Scripting Interactive Visualizations

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michael Meyer

im November 2006

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Dr. Tudor Gîrba

Institut für Informatik und angewandte Mathematik

Abstract

Data visualization is an important tool in reverse engineering. With a good visualization the interesting parts of the underlying data can be detected faster than by merely inspecting the raw data.

One peculiarity of the existing visualization tools is the fact that they implement a finite set of specific visualizations. These specialized tools are not flexible enough to support the user when a slightly or sometimes even drastically different visualization is needed. Often the user needs to be familiar with several visualization tools with each tool expecting a different input format. Usually a large amount of time is being invested into converting the data into the format that is expected by the visualization tool.

We propose a new visualization model that is designed to minimize the *time-to-solution*. We achieve this by working directly on the underlying data, by making *nesting* an integral part of the model and by defining a powerful scripting language that can be used to define visualizations. We support exploring data in an interactive way by providing hooks for various events. Users can register actions for these events in the visualization script. As a validation of our model we implemented the framework *Mondrian* and used it to implement several established visualizations.

Acknowledgements

First I wish to thank my supervisor Dr. Tudor Gîrba for this really interesting year. Being able to just walk into your office whenever I had a question was great. I also thank you for constantly providing me with new ideas and of course for the discussions we had about them. Thank you for challenging me ("This should be easy, right?") every so often. This made things a lot more interesting :-)

I thank Prof. Dr. Oscar Nierstrasz, head of the Software Composition Group, for giving me the opportunity to work in his group and especially for providing me with a solution for my administrative difficulties.

I thank Prof. Dr. Stéphane Ducasse, Prof. Dr. Oscar Nierstrasz and Orla Greevy for reading this master thesis. Thanks for all the constructive feedback.

I thank Adrian Kuhn for the interesting code optimization discussions we had. I also thank him for renaming his version of RectangleShape to RectangleFoo and for sending me a Youtube link every now and then.

I thank Rafael Wampfler for, well, hanging around in the students pool. "Working" was always fun when you were around. I would also like to thank all the other Software Composition Group members. I really enjoyed the discussions and lunches I had with all of you.

I thank Frédéric Pluquet for showing me how to implement sequence diagrams with Mondrian three months before I managed to do so. You really raised the bar.

Finally I would like to thank my parents for all the support they gave me in various occasions. It is great to know that there are people that always stand by you no matter what you do. I also thank my brother for all the things he taught me about brake fluid, valves and lambda sensors lately. The discussions combined with the practical exercises were more than welcome when my head was (once again) too full with algorithms, visualizations and clipping problems.

Michael Meyer
November 2006

Contents

1 Introduction

When confronted with a new kind of problem we often start with a small sketch on a piece of paper [Snyder, 2003; Bellin and Simone, 1997]. The sketch helps us arrange our ideas [Crapo *et al.*, 2000]. It seems that the human mind needs the additional visual aid for a better understanding of complex problems and coherences [Wade and Swanston, 2001].

The world is complex, dynamic, multidimensional; the paper is static, flat. How are we to present the rich visual world of experience and measurement on mere flatland? [Tufte, 1990]

Tufte describes the problem accurately: We have a complex, multilayered problem and have to condense it until it fits on a finite piece of paper or computer screen while still keeping its essence.

Visualization is an important tool in reverse engineering. Koschke reports that 80% of interviewed researchers consider visualizations as being important or absolutely necessary in software reverse engineering [Koschke, 2003]. Recently many new visualizations have been developed. Some focus on the changes of a system during time [Gîrba, 2005] others on email content visualization [Viégas *et al.*, 2006] and again others on runtime analysis [Greevy *et al.*, 2006; Reiss, 2006; Ducasse *et al.*, 2004a]. All these visualizations have the same goal: Making it easier to understand complex coherences.

The enormous interest in visualization as an aid for reverse engineering and problem detection can also be inferred from the large number of tools that have been developed for this purpose: Seesoft [Eick *et al.*, 1992], Rigi [Müller, 1986; Müller and Klashinsky, 1988], SHriMP [Storey and Müller, 1995; M.-A. D. Storey and Michaud, 2001], CodeCrawler [Lanza, 2003a; Lanza and Ducasse, 2005], etc.

1.1 Challenges for an information visualization framework

One peculiarity of the existing visualization tools is the fact that they implement a finite set of specific visualizations. The very nature of program optimization, problem detection and program understanding though, implies that the user might not know a priori what he is looking for, and therefore, what visualizations to use. Reiss identified the following two reasons for the slowness of the software developer community in adopting software visualization tools [Reiss, 2001]:

- Specialized tools are not flexible enough. They do not support the user when he needs a slightly or sometimes even drastically different visualization.
- The user needs to be familiar with several visualization tools with each tool expecting a different input format. Usually a large amount of time is being invested into converting the data into the format that is expected by the visualization tool: by having the model classes implement a certain interface, by adding wrapper objects around the model classes or by writing a tool that exports the model to a XML or TXT file.

We argue that a visualization framework that strives to be flexible, needs to consider the following factors:

The visualization engine should be domain independent. It is important that the framework be general enough to accommodate any data model.

Visualizations should be easily composed from simpler parts. The user of the framework should be able to define basic blocks that can be used to build more complex visualizations. One example of this is the way graph-layouting tools such as GraphViz[1] and aiSee[2] 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. Being able to define a visualization with several levels of detail adds flexibility to the framework. A possible use case is showing the details of a figure by representing other figures inside based on the characteristics of the represented object. The framework should also 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.

[1] http://www.graphviz.org
[2] http://www.aisee.com/

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. Another benefit is that the data model and the visualization model stay in sync when the data model changes.

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.

