# Chapter 1

# Applying Layout

---

**Note:**
This chapter was written with the participation of Peter Uhnák (i.uhnak@gmail.com).

---

A layout is the visualization representation along a two-dimensional plan of elements, typically edges and nodes. A layout makes a visualization not only aesthetic but also comprehensible. Roassal offers numerous different layouts and supports the composition of layouts.

Consider the following example (Figure 1.1):

```
view := RTView new.
es := (RTEllipse new size: 12; color: Color blue)
   elementsOn: (1 to: 30).
view addAll: es.

RTEdgeBuilder new
   view: view;
   elements: es;
   connectFrom: [ :value | value // 2 ].

es do: [ :each | each translateTo: (250 atRandom) @ (250 atRandom) ].
view
```

The script builds some elements and edges that link these elements. Each element is translated to a random location. Applying the RTTreeLayout layout highlights the structure from the connection between the elements (Figure 1.2):

```
view := RTView new.
es := (RTEllipse new size: 12; color: Color blue)
```

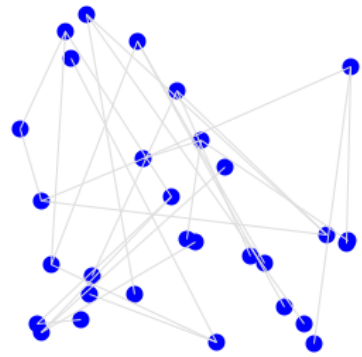

Figure 1.1: Graph with randomly placed nodes.

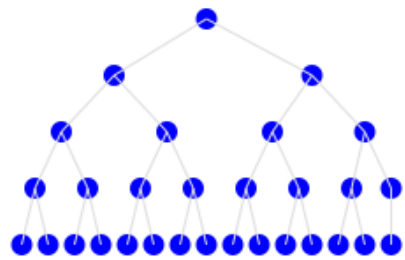

Figure 1.2: Tree layout.

```
    elementsOn: (1 to: 30).
view addAll: es.

RTEdgeBuilder new
    view: view;
    elements: es;
    connectFrom: [ :value | value // 2 ].

es do: [ :each | each translateTo: (250 atRandom) @ (250 atRandom) ].
RTTreeLayout on: es.
view
```

The layout reveals that the elements form a binary tree and one can distinguish what the top root node is and what the leaves are.

Roassal offers over 30 different layouts, each useful in its own rights.

## 1.1 Element-based Layouts

Element-based layouts are a particular set of layouts in Roassal which do not consider edges to determine element locations. Instead of using edges, an element-based layout uses element size, shape or position within a group. Edges are not forbidden: they are simply not used by the layout.

### Circle Layouts

Circle layouts arrange elements along a geometrical circle. The order of the elements, as with all circle-based layouts, is the same as the collection on which the layout operates.

```
v := RTView new.

es := (RTEllipse new size: 12) elementsOn: (1 to: 10).
v addAll: es.
es @ RTLabeled.

RTMetricNormalizer new
   elements: es;
   normalizeColor: #yourself using: (Array with: Color red with: Color lightRed).

RTEdgeBuilder new
   view: v;
   elements: es;
   connectFrom: [ :value | value // 2 ].

RTCircleLayout new on: es.
v
```

A circular layout may be parametrized along several properties:

```
RTCircleLayout new
   initialIncrementalAngle: 30 degreesToRadians;
   initialAngle: 15 degreesToRadians;
   initialRadius: 200;
   on: es.
```

Without any option, the circle layout distributes all the elements evenly along a circle (2pi/elements size). Additionally radius can be either set absolutely via initialRadius:, or as a scalable factor scaleBy: - then the radius will be elements size * scaleFactor.

It is important to note that RTCircleLayout does not take into consideration the size of the elements; this is enough when the elements are uniform, however if their sizes vary, different layouts may be considered.

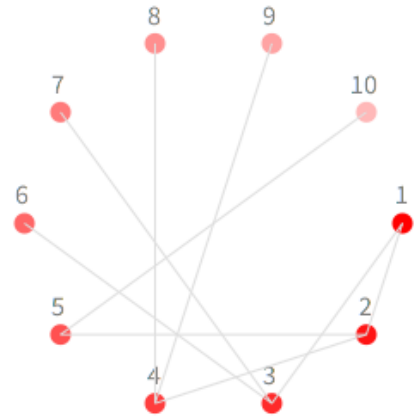


Figure 1.3: Circle layout applied on some elements .

