Gaphor Documentation

Release 1.0.0rc1

Arjan J. Molenaar

Mar 22, 2019

Contents

Some highlights of the documentation:

- A *manual*. It outlines some of the ideas in and behind Gaphor.
- The *Tech section* contains some interesting articles about the technology that drives Gaphor and Gaphas, Gaphor's canvas widget.

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

CHAPTER 1

The Gaphor manual

This section of the site is dedicated to some user documentation. Although most users will have previous experience with Gaphor, it is evident that some aspects of the application need a little more explanation.

1.1 Introduction

Welcome to the Gaphor!

Gaphor is a UML 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 designed around the following principles:

- Simplicity The application should be easy to use. Only some basic knowledge of UML is required.
- Consistency UML is a graphical modeling language, so all modeling is done in a diagram[1] .
- Workability The application should not bother the user every time they do something non-UML-ish.

This manual serves as a reference for all Gaphor has to offer. So, you may read it from start to finish, or jump to a section that interests you.

1.2 Working with Gaphor

Once Gaphor is launched, it provides you an almost empty user interface. A new model is already created and the diagram is opened.

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

- Navigation
- Main Canvas Diagram

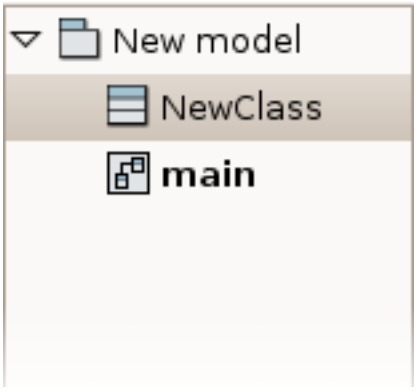
[1] We take this quite literally: even stereotypes are modeled in diagrams.

- Diagram Element Toolbox
- Properties pane

Each section has its own specific function.

1.2.1 Navigation

The navigation section of the interface displays a hierarchical view of your model. Every model element you create will be inserted into the navigation section. 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 on the left, you will see that there are three elements in the navigation view. 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 *NewClass* element is a class and the *main* element is a diagram.

In the navigation view, you can also right-click the model elements to get a context menu. This context menu allows you to delete model elements, and to refresh the navigation view.

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

1.2.2 Diagram section

The diagram section takes up the most space. Multiple diagrams can be opened at once: they are shown in tabs. Tabs can be closed from the file menu (Close) and by pressing *Ctrl+w*.

Most elements have *hot zones*, shown as gray rectangles that are only visible when the item is selected. Double clicking on those rectangles allows you to directly edit the item. E.g. change its name or change the name of an association end.

1.2.3 Toolbox

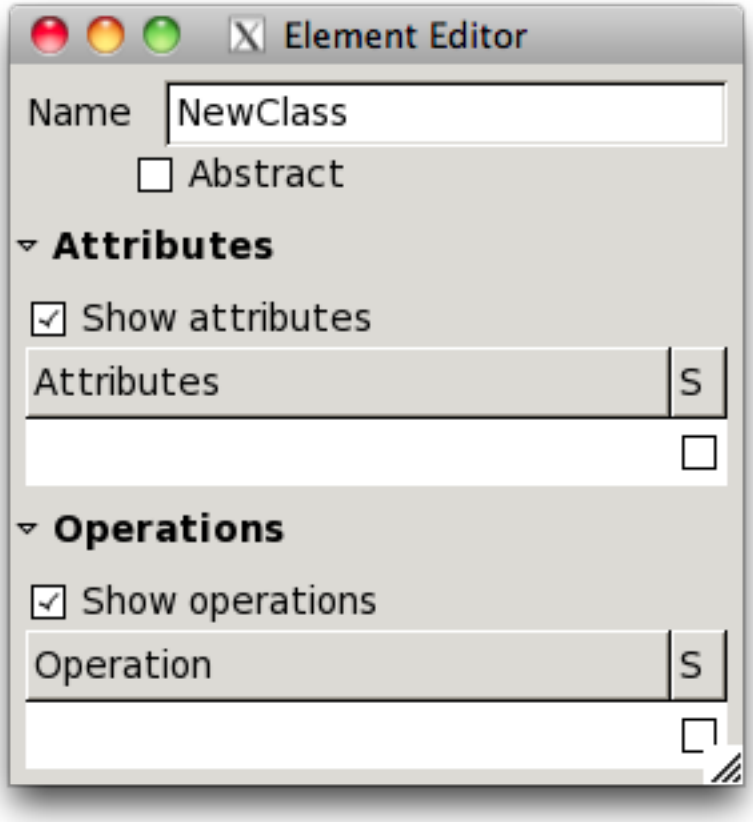
The toolbox is mainly used to add new diagram items to a diagram. Select the element you want to add by clicking on it. When you click on the diagram, the selected element is created. The arrow is selected again, so the element can be manipulated.