1.2 Our solution in a nutshell

Often it is not clear from the outset how the given data should be visualized. We argue that the users do not know up front in detail how their solution should appear in the end. This means that they need a tool that allows them to try several ideas with little effort. In this thesis we present a new visualization model that focuses on fast prototyping of visualizations. Our model can also be used as basis for specialized visualization tools.

Figure 1.1 (p.4) shows the essence of our solution. In our model a visualization is defined in a declarative way using scripts. In the script we refer to a *model*. The *model* provides two methods: the `#allClasses` method which returns a collection of class definitions and the `#allInheritances` method which returns a collection of inheritance definitions.

In the example we render a class hierarchy. We use the classes as nodes and the inheritance definitions as edges. The script is structured as follows: We start by creating a *ViewRenderer*. The *ViewRenderer* is used to build and render the visualization. We specify that a node should be created for each class and that the nodes should be painted as bordered rectangles. Next we specify that the inheritance definitions should be represented as edges and that they should be painted as lines. Finally we apply a tree layout to the nodes and render the visualization.

Each inheritance definition provides the methods `#superclass` and `#subclass`. In the third line of the script we specify that for each inheritance definition the edge

```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder.
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```

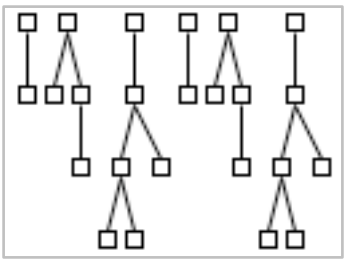


Figure 1.1: A simple visualization based on the Mondrian framework. The script shows the essence of our solution.

should be added between the result of the `#superclass` method and the result of the `#subclass` method.

Our framework works directly on the underlying data. We do not enforce the model to implement a certain interface and we do not need to put wrappers around the model classes. Instead the *shapes* act as a translator between the model interface and the view interface. The advantage of working directly on the underlying data is the fact that we do not need to recreate the visualization model when the data model changes.

Our model is *graph*-based [Battista *et al.*, 1999]. We have a graph behind every *figure*. A graph may contain other figures (*e.g.,* nodes and edges). In this way we support "infinite" nesting. In our approach, a figure does not have a visual representation. The reason why we still need a figure is that the underlying graphical framework usually expects a visual component to inherit from a special class (*e.g.,* VisualPart in Smalltalk or JComponent in Java).

The figures are decorated with *shapes*. Unlike a figure, a shape has a visual representation. The framework provides basic shapes like *RectangleShape* or *LineShape*. Shapes hold no data model specific state. Thus we are able to use the same shape to paint several figures. We decided to use stateless objects because one of our design goals was to minimize the overhead in terms of memory and time introduced by our framework.

All shapes have reasonable default values. Like this the shapes can be created

with the default constructor. We believe that this is an important feature for fast prototyping. Most shapes also have specific properties that the user may set if required. The *RectangleShape*, for example, supports the properties `color`, `width` and `height` and the *LineShape* supports the properties `color` and `lineWidth`.

The user specifies the visualization using a script. Like this the entire visualization can be defined in a single method. This allows the user to focus on the visualization he would like to achieve. In contrast to our script-based approach to visualization, frameworks that rely on subclassing have a tendency of becoming complicated and one usually ends up with spending more time in creating the classes that the framework wants than in creating the visualization.

1.3 Contributions

In this thesis we define a model [Meyer *et al.*, 2006] that is flexible enough to accommodate many different kinds of visualizations while keeping the complexity that is exposed to the user to a minimum. With our model the user can focus on his data and the visualization that he would like to build. The contributions of our work are:

1. Based on our analysis of the existing visualization tools and frameworks we identify a number of factors that are crucial for any flexible visualization model.
2. We implemented a full-fledged prototype of our model in Smalltalk and a basic version in Java.
3. We show how several established visualizations can be expressed with our approach.

1.4 Document structure

In **Chapter 2 (p.7)** we present the state of the art. In the first part we look at a variety of visualization tools and frameworks and in the second part we explore some established techniques for laying out components.

In **Chapter 3 (p.19)** we present our solution from a user point of view and we give a detailed introduction to our model.

In **Chapter 4 (p.35)** we validate our model by using *Mondrian* to implement various established visualizations.

In **Chapter 5 (p.53)** we present some of the problems that we had to solve and we look at implementing our model with different programming languages.

In **Chapter 6 (p.65)** we recapitulate the key features of our model.

2 State of the art

The main problem of software visualization is the fact that too many tools specialize on a limited number of visualizations. This leads to several problems. One problem is that a user cannot be familiar with all the tools. Another problem is that one usually has to convert the data in some way to make it compatible with the format that the tool expects which is time consuming. Furthermore we have to consider that it takes time to get familiar with a new tool.

What is needed is a single, cohesive visualization environment that can be readily adapted to a user's needs, so that he can learn the tool once and apply that knowledge to any problem [Bosch et al., 2000]

2.1 Approaches to Visualization

In this section we survey a number of existing visualization tools and frameworks. We are especially interested in the features that they provide with respect to our goal of providing a model for fast prototyping of visualizations.

2.1.1 GraphViz

GraphViz [Gansner and North, 2000] is a popular visualization tool that focuses on drawing directed graphs. The drawing can be described with a simple language[1] that supports graphs, nodes and edges. GraphViz supports subgraphs. This means that a node in a graph may contain another graph. This is an important feature since it allows us to express complex coherences that could not be expressed with an ordinary graph.

GraphViz includes a number of well-tuned layout algorithms [Gansner *et al.*, 1993]. Well-tuned means that these algorithms are very fast, they are capable of handling cycles in the graph definition and they take aesthetic measurements into account.

[1] http://www.graphviz.org/doc/info/lang.html

GraphViz also supports a large number of output formats (*e.g.,* GIF, PNG, SVG, VRML, PDF, PostScript, ...). Although some of the formats (*i.e.,* SVG and VRML) would be able to handle some kind of user interaction, GraphViz does not make use of this.

