Chapter 1

Layout

Note:
This chapter has been originally written by Peter Uhnák (i.uhnak@gmail.com)

Layouting is an important aspect of data visualization. It gives us aesthetically pleasing, and more comprehensible presentation, and also allows us by proper structuring to display internal aspects of the presented data.

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
```

This image does not convey much useful information. But by applying RTTreeLayout we get the following (Figure 1.2):

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

```

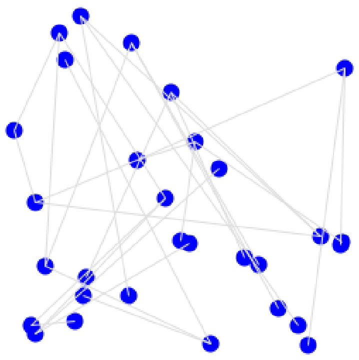


Figure 1.1: Graph with randomly placed nodes.

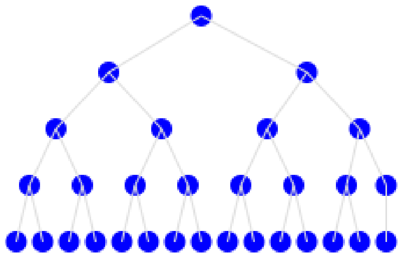


Figure 1.2: Layouted tree.

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

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

By mere repositioning of the elements we can immediately tell that it is a binary tree, what is the root node, and so on, and all that without the need of adding additional elements to the presentation.

There are many other layouting methods, each useful in its own rights.

1.1 Element-based Layouts

Element based layouts do not use edges to present relations between them. Instead they treat all elements based on their size, shape or position within a group. Of course the elements can still have edges.

Circle Layouts

Circle layouts arrange elements in a circle. The order of the elements, as in 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 @ RTLabelled.

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
```

Several properties can be configured for circular layouts:

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

If no options are provided, the algorithm will distribute all the elements evenly in 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 doesn't take into consideration size of the elements; this is enough when the elements are uniform, however if their sizes vary, different layout may be considered.

This is where **Equidistant** and **Weighted** layouts comes to rescue.

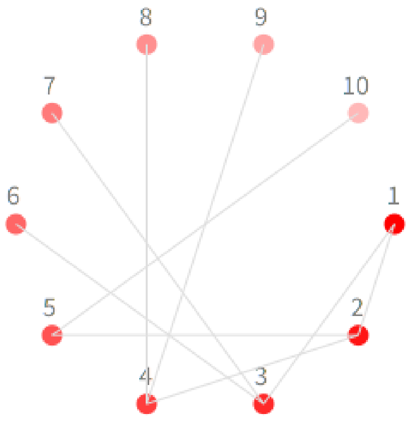
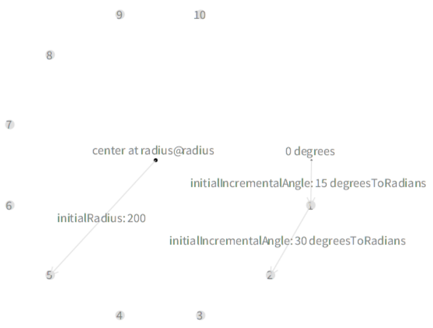


Figure 1.3: Circle layout applied on some elements .



9
10
8
7
center at radius@radius
0 degrees
initialIncrementalAngle: 15 degreesToRadians
6
initialRadius: 200
initialIncrementalAngle: 30 degreesToRadians
5
2
4
3

Figure 1.4: Some options of Circular Layout.

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

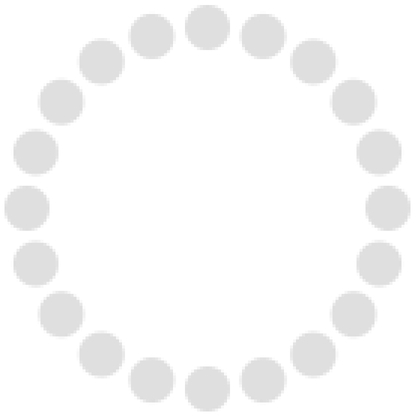


Figure 1.5: Uniform sizes.

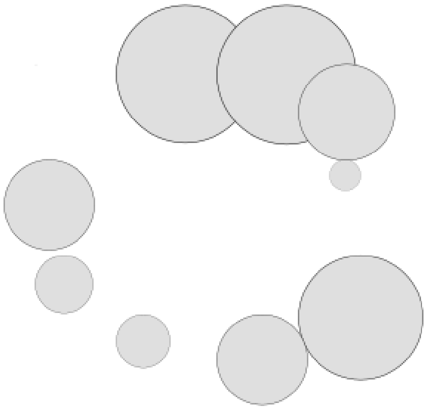


Figure 1.6: Non-uniform sizes.

