Mondrian: An Agile Information Visualization Framework

In Proceedings of ACM Symposium on Software Visualization (SoftVis 2006)

Michael Meyer*
Software Composition Group
University of Bern, Switzerland

Tudor Gîrba†
Software Composition Group
University of Bern, Switzerland

Mircea Lungu‡
Faculty of Informatics
University of Lugano, Switzerland

Abstract

Data visualization is the process of representing data as pictures to support reasoning about the underlying data. For the interpretation to be as easy as possible, we need to be as close as possible to the original data. As most visualization tools have an internal meta-model, which is different from the one for the presented data, they usually need to duplicate the original data to conform to their meta-model. This leads to an increase in the resources needed, increase which is not always justified. In this work we argue for the need of having an engine that is as close as possible to the data and we present our solution of moving the visualization tool to the data, instead of moving the data to the visualization tool. Our solution also emphasizes the necessity of reusing basic blocks to express complex visualizations and allowing the programmer to script the visualization using his preferred tools, rather than a third party format. As a validation of the expressiveness of our framework, we show how we express several already published visualizations and describe the pros and cons of the approach.

Keywords: software visualization, graph visualization, model transformation

1 Introduction

Visualization is an established tool to reason about data. Given a wanted visualization, we can typically find tools that take as input a certain format and that provide the needed visualization [Panas et al. 2005; Reiss 2001]. Typically, the visualization tools have an internal model and they translate the data in such a way that it fits their internal model.

The main advantage of this approach is that it can accommodate data provided by third party tools. This works perfectly well when it is enough to just generate the picture, or the animation without further inspection of the data. However, one drawback of the approach is that, when a deep reasoning is required, we need to refer back to the capabilities of the original tool that manipulates the actual data. A second drawback of the approach is that it actually duplicates the required resources: the data is present both in the original tool, and in the visualization tool. A third drawback is that when a programmer needs to visualize his own data, he is forced to leave his preferred environment and language and learn the language of the import/export format.

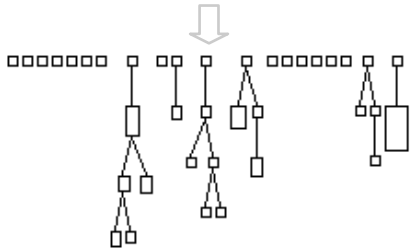
*e-mail:michael-meyer@students.unibe.ch
†e-mail:girba@iam.unibe.ch
‡e-mail:mircea.lungu@lu.unisi.ch

Several tools take a middle ground approach and choose to work close with the data by either offering integration with other services [Lanza and Ducasse 2005], or providing the services themselves [M.-A. D. Storey and Michaud 2001]. However, when another type of service is required, the integration is lost.

In this paper we propose a radically different approach: instead of moving the data to be visualized to the tool, we argue for moving the visualization tool to the data. Instead of providing a required data format, we provide a simple interface through which the programmer can easily script, in a declarative fashion, the visualization. This means that our solution works directly with the objects in the data model.

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. We provide a framework that puts all the emphasis on providing the needed basic pieces and that places the control in the hand of the programmer. The example below shows the essence of our approach: a script that creates a view, adds *nodes* representing some objects (in this case the *classes* in the *model*), and adds the *edges* representing some other objects (in this case the *inheritance relationships* in the *model*). The nodes and edges are represented *using shapes*. The result of executing the script on a given model is the tree shown below the script.

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



Mondrian, our framework, is written in Smalltalk, and this is why the previous script is written in Smalltalk (see the Appendix for details concerning the Smalltalk syntax). However, because we wanted the model to be easily implemented in other languages too, we strived to make it simple enough so it can be built on top of any graphical framework.

Paper structure. In the next section we detail the challenges that a visualization engine faces. Section 3 presents the framework from a user's point of view and then discusses the implementation. In Section 4 we validate our model by showing that the implementation of already known visualizations is straight-forward when based on the Mondrian framework. In Section 5 we show several characteristics of the user interface prototype. We position our approach with respect to the state of the art in Section 6, and we conclude in Section 7. We present the Smalltalk syntax in the Appendix.

2 Challenges for an information visualization engine

Reiss postulates two reasons for the slowness of the software developer community in adopting software visualization tools [Reiss 2001]. The first is the lack of flexibility: the tools which are designed for a specific purpose can not support the user when he needs a slightly (or sometimes even drastically) different visualization on the data related to the task at hand. The second reason is the difficulty of preparing and converting the data to the format understood by the visualization tool.

In this work we propose an approach which solves the second problem by bringing the visualization closer to the data and not viceversa, and solves the first problem by providing an extremely malleable visualization framework. Some of the factors that would be needed to provide flexibility to the framework and to support the proximity of the visualization to the data are:

The visualization engine should be domain independent. As on the one hand we want to bring the visualization to the data, and on the other hand we want to provide a *framework*, we will need to make sure that the framework is general enough to accommodate any data model.

Visualizations shoud be easily composed from simpler parts. The user of the framework should be able to define basic blocks that he can later for building more complex visualizations. One example is the way graph-layouting tools such as GraphViz and aiSee [GRA ; AIS] implement nested layouts: once a layout is defined, it can be used to layout the internal contents of a given node which is part of another graph which has yet another layout.

