Gaphor Documentation

Release 2.0.0

Arjan J. Molenaar

Aug 14, 2020

INSTALLATION

1 Gaphor on Linux 3

2 Gaphor on macOS 5

3 Gaphor on Windows 7

4 Framework 9

5 Service Oriented Architecture 11

6 Event System 15

7 Modeling Languages 17

8 Style Sheets 21

9 Transaction support 27

10 Items and Elements 31

11 UML and SysML Data Model 33

12 Stereotypes 35

13 Data Model 37

14 Connection Protocol 39

15 Saving and Loading models 41

16 File Format (XML) 43

17 Undo Manager 45

18 External links 47

Index 49

This documentation is aimed at those who would be interested in making contributions to Gaphor. For tutorials and how-to information, please visit the Gaphor Website.

In the future, we would like to split the documentation in to sections that focus on **explanation** (understanding-oriented) and **reference** (information-oriented). For now, this information is all together.

If you're into writing plug-ins for Gaphor you should have a look at our fabulous Hello world plug-in.

Setting up a development environment, and packaging Gaphor on different platforms:

CHAPTER

ONE

GAPHOR ON LINUX

1.1 Development Environment

To setup a development environment with Linux, you first need the latest stable version of Python. In order to get the latest stable 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. Finally 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.

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-3.0 libgirepository1.0-dev libcairo2-dev
```

Clone the repository.

```
$ cd gaphor
$ source venv
$ poetry run gaphor
```

1.2 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 3.34
```

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
```

1.3 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

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

CHAPTER

TWO

GAPHOR ON MACOS

2.1 Development Environment

To setup a development environment with macOS:

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

```
$ brew install python3 gobject-introspection gtk+3
```

Clone the repository.

```
$ cd gaphor
$ source ./venv
$ poetry run gaphor
```

2.2 Packaging for macOS

In order to create a dmg package for macOS, we utilize a custom bash script that picks up the required files, drops them in a bundle, and changes the link references.

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

```
$ cd macos-dmg
$ source package.sh
```

CHAPTER

THREE

GAPHOR ON WINDOWS

3.1 Development Environment

To setup a development environment in Windows:

1. Go to http://www.msys2.org/ and download the x86_64 installer
2. Follow the instructions on the page for setting up the basic environment
3. Run `C:\msys64\mingw64.exe` - a terminal window should pop up

```
$ pacman -Suy
$ pacman -S git mingw-w64-x86_64-gcc mingw-w64-x86_64-gtk3 \
mingw-w64-x86_64-pkg-config mingw-w64-x86_64-cairo \
mingw-w64-x86_64-gobject-introspection mingw-w64-x86_64-python \
mingw-w64-x86_64-python-gobject mingw-w64-x86_64-python-cairo \
mingw-w64-x86_64-python-pip
```

Clone the repository.

```
$ cd gaphor
$ source venv
$ poetry run gaphor
```

3.2 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.

1. Follow the instructions for settings up a development environment above
2. Run `C:\msys64\mingw64.exe` - a terminal window should pop up

```
$ cd win-installer
$ ./build-installer.sh
```

CHAPTER

FOUR

FRAMEWORK

4.1 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*.

4.2 Events

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*.

4.3 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.

4.4 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:

4.4.1 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.

4.4.2 file_manager

Loading and saving a model is done through this service.

4.4.3 element_factory

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

4.4.4 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

FIVE

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.

5.1 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.

5.2 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.

5.3 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"
```

5.4 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 (see gaphor/action.py).

5.5 Example plugin

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

The setup.py file contains an entry point:

```
entry_points = {
    'gaphor.services': [
        'helloworld = gaphor.plugins.helloworld: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(name='helloworld',                        # 5.
            label=_('Hello world'),
            tooltip=_('Every application ...'))
    def helloworld_action(self):
        main_window = self.main_window
        pass # gtk code left out
```

1. As stated before, a plugin should implement the `Service` interface. It also implements `ActionProvider`, saying it has some actions to be performed by the user.
2. The menu entry will be part of the "Tools" extension menu. This extension point is created as a service. Other services can also be passed as dependencies. Services can get references to other services by defining them as arguments of the constructor.
3. All action defined in this service are registered.
4. Each service has a `shutdown()` method. This allows the service to perform some cleanup when it's shut down.
5. The action that can be invoked. The action is defined and will be picked up by `add_actions()` method (see 3.)