```
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
```

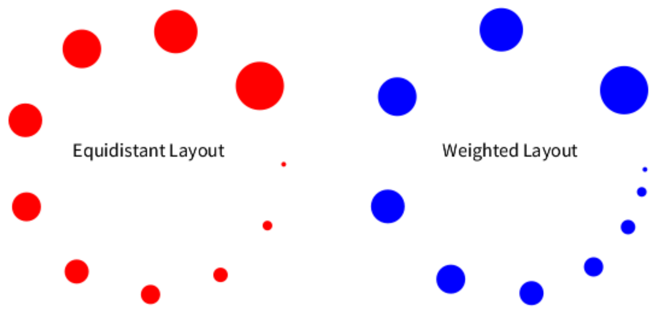


Figure 1.7: Equidistant (left) and Weighted (right) layout with non-uniform sizes.

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

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)).
```

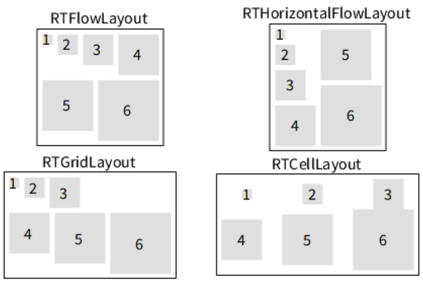


Figure 1.8: Flow and Grid Layouts.

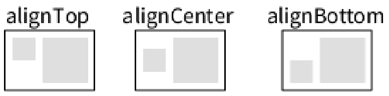


Figure 1.9: RTFlowLayout alignments.

```
            layout new on: group.
        ].

    RTCellLayout new gapSize: 10; on: es.

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

In Flow layouts elements are positioned in each line (column) based on their size, with defined total width of the containing area (maxWidth:). For Grid and Cell layout this limit is instead number of items in the line (lineItemsCount:. By default Flow will attempt to fill roughly rectangular area, while Grid will approximate golden ratio.

Alignment

Cells in Flow layouts can be aligned:

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

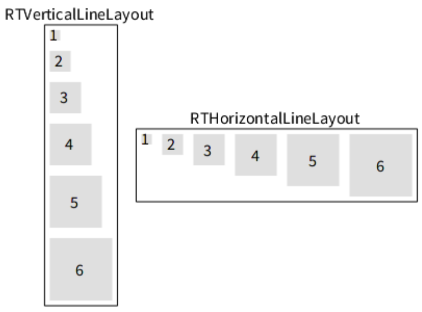


Figure 1.10: RTVerticalLineLayout and RTHorizontalLineLayout.

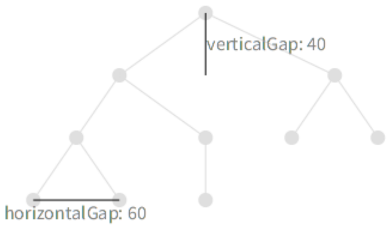


Figure 1.11: RTTreeLayout demonstrating gap sizes.

Line Layouts

Two line layouts are offered: RTVerticalLineLayout and RTHorizontalLineLayout to layout, vertically and horizontally, respectively (Figure 1.10).

1.2 Edge-driven layouts

Edge-driven layouts are an important layout familly: nodes location is determined by the connections between the nodes.

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. Alternative to Tree Layout is Horizontal

Tree Layout, the only difference is root on the left side with growing branches to the 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 in a circular structure. Since each new layer increases the radius of the circle, there is always more and more space available.

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

Dominance Tree Layout

This layout is especially useful for visualizing dependencies and flow charts, since it organizes elements in such 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.
```

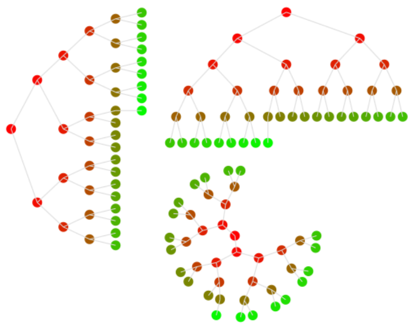


Figure 1.12: Comparison of Horizontal, Vertical and Radial Tree Layouts.

