Visualizing Software using Mondrian and Moose

Alexandre Bergel
University of Chile, Santiago, Chile
INRIA, France

http://www.bergel.eu

1 Installing Mondrian

Mondrian run on Visualworks[1], Squeak[2] and Pharo[3]. In this manual, Pharo will be used as the running platform for Mondrian.

If you do not have Pharo installed, then you ought to go to the Pharo website and follow the installation instructions. Then, open a workspace (World menu, `Tools/Workspace`). Type the following incantation:

```
ScriptLoader new installer ss project: 'Mondrian';
  install: 'MondrianLoader'.
ScriptLoader perform: #loadMondrian
```

select with the mouse, and press do-it. This should load the latest version of Mondrian. You're ready to jump to the next section.

Note on the Pharo version. Although a great of effort has been spent on making the Pharo/Squeak version compatible with the Visualworks version, some differences may exist. If you find one that is not referenced in this manual, please drop few lines to the developers.

A bit of history. Mondrian has been originally developed on Visualworks by Michael Meyer and Tudor Girba. A port has been initiated by Lukas Renggli. Alexandre Bergel took over and made significantly improved the Pharo version.

How to reach us. Currently, developers of Mondrian are:

- Alexandre Bergel: alexandre.bergel@inria.fr
- Tudor Girba: girba@iam.unibe.ch

The best way is to reach us is probably the moose mailing list. If you have an interest in Mondrian, Moose and other Smalltalk-based re- and reverse engineering techniques, https://www.iam.unibe.ch/mailman/listinfo/moose-dev is the place you want to go right away.

[1] www.cincomsmalltalk.com
[2] www.squeak.org
[3] www.pharo-project.org

2 First step

2.1 A first example

Open a Workspace, and do-it the following code excerpt:

```
| view |
view := MOViewRenderer new.
view nodes: (1 to: 20).
view open
```

You should see a new window with about 20 small boxes lined up in the top left corner. You've just rendered the numerical set between 1 and 20.

In the remaining of this section, we will intensively use Smalltalk reflection to make compelling examples. Let's add a variable contains all subclasses of the class Collection.

```
|view classes |
classes := Collection withAllSubclasses.
view := MOViewRenderer new.
view nodes: classes.
view open
```

If you execute the code given above you will not notice any graphical difference except the number of small boxes in the top left corner. This is expected since there are more than 20 subclasses of Collection in Pharo. All this boxes have the exact same shape.

2.2 Working with the Easel

So far, a visualization was rendered by explicitly instantiating MOViewRenderer and then sending the message open. This way of building visualization scripts does not shine by its interactivity with the designer. It is very close to the edit/compilation endless loop promoted by old-fashion programming languages such as C, C++ and Java (to some extend).

A nice dose of interactivity is accessible through an *easel*. An easel may be open by "doiting" the expression MOEasel open. A window similar to Figure 1 will popup.

Within an easel, you do not need to instantiate a view renderer and opening it up. Simply write your script without taking care of this initialization.

Note that you can press Cmd-S (as you would accept a Smalltalk method) to render the view.

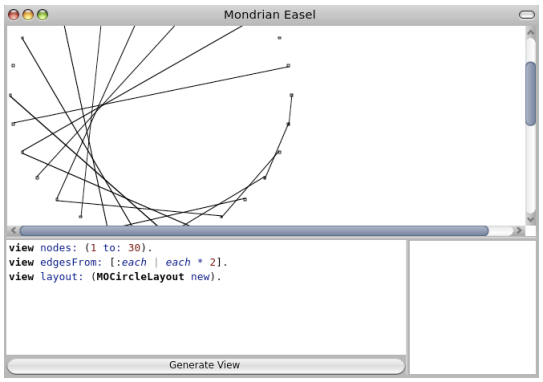

Figure 1. Scripting visualization with the Mondrian Easel.

3 The System Complexity View

Let's continue the example started in Section 2.1. All the boxes are similar so far. Let's make each small boxes reflect the shape of the class. For this, we need to add a shape. This shape will be a rectangle in which the height and the width is particularized. We enhance the previous example. Enter the following in the scripting pane of the Easel:

```
|classes |
classes := Collection withAllSubclasses.
view shape rectangle
  height: [:cls | cls methods size ];
  width: [:cls |  cls instVarNames size * 5 ]).
view nodes: classes.
```

For each class, a box is displayed. The width and the height of a box is computed by evaluating the provided blocks. We recall that a block in Smalltalk acts as a first-class function. The block is evaluated by providing an element, which is bound to cls. The block body may then be evaluated[4].

Each box is distinct from others by having a width and a height corresponding to the class it represents. The height of a box represents the number of methods. The multiplication stems from the fact that a class has comparatively a low number of instance variables.

Nodes do not have knowledge about how and when they will be rendered on the screen. A shape is therefore necessary. A number of shapes may be found in the Mondrian distribution.

Nodes are draggable. Moving the mouse cursor over a box will display the name of the class being represented.

