Chapter 1

Scripting Visualizations with Mondrian

Giving a meaning to a a large amount of data is challenging without adequate tools. Textual outputs are known to be limited in their expression and interactions.

Mondrian is a Domain Specific Language to script visualizations. Its latest implementation runs on top of Roassal (see Chapter **??**). It is made to visualize and interact with any arbitrary data, defined in terms of objects and their relationships. Mondrian is commonly employed in software assessment activities. Mondrian excels at visualizing software source code. This chapter introduces Mondrian's principles and describes its expressive commands to quickly make up your data. After its reading, you will be able to create interactive and visual representation.

1.1 Installation and first visualization

Mondrian is available via a Metacello configuration. Just open a workspace and type:

```
```

A First Visualization

You can get a first visualization by entering and executing the following code in a workspace. Each of these parts will be explain later.

```
| view |
view := ROMondrianViewBuilder new.
view shape rectangle
   width:  [ :cls | cls numberOfVariables * 5 ];
   height: #numberOfMethods;
   linearFillColor: #numberOfLinesOfCode within:  Collection withAllSubclasses.

view interaction action: #browse.

view nodes: ROShape withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
view open
```

1.2 Starting with Mondrian

Stéf ►I pasted the part from the roassal chapter◄ Stéf ►Mondrian is an older visualization framework developed by D. Girba and XXX and maintained by A. Bergel. It influenced the design of Roassal. It supports visualization scripting based on a dedicated DSL: the DSL is based on a stack model where elements on the top can be parametrized. Roassal is designed to be a flexible framework and it is not tight to a DSL. Now Roassal can be extended to support Mondrian-like DSL scripting. This is what we present in this Section.◄

A ROMondrianViewBuilder models the Mondrian DSL[1]. It is mostly compatible with the original Mondrian language.

A ROMondrianViewBuilder internally contains an instance of a ROView, called raw view. Its accessor is raw. All scripting using the ROMondrianViewBuilder result in creating ROElements with the shapes and interactions set by the script, and added to the raw view. To start a visualization with the builder, you can use the following code:

```
view := ROMondrianViewBuilder new.
view open.
```

A Mondrian builder can also be initialized with an instance of a ROView. However it is important to understand that this is not required, as the builder by default will create its own raw view. When working with the builder, is it possible to use the Mondrian DSL, sending messages to an instance of the ROMondrianViewBuilder, or directly with the raw view.

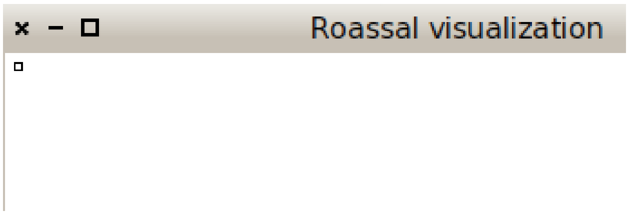
```
rawView := ROView new.
view := ROMondrianViewBuilder view: rawView.
view open.
```

[1] http://www.moosetechnology.org/tools/mondrian

A small summary of the Mondrian DSL is offered here. To more detailed information, please refer to the dedicated Moose book chapter [2].

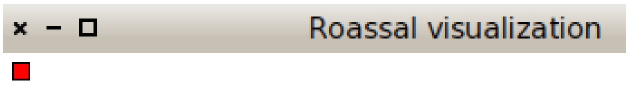
To add a node to the visualization, which be internally translated as a ROElement later on, use the selector node: with the object you want to represent on the instance of the builder.

```
view := ROMondrianViewBuilder new.
view node: 1.
view open.
```



To define shapes, use the shape message followed by the desired shape with its characteristics, before the node or nodes definition. This will locally define the shape for the nodes.

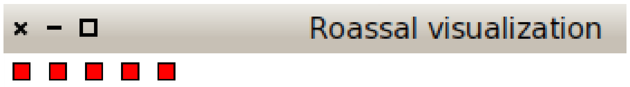
```
view := ROMondrianViewBuilder new.
view shape rectangle
   size: 10;
   color: Color red.
view node: 1.
view open.
```



By using the nodes: message with a collection of objects you can create several nodes.