Tools can be selected by simply clicking on them. By default the pointer tool is selected after every item placement. This can be changed by disabling the Diagram > Reset tool option. Tools can also be selected by a keyboard shortcut.

The actual character is displayed as part of the tooltip. Finally it is also possible to drag elements on the canvas from the toolbox.

1.2.4 Element Editor

The Element editor is not really part of the main screen, it's a utility window that shows all characteristics of the selected element. Things like name, attributes and stereotypes. It can be opened with *Ctrl+e*.

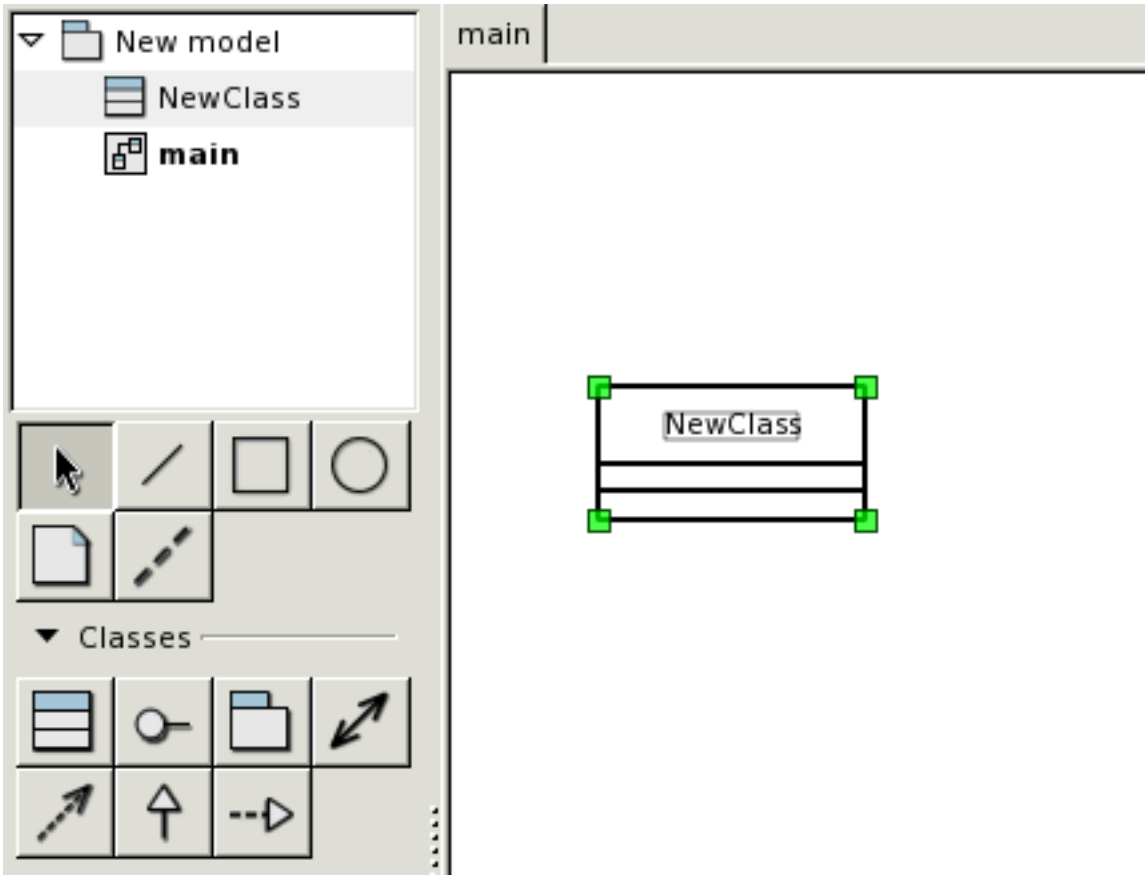


The properties shown depend on the element that is selected.

1.3 Creating models

Once Gaphor is started a new empty model is automatically created. The *main* diagram is already open in the *Diagram section*.

Select an element you want to place (e.g. 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 *Navigation*). The selected tool will reset itself to the Pointer tool if the option ''Diagram -> Reset tool" is selected.

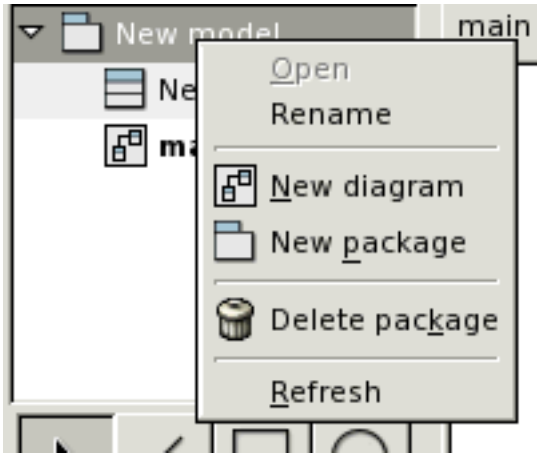


It's simple to add elements to a diagram.