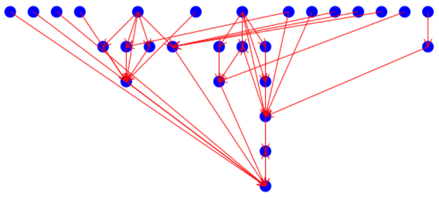


Figure 1.13: RTDominanceTreeLayout showing dependencies between RTShape classes.

```
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 1.14:.

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

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

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
```

Sugiyama

Sugyiama layout is a hierarchical layered layout. It places elements in hierarchical order such that the edge goes from higher layer to lower layer. At the same time the layout will try to minimize the amount of layers and edge crossings (Figure 1.15).

```
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
```

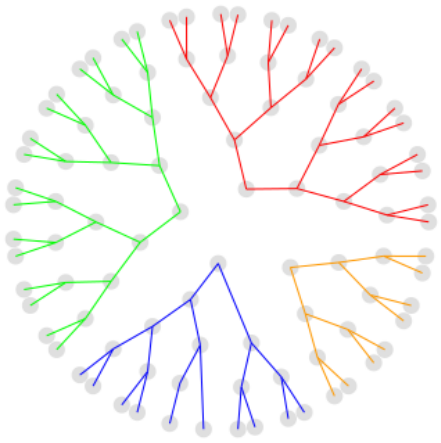


Figure 1.14: Four trees clustered together.

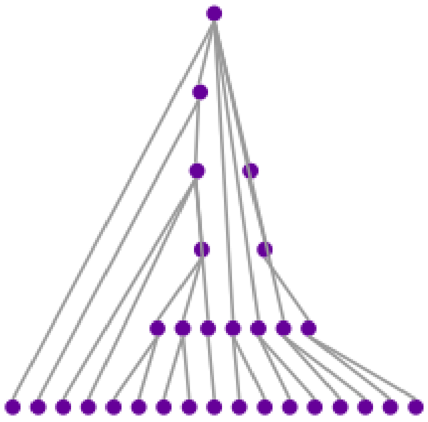


Figure 1.15: Sugiyama layout applied on hierarchy of 'RTLayout' classes.

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.

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

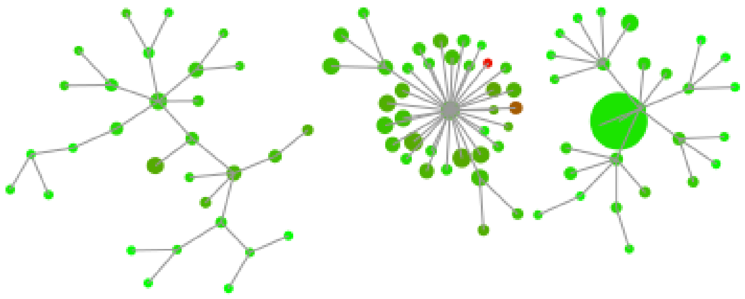


Figure 1.16: RTForceBasedLayout used to layout hierarchy of classes.

```
    withAllSubclasses.

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
```

Rectangle Pack

RTRectanglePackLayout packs all the elements as tightly as possible. It uses 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 view 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.
```

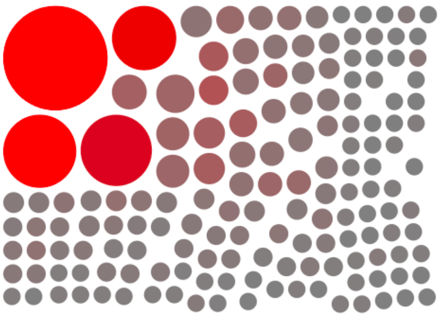


Figure 1.17: Pack of different elements.

```
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
```

1.3 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.
```

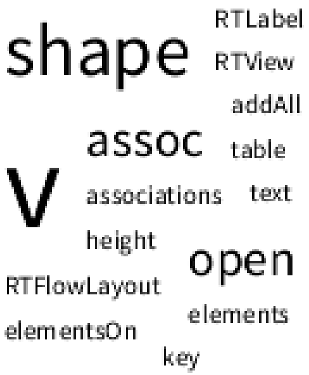


Figure 1.18: RTRectanglePackLayout used to layout a name cloud.

1. doInitialize: can be used for elements 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: can do some finishing touches after the layout has been performed.