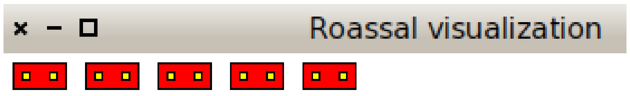
[2]http://themoosebook.org/book/internals/mondrian

```
view := ROMondrianViewBuilder new.
view shape rectangle
   size: 10;
   color: Color red.
view nodes: (1 to: 5).
view open.
```



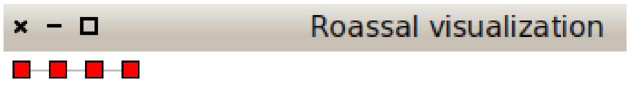
If the node or nodes have nested nodes, use the node:forIt: or nodes:forEach: message to add them. The second parameter is a block which will add the nested nodes, as the following code shows:

```
view := ROMondrianViewBuilder new.
view shape rectangle
   size: 10;
   color: Color red.
view
   nodes: (1 to: 5)
   forEach:[:each |
      view shape rectangle
         size: 5;
         color: Color yellow.
      view nodes: (1 to: 2).
   ].
view open.
```



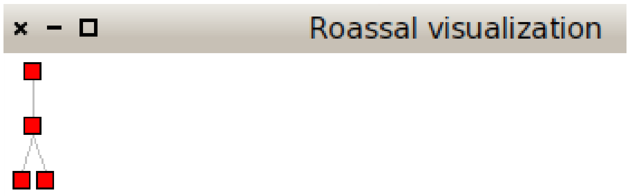
It is possible to create edges by using the edgesFromAssociations: message with a collection of associations between the model of the nodes.

```
view := ROMondrianViewBuilder new.
view shape rectangle
   color: Color red.
view nodes: (1 to: 4).
view
   edgesFromAssociations: (Array with: 1-> 2 with: 2 -> 3 with: 2 -> 4).
view open.
```



Similar to the Collection hierarchy example we need an appropriate layout. By default the builder applies a horizontal line layout and we need a tree layout. We use the treeLayout to apply it.

```
view := ROMondrianViewBuilder new.
view shape rectangle
   size: 10;
   color: Color red.
view nodes: (1 to: 4).
view edgesFromAssociations: (Array with: 1-> 2 with: 2 -> 3 with: 2 -> 4).
view treeLayout.
view open.
```



The Collection Hierarchy example

The Mondrian DSL allows a simpler scripting to the Collection hierarchy visualization than the one constructed through the chapter. By setting how each element and edge must be created, it is not necessary for us to create them by hand. The following code can be replaced for the earlier version:

```
view := ROMondrianViewBuilder new.
view shape rectangle
   width: [ :cls | cls instVarNames size ];
   height: [ :cls | cls methods size ].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
view open.
```

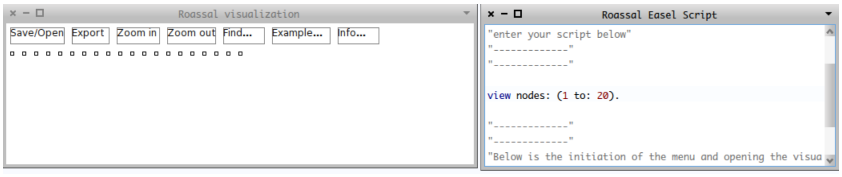
There are essentially two ways to work with Mondrian, either using the easel or a view renderer. The easel is a tool in which users may interactively and incrementally build a visualization by means of a script. The easel is particularly useful when prototyping. MOViewRenderer enables a visualization to be programmatically built, in a non-interactive fashion. You probably want to use this class when embedding your visualization in your application.

We will first use Mondrian in its easiest way, by using the easel. To open an easel, you can either use the World menu (it should contains the entry "Mondrian Easel") or execute the expression:

```
ROEaselMorphic open.
```

In the easel you have just opened, you can see two panels: the one on top is the visualization panel, the second one is the script panel. In the script panel, enter the following code and press the *generate* button:

```
view nodes: (1 to: 20).
```

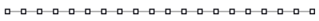


```
"enter your script below"
"-------------"
"-------------"

view nodes: (1 to: 20).

"-------------"
"-------------"
"Below is the initiation of the menu and opening the visua
```

You should see in the top pane 20 small boxes lined up in the top left corner. You have just rendered the numerical set between 1 and 20. Each box represents a number. The amount of interaction you can do is quite limited for now. You can only drag and drop a number and get a tooltip that indicates its value. We will soon see how to define interactions. For now, let us explore the basic drawing capabilities of Mondrian.

