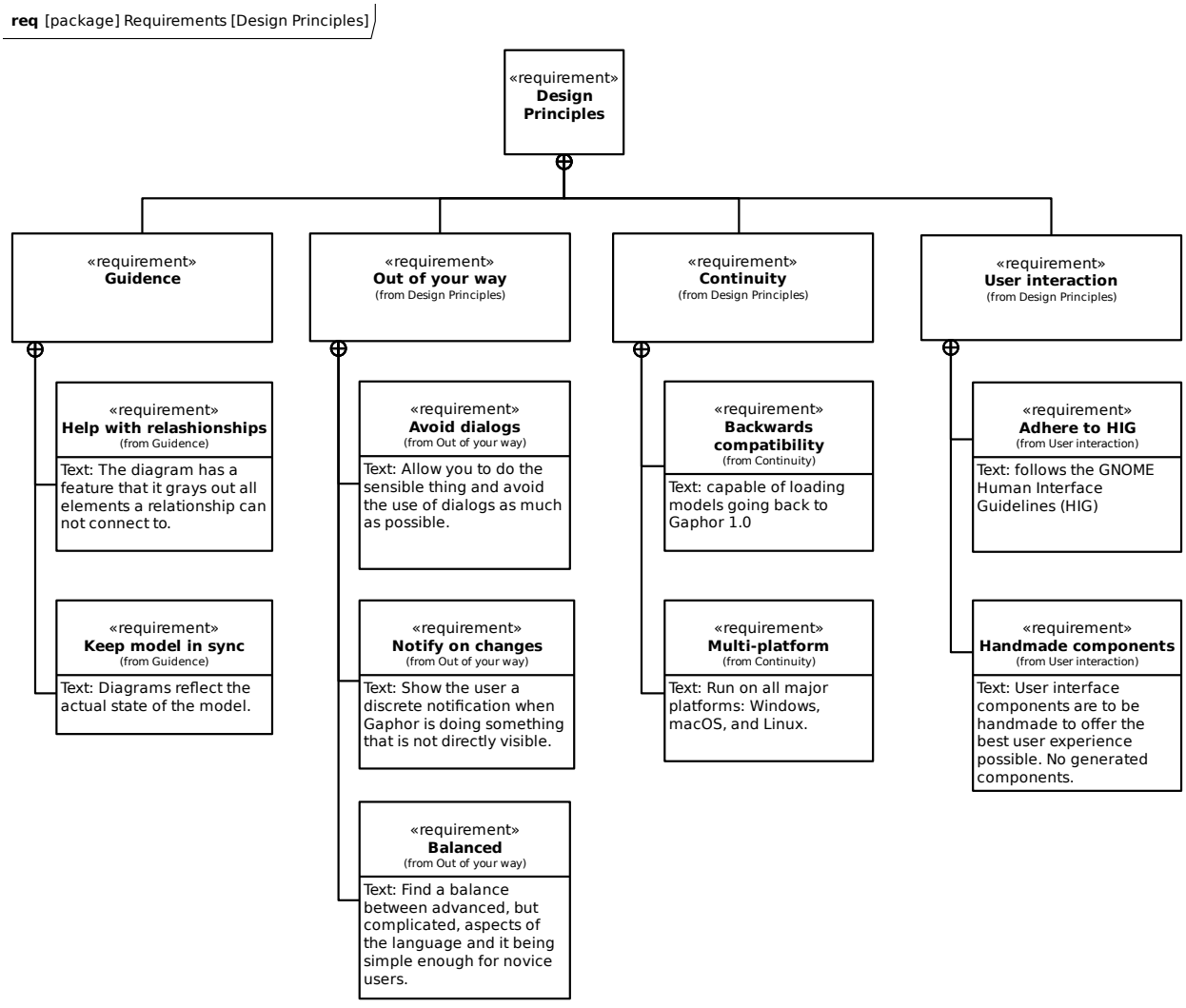
TWENTY

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.