The visualization should be definable at a fine grained level. Indeed, it is important to detail the visualization at any level of detail desired. For example, showing the details of a figure by representing inside other figures based on the characteristics of the represented object. From another perspective, the framework should support instance based representation as opposed to type based representation, as it is sometimes desired to not show all the objects of the same type with the same representation.

Object creation overhead should be kept to a minimum. Visualization engines typically have an internal meta-model, usually a graph-like one, in which they put the data and on which the visualization is defined. However, when the objects of the data model are already present, we do not need to duplicate them, but the visualization should work directly with those objects. This is important as it saves memory and time that is otherwise spent in building the internal model.

The visualization description should be declarative. The description of the visualization should be declarative, as it should only be a mapping between the data model and the visualization model. The benefit of a declarative approach is that it allows for generation of the descriptions from editors.

3 Scripting visualizations with Mondrian

In this section we show the usage of our framework by providing a step-by-step example of how to handle the main parts of the visualization, and we use the example to detail the internal structure of the framework, emphasizing the different design decisions.

3.1 Painting a view 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 38 classes. The task we propose is to provide a simple overview of the classes in the system and of some of their relationships.

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. 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. In the example below we represent each class using a Rectangle with border:

```
view := ViewRenderer new.
view nodes: model classes using: Rectangle withBorder.
view open.
```

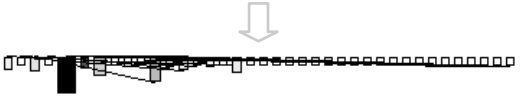
We directly support polymetric views on a Rectangle [Lanza and Ducasse 2003]. For example, if we want to specify the width and the height and the color, we need to set the different characteristics of the object to be used by the Rectangle:

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

NOA, NOM and LOC are methods in the object representing a class and return the value of the corresponding metrics: NOA stands for number of attributes, NOM stands for number of methods, and LOC stands for number of lines of code.

Adding edges. To show how classes inherit from each other, we can add an edge for each inheritance relationship. We use a similar instruction as for the nodes. In our example, supposing that we can ask the model for all the inheritance definitions between the classes in the model, and that each inheritance is represented as an object, the addition of the edges is illustrated in the example below:

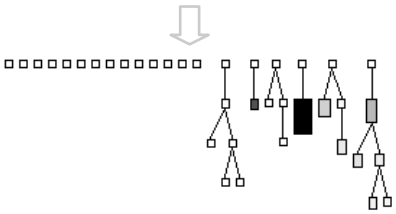
```
view := ViewRenderer new.
view nodes: model classes
    using: (Rectangle withBorder width: #NOA; height: #NOM;
                                 liniarColor: #LOC within: model classes).
view edges: model inheritances
    using: (Line from: #superclass to: #subclass).
view open.
```



Like in the case of the nodes, when specifying the shape, we made reference to methods that are defined in the inheritance object. Thus, given an inheritance object, we will create an edge between the node holding the superclass and the node holding the subclass.

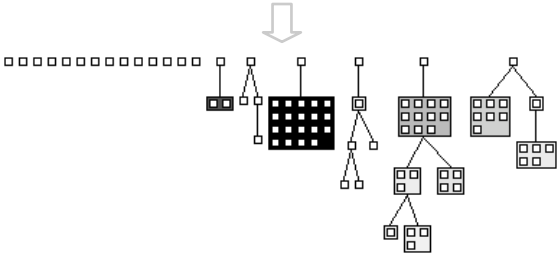
Layouting. To make the above graph understandable, we layout the nodes in a tree. By default, the nodes are arranged in a horizontal line. If another arrangement is desired, the programmer needs only to specify another supported layout. For example:

```
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.
```



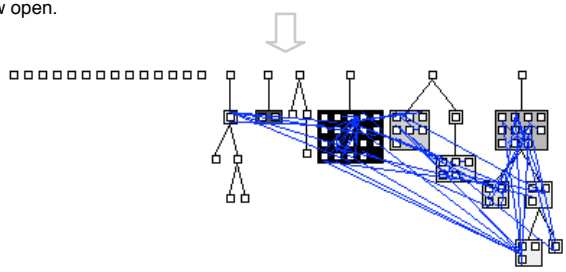
In the example, we use a Smalltalk construct which represents a closure, or a lamba. In our example, we use [:eachClass | ...]. This is equivalent to *(lamba(eachClass)(...))*. In Java, the closure can be modeled by using a Command pattern together with an anonymous class. A similar problem was solved in the SWT[1] framework of Eclipse, where the user interface needs to accommodate any type of objects using duck typing[2]. We use the closure to depict the nesting level: the code from inside the closure depicts the graph for each class.