We can add edges between nodes that we already drawn. Add a second line:

```
view nodes: (1 to: 20).
view edgesFrom: [ :v | v * 2 ].
```



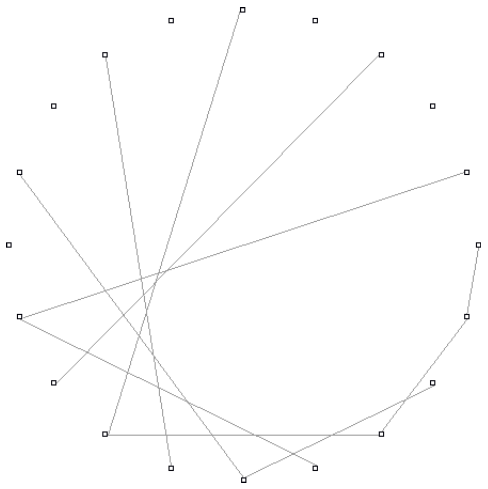
Each number is linked with its double. Not all the doubles are visible. For example, the double of 20 is 40, which is not part of the visualization. In that case, no edge is drawn.

The message edgesFrom: defines one edge per node, when possible. For each node that has been added in the visualization, an edge is defined between this node and a node lookup from the provided block.

Mondrian contains a number of layouts to order nodes. We use the circle layout:

```
view nodes: (1 to: 20).
view edgesFrom: [ :v | v * 2 ].
view circleLayout.
```

The visualization you obtain is:

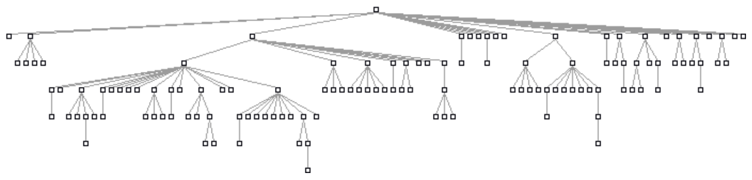


In the subsequent section we will visualize software code. Visualizing source code is often employed to discover patterns, useful when assessing code quality.

1.3 Visualizing the collection framework

We will now visualize Smalltalk classes. In the remaining of this section, we will intensively use the reflective capability of Pharo to introspect the collection class hierarchy. This will serve as compelling examples. Let's visualize the hierarchy of classes contained in the Collection framework:

```
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```



We have used a tree layout to visualize Smalltalk class hierarchies. This layout is particularly adequate since Smalltalk is single-inheritance oriented. Collection is the root class of the Smalltalk collection framework library. The message withAllSubclasses returns the list of Collection and its subclasses.

Classes are ordered vertically along their inheritance link. A superclass is above its subclasses.

1.4 Reshaping nodes

Mondrian visualizes graph of objects. Each object of the domain is associated to a graph element, a node or an edge. Graph element are not aware of their graphical representation. Graphical aspect is given by a shape.

So far, we have solely use the default shape to represent node and edges. The default shape of a node is a five pixels wide square and the default shape of an edge is a thin straight gray line.

A number of dimensions defines the appearance of a shape: the width and the height of a rectangle, the size of a line dash, border and inner colors, for example. We will reshape the nodes of our visualization to convey more information about the internal structure of the classes we are visualizing. Consider:

```
view shape rectangle
   width: [ :each | each instVarNames size * 3 ];
   height: #numberOfMethods.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

Figure 1.1 shows the result. Each class is represented as a box. The Collection class, the root of the hierarchy, is the top most box. The width of a class tells about the amount of instance variables it has. We multiply it by 3 for more contrasting results. The height tells about the number of methods. We can immediately spot classes with many more methods than others:

Collection, SequentiableCollection, String, CompiledMethod. Classes with more variables than others are: RunArray and SparseLargeTable.