Some elements are not directly visible. The section in the toolbox is collapsed and needs to be clicked first to reveal its contents.

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 restrict the user.

1.4 Creating new diagrams



To create a new diagram, use the Navigation. Select an element that can contain a diagram (a Package or Profile) and right-click[1]. Select *New diagram* and a new diagram is created.

1.5 Copy and paste

Items in a diagram can be copied and pasted (in the same diagram or another). A paste will create new items in the diagrams, the items they represent (e.g. the Class that's shown in the Navigation) is *not* copied (call it a shallow copy if you like).

1.6 Drag and drop

Adding an existing element to a diagram is simple: drag the element from the Navigation section onto a diagram. Not all elements that show up in the Navigation can be added: Diagrams and attribute/operations of a Class can not be added.

Elements can also be dragged within the Navigation section. This way classes and packages can be rearranged for example.

1.7 Writing plugins and services

There is little difference in writing a plugin or a service.

1.7.1 Accessing model related data

The datamodel classes are located in the *gaphor.UML* module. Data objects can be accessed through the ElementFactory. This is a special class for creating and managing data objects. Items can be queried using this element factory. It's registered in the application as *element_factory*. When writing a service or plugin the element factory can be injected into the service like this:

```
class MyThing(object):

    element_factory = inject('element_factory')

    def do_something(self):
        items = element_factory.select()
```

Note: In the console window services can be retrieved by using the *service()* function. E.g.:

```
ef = service('element_factory')
```

1.7.2 Querying the data model

Two methods are used for querying:

- *select(query=None)* -> returns an iterator

[1] Command-Click for MacOS X users running Gaphor in X11.

- *lselect(query=None)* -> returns a list

query is a lambda function with the element as parameter. E.g.:

```
element_factory.select(lambda e: e.isKindOf(UML.Class))
```

will fetch all Class instances from the element factory.

1.7.3 Traversing data instances

Once some classes are obtained It's common to traverse through a few related objects in order to obtain the required information. For example: to iterate through all parameters related to class' operations, one can write:

```
for o in classes.ownedOperation:
    for p in o.ownedParameter:
        do_something(p)
```

Using the `[:]` operator items can be traversed more easily[1]:

```
for o in classes.ownedOperation[:].ownedParameter:
    do_something(p)
```

It's also possible to provide a query as part of the selection:

```
for o in classes.ownedOperation['it.returnParameter'].ownedParameter:
    do_something(p)
```

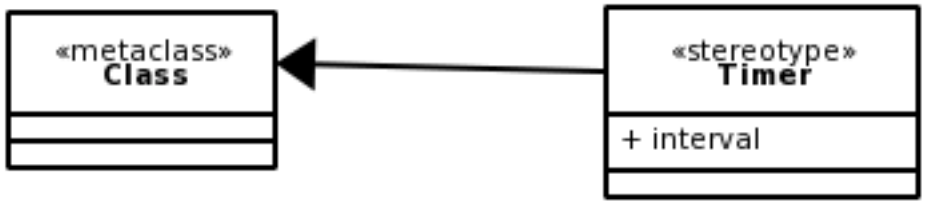
The variable `it` in the query refers to the evaluated object (in this case all operations with a return parameter are taken into account).

1.8 Extending models

The UML method to extend UML (basic) components with a special meaning is by using stereotypes. A stereotype defines a special usage of a model element. For example: a class that's used as a controller can be assigned a stereotype:

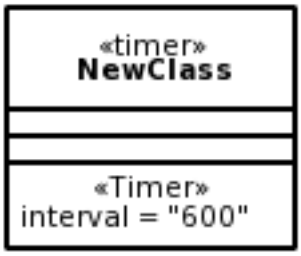
```
«controller»
```

Creating a stereotype starts by the creation of a Profile normally. Although stereotypes can be created in every package, it's a good habit to use Profiles for that. Next a Metaclass has to be created. The metaclass will tell the stereotype on which kind of elements it is applicable. A Stereotype can be connected to the Metaclass by means of an Extension relationship.