[1] http://www.eclipse.org/swt/

[2] http://www.coconut-palm-software.com/the_visual_editor/?p=25

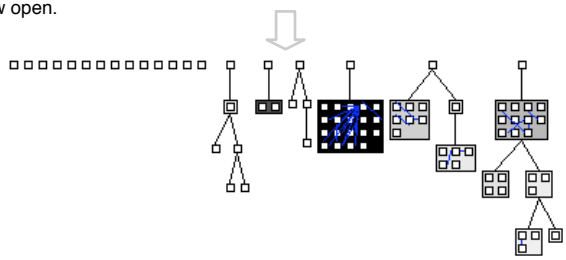
Adding inter-edges. The edges are created by specifying the *from* and the *to* objects. Because we can have the objects at various levels of nesting, it is important to specify the location from where the lookup of the objects should start. For example, if we want to add invocations as edges between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



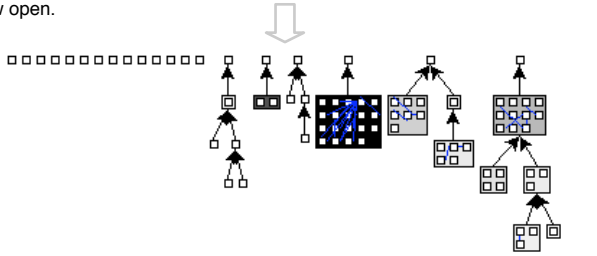
We defined the invocation edges in the outer graph, and as a result we obtained the edges between methods defined in different classes. However, if we want to restrict the edges only to the scope of one class, all we have to do is to move the instruction at the right nesting level:

```
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 [Lanza and Ducasse 2003]. However, when the user wants to specify an arrow at the end of the line, he can use decorations. We introduce the notion of decorations to allow the programmer compose the overall visual representation out of basic shapes. 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:

```
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.
```



Decorations can be applied to any figure. In fact, Rectangle withBorder is implemented as Rectangle new decoratedWith: Border new.

3.2 Mondrian internals

Overview. Figure 1 reveals the core structure of our framework.

Each Figure represents and holds an Object. The Figure, NodeFigure and EdgeFigure are implemented directly on top of the graphical framework, but they hold no specific value for the visualization (*e.g.,* label or the size of the shape). The entire responsibility of what gets drawn belongs to the Shape.

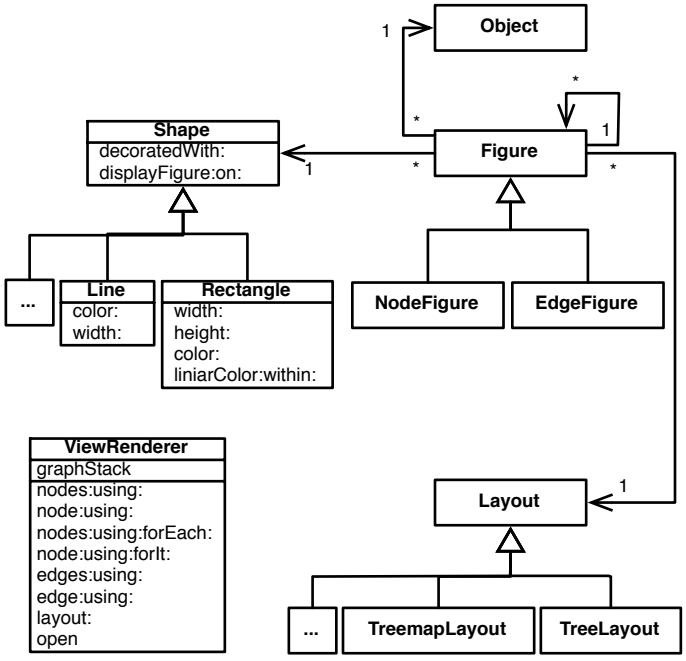


Figure 1: The internal model of Mondrian.

As we want to accommodate any Object, we cannot tie the implementation to a particular interface. That is why, the Figure talks to the Object through the Shape which acts like a translator between the visualization model and the data model. We can have several shapes (*e.g.,* Rectangle, Line), and depending on the Shape we can specify how to compute a certain visual characteristic via a closure. For example, to a Rectangle we can specify how to compute the width, height and color. As mentioned in the previous section, in Smalltalk, closures are first class objects that can be passed around and get executed with a specified context. Closures can be simulated in Java using anonymous classes that implement a command. A similar problem was addressed and solved in SWT.

The Shape is just 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. For example, the instruction view nodes: model classes using: Rectangle withBorder will create one NodeFigure for each class, but all those figures will share and be displayed according to the specification in Rectangle withBorder.

The Figure is implemented directly on top of the graphical framework. One goal of our framework was to create as less objects as possible for the visualization. The assumption is that the graphical framework creates an object for each visual figure, and we wanted to reuse that to model the graph. Our particular implementation was accomplished on top of the Smalltalk Hotdraw framework [Brant 1995]. We claim that such an implementation is not tied to Smalltalk: on the one hand Hotdraw is implemented in Java [3] too, on the other hand, the specific implementation is minimal. The three classes have 354 lines of code.

Each Figure is a graph and holds several NodeFigures and EdgesFigures. Furthermore, the Figure also knows the Layout to be applied on its children. The specific Layouts are implemented in subclasses of the Layout.

ViewRenderer. To make the script easy to write, we have designed the ViewRenderer to hide the internal details of the model. The intent of the ViewRenderer was to provide a script which is concise and which looks similar with a dedicated format, while still being an executable program[4].

In Figure 1 we show the main protocol of the ViewRenderer:

- two methods dealing with adding nodes – There are two methods for adding several nodes, and two for adding one node.
- two methods dealing with adding nodes with nested graphs – Again, we can add nodes specifying *for each* what goes inside, or we can add just one node specifying *for it* what goes inside.
- two methods dealing with adding edges – Like in the case of the nodes, we can add one or several edges.
- one method for specifying the layout – This method simply takes an instance of a Layout.
- one method for spawning the view

A particular implementation that dramatically increased the readability is the graphStack implemented internally. This is useful for expressing nested graphs. For example in the example below:

```
view nodes: model classes using: Rectangle withBorder forEach: [:eachClass
   view nodes: eachClass methods using: Rectangle withBorder
   view layout: CheckerboardLayout new.
].
```

[3] http://www.jhotdraw.org/

[4] The design of the ViewRenderer was inspired by the HtmlRenderer from Seaside [Ducasse et al. 2004]

we refer to the same variable view, both from the outer context and from the inner context. Internally, the implementation of view:using:forEach: puts the current graph (represented by a figure) on the stack and whenever view is called from within the nested context, the commands are attributed to the current graph. In this way the programmer feels like he is all the time creating the same view and by indenting correspondingly with the nesting levels, the script reads as the format of a GraphViz-like tool.

Reusing Scripts. To fully support the programmer, we want to allow him to place his views in methods, and then reuse them. For this we can pass the view as a parameter to a method that builds the visualization on the view. For example, the previous example script in the previous section can be implemented like:

```
view := ViewRenderer new.
view nodes: model classes using: Rectangle withBorder
   forEach: [:eachClass | eachClass viewMethodsIn: view]
...
view open
```

```
ModelClass>>viewMethodsIn: view
  view nodes: self methods using: Rectangle withBorder
  view layout: CheckerboardLayout new.
```

We implemented viewMethodsIn: to show on a view the methods from the ModelClass, and call it from within the graph inside a class node. This example shows how we can put the visualization near the data it represents and then reuse it in a larger context.

4 Case studies

To validate our model, we have implemented several already published visualizations. Each case study presented here exercises another part of the framework. Together with each resulting visualization we attach the code we used to generate it. The scripts presented are complete, except for the implementation of the data model. In our case, we assume we have a model object from which we can obtain the different other objects needed for the visualization (*e.g.,* classes, methods). The actual implementation makes use of the Moose reengineering environment [Nierstrasz et al. 2005] for getting access to the model data, but the framework can work with any model.

System Complexity View. In the example from Section 3 we show how we build polymetric views - graph views which map metrics on the characteristics of the nodes and edges [Lanza and Ducasse 2003]. One important polymetric view is the System Complexity View, which shows the class hierarchy. On the top part of Figure 3 we show the System Complexity View applied on ArgoUML (1405 classes) and then stretched to fit the page. Because the hierarchy is too wide, when stretched, the view becomes difficult to grasp.

However, when using a screen, we would like to have an overview of the system that uses the entire screen surface. That is why we designed a view we call Screen Filling System Complexity. In the bottom part of Figure 3 we show the same data as in the top part, but this time we put each hierarchy in a node and then arranged the hierarchies in a FlowLayout to fill the screen surface. Besides the hierarchies, we have also grouped the lonely classes in a box to save more space.

We also show the code required for the two views, to stress the fact that we only use simple layouts to produce a radically different visualization.

Class Blueprint. Class Blueprint is another polymetric view displaying the internals of the class [Ducasse and Lanza 2005]. It splits the class into five layers: the initialization methods, the public interface methods, the internal implementation methods, the accessor methods and the attributes. The script below shows the straight forward implementation in Mondrian: we create an overall node for the class and inside of it we create five nodes each containing the five layers; the layers are arranged in a HorizontalLineLayout, while the nodes inside a layer are arranged in a VerticalLineLayout; moreover, we add the edges for invocations (blue) and for accesses (cyan).

```
view := ViewRenderer new.
view node: class
      using: (Rectangle new decoratedWith: (Label text: #name)
      forIt: [

  view node: class initMethods using: Rectangle withBorder forIt: [
    view nodes: class initMethods using: Shape forMethod.
    view layout: VerticalLineLayout new].

  view node: class interfaceMethods using: Rectangle withBorder forIt: [
    view nodes: class interfaceMethods using: Shape forMethod.
    view layout: VerticalLineLayout new].

  view node: class implementationMethods using: Rectangle withBorder forIt: [
    view nodes: class implementationMethods using: Shape forMethod.
    view layout: VerticalLineLayout new].

  view node: class accessorMethods using: Rectangle withBorder forIt: [
    view nodes: class accessorMethods using: Shape forMethod.
    view layout: VerticalLineLayout new].

  view node: class attributes using: Rectangle withBorder forIt: [
    view nodes: class attributes using: Shape forAttribute.
    view layout: VerticalLineLayout new].

  view edges: class invocations using: (Line from: #invokedBy to: #invoked
                                          color: #blue).
  view edges: class accesses using: (Line from: #accessedBy to: #accessed
                                          color: #cyan).