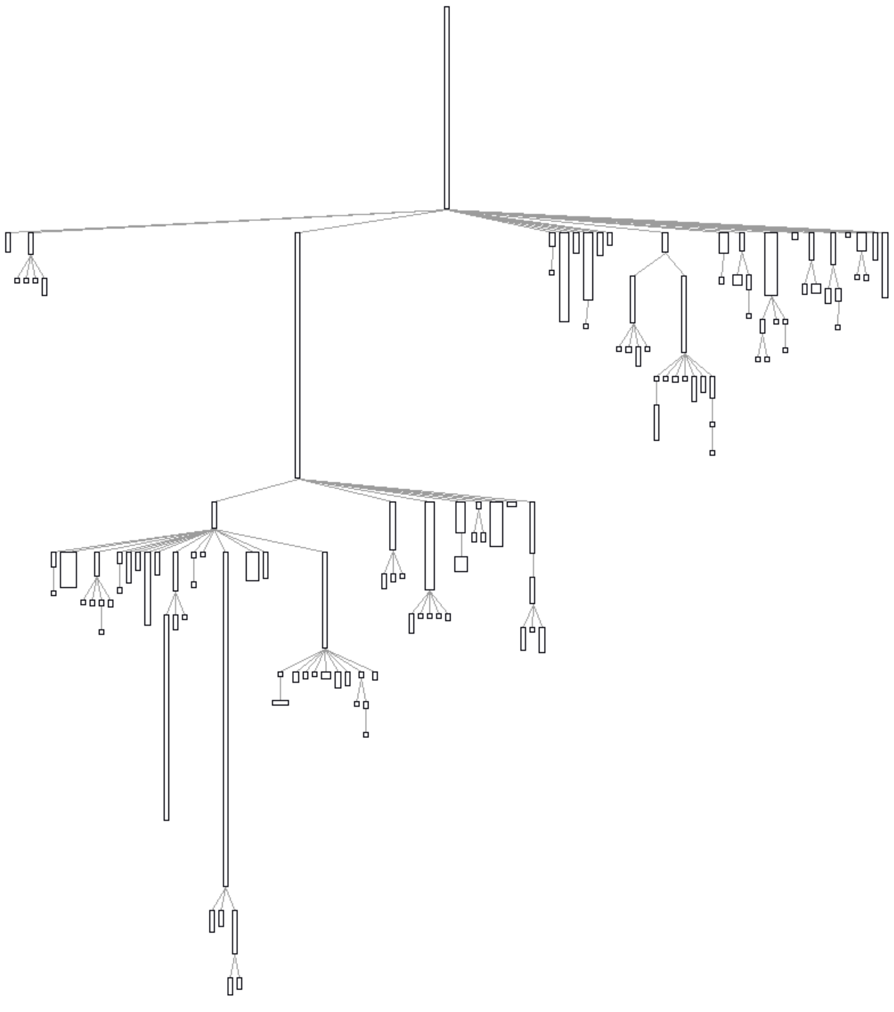


Figure 1.1: The system complexity for the collection class hierarchy.

1.5 Multiple edges

The message edgesFrom: is used to draw one edge at most per node. A variant of it is edges:from:toAll:. It support the definition of several edges starting from a given node. Consider the dependencies between classes. The script:

```
view shape rectangle
   size: [:cls | cls referencedClasses size ];
   withText.
view nodes: ArrayedCollection withAllSubclasses.
view shape arrowedLine.
view
   edges: ArrayedCollection withAllSubclasses from: #yourself toAll: #referencedClasses.
view circleLayout
```

The obtained visualization is given in Figure 1.2.

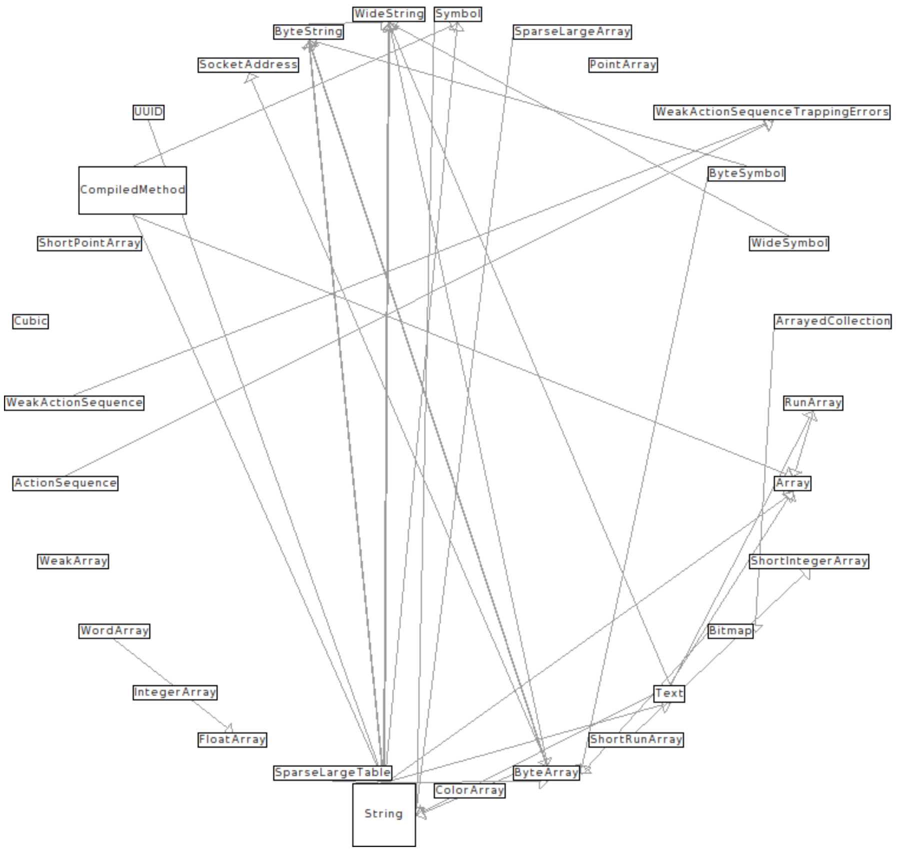


Figure 1.2: Direct references between classes.