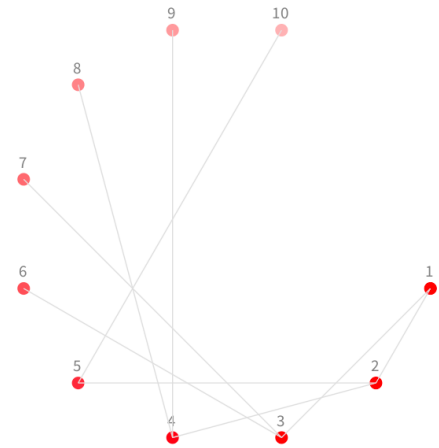


Figure 1.4: Some options of Circular Layout.

Variants of the default circle layout, named *equidistant* and *weighted* layouts, are also available:

- RTEquidistantCircleLayout makes sure that there is the same distance between each element.
- RTWeightedCircleLayout on the other hand adds spacing based on the

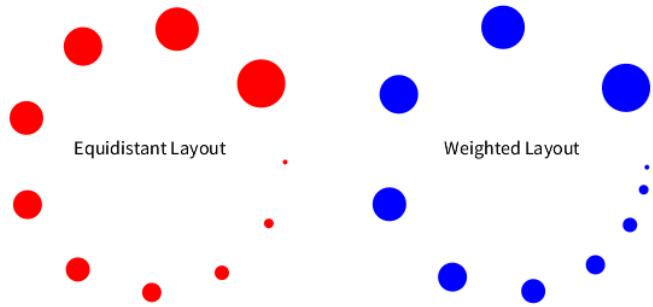


Figure 1.5: Equidistant (left) and Weighted (right) layout with non-uniform sizes.

size of the elements. Thus there will be less space between smaller elements, and more space between large ones.

So now if we apply layout on non-uniform elements we get:

```
v := RTView new.
elements := (RTEllipse new color: Color red; size: [:vv | vv * 4 ])
    elementsOn: (1 to: 10).
v addAll: elements.
RTEquidistantCircleLayout on: elements.
elements translateBy: -150 @ 0.
v add: ((RTLabel new elementOn: 'Equidistant Layout') translateTo: -40 @ 100).

es := (RTEllipse new color: Color blue; size: [:vv | vv * 4 ])
    elementsOn: (1 to: 10).
v addAll: es.
RTWeightedCircleLayout on: es.
es translateBy: 150 @ 0.
v add: ((RTLabel new elementOn: 'Weighted Layout') translateTo: 260 @ 100).
v
```

## Flow Layouts

A flow layout arranges elements in a 'flowing' manner. While we could consider circle layouts to be also a flow in a clockwise direction, layouts presented here provide flow by lines and columns.

### Flow and Grid Layouts

The flow layout arranges elements in lines, each line flowing from left to right; Horizontal flow on the other hand is in columns, flowing from top to bottom.

```
v := RTView new.

shape := RTCompositeShape new.
shape add: RTLabel new.
shape add: (RTBox new
   color: Color transparent; borderColor: Color black).
es := RTGroup
   with: (shape elementOn: RTFlowLayout)
   with: (shape elementOn: RTHorizontalFlowLayout)
   with: (shape elementOn: RTGridLayout)
   with: (shape elementOn: RTCellLayout).
v addAll: es.

RTNest new
   for: es
   inShape: #second
   add: [ :group :layout |
      group addAll: ((RTBox new size: [:m | m * 10])
         elementsOn: (1 to: 6)).
      layout new on: group.
   ].

RTCellLayout new gapSize: 10; on: es.

(v elements allButFirst: 4) @ (RTLabeled new
   color: Color black; center).
v
```

Flow layouts vertically line up elements according to their size. The layout may be configured with a maximum total width (maxWidth:). For Grid and Cell layout this limit is instead number of items in the line (lineItemsCount:). By default a flow layout try to fill a roughly rectangular area, while a grid layout approximate the golden ratio.

### Alignment

Cells in Flow layouts can be aligned:

To align RTHorizontalFlowLayout use alignTop for left, and alignBottom for right alignment.

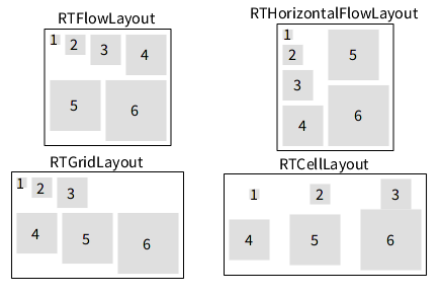


Figure 1.6: Flow and Grid Layouts.