].
view open.
```

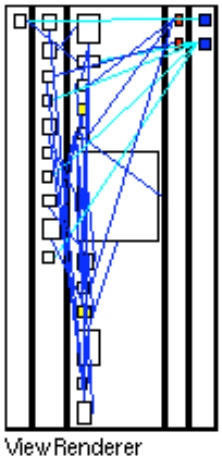


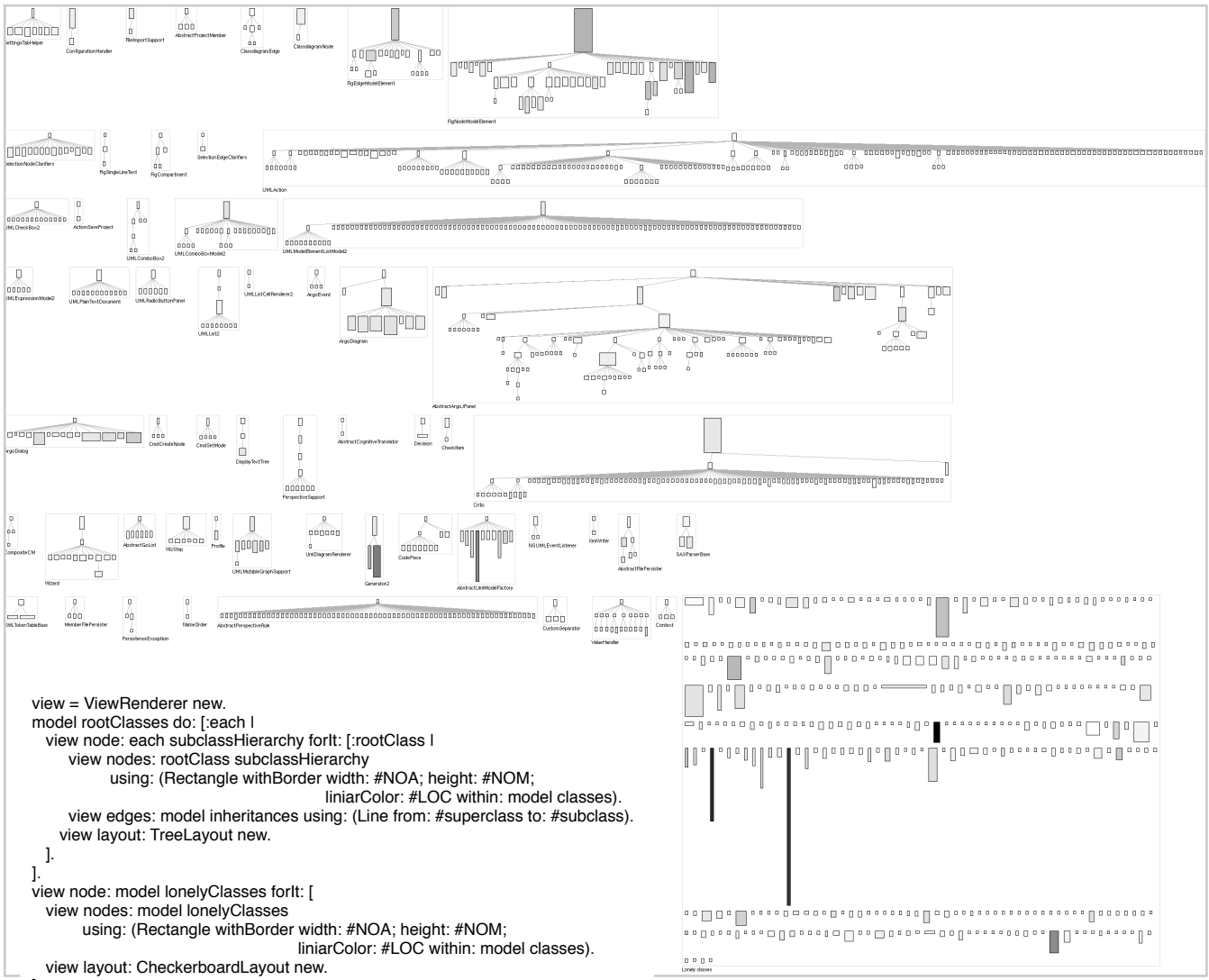
Figure 2: Example of a class blueprint.

Most often, the representation of a certain type of data can be reused in different contexts. In the above script, the shapes defining how methods and attributes should look like were factored out into: Shape forMethod, and Shape forAttribute.

From another perspective, we notice that the script is more complex then the one representing the System Complexity View, but this is due to the fact that the visualization itself is more complex.

```
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.
```

System Complexity



```
view = ViewRenderer new.
model rootClasses do: [:each |
  view node: each subclassHierarchy forIt: [:rootClass |
    view nodes: rootClass subclassHierarchy
          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 node: model lonelyClasses forIt: [
  view nodes: model lonelyClasses
        using: (Rectangle withBorder width: #NOA; height: #NOM;
                                  liniarColor: #LOC within: model classes).
  view layout: CheckerboardLayout new.
].
view layout: (FlowLayout withMaxWidth: 800).
view open.
```

Screen Filling System Complexity

Figure 3: When the hierarchy is too wide, the classic System Complexity gets too small when stretched to fit in one screen. Screen Filling System Complexity puts each hierarchy in a box and arranges the boxes to make use of the vertical space.

Spectographs. Figure 4 shows an example of spectographs [Wu et al. 2004] applied on the files in the CVS repository of JBoss sources (2094 files). The Figure shows each file on a line and colors a dot on that line with red when a commit affects the corresponding, yellow if the commit was nearby and green otherwise. The actual script traverses the CVS files in the model and for each month decides the color of the dot. The dots are arranged via a ScatterplotLayout.

