Gaphas

Release 0.5

Jul 21, 2018

Contents

Gaphas is Gaphor's diagram drawing widget.

Gaphas has been built with some extensibility in mind. It can be used for many drawing purposes, including vector drawing applications, diagram drawing tools and we even have a geographical map demo in our repository.

The basic idea is:

- Items (canvas items) can be added to a Canvas.
- A Canvas can be visualized by one or more Views.
- The canvas maintains the tree structure (parent-child relationships between items).
- A constraint solver is used to maintain item constraints and inter-item constraints.
- The item (and user) should not be bothered with things like bounding-box calculations.
- Very modular: e.g. handle support could be swapped in and swapped out.
- Rendering using Cairo. This implies the diagrams can be exported in a number of formats, including PNG and SVG.

Gaphas is released under the terms of the GNU Library General Public License (LGPL).

Gaphas has been build using *setuptools* and can be installed as Python Egg.

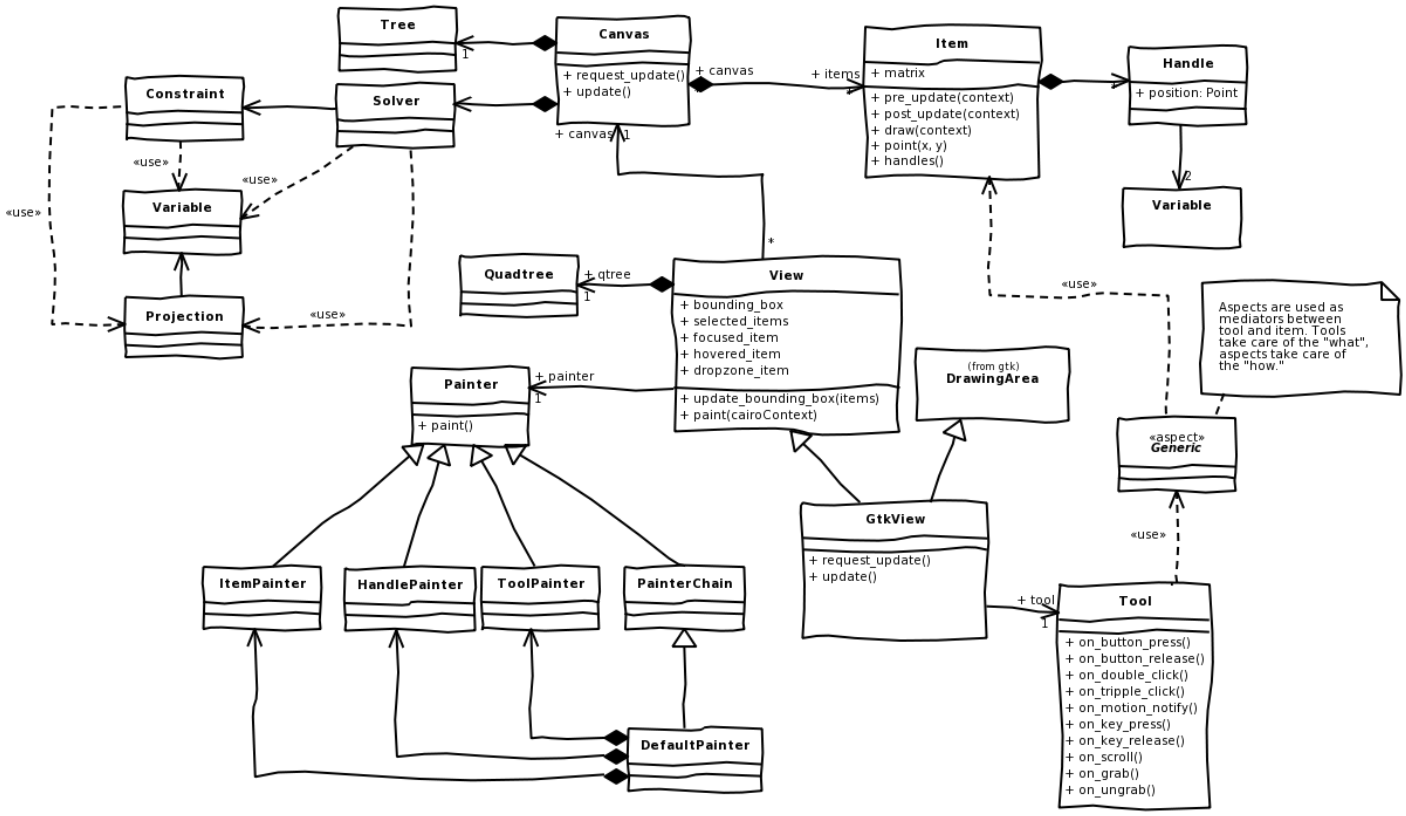
- Git repository: http://gitgub.com/amolenaar/gaphas
- Python Package index (PyPI): http://cheeseshop.python.org/pypi/gaphas

CHAPTER 1

Table of Contents

1.1 Class diagram

This class diagram describes the basic layout of Gaphas.



One-oh-one:

Canvas The main canvas class (container for Items)

Items Objects placed on a Canvas. Items can draw themselves, but not act on user events

Solver A constraint solver. Nice to have when you want to connect items together in a generic way.

View Base class that renders content (*paint()*). The view is responsible for the calculation of bounding boxes. This information is stored in a quadtree data structure for fast access.

GTK+ View A view to be used in GTK+ applications. This view class is interactive. Interaction with users is handled by Tools.