Figure 1.7: RTFlowLayout alignments.

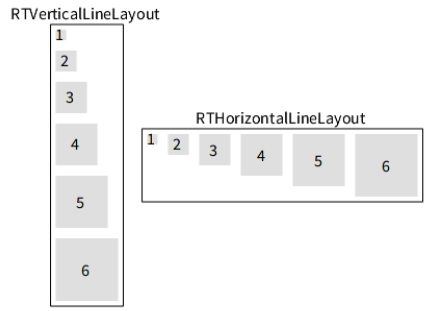


Figure 1.8: RTVerticalLineLayout and RTHorizontalLineLayout.

### Line Layouts

Two line layouts are offered: RTVerticalLineLayout and RTHorizontalLineLayout to layout, vertically and horizontally, respectively (Figure 1.8).

Here is an example of the horizontal line layout:

```
v := RTView new.
```

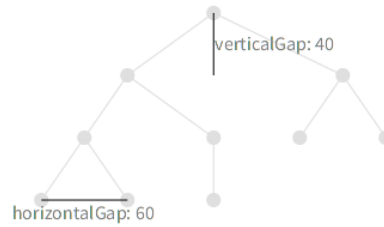


Figure 1.9: RTTreeLayout demonstrating gap sizes.

```
es := (RTBox new size: #yourself) elementsOn: (10 to: 40 by: 5).
v addAll: es.
RTHorizontalLineLayout on: es.
v
```

Replacing RTHorizontalLineLayout by RTVerticalLineLayout triggers the vertical line layout instead.

## 1.2 Edge-driven layouts

Edge-driven layouts determine the location of an element based on the edges linking these elements.

### Tree Layout

The beginning of this chapter gives an example of using the tree layout.

Note that in the picture above the horizontalGap is applied only to the leaves of the tree; distance between parents is then accommodated automatically, so no overlapping or crossing occurs. The tree Layout orients the tree vertically, while the RTHorizontalTreeLayout uses an original orientation, from left to right.

### Radial Tree Layout

One problem with trees is that they tend to have many leaves which often results in very wide visualizations. One way to deal with this problem is to present the tree circularly. Since each new layer increases the radius of the circle, the overall element structure accommodates the space better.

```
v := RTView new.
```

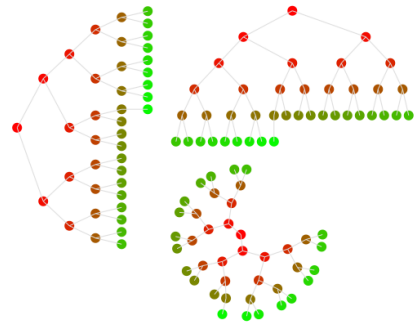

Figure 1.10: Comparison of Horizontal, Vertical and Radial Tree Layouts.

```
elsBuilder := [ | es |
    es := (RTEllipse new size: 12) elementsOn: (1 to: 40).
    v addAll: es.

RTEdgeBuilder new
    view: v;
    elements: es;
    connectFrom: [ :value | value // 2 ].

RTMetricNormalizer new
    elements: es;
    normalizeColor: #yourself using: (Array with: Color red with: Color green).
es
].

g := RTGroup with: elsBuilder value with: elsBuilder value with: elsBuilder value.

RTTreeLayout new on: g first.
RTHorizontalTreeLayout new on: g second.
RTRadialTreeLayout new on: g third.

RTRectanglePackLayout new gap: 0.1; on: g.
v
```

Note that RTRadialTreeLayout may produce an odd result in the presence of edge cycles.

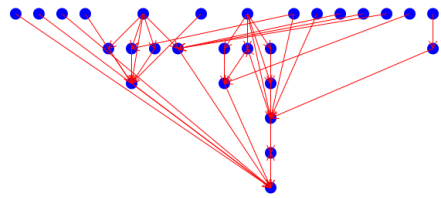

Figure 1.11: RTDominanceTreeLayout showing dependencies between RTShape classes.