String and CompiledMethod clearly shows up. These two classes contains many references to other classes. We also see that text: makes a shape contain a text.

Mondrian provides a whole bunch of utility methods to easily create elements. Consider the expression:

```
view edgesFrom: #superclass
```

edgesFrom: is equivalent to edges:from:to: :

```
view edges: Collection withAllSubclasses from: #yourself to: #superclass.
```

itself equivalent to

```
view
 edges: Collection withAllSubclasses
 from: [ :each | each superclass ]
 to: [ :each | each yourself ].
```

1.6 Colored shapes

A shape may be colored in various different way. Node shapes understand the message fillColor:, textColor:, borderColor:. Line shapes understands color:. Let's color the visualization of the collection hierarchy:

```
view shape rectangle
   size: 10;
   borderColor: [ :cls | ('*Array*' match: cls name)
                                    ifTrue: [ Color blue ]
                                    ifFalse: [ Color black ] ];
   fillColor: [ :cls | cls isAbstractClass ifTrue: [ Color lightGray ] ifFalse: [ Color white] ].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

The produced visualization is given in Figure **??**. It easily help identifying abstract classes that are not named as "Array" and the one that are abstract without having an abstract method.

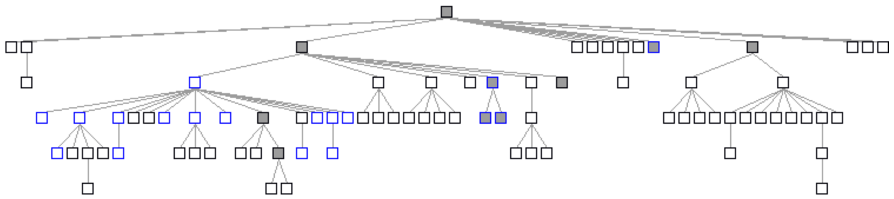


Figure 1.3: Abstract classes are in gray and classes with the word "Abstract" in their name are in blue.

Similarly than with height: and width:, messages to defines color either takes a symbol, a block or a constant value as argument. The argument is evaluated against the domain object represented by the graphical element (a double dispatch sends the message moValue: to the argument). The use of ifTrue:ifFalse: is not really practicable. Utilities methods are provided for that purpose to easily pick a color from a particular condition. The definition of the shape can simply be:

```
view shape rectangle
   size: 10;
   if: [ :cls | ('*Array*' match: cls name) ] borderColor: Color blue;
   if: [ :cls | cls isAbstractClass ] fillColor: Color lightGray.
...
```

The method isAbstractClass is defined on Behavior and Metaclass in Pharo. By sending the isAbstractClass to a class return a boolean value telling us whether the class is abstract or not. We recall that an abstract class in Smalltalk is a class that defines or inherits at least one abstract method (i.e., which contain self subclassResponsibility).

1.7 More on colors

Colors are pretty useful to designate a property (*e.g.*, gray if the class is abstract). They may also be employed to represent a continuous distribution. For example, the color intensity may indicate the result of a metric. Consider the previous script in which the node color intensity tells about the number of lines of code:

```
view interaction action: #browse.
view shape rectangle
   width: [ :each | each instVarNames size * 3 ];
   height: [ :each | each methods size ];
   linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

Figure 1.4 shows the resulting picture. The message linearFillColor:within: takes as first argument a block function that return a numerical value. The second argument is a group of elements that is used to scale the intensity of each node. The block function is applied to each element of the group. The fading scales from white (0 line of code) to black (the maximum lines of code). The maximum intensity is given by the maximum #numberOfLinesOfCode can take for all the subclasses of Collection. Variants of linearFillColor:within: are linearXXXFillColor:within:, where XXX is one among Blue, Green, Red, Yellow.

The visualization[3] you now obtain put in relation for each class the number of methods, the number of instance variables and the number of lines of code. Differences in size between classes might suggest some maintenance activities.