```
digraph G {
  a -> b -> c;
  b -> d;
  d -> a

  a [shape=polygon, sides=5, peripheries=3,
     color=deepskyblue, style=filled];
  b [shape=polygon, sides=4, skew=0.4,
     label="b aka B",color=greenyellow,style=filled];
  c [shape=polygon, sides=3, color=orange,
     style=filled];
  d [shape=circle, color=gold1, style=filled];
}
```

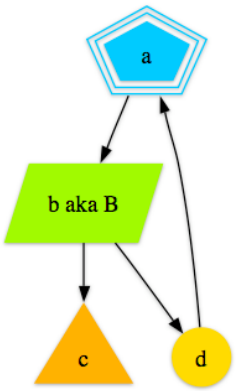


Figure 2.1: GraphViz definition of a graph and the resulting drawing. The graph consists of the nodes a to d. Notice how all the nodes have different colors and shapes and how the edge between node d and node a is slightly bent to avoid intersection with node b.

In GraphViz the user has, within bounds, the possibility to influence the way that a node or edge is drawn. A node can be drawn as box, circle, ellipse or as polygon and edges support several attributes like color, dotted (paints an edge as dotted line) or stroked (paints an edge as stroked line). GraphViz also supports more complicated node representations by allowing the user to compose boxes and labels in an html-table kind of way. Figure 2.1 (p.8) shows an example of a GraphViz graph definition and the resulting drawing.

Since GraphViz only creates static drawings there is no way to explore the data in an interactive way. Another drawback is the fact that GraphViz does not work on the underlying data directly. By transforming the data model to the format that GraphViz expects we duplicate the needed resources.

2.1.2 SHriMP

SHriMP (Simple Hierarchical Multi-Perspective) [Storey and Müller, 1995; M.-A. D. Storey and Michaud, 2001] is a tool for visualizing and analyzing hierarchical data. Though mainly used for analyzing and visualizing large software systems it

is not limited to this domain. The tool is graph-based and represents nodes as rectangles and edges as lines. SHriMP supports nesting. Nesting is a more general term for what we introduced as subgraphs in Section 2.1.1 (p.7). The term nesting is the more common one in the visualization community since it moves the focus from graph theory to the visualization field. When talking about nesting we mean that it is possible to embed an entire visualization inside a node as opposed to embedding a graph inside a node. The authors have implemented several layouts. Especially the SpringLayout and the RadialLayout produce aesthetically pleasing results.

A interesting feature that SHriMP offers is semantic zooming [Bederson and Hollan, 1994]. This is particularly interesting when exploring data in an interactive way. Semantic zooming means that we can define several levels of detail for our data. Figure 2.2 (p.9) shows an example of semantic zooming. We start with the Java package *bingo.game*. Then we display all the classes that are in the package and for the class *NotaryPublic* we also show the methods. So basically semantic zooming is nothing other than an application of nesting.

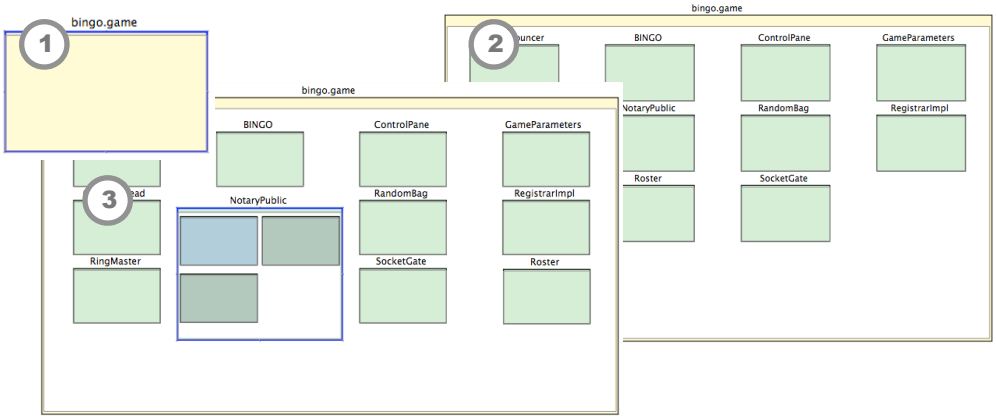


Figure 2.2: An example for semantic zooming. (1) shows the Java package *bingo.game*. Then we add the classes that the package contains (2) and finally we add the methods for the class *NotaryPublic* (3).

The standalone version of SHriMP loads its data from a file with a graph-like structure (*e.g.,* GXL, RSF, XML, XMI). There are also two sub-projects: Jambalaya is a plugin for Protégé[2] that can be used for visualizing ontologies and knowledge-base data and Creole is an eclipse plugin for source code visualization.

[2]http://protege.stanford.edu/

SHriMP does not work on the model directly. This means that we have to export our model each time we perform changes on it. The need to export and import the data leads to a very slow iteration loop. Semantic zooming only works if we export all the necessary data. It also seems that SHriMP is not meant to be easily extended with custom figures. So, as a typical user, we are limited to the basic shapes that SHriMP offers.

2.1.3 GEF

The Graphical Editing Framework (GEF) is part of the Eclipse Modeling Project [Moore *et al.*, 2004]. GEF assumes that you have a model that you would like to display and edit graphically. Unlike the previous tools GEF works with any kind of model and close to the original data. The framework makes use of the Model - View - Controller pattern. The controller classes are subclasses of EditPart. EditParts handle the communication between the model and the view. In GEF the communication works in both ways. This means that the EditPart propagates changes from the model to the view and also from the view to the model. The EditPart is also responsible for creating the visual representation of the model object. To create the EditParts the developer needs to provide a factory that returns the proper EditPart for each model entity. Based on this it is possible to have an instance-based representation of the model objects. The framework also supports nesting and the user has a lot of freedom when it comes to defining the visual representation of a model entity since he has access to the entire Java 2D API.

The Graphical Editing Framework is a powerful and easy to use framework with good interaction support. With interaction we mean that the user has the possibility of resizing figures with the mouse, that it is possible to move figures around with the mouse, that we have access to the model data behind a figure and that we can add and remove figures on the fly.

A problem of the GEF framework is the linking between the model and the controller. It is not possible to store a state in the controller although the framework creates a controller for every model object. So when we need to store a state like the x, y position of a figure we can either store this information in the model or we need to add a wrapper around the model object. This leads to a large number of objects. In the worst case we have a model object, a controller object and a wrapper object. So basically the framework triples the number of required objects.

2.1.4 CodeCrawler

CodeCrawler [Lanza, 2003a; Lanza and Ducasse, 2005] is a tool specialized for understanding object-oriented software systems. To work with CodeCrawler no programming skills are required. The tool comes with a rich and intuitive user interface and is capable of working with data provided by the Moose reengineering environment [Ducasse *et al.*, 2005; Nierstrasz *et al.*, 2005].

Defining a visualization in CodeCrawler is straightforward. The complete visualization can be defined at runtime with the mouse. Like most visualization tools CodeCrawler is graph-based. The user can define which model entities should represent the nodes and which model entities should be used as edges. The user also has the possibility to map metrics that the model provides to properties that the view needs. The rectangle implementation, for example, has the properties width, height, fill-color, border-width and border-color. These kind of visualizations where one maps model metrics to view properties are known as polymetric views [Lanza and Ducasse, 2003].

CodeCrawler also comes with a set of ready to use visualizations such as the *Class Hierarchy* and the *Class Blueprint* to name two of the well-established ones and several developers have used CodeCrawler as a basis for their own visualizations [Gîrba *et al.*, 2005; D'Ambros and Lanza, 2006b; D'Ambros and Lanza, 2006a; Arévalo, 2005].

Although CodeCrawler has been used as a base for custom visualizations the process of adding a new visualization is not trivial. Also CodeCrawler, like GEF, puts wrappers around the model entities which doubles the needed resources and it requires the visualization model to be recreated when the data model changes. Moreover CodeCrawler only supports type-based representations.