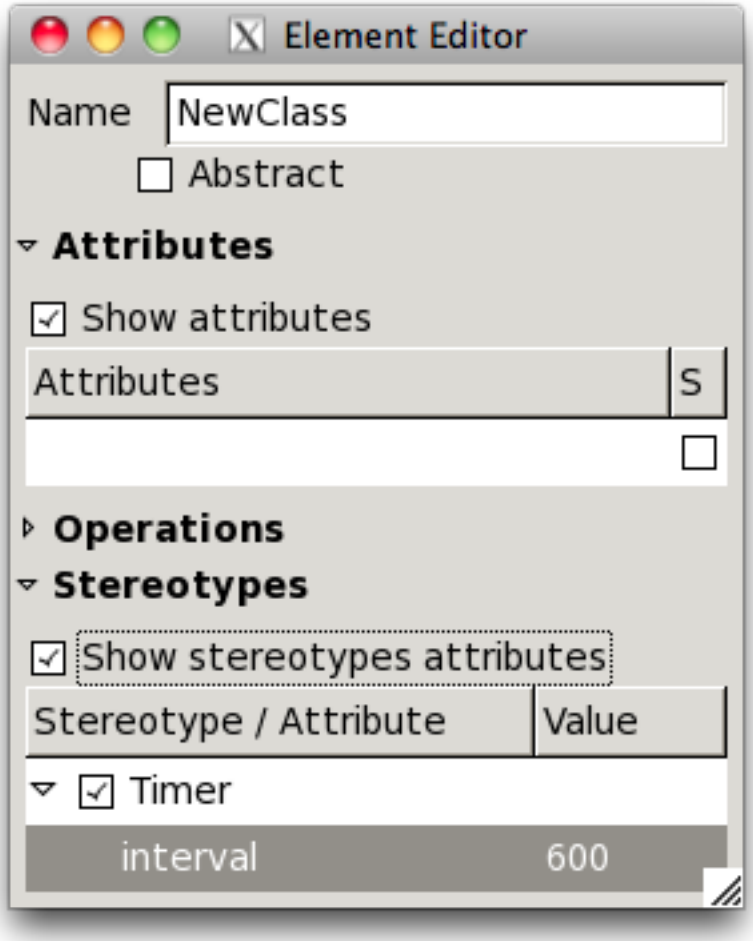


[1] See http://github.com/amolenaar/gaphor/blob/master/gaphor/misc/listmixins.py

Stereotypes can be applied to basically all elements in a model[1].



Stereotypes can contain attributes, as shown in the diagram above. Those attributes can be filled in the Element Editor. This allows for enormous flexibility. In most cases, especially if some sort of program logic has to be generated from the models, it is very handy to define special behaviours to classes and other elements by means of stereotypes.



Running Gaphor on different platforms:

[1] If for some reason the stereotype doesn't show in the diagram, raise a ticket ;)

CHAPTER 2

Gaphor on Windows

CHAPTER 3

Gaphor on Linux

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

- http://cvs.pld-linux.org/cgi-bin/cvsweb/SPECS/python-gaphas.spec
- http://cvs.pld-linux.org/cgi-bin/cvsweb/SPECS/gaphor.spec

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

3.1 Dependencies

Gaphor depends on Zope libraries:

- component
- interface

and:

- gaphas (of course :)
- pygtk

Above Zope modules require

- deferredimport
- deprecation
- event
- proxy
- testing

CHAPTER 4

Gaphor on macOS

CHAPTER 5

Custom Python Installation Location

This page is based on *custom installation locations <http://peak.telecommunity.com/DevCenter/EasyInstall#custom-installation-locations>*, from the PEAK site.

5.1 Unix/Linux

1. Create *$HOME/.pydistutils.cfg*:

```
[install]
install_lib = ~/.py-site-packages
install_scripts = ~/bin
```

2. Create (or extend) the *PYTHONPATH* environment variable (for (ba)sh):

```
export PYTHONPATH=~/.py-site-packages
```

3. Run *setup.py* script to fetch and install dependencies

```
python setup.py install
```

Prefix *~/.py-site-packages* can be changed to something more suitable for your setup.

> **Note for Linux users:** Make sure you have the *python-dev* package installed for your Python version, as some code needs to be compiled (those are packages Gaphor depends on, not Gaphor itself).
>
> **Note for Ubuntu Linux users:** Make sure you have the *build-essential* package installed. This package installs header files and what more, required to compile the C-extensions of *zope.interface*.

5.2 Windows

> **NOTE:** For Windows users it may be simpler to just forget about custom installation locations. Just follow the instructions on [wiki:Win32] and you should be set.

The Windows installation is almost the same as for Unix.

Replace *yourname* with your login name.

1. Distutils requires a HOME variable where it can find the configuration file. So in your Control Panel -> System -> Advanced -> Environment Variables add the following:

```
HOME=C:\Documents and Settings\yourname\Home
```

2. Create a directory *C:Documents and SettingsyournameHome*. Also create *%HOME%py-site-packages*.
3. Eventually add the Python directory to your *PATH* (default is *C:Python26*)
4. Create (or extend) *PYTHONPATH* variable:

```
PYTHONPATH=%HOME%\py-site-packages
```

5. Create a file *%HOME%pydistutils.cfg* with the following content:

```
[install]
install_lib=$home\py-site-packages
install_scripts=$home\bin

[build]
compiler=mingw32
```

Now you should be able to do *python setup.py install* from the command line.