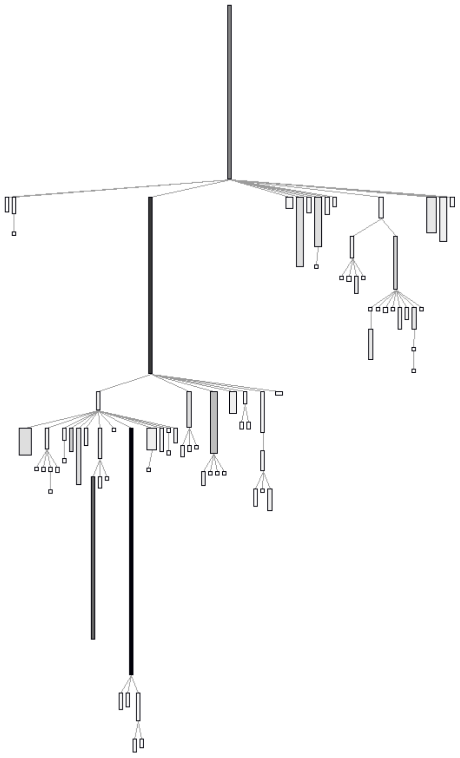


Figure 1.4: The system complexity visualization: nodes are classes; height is the number of lines of methods; width the number of variables; color tells about the number of lines of code.

A color may be assigned to an object identity using identityFillColorOf:. The argument is either a block or a symbol, evaluated against the domain object. A color is associate to the result of the argument.

[3]This visualization is named 'System complexity', if you wish to know more about it, you can refer to 'Polymetric Views—A Lightweight Visual Approach to Reverse Engineering' (Transactions on Software Engineering, 2003).

1.8 Popup view

Let's jump back on the abstract class example. The following script indicates abstract classes and how many abstract methods they define:

```
view shape rectangle
   size: [ :cls | (cls methods select:  #isAbstract ) size * 5 ] ;
   if: #isAbstractClass fillColor: Color lightRed;
   if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

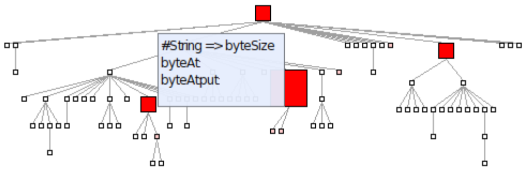


Figure 1.5: Boxes are classes and links are inheritance relationships. The amount of abstract method is indicated by the size of the class. A red class defines abstract methods and a pink class inherits from an abstract class solely.

Figure 1.5 indicate classes that are abstract either by inheritance or by defining abstract methods. Class size indicates the amount of abstract method defined.

The popup message can be enhanced to list abstract methods. Putting the mouse above a class does not only give its name, but also the list of abstract methods defined in the class. The following piece of code has to be added at the beginning:

```
view interaction popupText: [ :aClass |
  | stream |
  stream := WriteStream on: String new.
  (aClass methods select: #isAbstract thenCollect: #selector)
    do: [:sel | stream nextPutAll: sel; nextPut: $ ; cr].
  aClass name printString, ' => ', stream contents ].
...
```

So far, we have seen that an element has a shape to describe its graphical representation. It also contains an *interaction* that contains event handlers. The message popupText: takes a block as argument. This block is evaluated with the domain object as argument. The block has to return the popup text content. In our case, it is simply a list of the methods.

In addition to a textual content, Mondrian allows a view to be popped up. We will enhance the previous example to illustrate this point. When the mouse enters a node, a new view is defined and displayed next to the node.

```
view interaction popupView: [ :element :secondView |
   secondView node: element forIt: [
     secondView shape rectangle
       if: #isAbstract fillColor: Color red;
       size: 10.
     secondView nodes: (element methods sortedAs: #isAbstract).
     secondView gridLayout gapSize: 2
   ] ].

view shape rectangle
   size: [ :cls | (cls methods select:  #isAbstract ) size * 5 ] ;
   if: #isAbstractClass fillColor: Color lightRed;
   if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

The argument of popupView: is a two argument block. The first parameter of the block is the element represented by the node located below the mouse. The second argument is a new view that will be opened.

In the example, we used sortedAs: to order the nodes representing methods. This method is defined on Collection and belongs to Mondrian. To see example usages of sortedAs:, browse its corresponding unit test:

```
ToolSet browse: MOViewRendererTest selector: #testSortedAs
```

This last example uses the message node:forIt: in the popup view to define a subview.

