# Mondrian: Scripting Visualizations

Michael Meyer, Tudor Gîrba

Software Composition Group, University of Bern, Switzerland

## 1 Introduction

Visualization is representing data into pictures for supporting reasoning. For the interpretation to be as easy as possible, we need to be as close as possible to the original data.

The primary focus of our approach is to offer the *programmer* the possibility of visualizing his data model while using his preferred environment and tools. That is why, we have built Mondrian, an engine that puts all the emphasis on providing the needed basic pieces and that places the control in the hand of the programmer.

## 2 Scripting visualizations with Mondrian

In this section we give a simple step-by-step example of how to script visualizations using Mondrian. The example builds on a small *model* of a source code with 32 classes. The task we propose is to provide a simple overview of the classes in the system.

**Creating a view.** To make the things as easy as possible for the programmer, we have designed Mondrian to work like a view the programmer paints. Hence, the first thing we do is to create an empty view:

```
view := ViewRenderer new.
view open.
```

**Adding nodes.** Suppose we can ask the model object for the classes. We can add those classes to the visualization by creating a node for each class, where each node is represented as a Rectangle.

```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder width: #NOA; height: #NOM;
                            liniarColor: #LOC within: model classes).
view open.
```
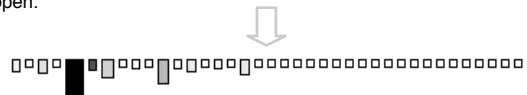


In the case above, NOA, NOM and LOC are methods in the object representing a class and return the value of the corresponding metrics.

**Adding edges.** To show how classes inherit from each other, we can add an edge for each inheritance relationship. In our example, supposing that we can ask the model for all the inheritance objects between the classes in the model, given an inheritance object, we will create an edge between the node holding the superclass and the node holding the subclass. Like in the case of the nodes, when specifying the shape, we made reference to methods that are defined in the inheritance object:

```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder width: #NOA; height: #NOM;
                        liniarColor: #LOC within: model classes).
view edges: model inheritances
    using: (Line from: #superclass to: #subclass).
view open.
```

**Layouting.** To make the above graph understandable, we layout the nodes in a tree. By default, the nodes are arranged in a horizontal line.

```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder width: #NOA; height: #NOM;
                                    liniarColor: #LOC within: model classes).
view edges: model inheritances
    using: (Line from: #superclass to: #subclass).
view layout: TreeLayout new.
view open.
```
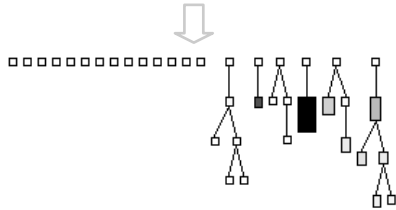


**Nesting.** To obtain more details for the classes, we would like to see which are the methods inside. To nest we specify for each node the view that goes inside. Supposing that we can ask each class in the model about its methods, we can add those methods to the class by specifying the view for each class:
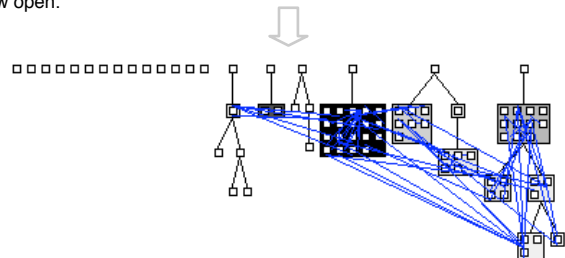
```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder liniarColor: #LOC within: model classes).
    forEach: [:eachClass |
        view nodes: eachClass methods using: Rectangle withBorder.
        view layout: CheckerboardLayout new.
    ].
view edges: model inheritances
    using: (Line from: #superclass to: #subclass).
view layout: TreeLayout new.
view open.
```
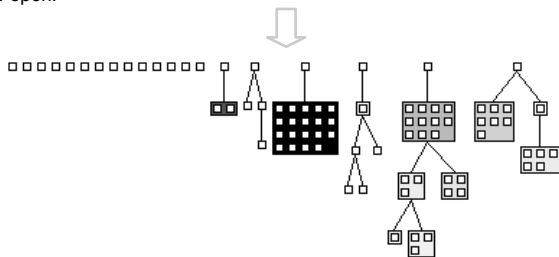


**Adding inter-edges.** The edges are created by specifying the *from* and the *to* objects. Because we can have the objects at various levels of nesting, it is important to specify the location from where the lookup of the objects should start. For example, if we want to add invocations edges between the methods, and if we suppose that we can ask the model object about

those invocations we can add them like we added inheritances:

```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder liniarColor: #LOC within: model classes).
    forEach: [:eachClass |
        view nodes: eachClass methods using: Rectangle withBorder.
        view layout: CheckerboardLayout new].
view edges: model invocations
    using (Line from: #invokedBy: to: #invoked).
view edges: model inheritances
    using: (Line from: #superclass to: #subclass).
view layout: TreeLayout new.
view open.
```