If you are a developer you should definitely install MinGW from http://mingw.org and add MinGW's *bin* directory to your path.

For a good Subversion client for Windows have a look at *TortoiseSVN <http://tortoisesvn.tigris.org/>*.

5.3 Mac OS X

Mac OS X is quite simple: place the following in your *$HOME/.pydistutils.cfg*:

```
[install]
install_lib = ~/Library/Python/$py_version_short/site-packages
install_scripts = ~/bin
```

CHAPTER 6

Tech section

6.1 Gaphor's Framework

Gaphor is built in a light, service oriented fashion. The application is split in a series of services, such as a file manager, undo manager and GUI manager. Those services are loaded based on entry points defined in Python Eggs (see *Service oriented design*).

Objects communicate with each other through events. Whenever something of importance happens (e.g. an attribute of a model element changes) an event is sent. Whoever is interested (a diagram item for example) receive notification once it has registered an event handler for that event type. Events are emitted though a central broken (zope.component in our case), so you do not have to register on every individual element that can send an event you're interested in (so the diagram item should check if the element that sent the event is actually the event the item is representing).

Gaphor is transactional. Transactions work simply by sending an event when a transaction starts and sending another when a transaction ends. E.g. undo management is transactional.

It all starts with an Application. Only one Application instance is permitted. The Application will look for services defined as *gaphor.services*. Those services are loaded and initialized.

The most notable services are:

gui_manager The GUI manager is one of the major services that have to be loaded. The GUI manager is responsible for loading the main window and displaying it on the screen.

This by itself doesn't do a thing though. The Action manager (*action_manager*) is required to maintain all actions users can perform. Actions will be shown as menu entries for example.

file_manager Loading and saving a model is done through this service.

element_factory The *data model* itself is maintained in the element factory. This service is used to create model elements and can be used to lookup elements or query for a set of elements.

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.

element_dispatcher Although Gaphor makes use of a central dispatch engine, this solution is not efficient when it comes to dispatching events of UML model elements. For this purpose the *element_dispatcher* can help out. It maintains a path of elements reaching from the root (e.g. from a diagram item) to the element of interest and will only signal in case this element changes. This makes complex dispatching very efficient.

6.2 UML datamodel

Gaphor uses the UML metamodel specs as guidelines for its data storage. In fact, the datamodel is generated for a model file.

The model is built using smart properties (descriptors). Those descriptors fire events when they're changed. This allows the rest of the application (visuals, undo system) to update their state accordingly. The events are send using Zope (3)'s signalling mechanism, called Handlers.

6.2.1 Model details

Pay attention to the following changes/additions with respect to the official model:

- Additions to the model have been put in the package AuxilaryConstructs.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 specs (2.2) are not clear on that subject.
- The Slot.value reference is singular.

Note: 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.

6.3 Stereotypes

Stereotypes are quite another story. In order to create a stereotype one should create a Profile. Within this profile a Diagram can be created. This diagram should accept only items that are useful within a profile:

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

and of course the usual: Comment, Association, Generalization and Dependency.

Thoughts:

- Profiles are reusable and its common to share them with different models.
- Stereotypes can only be owned by Profiles, not by (normal) Packages.
- We have to do a lookup if a MetaClass is actually part of the model.
- a stereotype can contain an image, that can be used instead of its name
- Profiles should be saved with the model too. And it should be possible to "update" a profile within a model.

Maybe it would be nice to create Stereotypes without creating the diagrams. Via a dialog once can select which class (Operation, Class, etc.) is stereotyped, which extra constraints apply and/or if you inherit from an already existing stereotype. This way it's easy to save your stereotypes apart from the model (and possibly in the model too) so they can be reused in other models.

I could create a special diagram window too that can be used to create profiles. Profiles should be added to packages within the model.

This window should contain: 1. Name of the stereotype 2. Metaclass it applies to (Class, Operation, etc.) 3. If it is a subclass of an already existing metaclass 4. Constraints 5. Description 6. The profile it belongs to.

When a stereotype is used, an instance is created of the Stereotype (meta)class. This is not really possible for our application. Gaphor should figure out another way to do this. Should check XMI and see how they do it (maybe a property is enough).

It looks like the stereotypes are more a concept than something that is implementable in an application. The point is that the stereotypes you define (instances of Stereotype) 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.

6.4 Description of Gaphors data model

Gaphor is an UML tool. In order to keep as close as possible to the UML specification the data model is based on the UML Metamodel. Since the OMG has an XMI (XML) specification of the metamodel, the easiest way to do that is to generate the code directly from the model. Doing this raises two issues:

1. There are more attributes defined in the data model than we will use.
2. How do we check if the model is consistent?

The first point is not such a problem: attributes we don't use don't consume memory.