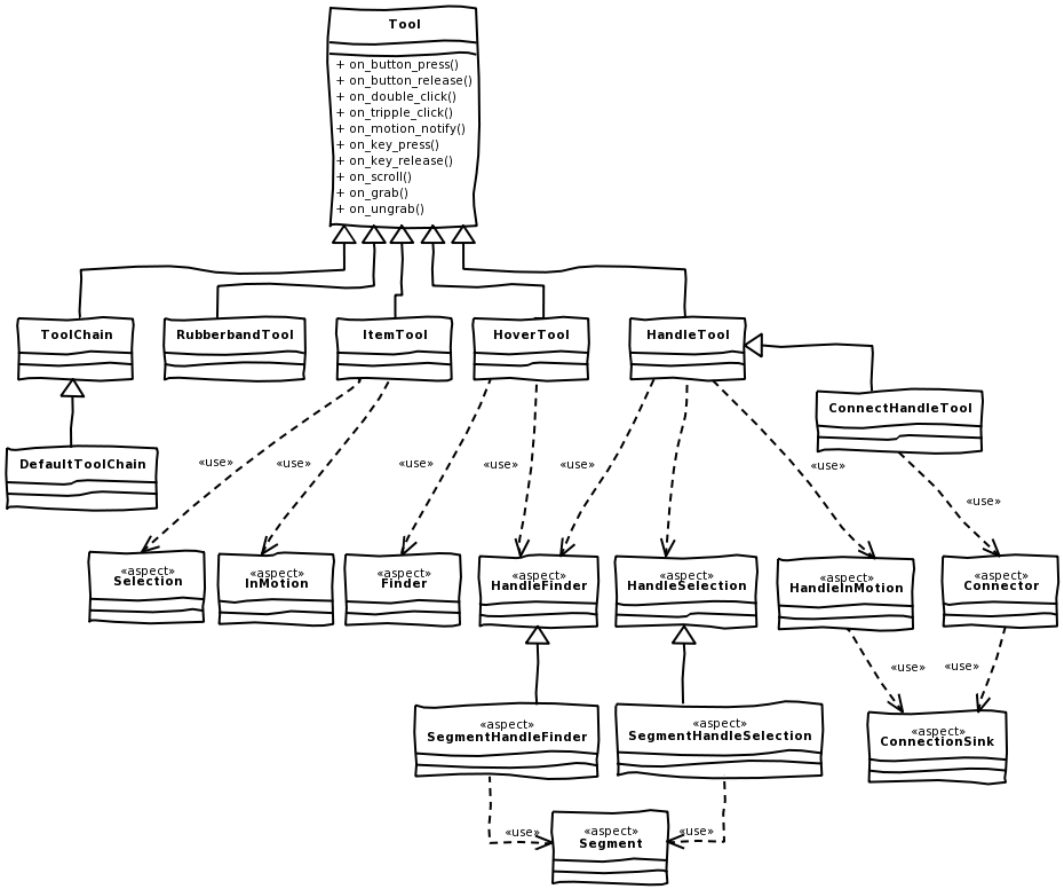
Painters Painters are the workers when it comes to painting items.

Tools Tools are used to handle user events (such as mouse movement and button presses).

Aspects Tools do not modify the items directly. They use aspects (not the AOP kind) as intermediate step. Since aspects are defined as generic functions, the behaviour for each diagram item type can be slightly different.

1.1.1 Tools

Several tools are used to make the overall user experience complete.



1.2 Interacting with diagrams

Tools are used to handle user actions, like moving a mouse pointer over the screen and clicking on items in the canvas.

Tools are registered on the `View`. They have some internal state (e.g. when and where a mouse button was pressed). Therefore tools can not be reused by different views[1].

For a certain action to happen multiple user events are used. For example a click is a combination of a button press and button release event (only talking mouse clicks now). In most cases also some movement is done. A sequence of a button press, some movement and a button release is treated as one transaction. Once a button is pressed the tool registers itself as the tool that will deal with all subsequent events (a `grab`).

Several events can happen based on user events. E.g.:

- item is hovered over (motion)
- item is hovered over while another item is being moved (`press`, `motion`)
- item is hovered over while dragging something else (DnD; `press`, `move`)
- grabbed (button press on item)
- handle is grabbed (button press on handle)
- center of line segment is grabbed (will cause segment to split; button press on line)
- ungrabbed (button release)
- move (item is moved -> hover + grabbed)
- key is pressed
- key is released
- modifier is pressed (e.g. may cause mouse pointer to change, giving a hit about what a grab event will do.

There is a lot of behaviour possible and it can depend on the kind of diagrams that are created what has to be done.

To organize the event sequences and keep some order in what the user is doing Tools are used. Tools define what has to happen (find a handle nearly the mouse cursor, move an item).

Gaphas contains a set of default tools. Each tool is ment to deal with a special part of the view. A list of responsibilities is also defined here:

HoverTool First thing a user wants to know is if the mouse cursor is over an item. The `HoverTool` makes that explicit. - Find a handle or item, if found, mark it as the `hovered_item`

HandleTool and ConnectHandleTool Handles are an important means to interact with items. They are used to resize element, move lines and (in case of `ConnectHandleTool`) establish connections between items. Handles are rendered on top of items so it makes sense to deal with them before you deal with items.

- On click: find a handle, if found become the grabbed tool and set focus on the selected item. Deselected current selection based on modifier keys.
- On motion: move the handle
- On release: release grab and release the handle

ItemTool Items are the elements that are actually providing any (visual) meaning to the diagram. ItemTool deals with moving them around. The tool makes sure the right subset of selected elements are moved (e.g. you don't want to move a nested item if its parent item is already moved, this gives funny visual effects)

- On click: find an item, if found become the grabbed tool and set the item as focused. Some extra behaviour regarding multiple select is also done here.
- On motion: move the selected items (only the ones that have no selected parent items)

[1] as opposed to versions < 0.5, where tools could be shared among multiple views.

- On release: release grab and release item

RubberBandTool If no handle or item is selected a rubberband selection is started.

PanTool and ZoomTool Handy tools for moving the canvas around and zooming in and out. Convenience functionality, basically.

TextEditTool An experimental tool for editing onscreen text.

All tools mentioned above are linked in a `ToolChain`. Only one tool can deal with a use event.

There is one more tool, that has not been mentioned yet:

PlacementTool A special tool to use for placing new items on the screen.

As said, tools define *what* has to happen, they don't say how. Take for example finding a handle: on a normal element (a box or something) that would mean find the handle in one of the corners. On a line, however, that may also mean a not-yet existing handle in the middle of a line segment (there is a functionality that splits the line segment).

The *how* is defined by so called aspects[2].