2.1.5 Seesoft

Seesoft [Eick *et al.*, 1992] is a tool that is specialized for visualizing large amounts of data (50000 lines of code or more). The tool expects version control data or the result of a static or dynamic analysis as input data. The tool will map every line of the input data to a thin line in the visualization. The color of the line will be set based on a user criterion. When working with version control data it uses red for all lines that have been changed a short while ago and blue for all lines that have not changed for a long while.

Seesoft does not support any other kind of visualization. The advantage of this approach is that the developers are able to optimize the code for this specific task

which is desirable, from a user point of view, when working with a huge amount of data. It can be expected that Seesoft is faster than a more generalized tool that offers the same visualization. The downside of the specialized tool approach is that users who need several different visualizations will need to know several different tools. In the worst case each of these tools expects the input data in a different format.

2.1.6 BLOOM

BLOOM [Reiss, 2001] is a tool that is optimized for software visualization. It offers a large number of different visualizations, 2D as well as 3D, and supports various data formats. Data can be obtained from a database or by parsing the source code and there are even facilities for collecting Java and C++ traces. BLOOM then provides a powerful query language to collect the data that should be visualized from these sources. Then BLOOM starts a search and looks for all defined visualizations that could be applied to the provided data. The end result of this process is a XML file that can be visualized by the BLOOM rendering engine.

BLOOM provides a plugin mechanism that makes it possible to add new visualizations hence the fact that BLOOM already offers a large number of visualizations.

A feature that BLOOM is missing is nesting. Furthermore since the focus of the project is on software visualization, it seems unlikely that the tool could be easily adapted to other domains.

2.1.7 Vizz3D

As the name suggests Vizz3D [Panas *et al.*, 2005] is a tool that focuses on 3D visualizations. Vizz3D is intended for users and not primarily for developers. Vizz3D has an easy-to-use user interface and the user can create the entire visualization with the mouse. The tool supports polymetric views and comes with a rich set of available visualizations.

Vizz3D focuses on software visualization. But adding new visualizations and data bindings is possible which makes Vizz3D not only a tool but also a framework. To add new bindings and visualizations programming skills are needed though. We did not find any evidence that Vizz3D supports nesting and it is not clear what kind of input data Vizz3D expects.

2.1.8 MetricView

MetricView [Termeer *et al.*, 2005] is a software visualization tool that turns traditional UML diagrams into polymetric views. The starting point is a plain UML diagram. The user can then define an "invisible" grid on top of the UML figure and map a metric to each cell. The visual representation of the metric is set by the user. It is possible to map a metric to an icon or to a rectangle where width and height can be based on the value of the metric. No programming skills are required to use MetricView. The entire visualization can be defined with the mouse.

MetricView loads the UML data and the metric data from two different files. The UML data is loaded from XMI (XML Metadata Interchange) files. The XMI format is a standard that is supported by many UML modeling tools such as *Rational Rose*[3] and *Together*[4]. The metric data can be loaded from files created by a tool called *Software Architecture Analysis Tool* [Muskens, 2002; Lange, 2003]. According to the authors loading metrics from other sources can be achieved with little effort.

MetricView is an end user tool that focuses on the various UML diagrams. This specialization does not make MetricView an ideal base for custom visualizations. Additionally we did not find any evidence that the diagrams that MetricView creates support interaction.

2.1.9 Softwarenaut

Softwarenaut [Lungu *et al.*, 2006] is a reverse engineering tool for package-based systems that uses Moose [Ducasse *et al.*, 2005] as data provider. Figure 2.3 (p.14) shows a screenshot of Softwarenaut while exploring the BitTorrent client Azureus[5]. Softwarenaut displays three levels of detail at the same time which is helpful for exploring a system in an interactive way. The *Exploration Perspective* is the main view where the user can collapse and expand packages. The *Map Perspective* gives an overview over the complete package hierarchy and highlights the nodes that are currently visible in the exploration perspective. The *Detail Perspective* provides details on the package which has the focus in the exploration perspective.

Softwarenaut is easy to use. When selecting a package in the exploration perspective the detail perspective gets updated immediately. One can right click on an arbitrary node in the map perspective and make that package the current selection

[3] www.ibm.com/software/rational
[4] http://www.borland.com/de/products/together
[5] http://azureus.sourceforge.net

in the exploration perspective. As a navigation aid Softwarenaut indicates which packages are interesting to explore based on package patterns.

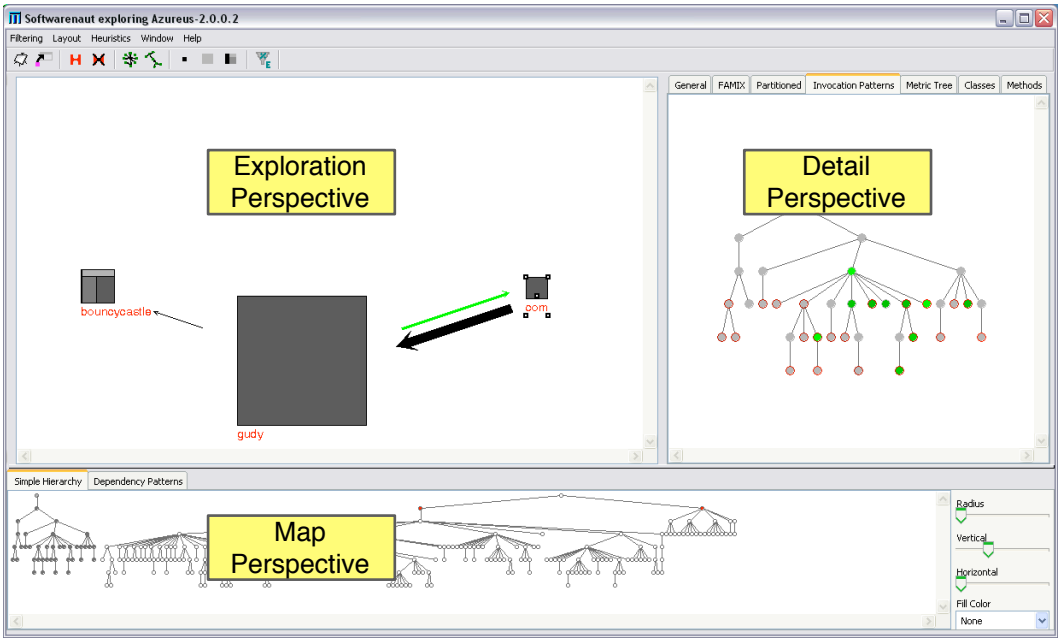


Figure 2.3: Softwarenaut is a reverse engineering tool for package-based systems. The tool focuses on interactive navigation.

While Softwarenaut is mainly used for analyzing package-based software systems it has also been used for navigating clusters [Lungu *et al.*, 2005].

Softwarenaut is a dedicated tool. With the three perspectives (*i.e.*, Exploration, Map and Detail perspective) even the user interface is highly specialized. The interaction between the three perspectives is hard coded. This makes it difficult to extend Softwarenaut with custom visualizations. Softwarenaut does not work with the data model directly. Instead it puts wrappers around the model objects. This duplicates the number of needed objects and we need to recreate the visualization model when the underlying data changes.

2.2 Declarative ways for composing components

Since we want to be able to script the visual representation of *figures* by composing *shapes* we explored some established ways of composing components.

2.2.1 CSS3

In the early days of the web it was common to have document content and document presentation in the same file. This was impractical for several reasons, the most serious one being that each file had to be edited if one wanted to change something like the font size of a title or the color of an emphasized word in a consistent way. This is why CSS[6] (Cascading Style Sheets) was introduced. CSS allows a complete separation of document content and document presentation.

While formating text works well with CSS it is still difficult to express complex layouts. This is why it is still common to use html-tables to define the overall look of a document. CSS3, the upcoming version of CSS, will have a special module to solve this problem. The module is called *CSS3 Advanced Layout Module*[7] and it offers all the functionality that the classical html table provides.