20.1 Guidance

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

20.1.1 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.

20.1.2 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.

20.2 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".

20.2.1 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.

20.2.2 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`.

20.2.3 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.

20.3 Continuity

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

20.3.1 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.

20.3.2 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.

20.4 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.

20.5 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

TWENTYONE

FRAMEWORK

21.1 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*.

21.2 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*.

21.3 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.

21.4 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:

21.4.1 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.

21.4.2 file_manager

Loading and saving a model is done through this service.

21.4.3 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.

21.4.4 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

TWENTYTWO

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.

22.1 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.

22.2 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.

22.3 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"
```

22.4 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).

CHAPTER

TWENTYTHREE

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

TWENTYFOUR

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?

Three functionalities are exposed by a *`ModelingLanguage` instance*:

- A data model (elements) and diagram items
- Diagram types
- A toolbox definition

Other functionalities can be extended by adding handlers to the respective generic functions:

- *Connectors*, allow diagram items to connect
- *Format/parse* model elements to and from a textual representation
- *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

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

24.1 Modeling language

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 property element_types: Iterable[ElementCreateInfo]

Iterate element 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.

24.2 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** – connecting item
- **element** – connectable item

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

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

Determine if items can be connected.

Is the connection allowed at all (during mouse movement for example)?

Returns *True* if connection is allowed.

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

Connect to an element.

Establish a connection between element and line. Also takes care of disconnects, if required (e.g. 1:1 relationships).

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.

Break connection, called when dropping a handle on a point where it can not connect.

get_connected(*handle: Handle*) → *Presentation*[*Element*] | None

Get item connected to a handle.

24.3 Format and parse

Model elements can be formatted to a simple text representation. For example, This is used in the Model Browser. It isn't a full serialization of the model element.

In some cases it's useful to parse a text back into an object. This is done when you edit attributes and operations on a class.

Not every `format()` needs to have an equivalent `parse()` function.

gaphor.core.format.**format**(*element:* Element) → str

Returns a human readable representation of the model element. In most cases this is just the name, however, properties (attributes) and operations are formatted more extensively:

```
+ attr: str
+ format(element: Element): string
```

gaphor.core.format.**parse**(*element:* Element, *text: str*) → None

Parse `text` and populate `element`. The element is populated with elements from the text. This may mean that new model elements are created as part of the parse process.

24.4 Copy and paste

Copy and paste works out of the box for simple items: one diagram item with one model element (the `subject`). It leverages 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. A copy function normally copies only the element and mandatory related elements. E.g. an Association needs two association ends.

gaphor.diagram.copypaste.**paste**(*copy_data: Opaque*, *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 provides some convenience functions:

gaphor.diagram.copypaste.**copy_full**(*items: Collection[*Element*]*, *lookup: Callable[[Id],* Element | *None]* | *None* = *None*) → CopyData:

Copy `items`. The `lookup` function is used to look up owned elements (shown as child nodes in the Model Browser).

gaphor.diagram.copypaste.**paste_link**(*copy_data: CopyData*, *diagram:* Diagram) → set[~gaphor.core.modeling.Presentation]:

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

gaphor.diagram.copypaste.**paste_full**(*copy_data: CopyData*, *diagram:* Diagram) → set[~gaphor.core.modeling.Presentation]:

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

24.5 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.

24.6 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`.

24.7 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.

24.8 Property Editor 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() → Gtk.Widget | None

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

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

24.9 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:* EventManager, *pos: tuple[int, int] | None = None*) → bool

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

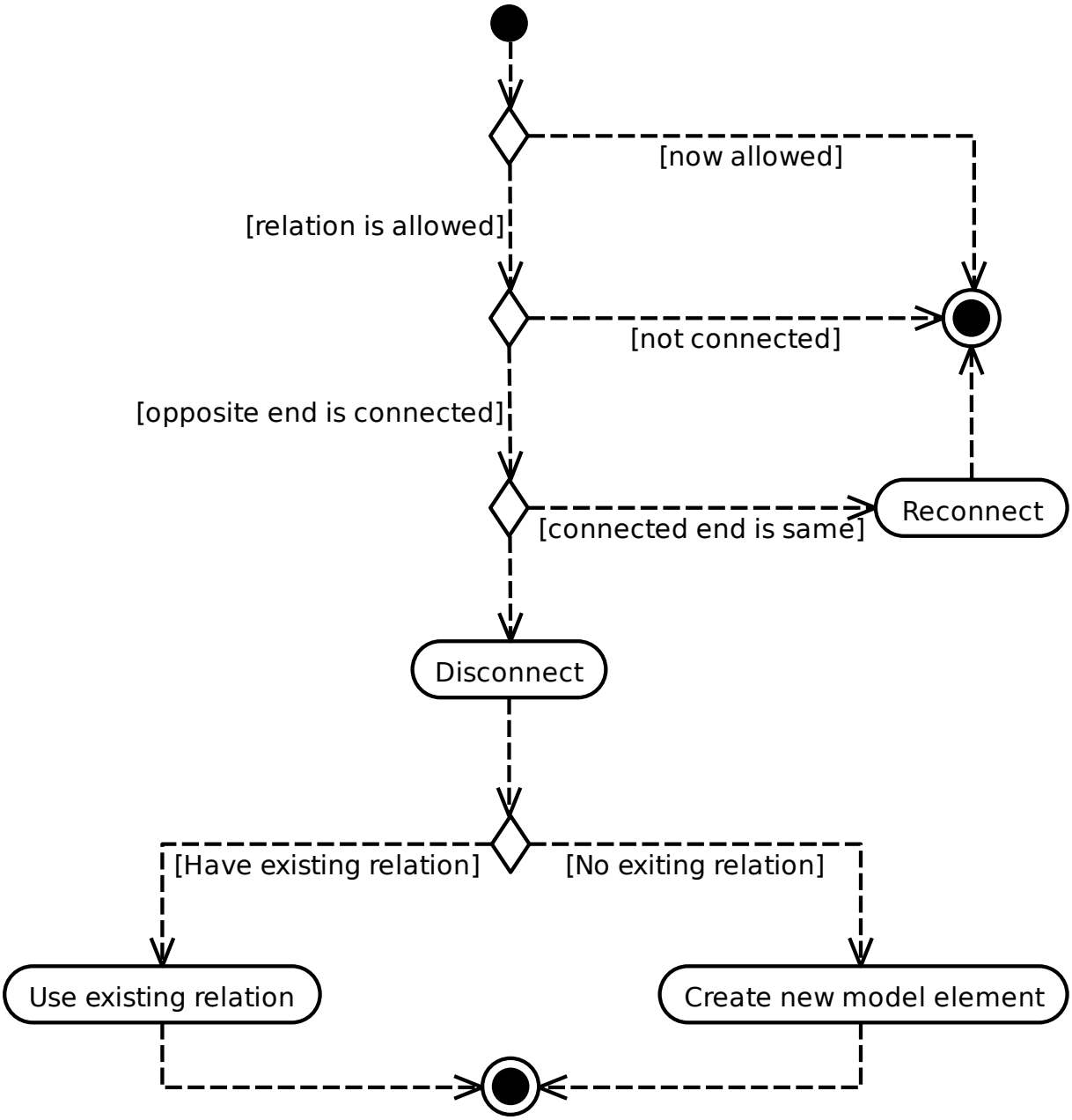
In case of a mouse press event, the mouse position (relative to the element) are also provided.

CHAPTER

TWENTYFIVE

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

TWENTYSIX

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 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)

```
    </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

TWENTYSEVEN

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.

27.1 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.

27.2 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.

27.3 Successful Transaction

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

27.4 Failed Transaction

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

27.5 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

TWENTYEIGHT

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

H

I

L

M

N

O

P

R

S

T

U

W