There are no consistency rules in the XML definition, we have to get them from the UML syntax description. It is probably best to create a special consistency module that checks the model and reports errors.

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

The data model is described in Python. 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 parents dictionary). If it does, the value is assigned, if it doesn't an exception is raised.

6.4.1 Bidirectional references

But how, you might wonder, do you handle bidirectional references (object one references object two and vice versa)? Well, this is basically the same as the uni-directional reference. Only now we need to add some extra information to the dictionary at class level. We just define an extra field that gives us the name of the opposite reference and voila, we can create bi-directional references. You should check out the code in `gaphor/UML/element.py` for more details.

6.4.2 Implementation

This will allow 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]'
```

All this wisdom is defined in the data-models base class: `Element`. The datamodel itself code is generated.

6.4.3 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.

6.4.4 UML.Element

6.5 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 it all starts with a button release event.

With "item" I refer to objects in a diagram (graphical), with "element" I refer to semantic (model) objects.

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)

6.6 Actions, menu items, toolbars and toolboxes

The GTK+ structure is like this:

```
 UIManager
     |
ActionGroup
     |
  Action
```

The do_activate signal is emitted when an action is activated.

Where it should lead:

- **Actions should be coded closer to the object they are working on**
  - main actions - not dependent on GUI components (load/save/new/quit)
  - main actions dependent on GUI component (sub-windows, console)
  - Item actions, act either on a diagram, the selected or focused item or no item.
  - diagram actions (zoom, grid) work on active diagram (tab)
  - menus and actions for diagram items through adapters
- Actions should behave more like adapters. E.g. when a popup menu is created for an Association item, the menu actions should present themselves in the context of that menu item (with toggles set right).
  - Could be registered as adapters with a name.
- Each window has its own action group (every item with a menu?)
- One toplevel UIManager per window or one per application/gui_manager?
- Normal actions can be modeled as functions. If an action is sensitive or visible depends on the state in the action. Hence we require the update() method.
- create services for for "dynamic" menus (e.g. recent files/stereotypes)

6.6.1 Solution for simple actions

For an action to actually be useful a piece of menu xml is needed.

Hence an interface IActionProvider has to be defined:

```
class IActionProvider(interface.Interface):
    menu_xml = interface.Attribute("The menu XML")
    action_group = interface.Attribute("The accompanying ActionGroup")
```

Support for actions can be arranged by decorating actions with an *@action* decorator and let the class create an ActionGroup using some actionGroup factory function (no inheritance needed here).

Note that ActionGroup is a GTK class and should technically only be used in the gaphor.ui package.

Autonom controllers can be defined, which provide a piece of functionality. They can register handlers in order to update their state.

Maybe it's nice to configure those through the egg-info system. I suppose gaphor.service will serve well (as they need to be initialized anyway)

- also inherit IActionProvider from IService?

```
[gaphor.services]
xx = gaphor.actions.whatever:SomeActionProviderClass
```

6.6.2 Solution for context sensitive menus

Context sensitive menus, such as popup menus, should be generated and switched on and off on demand.

Technically they should be enabled through services/action-providers.

It becomes even tougher when popup menus should act on parts of a diagram item (such as association ends). This should be avoided. It may be a good idea to provide such functionality through a special layer on the canvas, by means of some easy clickable buttons around the "hot spot" (we already have something like that for text around association ends).

Scheme:

1. User selects an item and presses the right mouse button: a popup menu should be displayed.
2. Find the actual item (this may be a composite item of the element drawn). Use an IItemPicker adapter for that (if no such interface is available, use the item itself).
3. Find a IActionProvider adapters for the selected (focused) item.
4. Update the popup menu context (actions) for the selected item.

6.7 Gaphor Diagram Items

The diagram items (or in short *items*) represent UML metamodel on a diagram. The following sections present the basic items.

6.7.1 DiagramItem

Basic diagram item supporting item style, text elements and stereotypes.

6.7.2 ElementItem

Rectangular canvas item.

6.7.3 NamedItem

NamedElement (UML metamodel) representation using rectangular canvas item.

6.7.4 CompartmentItem

An item with compartments (i.e. Class or State)

6.7.5 ClassifierItem

Classifer (UML metamodel) representation.

6.7.6 DiagramLine

Line canvas item.

6.7.7 NamedLine

NamedElement (UML metamodel) representation using line canvas item.

6.7.8 FeatureItem

Diagram representation of UML metamodel classes like property, operation, stereotype attribute, etc.

6.8 Services

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

Since Gaphor already uses the *zope.component* adapters, it seems like a good choice to use *zope.interface* too as starting point for services.

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