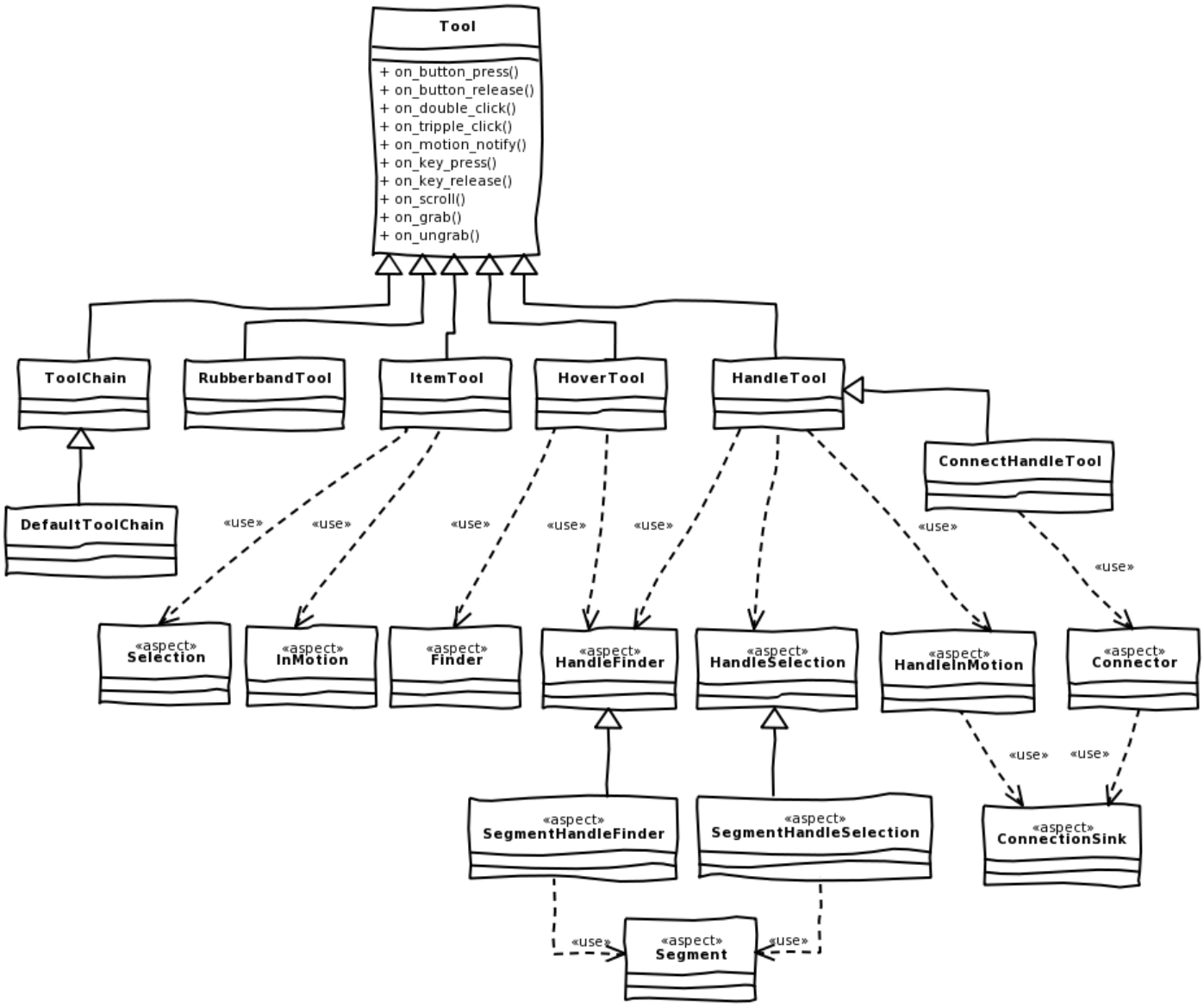
1.2.1 Separating the *What* from the *How*

The *what* is decided in a tool. Based on this the *how* logic can be applied to the item at hand. For example: if an item is clicked, it should be marked as the focused item. Same for dragging: if an item is dragged it should be updated based on the event information. It may even apply this to all other selected items.

The how logic depends actually on the item it is applied to. Lines have different behaviours than boxes for example. In Gaphas this has been resolved by defining a generic methods. To put it simple: a generic method is a factory that returns a specific method (or instance of a class, as we do in gaphas) based on its parameters.

The advantage is that more complex behaviour can be composed. Since the decision on what should happen is done in the tool, the aspect which is then used to work on the item ensures a certain behaviour is performed.

[2] not the AOP term. The term aspect is coming from a paper by Dick Riehe: The Tools and Materials metaphore <url...>.



The diagram above shows the relation between tools and their aspects. Note that tools that delegate their behaviour to aspects have more than one aspects. The reason is that there are different concerns involved in defining what the tools should do. Typically `ItemTool` will be selecting the actual item and takes care of moving it around as well. `HandleTool` does similar things for handles.

Big changes from Gaphas 0.4 tool include:

- Tools can contain state and should be used for one view only.
- Grabbing is done automatically for press-move-release event sequence.
- The _What_ is separated from the _How_, leaving less tools and less overhead (like finding the item under the mouse pointer).

1.3 Ports

Port is a part of an item, which defines connectable part of an item. This concept has been introduced Gaphas version 0.4.0 to make Gaphas connection system more flexible.

1.3.1 Port Types

There are two types of ports implemented in Gaphor (see *gaphas.connector* module).

First one is point port, as is often use in EDA applications, a handle connecting to such port is always kept at specific position, which equals to port's position.

Other port type is line port. A handle connecting to such port is kept on a line defined by start and end positions of line port.

Line port is used by items provided by *gaphas.item* module. *Element* item has one port defined at every edge of its rectangular shape (this is 4 ports). *Line* item has one port per line segment.

Different types of ports can be invented like circle port or area port, they should implement interface defined by *gaphas.connector.Port* class to fit into Gaphas' connection system.

1.3.2 Ports and Constraints (Covering Handles)

Diagram items can have internal constraints, which can be used to position item's ports within an item itself.

For example, *Element* item could create constraints to position ports over its edges of rectanglular area. The result is duplication of constraints as *Element* already constraints position of handles to keep them in a rectangle.

Therefore, when port covers handles, then it should reference handle positions.