```
<style type="text/css">
  body {
    display: "a.b.c" (2em)
             "....." (1em)
             "d.e.f"
             "....." (1em)
             "g.h.i" (2em)
             5em 1em * 1em 10em
  }
  #logo {position: a}
  #motto {position: b}
  #date {position: c}
  #main {position: e}
  #adv {position: f}
  #copy {position: g}
  #about {position: h}
</style>
```

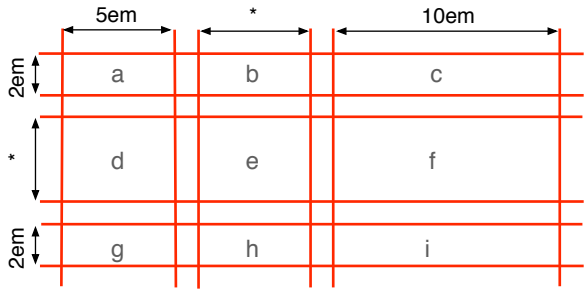


Figure 2.4: Example of a CSS3 layout definition and an outline of the grid that the definition creates.

Figure 2.4 (p.15) shows an example of a CSS3 layout definition and an outline of the grid that the definition creates. In the example we create a grid that consists of

[6] http://www.w3.org/Style/CSS/
[7] http://www.w3.org/TR/css3-layout/

five columns and five rows. The letters a to i are placeholders for a component and a point defines a column or row that will not contain a component. As distance unit we use *em*. This unit depends on the used font and is helpful for providing a similar look on different platforms. The middle row and the middle column are flexible. This means that they will take up all remaining space that the fixed rows and columns do not need. The first column is 5em wide, the second column 1em and the last column 10em.

After defining the grid we can assign html tags to the slots. CSS3 offers a new *position* property for this. In Figure 2.4 (p.15) we assign the tag with id `#logo` to slot a. A component can span several slots if we give the same name (*e.g.,* a) to more than one slot. If we wanted the logo to span the entire first row we could replace b and c by an a in Figure 2.4 (p.15).

Positioning and aligning components in a grid is straightforward. The table metaphor can also be used in the visualization domain for composing shapes and defining the visual representation of figures. In a simplified form this has already been done by GraphViz (Section 2.1.1 (p.7)) where it is possible to compose boxes and labels in a html-table kind of way.

What is not covered by the *CSS3 Advanced Layout Module* is resize behaviour. Web pages do not need this functionality since *web browsers* have scrollbars. However, in an interactive visualization model resize functionality is needed.

2.2.2 Form Layout

When building a GUI with Java one makes use of a *LayoutManager*. The LayoutManager is responsible for position and size of the components inside a container. While components can indicate their preferred size and alignment the LayoutManager has the final say. The LayoutManager is also responsible for resizing the components if the size of the container changes.

One of the most flexible and also most complicated LayoutManagers that Java offers is the GridBagLayout[8]. An alternative to the GridBagLayout is the FormLayout [Lentzsch, 2004]. The FormLayout is offered free of charge by JGoodies [9], a company specialized on advanced Java UI design and usability. The FormLayout provides the same flexibilty as the GridBagLayout but is easier to use.

The FormLayout works with rows and columns. After defining the grid we can start adding the components to the slots. A component can span several rows and

[8] http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html
[9] http://www.jgoodies.com