CHAPTER

SIX

EVENT SYSTEM

The Generic library provides the `generic.event` module which is used to implement the event system in Gaphor. This event system in Gaphor provides an API to *subscribe* to events and to then *handle* those events so that previously subscribed *handlers* are executed.

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.

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.

For more information about how the Generic library handles events see the Generic documentation.

CHAPTER

SEVEN

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.

A ModelingLanguage 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. However, the responsibilities of a ModelingLanguage do not stop there. Parts of functionality will be implemented by registering handlers to a set of generic functions.

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

- A data model (elements)
- Diagram items
- A toolbox definition
- *Connectors*, allow diagram items to connect
- *Grouping*
- *Editor pages*, shown in the collapsible pane on the right side
- *Inline (diagram) editor popups*
- *Copy/paste* behavior when element copying is not trivial, for example with more than one element is involved

We expose the first three by methods defined on the ModelingLanguage class. We then expose the others by adding handlers to the respective generic functions.

class gaphor.abc.**ModelingLanguage**

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

abstract lookup_diagram_item(*name: str*) → Optional[Type[Presentation]]

Look up a diagram item type (class) by name.

abstract lookup_element(*name: str*) → Optional[Type[Element]]

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

abstract property name

Human readable name of the model.

abstract property toolbox_definition

Get structure for the toolbox.

7.1 Connectors

Connectors are used to connect one element to another.

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

class gaphor.diagram.connectors.**BaseConnector**(*element: gaphor.core.modeling.presentation.Presentation[gaphor.c*
line: gaphor.core.modeling.presentation.Presentation[gaphor.core.

Connection adapter for Gaphor diagram items.

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

Parameters

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

The following methods are required to make this work:

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

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

allow(*handle: gaphas.connector.Handle*, *port: gaphas.connector.Port*) → bool
Determine if items can be connected.

Returns *True* if connection is allowed.

connect(*handle: gaphas.connector.Handle*, *port: gaphas.connector.Port*) → bool
Connect to an element. Note that at this point the line may be connected to some other, or the same element. Also the connection at model level still exists.

Returns *True* if a connection is established.

disconnect(*handle: gaphas.connector.Handle*) → None
Disconnect model level connections.

get_connected(*handle: gaphas.connector.Handle*) → Optional[gaphor.core.modeling.presentation.Presentation[gaphor.core
Get item connected to a handle.

get_connection(*handle: gaphas.connector.Handle*) → Optional[gaphas.canvas.Connection]
Get connection information

7.2 Grouping

Grouping is done by dragging one item on top of another.

Grouping dispatch objects are normally inheriting from `AbstractGroup`.

class gaphor.diagram.grouping.**AbstractGroup**(*parent: gaphor.core.modeling.presentation.Presentation*,
item: gaphor.core.modeling.presentation.Presentation)

Base class for grouping model elements, i.e. interactions contain lifelines and components contain classes objects.

Parameters

- **parent** (*Presentation*) – Parent item, which groups other items.
- **item** (*Presentation*) – Item to be grouped.

can_contain() → bool
Determine if parent can contain item.

abstract group() → None
Perform grouping of items.

abstract ungroup() → None
Perform ungrouping of items.

7.3 Editor property pages

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

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

class gaphor.diagram.propertypages.**PropertyPageBase**
A property page which can display itself in a notebook

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

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

7.4 Inline (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.inlineeditors.**InlineEditor**(*item: Item*, *view*, *pos: Optional[Tuple[int, int]] = None*) → bool
Show a small editor popup in the diagram. Makes for easy editing without resorting to the Element editor.