For example, an item, which is a horizontal line could be implemented like:

```
class HorizontalLine(gaphas.item.Item):
    def __init__(self):
        super(HorizontalLine, self).__init__()

        self.start = Handle()
        self.end = Handle()

        self.port = LinePort(self.start.pos, self.end.pos)

        self.constraint(horizontal=(self.start.pos, self.end.pos))
```

In case of *Element* item, each line port references positions of two handles, which keeps ports to lie over edges of rectangle. The same applies to *Line* item - every port is defined between positions of two neighbour handles. When *Line* item is orthogonal, then handle and ports share the same constraints, which guard line orthogonality.

This way, item's constraints are reused and their amount is limited to minimum.

1.3.3 Connections

Connection between two items is established by creating a constraint between handle's postion and port's positions (positions are constraint solver variables).

ConnectHandleTool class provides functionality to allow an user to perform connections between items.

1.3.4 Examples

Examples of ports can be found in Gaphas and Gaphor source code

- *gaphas.item.Line* and *gaphas.item.Element* classes
- *gaphas.examples.PortoBox* class has an example of movable port
- Gaphor interface and lifeline items have own specific ports

1.4 State management

A special word should be mentioned about state management. Managing state is the first step in creating an undo system.

The state system consists of two parts:

1. A basic observer (the `@observed` decorator)
2. A reverser

1.4.1 Observer

The observer simply dispatches the function called (as `<function ..>`, not as `<unbound method..>`!) to each handler registered in an observers list.

```
>>> from gaphas import state
>>> state.observers.clear()
>>> state.subscribers.clear()
```

Let's start with creating a Canvas instance and some items:

```
>>> from gaphas.canvas import Canvas
>>> from gaphas.item import Item
>>> canvas = Canvas()
>>> item1, item2 = Item(), Item()
```

For this demonstration let's use the Canvas class (which contains an add/remove method pair).

It works (see how the add method automatically schedules the item for update):

```
>>> def handler(event):
...     print 'event handled', event
>>> state.observers.add(handler)
>>> canvas.add(item1)
event handled (<function add at ...>, (<gaphas.canvas.Canvas object at 0x...>,
↪<gaphas.item.Item object at 0x...>, None, None), {})
>>> canvas.add(item2, parent=item1)
event handled (<function add at ...>, (<gaphas.canvas.Canvas object at 0x...>,
↪<gaphas.item.Item object at 0x...>, <gaphas.item.Item object at 0x...), {})
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>, <gaphas.item.Item object at 0x...>]
```

Note that the handler is invoked before the actual call is made. This is important if you want to store the (old) state for an undo mechanism.

Remember that this observer is just a simple method call notifier and knows nothing about the internals of the `Canvas` class (in this case the `remove()` method recursively calls `remove()` for each of it's children). Therefore some

careful crafting of methods may be necessary in order to get the right effect (items should be removed in the right order, child first):

```
>>> canvas.remove(item1)
event handled (<function _remove at ...>, (<gaphas.canvas.Canvas object at 0x...>,
↪<gaphas.item.Item object at 0x...>), {})
event handled (<function _remove at ...>, (<gaphas.canvas.Canvas object at 0x...>,
↪<gaphas.item.Item object at 0x...>), {})
>>> canvas.get_all_items()
[]
```

The `@observed` decorator can also be applied to properties, as is done in gaphas/connector.py's Handle class:

```
>>> from gaphas.solver import Variable
>>> var = Variable()
>>> var.value = 10
event handled (<function set_value at 0x...>, (Variable(0, 20), 10), {})
```

(this is simply done by observing the setter method).

Off course handlers can be removed as well (only the default revert handler is present now):

```
>>> state.observers.remove(handler)
>>> state.observers
set([])
```

What should you know:

1. The observer always generates events based on 'function' calls. Even for class method invocations. This is because, when calling a method (say Tree.add) it's the `im_func` field is executed, which is a function type object.
2. It's important to know if an event came from invoking a method or a simple function. With methods, the first argument always is an instance. This can be handy when writing an undo management systems in case multiple calls from the same instance do not have to be registered (e.g. if a method `set_point()` is called with exact coordinates (in stead of deltas), only the first call to `set_point()` needs to be remembered).

1.4.2 Reverser

The reverser requires some registration.

1. Property setters should be declared with `reversible_property()`
2. Method (or function) pairs that implement each others reverse operation (e.g. add and remove) should be registered as `reversible_pair()`'s in the reverser engine. The reverser will construct a tuple (callable, arguments) which are send to every handler registered in the subscribers list. Arguments is a `dict()`.

First thing to do is to actually enable the `revert_handler`:

```
>>> state.observers.add(state.revert_handler)
```

This handler is not enabled by default because:

1. it generates quite a bit of overhead if it isn't used anyway
2. you might want to add some additional filtering.

Point 2 may require some explanation. First of all observers have been added to almost every method that involves a state change. As a result loads of events are generated. In most cases you're only interested in the first event, since that one contains the state before it started changing.

Handlers for the reverse events should be registered on the subscribers list:

```
>>> events = []
>>> def handler(event):
...     events.append(event)
...     print 'event handler', event
>>> state.subscribers.add(handler)
```

After that, signals can be received of undoable (reverse-)events:

```
>>> canvas.add(Item())
event handler (<function _remove at ...>, {'item': <gaphas.item.Item object at 0x...>,
↪ 'self': <gaphas.canvas.Canvas object at 0x...>})
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>]
```

As you can see this event is constructed of only two parameters: the function that does the inverse operation of `add()` and the arguments that should be applied to that function.

The inverse operation is easiest performed by the function `saveapply()`. Of course an inverse operation is emitting a change event too:

```
>>> state.saveapply(*events.pop())
event handler (<function add at 0x...>, {'item': <gaphas.item.Item object at 0x...>,
↪'self': <gaphas.canvas.Canvas object at 0x...>, 'parent': None, 'index': 0})
>>> canvas.get_all_items()
[]
```

Just handling method pairs is one thing. Handling properties (descriptors) in a simple fashion is another matter. First of all the original value should be retrieved before the new value is applied (this is different from applying the same arguments to another method in order to reverse an operation).