```
init(application)
```

where *application* is the Application object for Gaphor (which is a service itself and therefore implements *IService* too.

Each service is allowed to define its own interface, as long as *IService* is implemented too.

Services should be defined as *entry_points* in the Egg info file.

Typically a service does some work in the background.

6.8.1 Example: ElementFactory

A nice example is the ElementFactory. Currently it is tightly bound to the gaphor.UML module. A default factory is created in __init__.py.

It depends on the undo_manager. However, on important events, events are emitted. (That is when an element is created/destroyed).

What you want to do is create an event handler for ElementFactory that stores the add/remove signals in the undo system.

The same goes for UML.Elements. Those classes (or more specific the properties) send notifications every time their state changes.

But.. where to put such information?

6.9 Plugins

Currently a plugin is defined by an XML file. This will change as plugins should be distributable as Eggs too. A plugin will contain user interface information along with its service definition.

See also:

Writing plugins

6.10 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: a service is responsible for loading/saving models, another for the menu structure, yet another service handles the undo system.

Service are defined as Egg entry points. With entry points applications can register functionality for specific purposes. Entry points are grouped in so called *entry point groups*. For example the *console_scripts* entry point group is used to start an application from the command line.

Entry points, as well as all major components in Gaphor, are defined around Zope interfaces. An interface defines specific methods and attributes that should be implemented by the class.

6.10.1 Entry point groups

Gaphor defines two entry point groups:

- gaphor.services
- gaphor.uicomponents

Services are used to add functionality to the application. Plug-ins should also be defined as services. E.g.:

```
entry_points = {
    'gaphor.services': [
        'xmi_export = gaphor.plugins.xmiexport:XMIExport',
    ],
},
```

There is a thin line between a service and a plug-in. A service typically performs some basic need for the applications (such as the element factory or the undo mechanism). A plug-in is more of an add-on. For example a plug-in can depend on external libraries or provide a cross-over function between two applications.

6.10.2 Interfaces

Each service (and plug-in) should implement the *gaphor.interfaces.IService* interface:

UI components is another piece of cake. The `gaphor.uicomponents` entry point group is used to define windows and user interface functionality. A UI component should implement the *gaphor.ui.interfaces.IUIComponent* interface:

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.interfaces.IActionProvider` interface:

Example plugin

A small example is provided by means of the *Hello world plugin*. Take a look at the files at Github.

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.

Also notice that, since the package is defined as `gaphor.plugins`, which is also a package in the default gaphor distribution, each package level contains a special statement in its `__init__.py` that tells setuptools its namespace declaration::

```
__import__('pkg_resources').declare_namespace(__name__)
```

Here is a stripped version of the hello world class:

```
from zope.interface import implementer

from gaphor.interfaces import IService, IActionProvider
from gaphor.core import _, inject, action, build_action_group


@implementer(IService, IActionProvider)              # 1.
class HelloWorldPlugin(object):

    gui_manager = inject('gui_manager')              # 2.

    menu_xml = """                                   # 3.
      <ui>
        <menubar name="mainwindow">
          <menu action="help">
            <menuitem action="helloworld" />
          </menu>
        </menubar>
      </ui>
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self):
        self.action_group = build_action_group(self)  # 4.

    def init(self, app):                               # 5.
        self._app = app

    def shutdown(self):                                # 6.
        pass

    @action(name='helloworld',                         # 7.
            label=_('Hello world'),
            tooltip=_('Every application ...'))
    def helloworld_action(self):
        main_window = self.gui_manager.main_window     # 8.
        pass # gtk code left out
```

1. As stated before, a plugin should implement the `IService` interface. It also implements `IActionProvider`, saying it has some actions to be performed by the user.
2. The plugin depends on the `gui_manager` service. The `gui_manager` can be referenced as a normal attribute. The first time it's accessed the dependency to the `gui_manager` is resolved.
3. As part of the `IActionProvider` interface an attribute `menu_xml` should be defined that contains some menu xml (see http://developer.gnome.org/doc/API/2.0/gtk/GtkUIManager.html#XML-UI).
4. `IActionProvider` also requires an `action_group` attribute (containing a `gtk.ActionGroup`). This action group is created on instantiation. The actions itself are defined with an `action` decorator (see 7).
5. Each `IService` has an `init(app)` method...
6. ... and a `shutdown()` method. Those can be used to create and detach event handlers for example.
7. The action that can be invoked. The action is defined and will be picked up by `build_action_group()` (see 4.)
8. The main window is retrieved from the `gui_manager`. At this point the "inject'ed" gui-manager reference is resolved.

6.11 Saving and loading Gaphor diagrams

Everything interesting is an ancestor of the Gaphor root tag.

The idea is to keep the file format as simple and extensible as possible: UML elements (including Diagram) are at top level, 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>
```

(continues on next page)

(continued from previous page)

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

6.12 UndoManager

Fine grained undo: undo specific properties.

Add undo-operation, e.g. Item.request_update() should be executed as part of an undo action. Such actions are normally called after the properties have been set right though. This is not a problem for idle tasks, but for directly executed tasks it is.