```
FormLayout layout = new FormLayout(
    "10dlu, pref, 4dlu, fill:pref:grow", //columns
    "pref, 2dlu, pref, 3dlu, pref"); //rows

PanelBuilder builder = new PanelBuilder(layout);
builder.setDefaultDialogBorder();
CellConstraints cc = new CellConstraints();

builder.addSeparator("General Information", cc.xyw(1, 1, 4));
builder.addLabel("Firstname", cc.xy (2, 3));
builder.add(new JTextField(), cc.xy (4, 3));
builder.addLabel("Lastname", cc.xy (2, 5));
builder.add(new JTextField(), cc.xy (4, 5));

this.getContentPane().add(builder.getPanel());
```

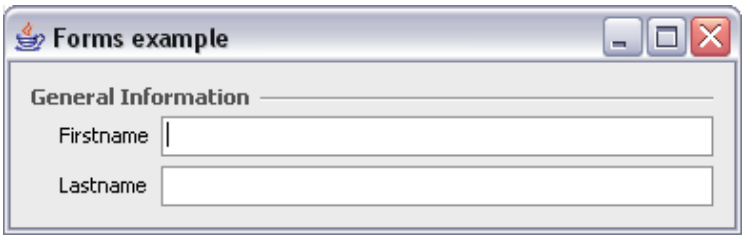


Figure 2.5: Screenshot of a Java UI and the code that creates it. In the code we make use of the FormLayout.

columns and we have a fine-grained way of influencing the behavior of a component when resizing the surrounding container.

Figure 2.5 (p.17) shows a screenshot of a Java UI and the code that creates it. The UI has been built with the FormLayout. In this example we define a grid with four columns and five rows. Most cells have a fixed width and height. As distance unit we use *dlu*. This unit, like *em*, is based on the used font and produces a better result if the UI is used on different platforms. For the width of the fourth column we use the term `fill:pref:grow`. The `pref` indicates that the column should use the preferred width of the contained component if possible. With the `fill` attribute we tell the layout that a component should use all available space. This means that a component will fill the entire cell, especially when the column width is larger than the preferred width of the component. Finally the `grow` attribute defines the resize behavior of the column. If we increase the width of the container, the columns with the `grow` attribute will distribute the newly available space among each other. After defining the grid we can start placing the components into the cells. A component can span several rows and columns.

The strength of the FormLayout is the compact layout definition and the manner in which the resize behaviour can be defined.

3 Scripting Visualizations with Mondrian

In the previous chapter we looked at several visualization tools and we introduced concepts like *nesting* and *instance-based representation* that are important in the field of visualization.

In this chapter we present our model with various examples. We will use *Mondrian*, the Smalltalk implementation of our model, to render the visualizations. After looking at the model from a user point of view we present the internal details of our model.

3.1 Building our first visualization: A class hierarchy

In the following examples we will use a simple model that contains classes and the inheritance relations between those classes. The method `#allClasses` returns a collection with all classes and the method `#allInheritances` returns a collection with all inheritance relations. We always present the script and the visualization that it produces.

Creating a view. We have designed Mondrian to work like a view that the user paints. We start by creating an empty view.

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

Adding nodes. Next we add a node for each class object to the visualization. We obtain the class objects by calling the model's `#allClasses` method. For the visual representation of the objects we choose a bordered rectangle.

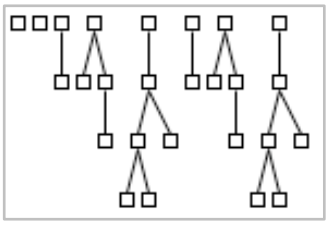
```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder.
view open.
```

Adding edges. In our example model the inheritance relations are modeled as first-class entities. We obtain the inheritance objects by calling the model's `#allInheritances` method. In the script below we make use of `#superclass` and `#subclass`. These are methods that the inheritance objects implement. The method `#superclass` returns the class object of the superclass and the method `#subclass` returns the class object of the subclass. For the visual representation of the edges we use a line.