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

7.5 Copy and paste

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

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

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

gaphor.diagram.copypaste.**copy**(*obj: Element*) → T
Create a copy of an element (or list of elements). The returned type should be distinct, so the *paste()* function can properly dispatch.

gaphor.diagram.copypaste.**paste**(*copy_data: T*, *diagram: Diagram*, *lookup: Callable[[str], Element]*) → object

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

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

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

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

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

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

CHAPTER

EIGHT

STYLE SHEETS

Since Gaphor 2.0, Gaphor 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.

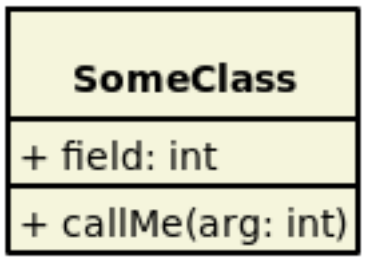
Its application goes well beyond web documents, though. Gaphor uses CSS to provide style elements to items in diagrams. CSS allows us, users of Gaphor, to change the visual appearance of our diagrams. Color and line styles can be changed to make it easier to read the diagrams.

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.

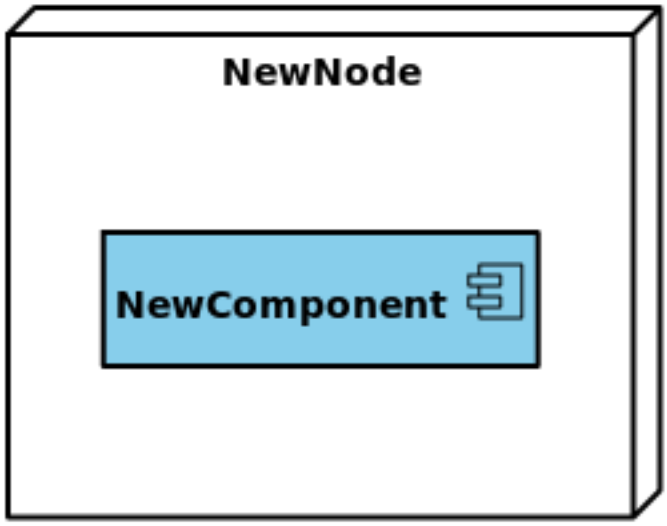
Here is a simple example of how to change the background color of a class:

```
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: black;
}

* {
  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.

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

- The official specification of CSS3 attribute selectors.
- We provide the `|=` attribute selector for the sake of completeness. It's probably not very useful in a Gaphor context.
- 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.

8.2 Style properties

Gaphor supports a subset of CSS properties and some Gaphor specific properties. The style sheet interpreter is relatively straight forward. We measure all widths, heights, and sizes in pixels. We don't support complex style declarations, like the `font` property in HTML/CSS which can contain font family, size, weight.

8.2.1 Colors

background-color	Examples: `background-color: azure;` `background-color: rgb(255, 255, 255);` `background-color: hsl(130, 95%, 10%);`
color	Color used for lines
highlight-color	Color used for highlight, e.g. when dragging an item over another item.
text-color	Color for text

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

8.2.2 Text and fonts

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

- `font-family` can be only one font name, not a list of (fallback) names, as is used for HTML.

8.2.3 Drawing and spacing

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

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

8.2.4 Diagram styles

Only a few properties can be defined on a diagram, namely `background-color` and `line-style`. We 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.

8.3 Ideas

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

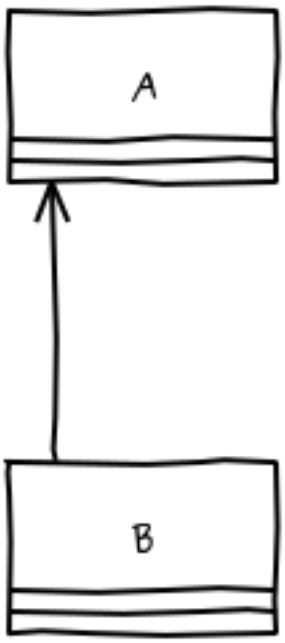
To create your own expression you may want to use the Console (Tools -> Console, in the Hamburger menu). Drop us a line on Gitter and we would be happy to help you.

8.3.1 The drafts package

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

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

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



8.3.2 Unconnected relationships

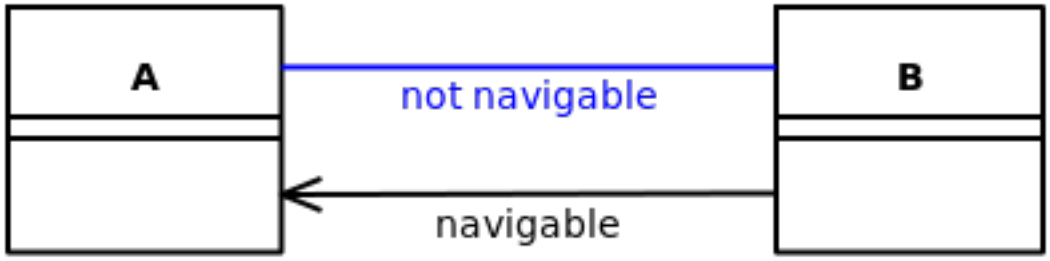
All items on a diagram that are not backed by a model element, should be drawn in a dark red color. This can be used to spot not-so-well connected relationships, such as Generalization, Implementation, and Dependency. These items will only be backed by a model element once you connect both line ends.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

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

8.3.3 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;
}
```