For this a `reversible_property` has been introduced. It works just like a property (in fact it creates a plain old property descriptor), but also registers the property as being reversible.

```
>>> var = Variable()
>>> var.value = 10
event handler (<function set_value at 0x...>, {'self': Variable(0, 20), 'value': 0.0})
```

Handlers can be simply removed:

```
>>> state.subscribers.remove(handler)
>>> state.observers.remove(state.revert_handler)
```

1.4.3 What is Observed

As far as Gaphas is concerned, only properties and methods related to the model (e.g. `Canvas`, `Item`) emit state changes. Some extra effort has been taken to monitor the `Matrix` class (which is from Cairo).

canvas.py: `Canvas`: `add()` and `remove()`

connector.py: `Position`: `x` and `y` properties

`Handle`: `connectable`, `movable`, `visible`, `connected_to` and `disconnect` properties

item.py: `Item`: `matrix` property

`Element`: `min_height` and `min_width` properties

`Line`: `line_width`, `fuzziness`, `orthogonal` and `horizontal` properties

solver.py: `Variable`: `strength` and `value` properties

`Solver`: `add_constraint()` and `remove_constraint()`

matrix.py: `Matrix`: `invert()`, `translate()`, `rotate()` and `scale()`

Test cases are described in undo.txt.

1.5 Undo - implementing basic undo behaviour with Gaphas

This document describes a basic undo system and tests Gaphas' classes with this system.

This document contains a set of test cases that is used to prove that it really works.

See state.txt about how state is recorded.

Contents

For this to work, some boilerplate has to be configured:

```
>>> from gaphas import state
>>> state.observers.clear()
>>> state.subscribers.clear()
```

```
>>> undo_list = []
>>> redo_list = []
>>> def undo_handler(event):
...     undo_list.append(event)
>>> state.observers.add(state.revert_handler)
>>> state.subscribers.add(undo_handler)
```

This simple undo function will revert all states collected in the undo_list:

```
>>> def undo():
...     apply_me = list(undo_list)
...     del undo_list[:]
```

(continues on next page)

(continued from previous page)

```
...     apply_me.reverse()
...     for e in apply_me:
...         state.saveapply(*e)
...     redo_list[:] = undo_list[:]
...     del undo_list[:]
```

1.5.1 Undo functionality tests

The following sections contain most of the basis unit tests for undo management.

tree.py: Tree

Tree has no observed methods.

matrix.py: Matrix

Matrix is used by Item classes.

```
>>> from gaphas.matrix import Matrix
>>> m = Matrix()
>>> m
Matrix(1, 0, 0, 1, 0, 0)
```

translate(tx, ty):

```
>>> m.translate(12, 16)
>>> m
Matrix(1, 0, 0, 1, 12, 16)
>>> undo()
>>> m
Matrix(1, 0, 0, 1, 0, 0)
```

scale(sx, sy):

```
>>> m.scale(1.5, 1.5)
>>> m
Matrix(1.5, 0, 0, 1.5, 0, 0)
>>> undo()
>>> m
Matrix(1, 0, 0, 1, 0, 0)
```

rotate(radians):

```
>>> def matrix_approx(m):
...     a = []
...     for i in tuple(m):
...         if -1e-10 < i < 1e-10: i=0
...         a.append(i)
...     return tuple(a)
```

```
>>> m.rotate(0.5)
>>> m
Matrix(0.877583, 0.479426, -0.479426, 0.877583, 0, 0)
>>> undo()
>>> matrix_approx(m)
(1.0, 0, 0, 1.0, 0, 0)
```

Okay, nearly, close enough IMHO...

```
>>> m = Matrix()
>>> m.translate(12, 10)
>>> m.scale(1.5, 1.5)
>>> m.rotate(0.5)
>>> m
Matrix(1.31637, 0.719138, -0.719138, 1.31637, 12, 10)
>>> m.invert()
>>> m
Matrix(0.585055, -0.319617, 0.319617, 0.585055, -10.2168, -2.01515)
>>> undo()
>>> matrix_approx(m)
(1.0, 0, 0, 1.0, 0, 0)
```

Again, rotate does not result in an exact match, but it's close enough.

```
>>> undo_list
[]
```

canvas.py: Canvas

```
>>> from gaphas import Canvas, Item
>>> canvas = Canvas()
>>> canvas.get_all_items()
[]
>>> item = Item()
>>> canvas.add(item)
```

The `request_update()` method is observed:

```
>>> len(undo_list)
1
>>> canvas.request_update(item)
>>> len(undo_list)
2
```

On the canvas only `add()` and `remove()` are monitored:

```
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>]
>>> item.canvas is canvas
True
>>> undo()
>>> canvas.get_all_items()
[]
>>> item.canvas is None
True
```

(continues on next page)

(continued from previous page)

```
>>> canvas.add(item)
>>> del undo_list[:]
>>> canvas.remove(item)
>>> canvas.get_all_items()
[]
>>> undo()
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>]
>>> undo_list
[]
```

Parent-child relationships are restored as well:

TODO!

```
>>> child = Item()
>>> canvas.add(child, parent=item)
>>> child.canvas is canvas
True
>>> canvas.get_parent(child) is item
True
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>, <gaphas.item.Item object at 0x...>]
>>> undo()
>>> child.canvas is None
True
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>]
>>> child in canvas.get_all_items()
False
```

Now redo the previous undo action:

```
>>> undo_list[:] = redo_list[:]
>>> undo()
>>> child.canvas is canvas
True
>>> canvas.get_parent(child) is item
True
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>, <gaphas.item.Item object at 0x...>]
```

Remove also works when items are removed recursively (an item and it's children):

```
>>> child = Item()
>>> canvas.add(child, parent=item)
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>, <gaphas.item.Item object at 0x...>]
>>> del undo_list[:]
>>> canvas.remove(item)
>>> canvas.get_all_items()
[]
>>> undo()
>>> canvas.get_all_items()
[<gaphas.item.Item object at 0x...>, <gaphas.item.Item object at 0x...>]
>>> canvas.get_children(item)
[<gaphas.item.Item object at 0x...>]
```