1.9 Subviews

A node is a view in itself. This allows for a graph to be embedded in any node. The embedded view is physically bounded by the encapsulating node. The embedding is realized via the keywords nodes:forEach: and node:forIt:.

The following example approximates the dependencies between methods by linking methods that may call each other. A method m1 is connected to a method m2 if m1 contains a reference to the selector #m2. This is a simple but effective way to see the dependencies between methods. Consider:

```
view nodes: MOShape withAllSubclasses forEach: [:cls |
   view nodes: cls methods.
   view edges: cls methods from: #yourself toAll: [ :cm | cls methods select: [ :rcm |  cm
      messages anySatisfy: [:s | rcm selector == s ] ] ].
   view treeLayout
].
view interaction action: #browse.
view edgesFrom: #superclass.
view treeLayout.
```

A subview contains its own layout. Interactions and shapes defined in a subview are not accessible from a nesting node (Figure 1.6).

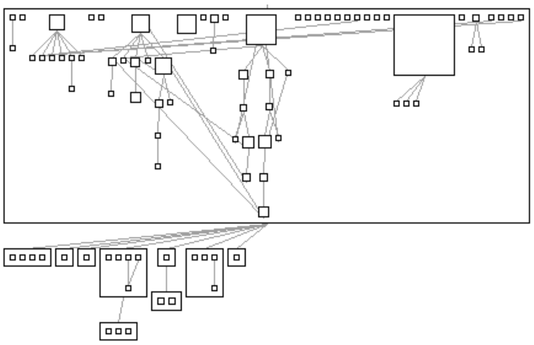


Figure 1.6: Large boxes are classes. Inner boxes are methods. Edges shows a possible invocation between the two.

1.10 Forwarding events

Method of the visualization given in the previous section may be moved by a simple drag and drop. However, it may be wished that the methods have a fixed position, and only the classes can be drag-and-dropped. In that case, the message forward has to be sent to the interaction. Consider:

```
view nodes: ROShape withAllSubclasses forEach: [:cls |
   view interaction forward.
   view shape rectangle
               size: #numberOfLinesOfCode.
   view nodes: cls methods.
   view edges: cls methods from: #yourself toAll: [ :cm | cls methods select: [ :rcm |  cm
      messages anySatisfy: [:s | rcm selector == s ] ] ].
   view treeLayout
].
view interaction action: #browse.
view edgesFrom: #superclass.
view treeLayout.
```

Moving a method will move the class instead. It is often convenient to drag and drop more than one element. As most operating systems, Mondrian offers multiple selection using the Ctrl or Cmd key. This default behavior is available for every nodes. Multiple selection allows for a group of node to be moved.

1.11 Events

Each mouse movement, click and keyboard keystroke corresponds to a particular event. Mondrian offers a rich hierarchy of events. The root of the hierarchy is MOEvent. To associate a particular action to an event, an handler has to be defined on the object interaction. On the following example, clicking on a class opens a code browser:

```
view shape rectangle
  width: [ :each | each instVarNames size * 5 ];
  height: [ :each | each methods size ];
  if: #isAbstractClass fillColor: Color lightRed;
  if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.

view interaction on: ROMouseClick do: [ :event | event model browse ].

view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

The block handler accepts one argument: the event generated. The object that triggered the event is obtained by sending modelElement to the event object.

1.12 Interaction

Mondrian offers a number of contextual interaction mechanisms. The interaction object contains a number of keywords for that purpose. The message highlightWhenOver: takes a block as argument. This block returns a list of the nodes to highlight when the mouse enters a node. Consider the example:

```
view interaction
  highlightWhenOver: [:v | {v - 1 . v + 1. v + 4 . v - 4}].
view shape rectangle
  width: 40;
  height: 30;
  withText.
view nodes: (1 to: 16).
view gridLayout gapSize: 2.
```

Entering the node 5 highlights the nodes 4, 6, 1 and 9. This mechanism is quite efficient to not overload with connecting edges. Only the information is shown for the node of interest.

A more compelling application of highlightWhenOver: is with the following example. A hierarchy of class is displayed on the left hand side. On the right hand size a hierarchy of unit tests is displayed. Locating the mouse pointer above a unit test highlights the classes that are referenced by one of the unit test method. Consider the (rather long) script:

```
"System complexity of the collection classes"
view shape rectangle
  width: [ :each | each instVarNames size * 5 ];
  height: [ :each | each methods size ];
  linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.