CHAPTER

NINE

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()
...
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()
...
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

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

Cleanup:

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

CHAPTER

TEN

ITEMS AND ELEMENTS

Diagram items, or just items for short, represent the UML metamodel on a diagram. In other words, they are the graphical objects in a diagram. `DiagramItem` supports item style, text elements, and stereotypes.

Elements refer to the semantic model objects. `Element` is the base class for the UML data classes.

CHAPTER

ELEVEN

UML AND SYSML DATA MODEL

Gaphor uses the UML and SysML Specifications as guidelines for its own data storage. The Python data model is generated from a Gaphor model file that describes the relationships between the supported UML elements.

The model is built using smart properties (descriptors). These properties emit events when they're changed. This allows the rest of the application, for example, the visuals and undo system, to update their state accordingly. The events are sent using a signaling mechanism, called handlers.

11.1 Model details

Pay attention to the following changes/additions with respect to the official Gaphor model, in the `models/UML.gaphor` file:

- Additions to the model have been put in the package AuxiliaryConstructs.Presentations and .Stereotypes.
- A Diagram element is added in order to model the diagrams.
- A special construct has been put into place in order to apply stereotypes to model elements. The current UML Specification is not clear on that subject.
- The Slot.value reference is singular.
- ValueSpecification is generated as if it were a normal attribute. As a result, its subclasses (Expression, OpaqueExpression, InstanceValue, LiteralSpecification and its Literal* subclasses) are not available.

CHAPTER

TWELVE

STEREOTYPES

UML defines a concept called a stereotype, which is how you can extend an existing metaclass. This allows you to create new notation that can add to or replace existing elements. In order to create a stereotype in Gaphor, first you need to create a Profile. A profile is a collection of stereotypes. Next a Diagram can be created within the profile. Although a diagram in Gaphor can accept any type of element, by convention, the profile diagram should only contain items that are useful within a profile:

- Classes, which will function as a `<<metaclass>>`.
- Stereotype, which will be defined as `<<stereotype>>`.
- Extensions, connecting metaclasses and stereotypes.

Comment, Association, Generalization, and Dependency can also be used within a profile diagram, just like other UML diagrams.

Some things to keep in mind when working with profiles:

- Profiles are reusable and its common to share them across different models.
- A stereotype can only be owned by a profile, it can not be in a normal Package.
- In order to make use of a stereotype, Gaphor has to perform a lookup if the MetaClass it is extended from is part of the model.
- A stereotype can contain an image, which can be used instead of its name.
- Profiles should be saved with the model too. It should also be possible to "update" a profile within a model.
- Stereotypes that you define should be instantiated in your model when you create a stereotyped class.

Note: There is no way to connect a stereotype with a class other than an Association.