As well as the reparent() method:

```
>>> canvas = Canvas()
>>> class NameItem(Item):
...     def __init__(self, name):
...         super(NameItem, self).__init__()
...         self.name = name
...     def __repr__(self):
...         return '<%s>' % self.name
>>> ni1 = NameItem('a')
>>> canvas.add(ni1)
>>> ni2 = NameItem('b')
>>> canvas.add(ni2)
>>> ni3 = NameItem('c')
>>> canvas.add(ni3, parent=ni1)
>>> ni4 = NameItem('d')
>>> canvas.add(ni4, parent=ni3)
>>> canvas.get_all_items()
[<a>, <c>, <d>, <b>]
>>> del undo_list[:]
>>> canvas.reparent(ni3, parent=ni2)
>>> canvas.get_all_items()
[<a>, <b>, <c>, <d>]
>>> len(undo_list)
1
>>> undo()
>>> canvas.get_all_items()
[<a>, <c>, <d>, <b>]
```

Redo should work too:

```
>>> undo_list[:] = redo_list[:]
>>> undo()
>>> canvas.get_all_items()
[<a>, <b>, <c>, <d>]
```

Undo/redo a connection: see gaphas/tests/test_undo.py

connector.py: Handle

Changing the Handle's position is reversible:

```
>>> from gaphas import Handle
>>> handle = Handle()
>>> handle.pos = 10, 12
>>> handle.pos
<Position object on (10, 12)>
>>> undo()
>>> handle.pos
<Position object on (0, 0)>
```

As are all other properties:

```
>>> handle.connectable, handle.movable, handle.visible
(False, True, True)
>>> handle.connectable = True
>>> handle.movable = False
```

(continues on next page)

(continued from previous page)

```
>>> handle.visible = False
>>> handle.connectable, handle.movable, handle.visible
(True, False, False)
```

And now undo the whole lot at once:

```
>>> undo()
>>> handle.connectable, handle.movable, handle.visible
(False, True, True)
```

item.py: Item

The basic Item properties are canvas and matrix. Canvas has been tested before, while testing the Canvas class.

The Matrix has been tested in section matrix.py: Matrix.

item.py: Element

An element has `min_height` and `min_width` properties.

```
>>> from gaphas import Element
>>> e = Element()
>>> e.min_height, e.min_width
(Variable(10, 100), Variable(10, 100))
>>> e.min_height, e.min_width = 30, 40
>>> e.min_height, e.min_width
(Variable(30, 100), Variable(40, 100))
```

```
>>> undo()
>>> e.min_height, e.min_width
(Variable(0, 100), Variable(0, 100))
```

```
>>> canvas = Canvas()
>>> canvas.add(e)
>>> undo()
>>> e.canvas
```

item.py: Line

A line has the following properties: `line_width`, `fuzziness`, `orthogonal` and `horizontal`. Each one of then is observed for changes:

```
>>> from gaphas import Line
>>> from gaphas.segment import Segment
>>> l = Line()
```

Let's first add a segment to the line, to test orthogonal lines as well.

```
>>> segment = Segment(l, None)
>>> _ = segment.split_segment(0)
```

```
>>> l.line_width, l.fuzziness, l.orthogonal, l.horizontal
(2, 0, False, False)
```

Now change the properties:

```
>>> l.line_width = 4
>>> l.fuzziness = 2
>>> l.orthogonal = True
>>> l.horizontal = True
>>> l.line_width, l.fuzziness, l.orthogonal, l.horizontal
(4, 2, True, True)
```

And undo the changes:

```
>>> undo()
>>> l.line_width, l.fuzziness, l.orthogonal, l.horizontal
(2, 0, False, False)
```

In addition to those properties, line segments can be split and merged.

```
>>> l.handles()[1].pos = 10, 10
>>> l.handles()
[<Handle object on (0, 0)>, <Handle object on (10, 10)>]
```

This is our basis for further testing.

```
>>> del undo_list[:]
```

```
>>> Segment(l, None).split_segment(0)
([<Handle object on (5, 5)>], [<gaphas.connector.LinePort object at 0x...>])
>>> l.handles()
[<Handle object on (0, 0)>, <Handle object on (5, 5)>, <Handle object on (10, 10)>]
```

The opposite operation is performed with the merge_segment() method:

```
>>> undo()
>>> l.handles()
[<Handle object on (0, 0)>, <Handle object on (10, 10)>]
```

Also creation and removal of connected lines is recorded and can be undone:

```
>>> canvas = Canvas()
>>> def real_connect(hitem, handle, item):
...     def real_disconnect():
...         pass
...     canvas.connect_item(hitem, handle, item, port=None, constraint=None,␣
↪callback=real_disconnect)
>>> b0 = Item()
>>> canvas.add(b0)
>>> b1 = Item()
>>> canvas.add(b1)
>>> l = Line()
>>> canvas.add(l)
>>> real_connect(l, l.handles()[0], b0)
>>> real_connect(l, l.handles()[1], b1)
>>> canvas.get_connection(l.handles()[0])
Connection(item=<gaphas.item.Line object at 0x...>)
```

(continues on next page)

(continued from previous page)

```
>>> canvas.get_connection(l.handles()[1])
Connection(item=<gaphas.item.Line object at 0x...>)
```

Clear already collected undo data:

```
>>> del undo_list[:]
```

Now remove the line from the canvas:

```
>>> canvas.remove(l)
```

The handles are disconnected:

```
>>> l.canvas
>>> canvas.get_connection(l.handles()[0])
>>> canvas.get_connection(l.handles()[1])
```

Undoing the remove() action should put everything back in place again:

```
>>> undo()
```

```
>>> l.canvas
<gaphas.canvas.Canvas object at 0x...>
>>> canvas.get_connection(l.handles()[0])
Connection(item=<gaphas.item.Line object at 0x...>)
>>> canvas.get_connection(l.handles()[1])
Connection(item=<gaphas.item.Line object at 0x...>)
```

solver.py: Variable

Variable's strength and value properties are observed:

```
>>> from gaphas.solver import Variable
>>> v = Variable()
>>> v.value = 10
>>> v.strength = 100
>>> v
Variable(10, 100)
>>> undo()
>>> v
Variable(0, 20)
```

solver.py: Solver

Solvers `add_constraint()` and `remove_constraint()` are observed.

```
>>> from gaphas.solver import Solver
>>> from gaphas.constraint import EquationConstraint
>>> s = Solver()
>>> a, b = Variable(1.0), Variable(2.0)
>>> s.add_constraint(EquationConstraint(lambda a,b: a+b, a=a, b=b))
EquationConstraint(<lambda>, a=Variable(1, 20), b=Variable(2, 20))
>>> list(s.constraints_with_variable(a))
[EquationConstraint(<lambda>, a=Variable(1, 20), b=Variable(2, 20))]
```

```
>>> undo()
>>> list(s.constraints_with_variable(a))
[]
```

```
>>> undo_list[:] = redo_list[:]
>>> undo()
>>> list(s.constraints_with_variable(a))
[EquationConstraint(<lambda>, a=Variable(1, 20), b=Variable(2, 20))]
```

1.6 Quadtree implementation

In order to find items and handles fast on a 2D surface, a geometric structure is required.