"Unit tests of the package CollectionsTest"
view shape rectangle withoutBorder.
view node: 'compound' forIt: [
  view shape label.
  view node: 'Collection tests'.

  view node: 'Collection tests' forIt: [
    | testClasses |
    testClasses := (PackageInfo named: 'CollectionsTests') classes reject: #isTrait.
    view shape rectangle
      width: [ :cls | (cls methods inject: 0 into: [ :sumLiterals :mtd | sumLiterals + mtd
        allLiterals size]) / 100 ];
      height: [ :cls | cls numberOfLinesOfCode / 50 ].
    view interaction
        highlightWhenOver: [ :cls | ((cls methods inject: #()
                    into: [:sum :el | sum , el allLiterals ]) select: [:v | v isKindOf: Association ]
        thenCollect: #value) asSet ].
    view nodes: testClasses.
    view edgesFrom: #superclass.
    view treeLayout ].

  view verticalLineLayout alignLeft
].
```

The script contains two parts. The first part is the ubiquitous system complexity of the collection framework. The second part renders the tests contained in the CollectionsTests. The width of a class is the number of literals contained in it. The height is the number of lines of code. Since the collec-

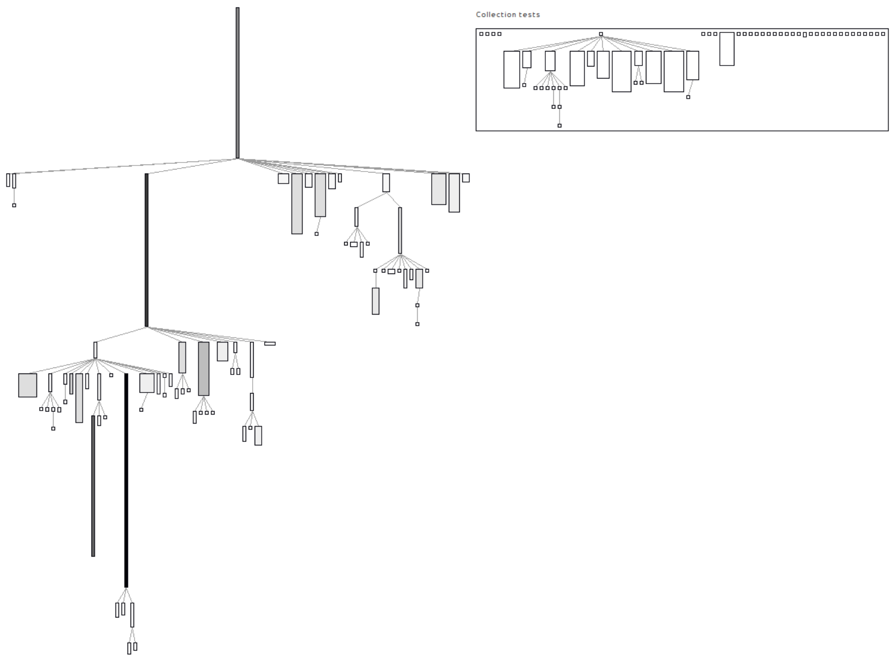


Figure 1.7: Interactive system complexity.

tion tests makes a great use of traits to reuse code, these metrics have to be scaled down. When the mouse is put over a test unit, then all the classes of the collection framework referenced in this class are highlighted.

Displaying edges when overing a particular node is a recurrent problem in visualization. The method dynamicEdge:using: is offered for that purpose. The following example draw arrowed lines from an element to the two others:

```
view interaction
   dynamicEdge: [ :model | (Array with: 10 with: 20 with: 30) copyWithout: model ]
   using: (ROLine arrowed color: Color red).
view shape rectangle size: 20.
view nodes: (Array with: 10 with: 20 with: 30).
view circleLayout.
```

1.13 Conclusion

Mondrian enables any graph of objects to be visualized. This chapter has reviewed the main features of Mondrian:

- The most common way to define nodes is with nodes: and edges with edgesFrom:, edges:from:to: and edges:from:toAll:.
- A whole range of layout is offered. The most common layouts are accessible by sending circleLayout, treeLayout, gridLayout to a view.
- A shape defines the graphical aspect of an element. Height and width are commonly set with height: and width:, respectively.
- A shape is colored with borderColor: and fillColor:.
- Information may be popped up with popupText: and popupView:.
- A subview is defined with nodes:forEach: and node:forIt:.
- Events of a sub node is forwarded to its parent with forward and forward:.
- Highlighting is available with highlightWhenOver:, which takes a one-arg block that has to return the list of nodes to highlight.

Acknowledgment We are very grateful to Nicolas Rosselot Urrejola for his review of an early draft of the chapter.