```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder.
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view open.
```

Layouting. By default the nodes are arranged in a horizontal line. Since we have hierarchical data a tree layout is more appropriate.

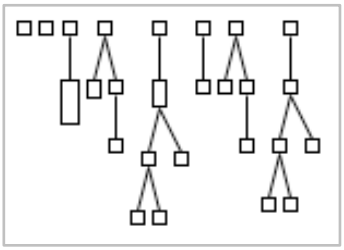
```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder.
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```



Polymetric size. Our model supports polymetric views [Lanza and Ducasse, 2003]. The class objects implement the methods `#noa` and `#nom`. The `#noa` method returns the number of attributes and the `#nom` method returns the number of methods in the class. In the next example we map the width of the rectangle to

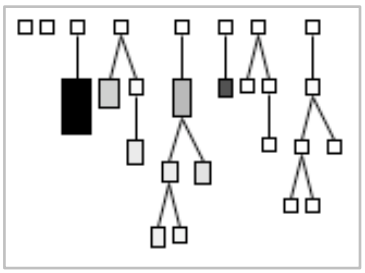
the result of the `#noa` method and the height of the rectangle to the result of the `#nom` method.

```
view := ViewRenderer new.
view nodes: model allClasses
     using: (RectangleShape withBorder width: #noa height: #nom).
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```



Polymetric color. We assign to each node a color between white and black based on the *number of lines of code*. In our model the *number of lines of code* can be obtained by executing the class object's `#loc` method. In the script we make use of a class called *LinearNormalizer*. The class takes as the context all objects that should get a color and as the command the name of the method that should be used for determining the color.

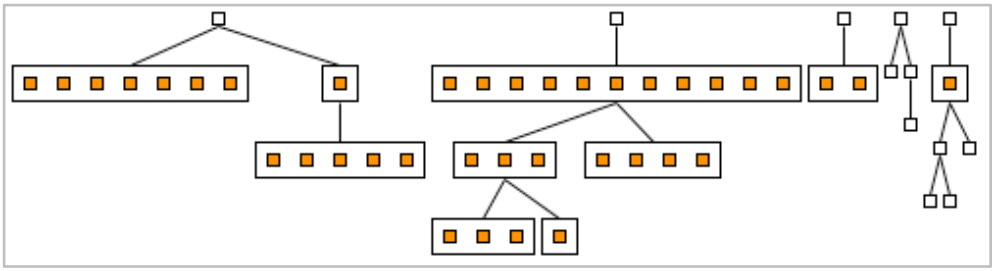
```
view := ViewRenderer new.
view nodes: model allClasses
     using: (RectangleShape withBorder width: #noa height: #nom color:
                (LinearNormalizer
                     context: model allClasses
                     command: #loc)).
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```



At this point we have produced a complete System Complexity view [Lanza, 2003b] with a script that consist of only five instructions.

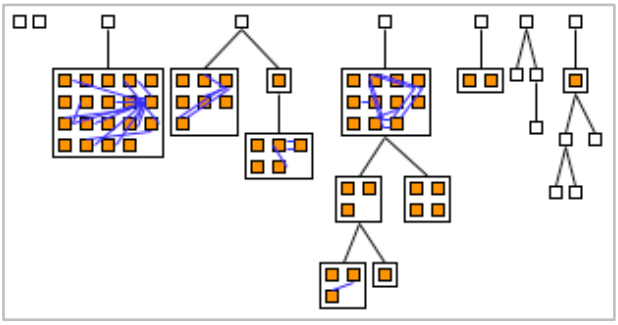
Nesting. The class objects implement the method `#allMethods` that returns a collection with method objects. We would like to visualize the methods as orange rectangles that are surrounded by a border. Until now we used the command `#nodes:using:` for adding the class objects. Now we use the command `#nodes:using:forEach:`. The code that we provide in the `forEach` block will be evaluated for each class object.

```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder
     forEach: [:class |
         view nodes: class allMethods using: RectangleShape orange.
     ].
view edges: model allInheritances from: #superclass to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```



Local edges. Our model allows edges to be added in a controlled way. Since we have methods in our visualization we can visualize the method invocations. Again we assume that in our model the invocations are first-class entities. The model's `#allInvocations` method returns a collection with invocation objects. Each invocation object implements the methods `#invoces` and `#invokedBy`. To emphasize the difference between the invocation edges and the inheritance edges we will use blue for the invocation edges and black for the inheritance edges.

```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder
     forEach: [:class |
        view nodes: class allMethods using: RectangleShape orange.
        view layout: GridLayout new.
        view edges: model allInvocations
             from: #invokedBy
             to: #invokes
             using: (LineShape color: Color blue).
     ].
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```

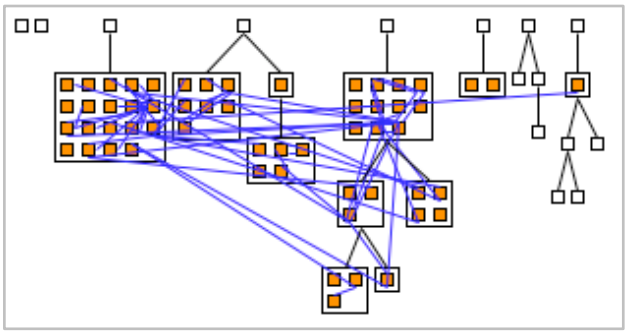


The full method to add edges is `#edges:from:to:using:fromGlobal:toGlobal:`. With the `fromGlobal:toGlobal` part of the method we can influence the scope of the edges. Setting `fromGlobal` or `toGlobal` to true means that all figures in the visualization will be taken into account. Setting them to false means that only figures on the same nesting level or below will be considered. In the method `#edges:from:to:using:` we set `fromGlobal` and `toGlobal` to false. Like this we can influence the scope of the edges by placing the call on the appropriate nesting level.