Edges may be trivially added.

```
|classes |
classes := Collection withAllSubclasses.
view shape rectangle
  height: [:cls | cls methods size ];
  width: [:cls |  cls instVarNames size * 5 ]).
view nodes: classes.
view edgesFrom: #superclass.
view treeLayout.
```

Edges are built from superclass links existing between subclasses of Collection. To really understand how edges are built, let's have a look at the definition of edgesFrom::

```
MOViewRenderer>> edgesFrom: aBlockOrSelector
  | domain |
  domain := self root nodes collect: [:each |each model ].
  self edges: domain from: aBlockOrSelector to: #yourself
```

Edges exists only for nodes that belong to what has been set as view nodes. For example, having view edgesFrom: [:each | Object] will not draw any edge since the class Object hasn't been provided in the view nodes: instruction.

We also added a layout. A layout is in charge of properly positioning the node according to some algorithm. The method treeLayout is defined as:

```
MOViewRenderer>>treeLayout
  ^ self layout: MOTreeLayout new.
```

A number of layout are provided in the Mondrian-Layouts class category.

So far, only 4 dimensions of a box have been explored: x, y, width, height. Colors may be easily added. Consider the following example that shows in blue all classes having the word *Array* in their name, and in green the ones having *Dictionary*.

```
|classes |
classes := Collection withAllSubclasses.
view shape rectangle
  height: [:cls | cls methods size ];
  width: [:cls |  cls instVarNames size * 5 ]).
  fillColor: [:cls |  (cls name matchesRegex: '.*Dictionary.*')
                ifTrue: [ Color green ]
                ifFalse: [ (cls name matchesRegex: '.*Array.*')
                              ifTrue: [ Color blue ]
                              ifFalse: [ Color white ] ] ].
view nodes: classes.
view edgesFrom: #superclass.
view treeLayout.
```

Color intensity may also be proportional to a numerical value. Consider:

```
|  classes numberOfLinesOfCode |
classes := Collection withAllSubclasses.
numberOfLinesOfCode := Dictionary new.
classes do: [:cls |
  numberOfLinesOfCode at: cls put:
   (cls methods
    inject: 0
    into: [:sum :aMethod |
          sum + aMethod getSourceFromFile lineCount ]) ].
view shape rectangle
  width: [:cls |  cls instVarNames size * 5];
  height: [:cls |  cls methods size];
  linearFillColor: [:cls |  numberOfLinesOfCode at: cls ]
        within: classes.
view nodes: classes.
view edgesFrom: #superclass.
view treeLayout
```

[4]Blocks are evaluating by sending the message value: to it. Note that the Mondrian framework takes care of evaluating provided blocks. There is nothing for you to do then.

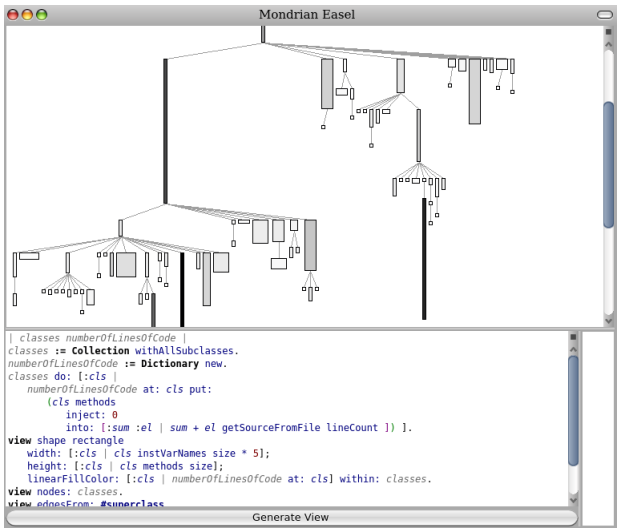

Figure 2. The System Complexity View on the collection framework in Pharo.

The message sent to the shape is linearFillColor: aBlock within: domains. The provided block should return a numerical value. In this case, we compute the number of lines of code of a class by summing all method lengths. The message methods returns a collection of CompiledMethod instances. Sending getSourceFromFile to a compiled method returns its source code as a String. Sending lineCount to a string returns the number of lines represented by its receiver. The method inject:into: iterates over a collection using an accumulator[5].

The view you obtained for the collection framework in Pharo (Figure 2) is called *System Complexity View* [LD03, Lan03].

4 Visualizing a Moose model

So far, we have visualized part of the Pharo library only. This section gives a very brief and informal introduction on the FAMIX meta model. Having a rudimentary background on how Moose models are composed of is necessary to visualize them.

Moose is a platform for software analyzes. FAMIX is an extensible meta-model used by Moose to represent program source code. We will concentrate on the subset of FAMIX that is closely linked to object-orientation.

To see what a model looks like, open a Moose open (just evaluate MoosePanel open in a workspace). Open a Moose finder on the LAN sample model. The LAN model is made of 679 FAMIX entities, comprising packages, accesses, invocations, classes, methods, just to name a few.

Select the line *All famixclass (38 FAMIXClasses)* and select the *Complexity* tab. You should now be familiar with what you see.

