Chapter 1

Domain-Specific Visualization Made Easy with Builders

The Roassal visualization engine is a generic purpose engine for visualizing data. Roassal has been designed to easily map any arbitrary Pharo object structure, metrics, properties to visual dimensions.

Frequently, visualizations have the same limitation than software artifact: if not properly designed, a visualization is difficult to extend and reuse in a context that is different from the one for which it had been originally designed. In software engineering, problems traditionally associated with software reuse, composition, and extension are tremendously alleviated by employing domain-specific languages: a reduced and specialized language for a particular application domain lowers the effort of designing and maintaining software.

Roassal supports a generic infrastructure, called *builder*, to efficiently build and reuse visualizations. Builder follows the key principles of domain-specific languages by supporting the definition of *domain-specific visualizations*, *i.e.,* visualizations tailored to a particular domain. Builder offers a set of reusable building blocks to easily define visual shapes, interaction and layout over any arbitrary domain. A builder encapsulates the logic of a visualization and maps a particular domain to visual Roassal elements.

The builder infrastructure is composed of five key classes, which are reviewed along this chapter.

This chapter was written with the participation of Yuriy Tymchuk (yuriy.tymchuk@me.com)

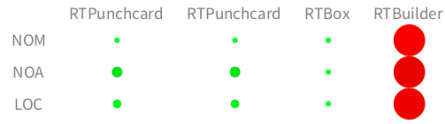

Figure 1.1: Second version of the punchcard builder.

1.1 Punch card example

This section describes the builder infrastructure in a tutorial-like fashion. The example we use throughout this paper is a punch card, as shown in Figure 1.1.

Metrics are vertically located on the left hand side and elements are horizontally located. The figure shows that the element RTBuilder and RTPunchcard have high values for the metrics NOM, NOA, and LOC.

1.2 Defining a builder with RTBuilder

The class RTBuilder translates a user-defined domain into low level instructions using the Roassal API. A builder essentially wraps a Roassal view and offers dedicated (sub-)builders, for edges, nodes, interaction, described in the following sections.

A punchcard is defined mapping objects and metrics to some visual dimensions (*e.g.,* elements size and colors). Computing the metrics over the objects results in a grid. The input of the builder will therefore be objects and metrics.

So far, most of the visualizations have been defined by typing in a playground. The playground is made to easily build script and try out relatively short pieces of code. As soon as "serious" code has to be written, such as a builder, you need to move away from the playground and use the Nautilus System Browser. The System Browser is open from the World menu.

The very first step, as for any code writing in Pharo, you need to define a package. The package Punchcard is created by right-clicking on the package pane (top-left textual pane) in a system browser.

A builder is created by subclass the class RTBuilder. We therefore create a subclass of it, called RTPunchcard :

```
RTBuilder subclass: #RTPunchcard
    instanceVariableNames: 'metrics objects'
    classVariableNames: ''
    package: 'Punchcard'
```

As seen in the chapter about the Pharo language, the class RTPunchcard contains two instance variables, called metrics and objects. These instance variables have to be initialized. Defining an initialize method in RTPunchChard is therefore our next step:

```
initialize
    super initialize.
    objects := OrderedCollection new.
    metrics := OrderedCollection new.
```

All the methods provided in this chapter are defined in the class we have just defined. An end-user defines a visualization by specifying objects and metrics. Two methods have to be provided for that purpose:

```
addMetric: blockOrSymbol named: aName
    metrics add: aName -> blockOrSymbol

addObject: anObject
    objects add: anObject
```

The method renderIn: is key in the builder framework. This is where the visualization is constructed, *i.e.,* Roassal elements are created and the view is filled. The method renderIn: is intended to be overridden when creating a new builder class. Here is a first version of the method renderIn: which displays a list of objects, and computes the metrics for each object:

```
renderIn: aView
    aView add: (RTLabel elementsOn: ' ').
    objects
        do: [ :object | aView add: (RTLabel elementOn: object) ].

    metrics do: [ :assoc |
        aView add: (RTLabel elementOn: assoc key).
        objects do: [ :object |
            | value |
            value := assoc value rtValue: object.
            aView add: ((RTEllipse new size: value) element) ] ].

    RTCellLayout new
        lineItemsCount: objects size + 1;
        on: aView elements
```

Line 1 inserts an empty label, used for the first column. Line 3 - 4 inserts a label for each object. Line 6 - 11 inserts the name of each metric and compute the metrics against the objects. Lines 13 - 15 layout the elements as a grid.

At that stage, the builder may be invoked as:

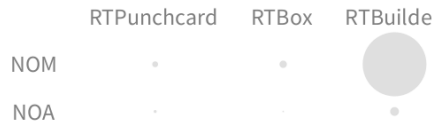

Figure 1.2: First version of the punchcard builder.

```
b := RTPunchcard new.
b addObject: RTPunchcard.
b addObject: RTBox.
b addObject: RTBuilder.
b addMetric: #numberOfMethods named: 'NOM'.
b addMetric: #numberOfVariables named: 'NOA'.
b
```

The objects used in the visualization are three classes: RTPunchcard, RTBox, RTBuilder. For each of these classes, the messages numberOfMethods and numberOfVariables will be sent to return a numerical value.

The produced rendering is given in Figure 1.2. At that stage, the visualization is very simplistic: circles may be excessively large if metric values are high and the exact value behind each circle is not accessible. We address this in the subsequent sections.

1.3 Specifying shapes with RTShapeBuilder

A shape builder is a particular object to configure shapes and is a factory of Roassal elements. The code renderIn: method uses the expression RTEllipse new size: value to define a circle with a size given by each metric value. Size of an ellipse is therefore linear to the value, which is not really optimal in case of large disparities between values.