CHAPTER

THIRTEEN

DATA MODEL

Gaphor is a UML and SysML tool. In order to keep as close as possible to the UML specification the Gaphor data model is based on the UML Metamodel. The Object Management Group (OMG), the not-for-profit technology standards consortium that governs UML, has a XML Metadata Interchange (XMI) file describing the metamodel. Therefore, the easiest way to keep Gaphor consistent with UML would be to to generate Gaphor's data model code directly from this UML metamodel in XMI. There are two challenges with this approach:

1. There are more attributes defined in the data model than we will use, unless Gaphor got to the point where it 100% implemented the UML specification.
2. There are no consistency rules in the UML XMI definition.

The first point ends up not being much of a problem: attributes we don't use don't consume memory.

For the second point, we have to get the model consistency rules directly from the UML Specification. Our approach is to create a special consistency module that checks the model and reports errors.

In the UML metamodel all classes are derived from `Element`. So we have created a substitute for `Element` that gives some behaviour to the data objects.

Gaphor's data model is implemented in Python like the rest of the application. Since the Python language doesn't make a difference between classes and objects, we can define the possible attributes that an object of a particular kind can have in a dictionary (name-value map) at class level. If a value is set, the object checks if an attribute exists in the class' dictionary (and the parent's dictionary). If the attribute exists, the value is assigned, if it doesn't exist then an exception is raised.

13.1 Bidirectional References

If two objects need to reference each other, a bidirectional reference is needed to be supported in Gaphor. This works very similar to the uni-directional reference that is stored in a dictionary at the class level. We add some extra information to the dictionary at class level to make relationship bidirectional. The information is stored in an extra field that gives us the name of the opposite reference. and voila, we can create bi-directional references. Please reference `gaphor/UML/element.py` for more details.

13.2 Implementation

Below is an example of the implementation that allows the user to assign a value to an instance of `Element` with name `name`. If no value is assigned before the value is requested, it returns and empty string '':

```
m = Class()
print(m.name)               # Returns ''
m.name = 'MyName'
print(m.name)               # Returns 'MyName'

m = Element()
c = Comment()
print(m.comment)              # Returns an empty list '[]'
print(c.annotatedElement)     # Returns an empty list '[]'
m.comment = c                 # Add 'c' to 'm.comment' and add 'm' to 'c.
↪annotatedElement'
print(m.comment)              # Returns a list '[c]'
print(c.annotatedElement)     # Returns a list '[m]'
```

This behavior is defined in the data model's base class: `Element`. The code for the data model is stored in uml2.py and is generated from the `uml2.gaphor` Gaphor model file.

13.3 Extensions to the Data Model

A few changes have been made to Gaphor's implementation of the metamodel. First of all some relationships have to be modified since the same name is used for different relationships. Some n:m relationships have been made 1:n. These are all small changes and should not restrict the usability of Gaphor's model.

The biggest change is the addition of a whole new class: Diagram. Diagram is inherited from Namespace and is used to hold a diagram. It contains a `gaphas.canvas.Canvas` object which can be displayed on screen by a `DiagramView` class.

13.4 UML.Element

class gaphor.core.modeling.**Element**(*id: Optional[Union[str, bool]] = None*, *model: Optional[gaphor.core.modeling.element.RepositoryProtocol] = None*)

Base class for all model data classes.

CHAPTER

FOURTEEN

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. Please reference the page on *Items and Elements* if you need a reminder on the difference between the two.

Is relation with this element allowed?

No: do nothing (not even glue should have happened as the same question is asked there).

Yes: connect_handle() Is opposite end connected?

No: Do nothing

Yes:

Does the item already have a subject element relation?

Yes:

Is the previous item the same as the current?

Yes: Do nothing

No: Let subject end point to the new element

No: Create relation or find existing relation in model

Search for an existing relation in the model:

Found: Use that relation

Nothing: Create new model elements and connect to item

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

FIFTEEN

SAVING AND LOADING MODELS

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: `Reference` and `Value`. Reference is used to point to other UML elements, Value has a value inside (an integer or a string).