## Dominance Tree Layout

This layout is especially useful for visualizing dependencies and flow charts, since it organizes elements in such a manner that the flow of the graph is emphasized.

```
v := RTView new.
v @ RTDraggableView @ RTZoomableView.
classes := RTShape withAllSubclasses asGroup.
es := (RTEllipse new size: 15; color: Color blue) elementsOn: classes.
v addAll: es.

eb := RTEdgeBuilder new.
eb view: v; objects: classes.
eb shape arrowedLine; color: Color red.
eb
   connectFrom: #yourself toAll: #dependentClasses.

RTDominanceTreeLayout new
   verticalGap: 30;
   horizontalGap: 15;
   on: es.
v
```

## Cluster Layout

Cluster is visually similar to radial tree; it groups related elements together (Figure 1.12:.

```
view := RTView new.
view @ RTDraggableView  @ RTZoomableView.

es := (RTEllipse new size: 12) elementsOn: (4 to: 100).
view addAll: es.
```

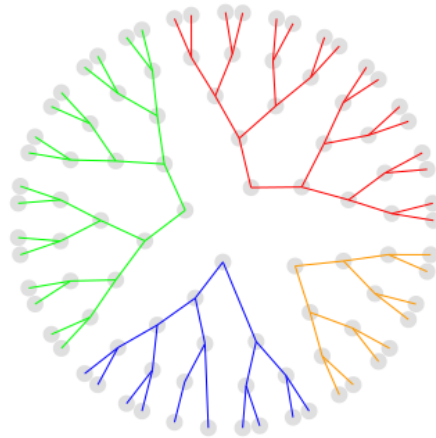

Figure 1.12: Four trees clustered together.

```
edges := RTGroup new.
es copy do: [ :e |
    | fromE |
    fromE := es elementFromModel: (e model // 2).
    fromE ifNotNil: [ edges add: (RTLine edgeFrom: fromE to: e) ].
    ].
view addAll: edges.

colorize := nil.
colorize := [ :root :color |
root outgoingEdges do: [ :edge |
    edge shape color: color.
    edge signalUpdate.
    colorize value: edge to value: color.
    ].
].

colorize value: es first value: Color red.
colorize value: es second value: Color green.
colorize value: es third value: Color blue.
colorize value: es fourth value: Color orange.

RTClusterLayout new on: es.
view
```

The difference between the cluster and radial layouts has to do with the layers forming circles.

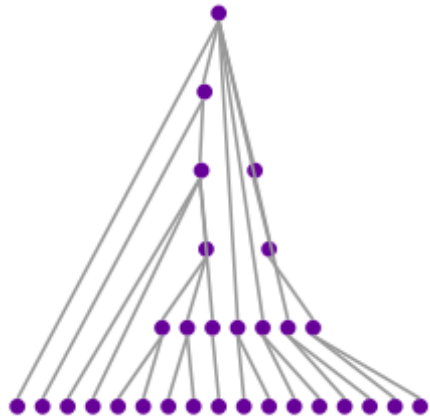

Figure 1.13: Sugiyama layout applied on hierarchy of 'RTLayout' classes.

## Sugiyama

Sugiyama layout is a hierarchical layered layout. It places elements in hierarchical order such that edges goes from higher layer to lower layer. At the same time the layout minimizes the amount of layers and edge crossings (Figure 1.13).

```
classes := RTLayout withAllSubclasses.
b := RTMondrian new.
b shape ellipse color: Color purple; size: 5.
b nodes: classes.
b edges connectFrom: #superclass.
b layout sugiyama.
b
```

## Force Based Layout

Force Based Layout applies force between related elements similar to electrical charge. Thus related elements will repulse each other. The charge is usually negative since it represents repulsion. Additionally to charge: you can also specify strength:, which is the strength of the bonds (edges) between elements.

```
v := RTView new.

es := RTEllipse elementsOn: Collection withAllSubclasses.
v addAll: es.
```