```
view := Mondrian.ViewRenderer new.
yPos := 5.
model files do: [:file | | xPos greenOk yellowOk|
    greenOk := false.
    yellowOk := false.
    xPos := 5.
    model months do: [:month |
        (file hasCommitIn: month) ifTrue: [
            view node: file using: (Rectangle x: xPos y: yPos
                                                color: #red).
            greenOk := false.
            yellowOk := true.
        ].
        ifFalse: [
            greenOk ifTrue: [
                view node: file using: (Rectangle x: xPos y: yPos
                                                color: #green)].
            yellowOk ifTrue: [
                view node: file using: (Rectangle x: xPos y: yPos
                                                color: #yellow).
                yellowOk := false.
                greenOk := true.
            ]
        ].
        xPos := xPos + 5.
    ].
    yPos := yPos + 1.
].
view layout: Mondrian.ScatterplotLayout new.
view open.
```

Figure 4: Example of a spectograph.

One striking characteristic of the presented script is that it is *not* as readable as the previous ones. First of all, the code is longer than in the rest of the examples presented in this paper, even though the visualization appears to be simple. When taking a second look we noticed that the view description is hampered by the actual algorithm of traversing the data model in a way that is not directly supported by the model.

We have encountered such a situation several times while experimenting, and every time the script was too long and difficult to grasp, we have come to the conclusion that the data model should be updated. Once we applied the enhancement, the code became much more readable. In our example, a lot of lines of code are dedicated to the computation of the color for a specified pixel (*i.e.,* red, green, yellow). If this information would have been provided by the data model directly, the script would have been much more compact.

This example also emphasizes the focus of our framework: there is an object behind every representation. While this can be suited for graph like visualizations, for this particular one, it generates a significant overhead in object creation: every month of a file is represented by an object figure. Definitely, the spectograph can be implemented in a much more concise way using either a more intelligent figure for a line, or by just creating one single figure for the entire view. However, the example does show that it is possible to prototype even such a visualization using our framework.

Scatterplot. Our model can be used for general information visualization as well. In Figure 5 we show how we build a scatterplot showing classes of Ant (500 classes) according to two metrics like in the polymetric views [Lanza and Ducasse 2003].