Diagram is a special case. Since this element contains a diagram canvas inside, it may become pretty big (with lots of nested elements). This is handled by the load and save function of the Diagram class. All elements inside a canvas have a tag `Item`.

```
<?xml version="1.0" ?>
<Gaphor version="1.0" gaphor_version="0.3">
  <Package id="1">
    <ownedElement>
      <reflist>
        <ref refid="2"/>
        <ref refid="3"/>
        <ref refid="4"/>
      </reflist>
    </ownedElement>
  </Package>
  <Diagram id="2">
    <namespace>
      <ref refid="1"/>
    </namespace>
    <canvas extents="(9.0, 9.0, 189.0, 247.0)" grid_bg="0xFFFFFFFF"
     grid_color="0x80ff" grid_int_x="10.0" grid_int_y="10.0"
     grid_ofs_x="0.0" grid_ofs_y="0.0" snap_to_grid="0"
     static_extents="0" affine="(1.0, 0.0, 0.0, 1.0, 0.0, 0.0)"
     id="DCE:xxxx">
      <item affine="(1.0, 0.0, 0.0, 1.0, 150.0, 50.0)" cid="0x8293e74"
         height="78.0" subject="3" type="ActorItem" width="38.0"/>
      <item affine="(1.0, 0.0, 0.0, 1.0, 10.0, 10.0)" cid="0x82e7d74"
         height="26.0" subject="5" type="CommentItem" width="100.0"/>
    </canvas>
  </Diagram>
  <Actor id="3">
    <name>
      <val><![CDATA[Actor]]></val>
    </name>
    <namespace>
      <ref refid="1"/>
    </namespace>
```

(continues on next page)

(continued from previous page)

```
  </Actor>
  <UseCase id="4">
    <namespace>
      <ref refid="1"/>
    </namespace>
  </UseCase>
  <Comment id="5"/>
</Gaphor>
```

CHAPTER

SIXTEEN

FILE FORMAT (XML)

Since Gaphor generates the Python data model from a Gaphor model file, it would be possible to also generate a Document Type Definition (DTD) as well.

These are the things that should be distinguished:

- Model elements
- Associations with other model elements (referenced by ID):
  - Relations with multiplicity of zero to one (0..1)
  - Relations with multiplicity of zero or more (0..*)
- Attributes, which always have a multiplicity of zero to one (0..1)
- Diagrams
  - One canvas
  - Several canvas items
- Derived attributes and associations are not saved
- Model elements should have their class name as tag name:

```
<Class id="1234-5678-...">
  ...
</Class>
<Package id="xxx...">
  ...
</Package>
```

- Support for the two types of Associations, single and multiple:

```
<Class id="xxx.xxx...">
  <package>
    <ref refid="xxx..."/>
  </package>
</Class>
<Package id="xxx...">
  <ownedClassifier>
    <reflist>
      <ref refid="xxx.xxx..."/>
      ...
    </reflist>
  </ownedClassifier>
</Package>
```

- Associations contain primitive data, this can always be displayed as strings:

```
<Class id="xxx.xxx...">
  <name>My name</name>
  <intvar>4</intvar>
</Class>
```

- Canvas is the tag in which all canvas related stuff is placed. This is the same way it is done now:

```
<Diagram id="...">
  <canvas>
    <item type="AssociationItem">
      <subject>
        <ref refid="..."/>
      </subject>
      <width><val>100.0</val></width>
    </item>
  </canvas>
</Diagram>
```

Most of the time you do not want to have anything to do with the canvas. The data stored there is specific to Gaphor. The model elements however, are interesting for other things such as code generators and conversion tools. Gaphor is also able to export diagrams into a SVG image, so that they can be used outside of the tool.

CHAPTER

SEVENTEEN

UNDO MANAGER

Undo is implemented in order to erase the last change done, reverting it to an older state or reversing the command that was done to the model being edited. With the possibility of undo, users can explore and work without fear of making mistakes, because they can easily be undone.

Gaphor makes use of the undo system in Gaphas, which is Gaphor's Canvas widget. Gaphas implements this undo system by storing the reverse operations to an undo list. For example, if you add a new item to the screen, it will save the remove operation to the undo list. If undo is then called, Gaphas will then remove the item from the screen that was added.