[5] As an example, the following code compute the factorial of 20: (1 to: 20) inject: 1 into: [:accu :el | accu * el]

Right click on the *All famixclass (38 FAMIXClasses)* line, and open the group of classes in a Mondrian easel. You can now happily exercise your freshly learned skills on creating visualization. In the script panel, a variable classGroup is available. This variable has been set by the Moose finder and contains a collection of FAMIXClass instances.

Try to reproduce the System Complexity View on this model. For a given FAMIXClass, you can access:

- superclass by sending the message superclass
- number of lines of code by sending the message numberOfLinesOfCode
- number of methods with numberOfMethods
- number of attributes (*i.e.,* instance variables) with numberOfAttributes

5 A note on edges

A renderer offers a number of convenient methods. Browse the class MOViewRender to have the complete list. Edges are declared from a number of elements, and two blocks that are used to compute from the elements the starting and ending point. Consider the following script:

```
view nodes: (1 to: 30).
view edges: (1 to: 30) from: #yourself to: [:aNumber |aNumber // 5 ].
view dominanceTreeLayout
```

The first line declares 30 nodes. The second line declares an edge for each of the nodes. The edge begins from the node itself (*i.e.,* having #yourself is equivalent to [:anElement | anElement]) and ends in another number computed with aNumber // 5. The method // aNumber is defined on the class Number. It returns the remaining of a division between the receiver and the division.

- edges: aCollection from: aFromBlock to: aToBlock – Draw at most *one* edge for each node. Two edges cannot start from a unique node.
- edges: aCollection from: aFromBlock toAll: aToBlock – Note that aToBlock must returns a collection. This method is useful to draw more than one edges starting from a node.
- edgesFrom: aSelector – Equivalent to edges: nodes from: aBlockOrSelector to: #yourself

The complete list may be find in the definition of MOViewRenderer.

A typical usage of edges:from:to: may be found in the following code excerpt:

```
view nodes: (1 to: 5).
view shape arrowedLine.
view edges: {1 -> 2. 1 -> 5 . 4 -> 3} from: #key to: #value.
view circleLayout
```

The expression {1 -> 2. 1 -> 5 . 4 -> 3} defines an array of associations. An association is obtained by sending the message -> to an object with a second object as argument. The receiver and the argument of this message may be retrieved by using the message key and value. Look for the implementor of -> to see how this works. Implementors may be obtained by pressing Cmd-m or Alt-m when selecting the a message name in a workspace.

6 Nesting

View nesting is supported by using the message nodes:forEach:. Within an easel open a moose model, try:

```
view nodes: classGroup forEach: [:each |
  view nodes: each methods.
  view gridLayout ].
view gridLayout
```

You can also insert label using a new shape:

```
view shape rectangle withoutBorder.
view nodes: classGroup forEach: [:each |
  view node: each forIt:
        [ view nodes: each methods .
          view gridLayout ] .
  view shape label text: #name.
  view node: each.
  view verticalLineLayout center ].
view gridLayout
```

7 A small exercise now

1. Open an easel on all famix classes of a Moose model
2. Use colors to distinguish stub and no-stub classes
3. Use colors to distinguish overridden methods from not overridden methods
4. Refine your visualization according to your imagination

8 Interactions

Event handlers may be attached to nodes. Event are actions triggered by the mouse (e.g., moving the cursor over a node, pressing a button, pressing a key...). In Pharo, events use announcement. Handler may be trivially attached to nodes as:

```
|view classes |
classes := Collection withAllSubclasses.
view := MOViewRenderer new.
view interaction
  on: MOMouseDown do: [:ann | ann element browse ];
  popupText.
view shape:
  (MORectangleShape new
        height: [:cls | cls methodDict size max: 5];
        width: [:cls | cls instVarNames size * 5 max: 5]).
view nodes: classes.
view edgesFrom: #superclass.
view treeLayout. view open
```

Pressing a box will now open a system browser. Most of the commonly events are handled. Unfortunately, some events related to the OS may be useful to grab in some situation. Window switching is one of them. For example, the zooming (accessible through the key + and -) is set in the method MOViewRenderer>>setDefaultHandler:

```
MOViewRenderer>>open
  self root interaction on: MOKeyDown do: [:ann |
        ann character asInteger = 24 ifTrue: [ self root increaseZoom ].
        ann character asInteger = 27 ifTrue: [ self root decreaseZoom ].
        systemWindow changed ].
```

In some case, one may want to change the shape of a node when a particular action occurs. This could be changing the color of a node when pressing on it for example. The method copyShapeAndDo: is useful for that purpose. Consider the following example:

```
view on: MOMouseEnter do: [:ann |
  ann element copyShapeAndDo: [:shape | shape fillColor: Color blue].
  view updateWindow].

view on: MOMouseLeave do: [:ann |
  ann element copyShapeAndDo: [:shape | shape fillColor: Color white].
  view updateWindow].
```

The updateWindow needs to be sent to the view in order to refresh the windows content.

9 Links

- Great Smalltalk manual: squeakbyexample.org
- Moose and associated tools: moose.unibe.ch

References

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Bern, May 2003.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.