```
view := ViewRenderer new.
view shape: Rectangle new.
view decorateShapeWith: HorizontalCoordinate new.
view decorateShapeWith: VerticalCoordinate new.
view node: model classes forIt: [
    view nodes: model classes
        using: (Rectangle withGrayBorder x: (each #averageLOC; y: #NOM;
                                        liniarColor: #TCC within: model classes).
    view layout: ScatterplotLayout new.
].
view open.
```

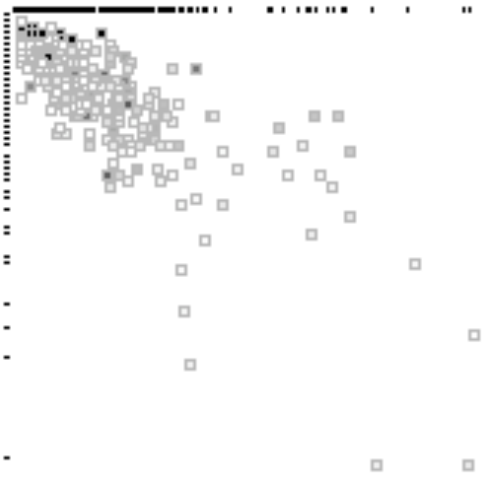


Figure 5: Example of a scatterplot.

The example shows how the decorations can be used in a way that bares information: the node surrounding the scatterplot is a rectangle decorated with coordinate lines (*i.e.,*HorizontalCoordinate and VerticalCoordinate) created by the projection of the nodes inside on the x and y [Tufte 2001]. Using decorations, the programmer can use basic parts to create more complex representations. As with the layouts, a library of decorations is needed.

5 Mondrian prototype

In this section we briefly sketch some characteristics of the prototype implementation of the Mondrian user interface.

As mentioned before, the Shape is just a specification of how the data model should be transformed to the visualization model. This approach allowed us to create editors that handle the mapping based on knowing the meta-model of the data.

For example, in Figure 5 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.

The view presents the hierarchy of Mondrian classes and the editor is built on top of the Moose environment [Ducasse et al. 2005]. The screenshot shows a selected class and the editor allows us to map metrics that Moose can compute on classes to the Rectangle and to the Border that represent the selected class. By changing the metrics in the editor we change the Shape object.

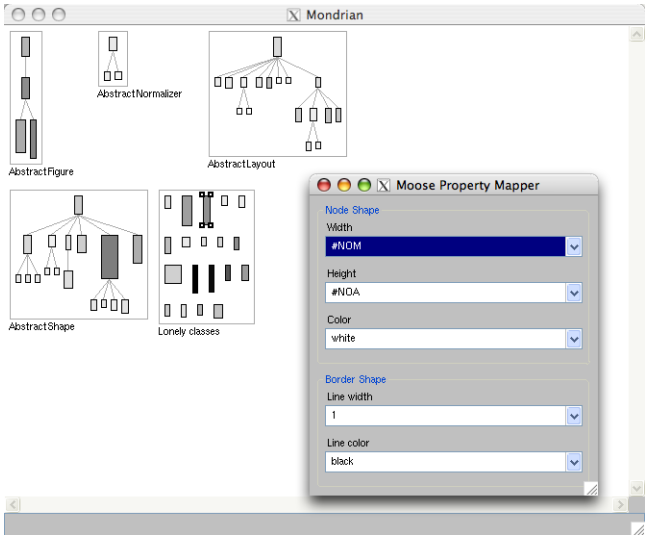


Figure 6: Mondrian editor based on the Moose meta-model.

In Section 2 we claimed that the visualization specification should be instance based. For example, in Figure 5 we show the same visualization as in the previous one, only now the ViewRenderer is shown using a Class Blueprint. With such a mechanism in place, we could implement a dynamic tool that allows for semantic zooming: each time we move the mouse over an object, we represent it with more details by specifying a different representation only for that particular object.

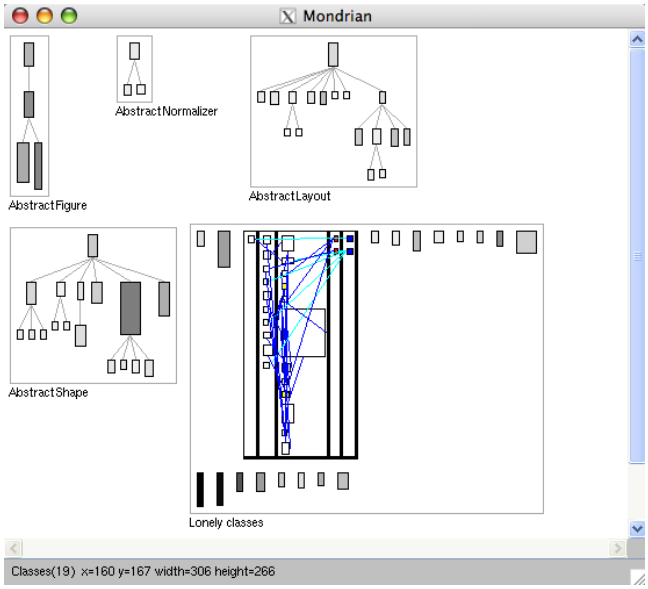


Figure 7: Mondrian showing instance based visualization.

6 Related Work

The importance of software visualization as a comprehension aid for reverse engineering and problem detection can be inferred from the large number of tools and techniques that have been developed for this purpose: Seesoft [Eick et al. 1992], Rigi [Müller 1986; Müller and Klashinsky 1988] and SHriMP [Storey and Müller 1995; M.-A. D. Storey and Michaud 2001], CodeCrawler [Lanza 2003; Lanza and Ducasse 2005], etc. In this paper we have shown how our visualization framework can be used to implement visualizations provided by some of these tools.

One peculiarity of these tools is that they implement a finite set of specific visualizations. However, the very nature program optimization, problem detection and program understanding implies that the user might not know a priori what exactly is he looking for, and therefore, what visualizations to use. In these cases, the possibility of easily generating new visualizations is valuable to him.

This is the reason why some tools provide more freedom to the user in defining his own visualizations. One of the first such tools is CodeCrawler [Lanza 2003; Lanza and Ducasse 2005] which is a tool that provides visualizations of combined metrics and structural information. The tool offers the user the possibility of online configuring the parameters of the visualization.

Another tool which emphasizes the importance of user customization is Vizz3D[Panas et al. 2005]. The difference between this and the preceding one is that Vizz3D provides more general visualizations while CodeCrawler is focused on metric-based visualizations.

The difference between our framework and CodeCrawler and Vizz3D is the target audience. While CodeCrawler and Vizz3D are aimed at less sophisticated end users who are satisfied to configure their visualizations via a user interface, Mondrian is aimed at the programer who needs the full power and flexibility of a programming language.

From this point of view, a tool which is more similar in intentions to ours is G See [Favre 2001]. G See provides a simple interface for data discovery and visualization services such as layouts and integration with external tools.

EVolve [Wang et al. 2003] is a tool and a framework for visualizing information derived from the run-time execution of a system. While the framework provides a rich set of visualizations and can be extended with new sources of visualization, it lacks the possibility of aggregating the various visualizations.

BLOOM [Reiss 2001] is another tool which tries to provide a flexible and extensible visualization environment. The difference between our tool and BLOOM is the granularity level: while bloom can be extended by dynamically loading separately compiled visualization components, in our case we provide code-level composition and extensibility facilities.

Another class of tools which are similar to ours are the more general graph visualization tools [GRA ; AIS]. Both provide the possibility of scripting visualizations and having nested graphs. However, because they are totally separated from the model, the possibility of interacting with the model is inexistent. Moreover, as we have shown, our tool is aimed at representing more than nested graphs.

7 Conclusions

We have presented a visualization framework that is aimed at providing the programmer with the necessary abstractions and tools for quickly drafting new visualizations by easily combining simpler ones. As a validation of our approach, we have presented the natural way in which several well-known visualizations can be expressed using our framework.