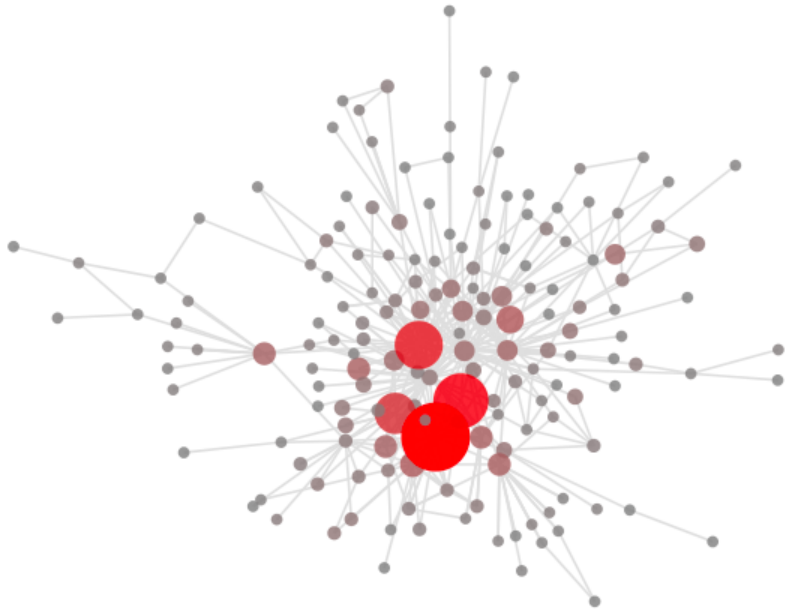

Figure 1.14: RTForceBasedLayout used to layout hierarchy of classes.

```
RTEdgeBuilder new
    elements: es;
    view: v;
    moveBehind;
    connectToAll: #dependentClasses.

RTForceBasedLayout on: es.

RTMetricNormalizer new
    elements: es;
    normalizeColor: #numberOfMethods;
    normalizeSize: #numberOfMethods.
v
```

The force based layout is appealing due to its simplicity of use and nice results that it produces. However, it badly scales. Performing a force based layout on hundreds of nodes and edges may take hours (literally). Before performing this layout on a large graph, one may want to try it on a reduced graph.

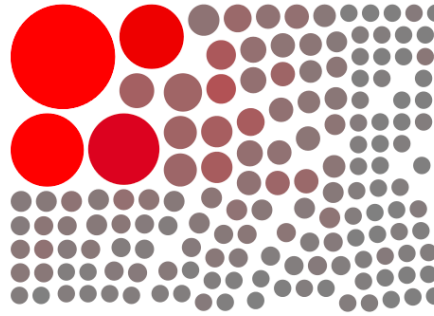

Figure 1.15: Pack of different elements.

## Rectangle Pack

RTRectanglePackLayout packs all the elements as tightly as possible. It uses an element's bounding box, so using circles or polygons instead of boxes will have no effect. One use for this layout is to provide comparative views of some elements — name clouds, source code size of classes, etc.

```
v := RTView new.
v addAll: ((RTEllipse new color: (Color red alpha: 0.3)) elementsOn: Collection
    withAllSubclasses) @ RTPopup.
RTMetricNormalizer new
    elements: v elements;
    normalizeSize: #numberOfLinesOfCode min: 10 max: 60;
    normalizeColor: #numberOfMethods using: (Array with: Color gray with: Color
    red ).
RTRectanglePackLayout on: v elements.
v
```

RTNameCloud also internally uses RTRectanglePackLayout

```
b := RTNameCloud new
    addString: 'open
    | v shape |
    v := RTView new.
    shape := RTLabel new height: [ :assoc | assoc value ]; text: #key.
    v addAll: (shape elementsOn: table associations).
    RTFlowLayout on: v elements.
    v open'.
b
```

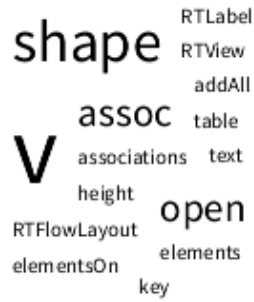


Figure 1.16: RTRectanglePackLayout used to layout a name cloud.

## 1.3 Layout builder

The class RTLayoutBuilder offers expressive ways to build and compose layouts. The layout builder is offered by Mondrian. Consider the following Mondrian script (Figure 1.17):

```
b := RTMondrian new.
b nodes: RTShape withAllSubclasses.
b shape line color: Color blue trans.
b edges
   connectToAll: [ :c | c dependentClasses copyWithout: c superclass ].
b layout
   force.
b
```

The script represents each subclass of RTShape as a box. Edges represents dependencies between classes, without considering inheritance. We made this example to produce lonely classes. It often happens that lonely classes need a particular layout.