There are two popular variants: Quadtrees and R-trees. R-trees are tough and well suited for non-moving data. Quadtrees are easier to understand and easier to maintain.

Idea:

- Divide the view in 4 quadrants and place each item in a quadrant.
- When a quadrant has more than ‘’x” elements, divide it again.
- When an item overlaps more than one quadrant, it’s added to the owner.

Gaphas uses item bounding boxed to determine where items should be put.

It is also possible to relocate or remove items to the tree.

The Quadtree itself is added as part of Gaphas’ View. The view is aware of item’s bounding boxes as it is responsible for user interaction. The Quadtree is set to the size of the window. As a result items which are part of the diagram, may be placed outside the window and thus will not be added to the quadtree. Item’s that are partly in- and partly outside the window will be clipped.

1.6.1 Interface

The Quadtree interface is simple and tailored towards the use cases of gaphas.

Important properties:

- bounds: boundaries of the canvas/view

Methods for working with items in the quadtree:

- *add(item, bounds)*: add an item to the quadtree
- *remove(item)*: remove item from the tree
- *update(item, new_bounds)*: replace an item in the quadtree, using it’s new boundaries.
- Multiple ways of finding items have been implemented: 1. Find item closest to point 2. Find all items within distance *d* of a point 3. Find all items inside a rectangle 4. Find all items inside or intersecting with a rectangle

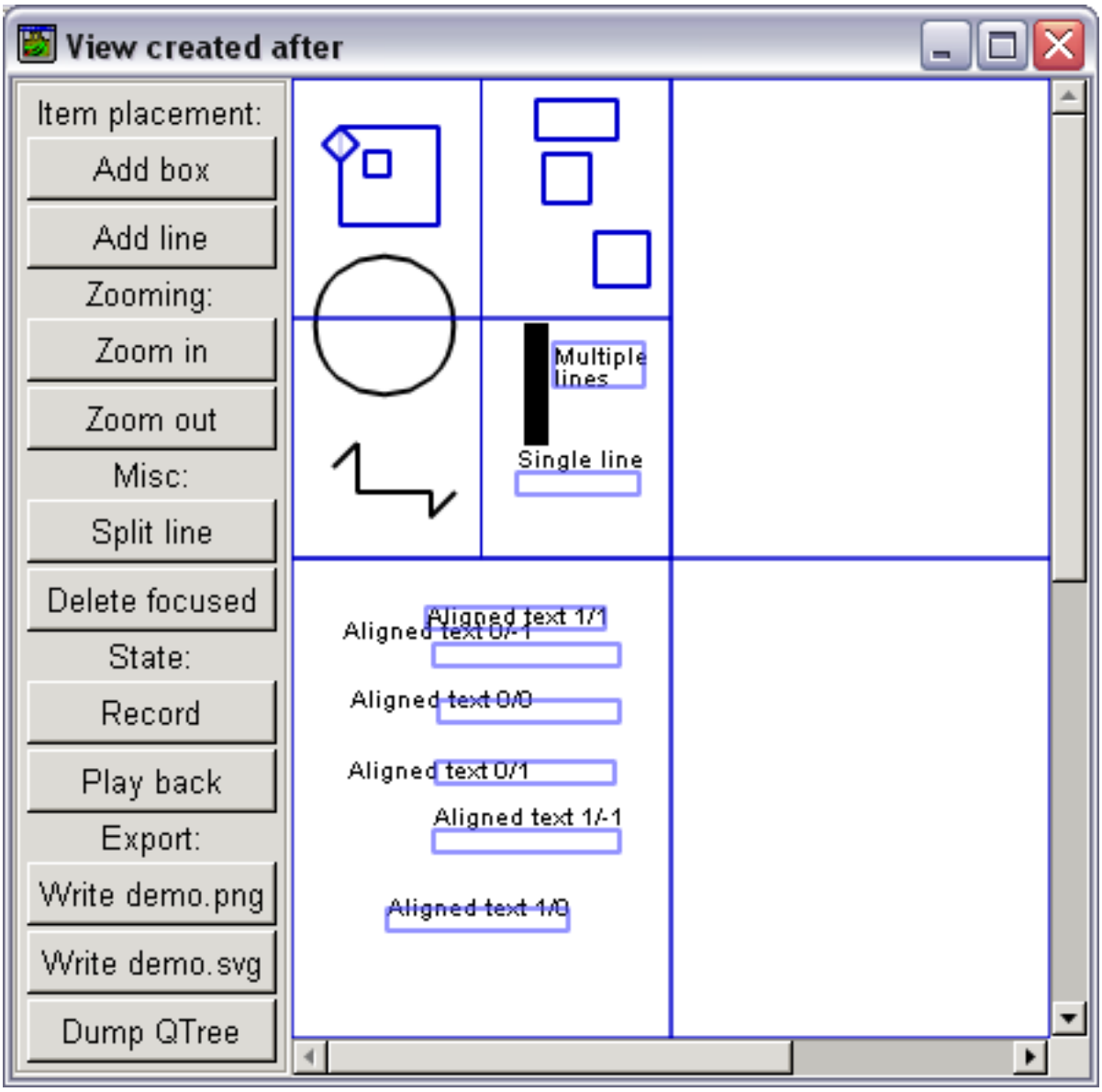
Methods working on the quadtree itself:

- *resize(new_bounds)*: stretch the boundaries of the quadtree if necessary.