At this stage, the Mondrian framework is not intended to be a replacement for an existing end-user tool. An end-user tool has to provide interaction facilities, predefined and meaningful combinations of visualizations, coupled perspectives, and might also provide data collection besides the data visualization services, *etc.*. Instead, the primary focus of Mondrian is providing researchers the possibility to build the visualizations representing their models by using their preferred tools and environment.

We have developed the framework in parallel with experimenting with it, and we are continuously evolving it. The assumption of the approach is that the data model exists already. During our experiments, the most important lesson learnt is that the visualization description can be straight-forward when the model does provide the right information, and when it does not, it is often the case that conceptual enhancements of the data model are needed.

In essence, Mondrian is a meta-model of the different visualizations presented, and the scripts are in fact transformations of the data model into the visualization model. One particular part we put much emphasis on is the declarative manner of the scripts. This is an important part, as in the future we intend to create generic user interfaces to generate such scripts.

Acknowledgments. Gîrba and Meyer gratefully acknowledge the financial support of the Swiss National Science Foundation for the project Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

References

AiSee, Graph Layout Software. http://www.aisee.com/.

BRANT, J. 1995. *HotDraw*. Master's thesis, University of Illinois at Urbana-Chanpaign.

DUCASSE, S., AND LANZA, M. 2005. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering 31*, 1 (Jan.), 75–90.

DUCASSE, S., LIENHARD, A., AND RENGGLI, L. 2004. Seaside — a multiple control flow web application framework. In *Proceedings of ESUG Research Track 2004*, 231–257.

DUCASSE, S., GÎRBA, T., LANZA, M., AND DEMEYER, S. 2005. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series. Franco Angeli, Milano, 55–71.

EICK, S. G., STEFFEN, J. L., AND ERIC E., JR., S. 1992. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering 18*, 11 (Nov.), 957–968.

FAVRE, J.-M. 2001. Gsee: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension*, IEEE, 233–244.

Graphviz, Graph Visualization Software. http://www.graphviz.org/.

LANZA, M., AND DUCASSE, S. 2003. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering 29*, 9 (Sept.), 782–795.

LANZA, M., AND DUCASSE, S. 2005. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series. Franco Angeli, Milano, 74–94.

LANZA, M. 2003. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, IEEE Press, 409–418.

M.-A. D. STOREY, C. B., AND MICHAUD, J. 2001. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*.

MÜLLER, H. A., AND KLASHINSKY, K. 1988. Rigi – a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, IEEE Computer Society Press, 80–86.

MÜLLER, H. A. 1986. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University.

NIERSTRASZ, O., DUCASSE, S., AND GÎRBA, T. 2005. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, ACM Press, New York NY, 1–10. Invited paper.

PANAS, T., LINCKE, R., AND LÖWE, W. 2005. Online-configuration of software visualization with Vizz3D. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS 2005)*, 173–182.

REISS, S. P. 2001. An overview of bloom. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ACM Press, New York, NY, USA, 2–5.

STOREY, M.-A. D., AND MÜLLER, H. A. 1995. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, IEEE Computer Society Press, 275–284.

TUFTE, E. R. 2001. *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press.

WANG, Q., WANG, W., BROWN, R., DRIESEN, K., DUFOUR, B., HENDFREN, L., AND VERBRUGGE, C. 2003. EVolve: an open extensible software visualization framework. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS 2003)*, 37–49.

WU, J., HOLT, R., AND HASSAN, A. 2004. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, IEEE Computer Society Press, Los Alamitos CA, 80–89.

A APPENDIX: Smalltalk Syntax in a Nutshell

In Smalltalk everything is an object and all the computation is done through objects sending messages to other objects. There are only four types of expressions: literals, variables, messages and blocks.

Messages. Smalltalk is based entirely on sending messages to objects. In response to every message, some object is returned (often, the returned object is the one to which the message was sent). There are three types of messages in Smalltalk:

1. Unary messages, which have no parameters
   Example: model classes (sends the classes message to the model object).

2. Bynary messages, which usually are used for arithmetical operations
   Example: 2 + 3 (sends the message + with parameter 3 to the 2 object.

3. Keyword messages, which use keywords to organize partameters
   Example: myArray at: 1 put: 5 (sends the at:put: message to myArray with the parameters 1 and 5).

The precedence of the operations is unary, bynary and keywords.

Statements. Every statement can be one of two types:

1. A message send (*e.g.,*model classes.).
2. An assignment (*e.g.,*greenOK := false.).

Blocks. The *Smalltalk blocks* are the counterparts of the LISP closures. They are also objects and represent a sequence of statements, separated by periods, delimited by brackets. A block specifies a computation that is deferred until the block is evaluated. The local variables inside a block are defined between vertical bars.

Example: [:param| |x| param doSomething] (a block defined with one parameter, param, and one local variable *x* which is not used).

Control Structures. The Boolean objects have the method ifTrue:ifFalse:, which can be used to build selection. Thus, instead of having language constructs for if then else, there are just plain messages sent to a Boolean object. For example:

```
(total = 0)
    ifTrue: [...]
    ifFalse: [...]
```

(if total equals 0 the first block is evaluated, else the second is evaluated)