17.1 Overview of Transactions

Gaphor adds on to what Gaphas provides with an undo service, called the Undo Manager. The Undo Manager works transactionally. This means that if something is being updated by the user in a model, each change is divided into operations called transactions. Each operation 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 decorator called `@transactional`. Methods then make use of this decorator, and the undo data is stored in a transaction once the method is called. For example, pasting data in the model using the `copyservice` module and setting a value on an object's property page both create new transactions.

When transactions take place, they also 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. Please see the next sections for more detail on how these event notifications work during a transaction.

17.2 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.

17.3 Successful Transaction

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

17.4 Failed Transaction

1. A `TransactionRollback` event is emitted. TransactionRollback event is emitted
2. The `UndoManager` plays back all the recorded actions and then stops listening.

17.5 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

EIGHTEEN

EXTERNAL LINKS

- You should definitely check out Agile Modeling including these pages:
    1. UML Diagrams (although Gaphor does not see it that black-and-white).
    2. http://www.agilemodeling.com/essays/
- The official UML specification. This ‘’is” our data model.

INDEX

A

AbstractGroup (*class in gaphor.diagram.grouping*), 18
ActionProvider (*class in gaphor.abc*), 12
allow() (*gaphor.diagram.connectors.BaseConnector method*), 18

B

BaseConnector (*class in gaphor.diagram.connectors*), 18
built-in function
    gaphor.diagram.copypaste.copy(), 19
    gaphor.diagram.copypaste.deserialize(), 20
    gaphor.diagram.copypaste.paste(), 19
    gaphor.diagram.copypaste.serialize(), 20
    gaphor.diagram.inlineeditors.InlineEditor(), 19

C

can_contain() (*gaphor.diagram.grouping.AbstractGroup method*), 19
close() (*gaphor.ui.abc.UIComponent method*), 12
connect() (*gaphor.diagram.connectors.BaseConnector method*), 18
construct() (*gaphor.diagram.propertypages.PropertyPageBase method*), 19

D

disconnect() (*gaphor.diagram.connectors.BaseConnector method*), 18

E

Element (*class in gaphor.core.modeling*), 38
EventManager (*class in gaphor.core.eventmanager*), 15

G

gaphor.diagram.copypaste.copy()
    built-in function, 19
gaphor.diagram.copypaste.deserialize()
    built-in function, 20
gaphor.diagram.copypaste.paste()
    built-in function, 19
gaphor.diagram.copypaste.serialize()
    built-in function, 20
gaphor.diagram.inlineeditors.InlineEditor()
    built-in function, 19
get_connected() (*gaphor.diagram.connectors.BaseConnector method*), 18
get_connection() (*gaphor.diagram.connectors.BaseConnector method*), 18
group() (*gaphor.diagram.grouping.AbstractGroup method*), 19

H

handle() (*gaphor.core.eventmanager.EventManager method*), 15

L

lookup_diagram_item() (*gaphor.abc.ModelingLanguage method*), 17
lookup_element() (*gaphor.abc.ModelingLanguage method*), 17

M

ModelingLanguage (*class in gaphor.abc*), 17

N

name() (*gaphor.abc.ModelingLanguage property*), 17

O

open() (*gaphor.ui.abc.UIComponent method*), 12

P

PropertyPageBase (*class in gaphor.diagram.propertypages*), 19

S

Service (*class in gaphor.abc*), 12
shutdown() (*gaphor.abc.Service method*), 12

shutdown() (*gaphor.core.eventmanager.EventManager method*), 15
shutdown() (*gaphor.ui.abc.UIComponent method*), 12
subscribe() (*gaphor.core.eventmanager.EventManager method*), 15

T

toolbox_definition() (*gaphor.abc.ModelingLanguage property*), 17

U

UIComponent (*class in gaphor.ui.abc*), 12
ungroup() (*gaphor.diagram.grouping.AbstractGroup method*), 19
unsubscribe() (*gaphor.core.eventmanager.EventManager method*), 15