By using a shape builder, the shape may be defined *externally* from the renderIn: method.

A builder offers the method createShapeBuilder to initialize the shape builder used when creating elements, using elementOn:. In our case, we are interested in having circle shapes of a size given by the element's object model. Overriding the method createShapeBuilder allows one to initialize the shape builder associated to a builder.

```
createShapeBuilder
    | sb |
    sb := super createShapeBuilder.
```

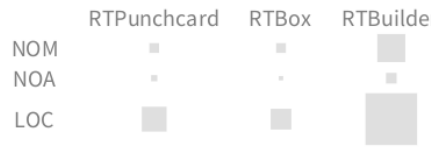

Figure 1.3: Second version of the punchcard builder.

```
    sb ellipse size: #yourself.
    ^ sb
```

The method renderIn: directly uses RTEllipse, at Line 11. Having this use of the shape prevents one from changing the shape. Instead, this method can be rewritten:

```
renderIn: aView
    aView add: (RTLabel elementsOn: ' ').
    objects
        do: [ :object | aView add: (RTLabel elementOn: object) ].

    metrics do: [ :assoc |
        aView add: (RTLabel elementOn: assoc key).
        objects do: [ :object |
            | value |
            value := assoc value rtValue: object.
            aView add: (self elementOn: value) ] "New line"].

    RTCellLayout new
        lineItemsCount: objects size + 1;
        on: aView elements
```

The behavior as shown in the previous example is preserved. However, by using elementOn: to create elements, visual shapes may be particularized by the end-user. Consider the following example:

```
b := RTPunchcard new.
b shape rectangle
    size: [ :v | v sqrt * 3 ].
b addObject: RTPunchcard.
b addObject: RTBox.
b addObject: RTBuilder.
b addMetric: #numberOfMethods named: 'NOM'.
b addMetric: #numberOfVariables named: 'NOA'.
b addMetric: #numberOfLinesOfCode named: 'LOC'.
b
```

Circles have been replaced with boxes. Each box reflects the square root of a metric value and the minimum size is set to 5 pixels. The visualization is still limited. One cannot compare metrics across the different classes. Values have to be normalized for this.

1.4 Data Normalization

The shape and colors of elements may be resized to reflect a particular property when compared with other elements. We call this *normalization*. Consider the following improvement of renderIn:.

```
renderIn: aView
    | el |
    aView add: (RTLabel elementsOn: ' ').
    objects do: [ :object |
        | objElement |
        objElement := RTLabel elementOn: object.
        aView add: objElement ].

    metrics do: [ :assoc |
        aView add: (RTLabel elementOn: assoc key).
        self resetCreatedElements.
        objects do: [ :object |
            | value |
            value := assoc value rtValue: object.
            el := self elementOn: value.
            aView add: el ].
        self normalizer elements: self createdElements. "New line"
        self normalizer build "New line" ].
    RTCellLayout new
        lineItemsCount: objects size + 1;
        on: aView elements
```

The builder keeps track of the objects that are created thanks to the call self elementOn: value. Having the list of created objects is useful to configure the normalizer. In this case, we consider a line as the scope of the normalization.

A more elaborated example is:

```
b := RTPunchcard new.
b normalizer
    normalizeSize;
    normalizeColorUsing: { Color green . Color red }.

b addObject: RTPunchcard.
b addObject: RTPunchcard.
```

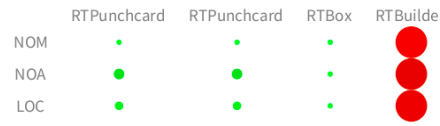

Figure 1.4: Third version of the punchcard builder.

```
b addObject: RTBox.
b addObject: RTBuilder.
b addMetric: #numberOfMethods named: 'NOM'.
b addMetric: #numberOfVariables named: 'NOA'.
b addMetric: #numberOfLinesOfCode named: 'LOC'.
b
```

Result of the example is given in Figure 1.4. Size and color reflect the metric value, which enable an easy comparison.

1.5 Specifying interaction using RTInteractionBuilder

Often a visualization has to offer some interaction facilities to let the user get details or allow for a navigation. The builder class offers the method setUpInteractionFor: that lets the user specify interaction with the elements:

```
renderIn: aView
    | el |
    aView add: (RTLabel elementsOn: ' ').
    objects do: [ :object |
        | objElement |
        objElement := RTLabel elementOn: object.
        self setUpInteractionFor: objElement. "New line"
        aView add: objElement ].

    metrics do: [ :assoc |
        aView add: (RTLabel elementOn: assoc key).
        self resetCreatedElements.
        objects do: [ :object |
            | value |
            value := assoc value rtValue: object.
            el := self elementOn: value.
            aView add: el ].
        self normalizer elements: self createdElements.
        self normalizer build ].
    RTCellLayout new
```

```
        lineItemsCount: objects size + 1;
        on: aView elements
```

Similarly to shapes and normalizer, a list of interactions may be set when defining a visualization, shown as follows:

```
b := RTPunchcard new.
b interaction
    popupText;
    highlightColored: Color blue trans;
    action: #browse.

b normalizer
    normalizeSize;
    normalizeColorUsing: { Color green . Color red }.

b addObject: RTPunchcard.
b addObject: RTBox.
b addObject: RTBuilder.
b addMetric: #numberOfMethods named: 'NOM'.
b addMetric: #numberOfVariables named: 'NOA'.
b addMetric: #numberOfLinesOfCode named: 'LOC'.
b
```

Right-clicking on the class name will open the menu to open a system browser on that class (*i.e.,* the result of sending browse to a class).