1.6.2 Implementation

The implementation of gaphas' Quadtree can be found at http://github.com/amolenaar/gaphas/trees/blob/gaphas/quadtree.py.

Here's an example of the Quadtree in action (Gaphas' demo app with *gaphas.view.DEBUG_DRAW_QUADTREE* enabled):



The screen is divided into four equal quadrants. The first quadrant has many items, therefore it has been divided again.

References

(Roids) http://roids.slowchop.com/roids/browser/trunk/src/quadtree.py

(???) http://mu.arete.cc/pcr/syntax/quadtree/1/quadtree.py

(!PyGame) http://www.pygame.org/wiki/QuadTree?parent=CookBook

(PythonCAD) http://www.koders.com/python/fidB93B4D02B1C4E9F90F351736873DF6BE4D8D5783.aspx

1.7 Constraint Solver

Gaphas' constraint solver can be consider the hart of the Canvas (The Canvas instance holds and orders the items that are drawn on it and in the end will be displayed to the user).

The constraint solver ('solver' for short) is called during the update of the canvas.

A solver contains a set of constraints. Each constraint in itself is pretty straightforward (e.g. variable ''a" equals variable ''b''). Did I say variable? Yes I did. Let's start at the bottom and work our way to the solver.

A Variable is a simple class, contains a value. It behaves like a ''float" in many ways. There is one typical thing about Variables: they can be added to Constraints.

A Constraint is an equation. The trick is to make all constraints true when solving a set of constraints. That can be pretty tricky, since a Variable can play a role in more than one Constraint. Constraint solving is governed by the Solver (ah, there it is).

Constraints are instances of Constraint class (more specific: subclasses of the Constraint class). A Constraint can perform a specific trick: e.g. centre one Variable between two other Variables or make one Variable equal to another Variable.

It's the Solver's job to make sure all constraint are true in the end. In some cases this means a constraint needs to be resolved twice, but the Solver sees to it that no deadlocks occur.

1.7.1 Variables

When a variable is assigned a value it marks itself ''dirty''. As a result it will be resolved the next time the solver is asked to.

Each variable has a specific ''strength''. String variables can not be changed by weak variables, but weak variables can change when a new value is assigned to a stronger variable. The Solver always tries to solve a constraint for the weakest variable. If two variables have equal strength, however, the variable that is most recently changed is considered slightly stronger than the not (or earlier) changed variable.

1.7.2 Projections

There's one special thing about Variables: since each item has it's own coordinate system ((0, 0) point, handy when the item is rendered), it's pretty hard to make sure (for example) a line can connect to a box and ''stays" connected, even when the box is dragged around. How can such a constraint be maintained? This is where Projections come into play. A Projection can be used to project a variable on another space (read: coordinate system).

The default projection (to canvas coordinates) is located in *gaphas.canvas* and is known as *CanvasProjection*.

When a constraint contains projections, it is most likely that this constraint connects two items together. At least the constraint is not entirely bound to the item's coordinate space. This knowledge is used when an item is moved. A move operation typically only requires a change in coordinates, relative to the item's parent item (this is why having a (0,0) point per item is so handy). This means that constraints local to the item not not need to be resolved. Constraints with links outside the item's space should be solved though. Projections play an important role in determining which constraints should be resolved.

The Solver can be found at: http://github.com/amolenaar/gaphas/trees/blobs/gaphas/solver.py, along with Variables and Projections.

1.8 API reference

This part describes the API of Gaphas.

The API can be separated into three parts. First of all there's the model (canvas and items). Then there's the view (view) and controllers (tools).

Some more generic stuff is also described here.

1.8.1 Canvas and items

Canvas

This part describes the API of Gaphas.

mod *gaphor.canvas*

Items

Items are put on a Canvas.

mod *gaphor.item*

Handles and ports

This part describes the API of Gaphas.

mod *gaphor.connector*

Solver

This part describes the API of Gaphas.

mod *gaphor.solver*

Constraints

The default constraint classes provided by Gaphas

mod *gaphor.constraint*

TODO: And the rest

Various utility functions

This part describes the API of Gaphas.

mod *gaphor.util*

1.8.2 View and tools

Everything related to displaying the canvas and interacting with it.

View

The View is the base class for viewing related operations. A good example is the `GtkView`, which provides a GTK+ widget for viewing (and editing) the canvas. Views are also used for rendering to images (for example SVG or PNG).

mod *gaphor.view*

GTK+ View

This part describes the API of Gaphas.

mod *gaphor.view*

Painters

This part describes the API of Gaphas.

mod *gaphor.painter*

TODO: Add the rest

Interacting with the canvas is done through tools. Tools tell _what_ has to be done (like moving). To make an element move aspects are defined. Aspects tell _how_ the behaviour has to be performed.

Tools

This part describes the API of Gaphas.

mod *gaphor.tool*

Aspects

TODO: Explain aspects

mod *gaphor.aspect*

Find elements on a canvas

Perform selection of elements

Move items around

Select a handle

Perform handle motion

Connect and disconnect a handle

Perform behaviour for elements a handle is connected to.

... autoclass:: PaintFocused

Special aspect that can perform paint behaviour for the *FocusedItemPainter*.

Extended behaviour

By importing the following modules, extra behaviour is added to the default view behaviour.

Line segmentation

This part describes the API of Gaphas.

mod *gaphor.segment*

Alignment helper: Guide

By importing this module Guide behaviour will be added to all views. Guides will help you align items next to each other.

The guide module consists of a few aspects, triggered when items are moved, as well as a painter, so guides will be drawn.

mod *gaphor.guide*

1.8.3 Miscellaneous

Tree structure

This part describes the API of Gaphas.

mod *gaphor.tree*

Matrix

The Matrix class is used to define transformations on an item, relative to the parent Item. This is basically the same implementation as cairo.Matrix, only notifications are sent on state changes (see the state module).

mod *gaphor.matrix*

Table structure

This part describes the API of Gaphas.

mod *gaphor.table*

Quadtree

This part describes the API of Gaphas.

mod *gaphor.quadtree*

Lines, points and rectangles

Decorators

mod *gaphor.decorators*

- genindex
- modindex
- search

Python Module Index

g

Index

G