**Decorating shapes.** By default, the sense of the edges is shown by the convention that edges leave from the bottom-right of the node and end on the top-left of the node. However, when the user wants to specify an arrow at the end of the line, he can use decorations. For example, when we want to show the arrows on the inheritances all we have to do is to decorate the Line with an Arrow. Decorations can be applied to any figure. In fact, Rectangle withBorder is implemented as Rectangle new decoratedWith: Border new.

```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder liniarColor: #LOC within: model classes).
    forEach: [:eachClass |
        view nodes: eachClass methods using: Rectangle withBorder.
        view layout: CheckerboardLayout new
        view edges: model invocations
            using (Line from: #invokedBy to: #invoked)].
view edges: model inheritances
    using: ((Line from: #superclass to: #subclass)
            decoratedWith: Arrow new).
view layout: TreeLayout new.
view open.
```
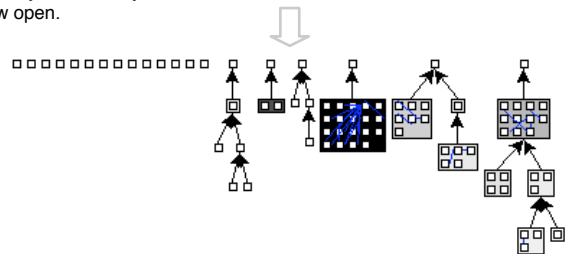
# 3 Mondrian prototype

**Design.** Figure 1 reveals the core structure of our framework. Each Figure holds an Object. The Figures are implemented directly on top of the graphical framework, but they hold no specific value for the visualization. The entire responsibility of what gets drawn belongs to the Shape, which is a specification of how the Figure should be displayed on a canvas (via displayFigure:on:). The Shape holds no state, and thus it is possible to share a Shape between several nodes or edges. Furthermore, each Figure is a graph and holds several other Figures, and it also knows the Layout to be applied on its children.
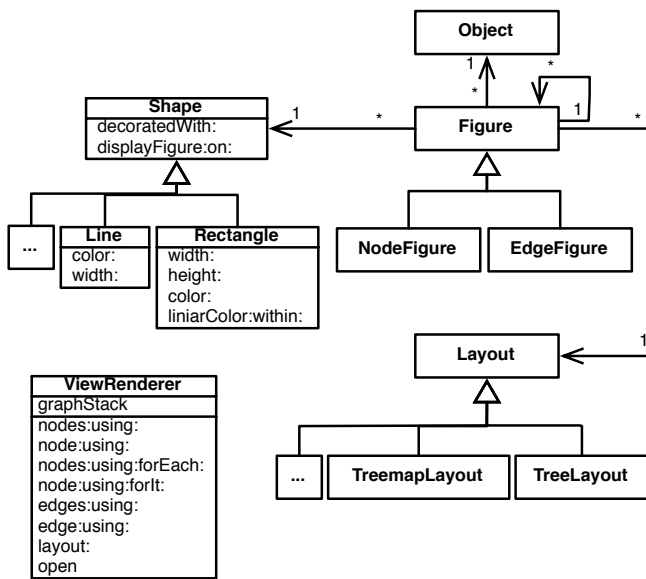


Figure 1: The internal model of Mondrian.

To make the script easy to write, we have designed the ViewRenderer to hide the internal details of the model. The structure of the ViewRenderer was inspired by the Renderer of Seaside.

**User Interface.** Having an explicit model of the visualization, allowed us to create an editor that maps model characteristics to the visualization. For example, the below image we show a screenshot of such an editor that makes use of the *a priori* knowledge of the structure of the data model and for a given selected figure, generates an editor that asks for each representation shape the different visualization characteristics.
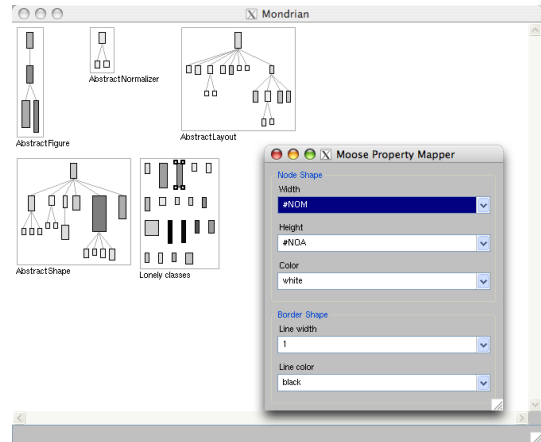


Figure 2: Mappings editor based on Moose.

In our example, the view presents the hierarchy of Mondrian classes and the editor is built on top of the Moose environment and allows us to map metrics to the Rectangle and to the Border selected.

Visualization specification should be instance based. For example, in the next screenshot we show the same visualization as in the previous one, only now the ViewRenderer is shown using a Class Blueprint.
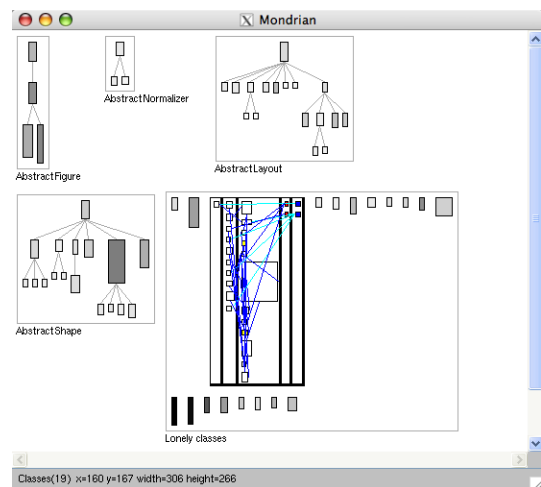


Figure 3: Instance based visualization.

3