The message ifNotConnectedThen: takes as argument a layout (*i.e.,* an instance of a subclass of RTLayout). The provided layout is applied only to elements that are not connected (*i.e.,* with no incoming or outgoing edges).

```
b := RTMondrian new.
b nodes: RTShape withAllSubclasses.
b shape line color: Color blue trans.
b edges
   connectToAll: [ :c | c dependentClasses copyWithout: c superclass ].
```

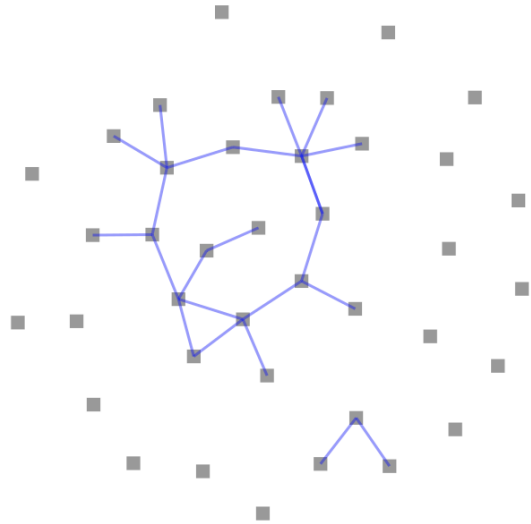

Figure 1.17: Dependencies between classes.

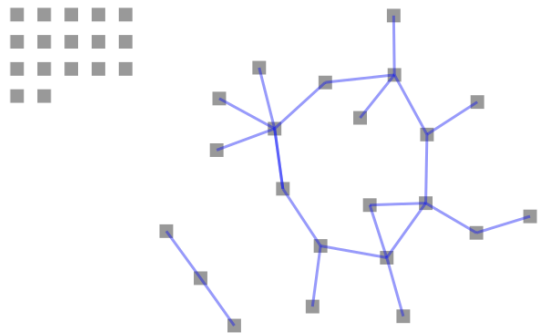

Figure 1.18: Conditional layout.

```
b layout
    force;
    ifNotConnectedThen: RTGridLayout new.
b
```

Using the composition of layout and the layout builder (Figure 1.19):

```
classes := RTLayout withAllSubclasses, RTBuilder withAllSubclasses, RTShape
    withAllSubclasses.
```

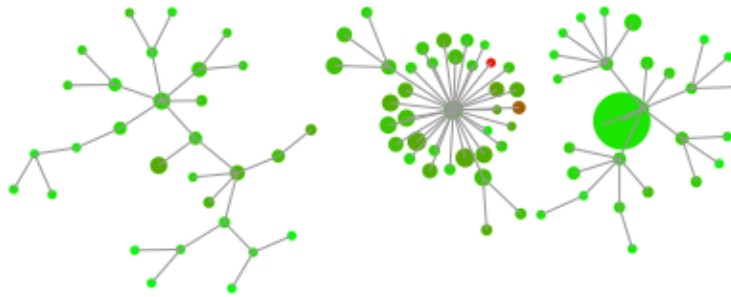

Figure 1.19: RTForceBasedLayout used to layout hierarchy of classes.

```
b := RTMondrian new.
b shape circle size: 5.
b nodes: classes.
b edges connectFrom: #superclass.

b normalizer
    objects: classes;
    normalizeSize: #numberOfMethods min: 5 max: 30;
    normalizeColor: #numberOfLinesOfCode using: (Array with: Color green with:
    Color red ) using: #sqrt.

b layout
    for: [ :c | c includesBehavior: RTLayout ] use: RTForceBasedLayout new;
    for: [ :c | c includesBehavior: RTBuilder ] use: RTForceBasedLayout new;
    for: [ :c | c includesBehavior: RTShape ] use: RTForceBasedLayout new;
    flow.
b
```

Several layouts are available:

- force for a force based layout (also called spring layout)
- horizontalLine for the horizontal line layout
- verticalLine to vertically line up elements
- tree to have a vertical tree
- horizontalTree for an horizontal tree, roots are on the left-hand side and leaves on the right-hand side
- cluster and radial for radial-based layout

- sugiyama for the Sugiyama layout, which is a vertical tree layout that minimizes edges crossing

## 1.4 Creating custom layout

If you want to add your own layout you just need to subclass RTLayout and implement RTLayout>>doExecute: elements.

For more fine-graded control you have three main methods available.

```
RTLayout>>executeOnElements: elements
  self doInitialize: elements.
  self doExecute: elements asOrderedCollection.
  self doPost: elements.
```

1. doInitialize: can be used for element preprocessing, if needed. For example RTAbstractGraphLayout uses this for removing cycles from the graph, so the layouts can work only with trees.
2. doExecute: is the main method, and the only method that must be implemented. Perform your layout here.
3. doPost: allows one to insert post-processing actions.