Should only need to save the originals, values calculated, e.g. during an update shouldn't have to be calculated -> use undo-operations to trigger updates.

To update:

Handle: x, y (solvable) connectable (attr) visible (attr) movable (attr) connection status (solver?)

Item: matrix canvas is managed from Canvas

Element: handles width, height min_width, min_height (solvable?)

Line: handles line_width fuzzyness (attr) orthogonal (boolean) horizontal (boolean)

Canvas:

tree: add() remove()

request_update() (should be performed as part of undo action when called)

Solver (?): add_constraint() remove_constraint() Variable state

In Gaphor, connecting two diagram items is considered an atomic task, performed by a IConnect adapter. This operation results in a set of primitive tasks (properties set and a constraint created).

For methods, it should be possible to create a decorator (@reversible) that schedules a method with the same signature as the calling operation, but with the inverse effect (e.g. the gaphas.tree module):

```
class Tree(object):

  @reversable(lambda s, n, p: s.remove(n))
  def add(self, node, parent=None):
      ... add

  @reversable(add, self='self', node='node', parent='self.get_parent(node)')
  def remove(self, node):
      ... remove
```

Okay, so the second case is tougher...

So what we did: Add a StateManager to gaphas. All changes are sent to the statemanager. Gaphor should implement its own state manager.

- all state changes can easily be recorded
- fix it in one place
- reusable throughout Gaphas and subtypes.

6.12.1 Transactions

Gaphor's Undo manager works transactionally. Typically, methods can be decorated with @transactional and undo data is stored in the current transaction. A new tx is created when none exists.

Although undo functionality is at the core of Gaphor (diagram items and model elements have been adapted to provide proper undo information), the UndoManager itself is just a service.

Transaction support though is a real core functionality. By letting elements and items emit event notifications on important changed other (yet to be defined) services can take benefit of those events. The UML module already works this way. Gaphas (the Gaphor canvas) also emits state changes.

When state changes happen in model elements and diagram items an event is emitted. Those events are handled by special handlers that produce "reverse-events". Reverse events are functions that perform exactly the opposite operation. Those functions are stored in a list (which technically is the Transaction). When an undo request is done the list is executed in LIFO order.

To start a transaction:

1. A Transaction object has been created.
2. This results in the emission of a TransactionBegin event.
3. The TransactionBegin event is a trigger for the UndoManager to start listening for IUndoEvent actions.

Now, that should be done when a model element or diagram item sends a state change:

1. The event is handled by the "reverse-handler"

2. Reverse handler generates a IUndoEvent signal
3. The signal is received and stored as part of the undo-transaction.

(Maybe step 2 and 3 can be merged, since only one function is not of any interest to the rest of the application - creates nasty dependencies)

If nested transaction are created a simple counter is incremented.

When the topmost Transaction is committed:

1. A TransactionCommit event is emitted
2. This triggers the UndoManager to close and store the transaction.

When a transaction is rolled back:

1. The main transaction is marked for rollback
2. When the toplevel tx is rolled back or committed a TransactionRollback event is emitted
3. This triggers the UndoManager to play back all recorded actions and stop listening.

6.12.2 References

A Framework for Undoing Actions in Collaborative Systems http://www.eecs.umich.edu/~aprakash/papers/undo-tochi94.pdf

Undoing Actions in Collaborative Work: Framework and Experience https://www.eecs.umich.edu/techreports/cse/94/CSE-TR-196-94.pdf

Implementing a Selective Undo Framework in Python http://www.python.org/workshops/1997-10/proceedings/zukowski.html

6.13 Gaphor XML format

This format is meant to be a shorter and more obvious version of Gaphor's file format. The current format makes it pretty hard to do some decent XSLT parsing. In the current file format one has to compare the @name attribute with the model element name one wishes.

Since the data model is generated from a Gaphor (0.2) model it would be a piece of cake to generate a DTD too.

These are the things that should be distinguished: - model elements - associations with other model elements (referenced by ID):

- 0..1 relations
- 0..* relations
- attributes (always have a multiplicity of 0..1)
- diagrams
  - one canvas
  - several canvas items
- derived attributes and associations are not saved of course.

Model elements should have their class name as tag name, e.g.:

```
<Class id="DCE:xxx.xxx...">
  ...
</Class>
<Package id="DCE:xxx...">
  ...
</Package>
```

Associations are in two flavors: single and multiple:

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

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

```
<Class id="DCE:xxx.xxx...">
  <name>
    <![CDATA[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="DCE:..."/>
      </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 has export filters for SVG graphics, so diagrams can be exported in a independant way.

6.14 External links

- You should definitely check out http://www.agilemodeling.com.
  - The UML diagrams (although Gaphor does not see it that black-and-white).
  - http://www.agilemodeling.com/essays/

- The official UML metamodel specification. This ‘’is” our data model.