In the previous script we only see the method invocations within a class since the `#edges:from:to:using:` call was inside the `forEach` block. We call this kind of edge with a narrow scope a *local edge*.

Global edges. To see the method invocations between different classes we move the `#edges:from:to:using:` call to the outermost position. These top-level edges have the broadest possible scope and thus we call them *global edges*.

```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder
     forEach: [:class |
        view nodes: class allMethods using: RectangleShape orange.
        view layout: GridLayout new.
     ].
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view edges: model allInvocations
     from: #invokedBy
     to: #invokes
     using: (LineShape color: Color blue).
view layout: TreeLayout new.
view open.
```



Saving a reusable script. Another feature of our model is the fact that scripts can be reused. For this we take another look at the previous example. In that example we have one large script where we add classes, methods and edges. The script can be split into two visualizations.

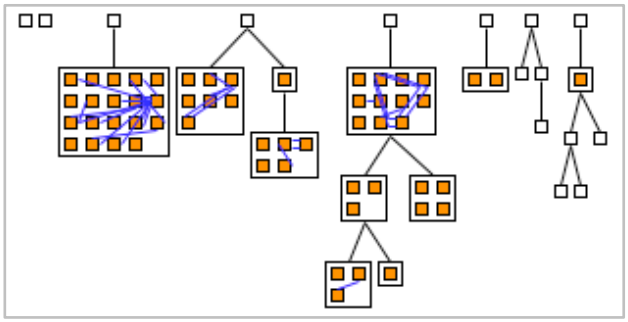
When we want a script to be reusable we store it in a method. The only constraint to a reusable script is that it is not allowed to create its own instance of *ViewRenderer*. The method should accept an existing instance of a *ViewRenderer* as parameter.

In the following script we add the method `#addMethodsView:` directly to our data model. Keeping the data and the visualizations close together is quite handy but our model does not enforce this. If we do not want to "pollute" our model with visualization scripts we can create a library class where we store our scripts.

```
Class>>addMethodsTo: view
    view nodes: self allMethods using: RectangleShape orange.
    view layout: GridLayout new.
    view edges: self allInvocations
         from: #invokedBy
         to: #invokes
         using: (LineShape color: Color blue).
```

Reusing a script. In the next example we will produce the same result as in the *local edges* example but this time by reusing the script we have defined in the method `#addMethodsView:`.

```
view := ViewRenderer new.
view nodes: model allClasses using: RectangleShape withBorder
     forEach: [:class | class addMethodsTo: view].
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```



Notice how the script becomes smaller and easier to read. By allowing the user to reuse existing visualizations it is possible to build complex visualizations by composing several simple visualizations. On the other hand a user can also choose to decompose a complex visualization into several simpler blocks which helps to keep the whole visualization maintainable.

3.2 Adding interaction to the Visualization

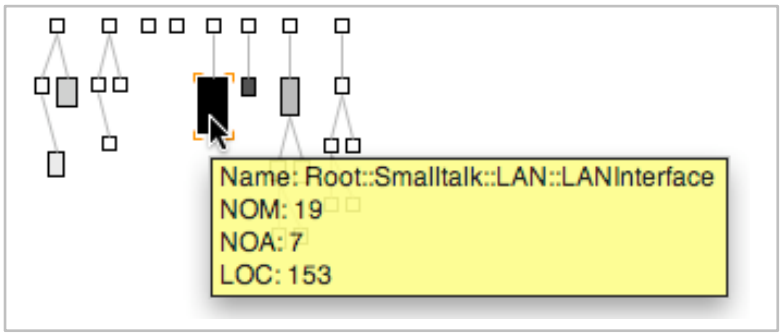
Our model supports interaction in a fine-grained way. On one hand we offer hooks for several events, and on the other hand the user can decide for each figure which events he would like to register.

We support a number of different events. The `onClick` and `onDoubleClick` events occur when a user clicks on a figure. When a user selects or deselects a figure we raise an `onSelect` or an `onDeselect` event. Opening a figures menu with a right click raises an `onMenuOpen` event and when the mouse moves over a figure an `onMouseOver` event occurs.

The *ViewRenderer* has the method `#interaction` for registering events. When calling `#interaction` the *ViewRenderer* returns an *EventHandler* instance that will be available until the next call to a `#nodes:using:..` or an `#edges:using:..` method. The reasoning behind this is that we want to give the user the possibility to define different behaviour for different figures and at different nesting levels.

Popup text. A possible use case is shown in the next script where we show a popup window with class-specific information when the mouse moves over a class figure and nothing when we move over an edge figure.

```
view := ViewRenderer new.
view interaction popupText: [:entity |
                    'Name: ', entity longName, Character cr,
                    'NOM: ', entity nom, Character cr,
                    'NOA: ', entity noa, Character cr,
                    'LOC: ', entity loc]).
view nodes: model allClasses using: Shape forClass.
view edges: model allInheritances from: #superclass to: #subclass
     using: LineShape new.
view layout: TreeLayout new.
view open.
```



The method `#popupText:` is a utility method that the *EventHandler* provides. The method is actually a shortcut for defining a specific `#onMouseEnter:` event. The

next script shows the implementation of `#popupText:`.

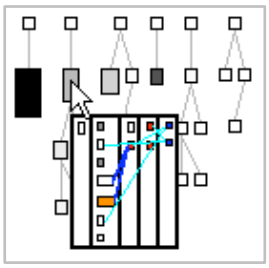
```
popupText: aBlock
    self onMouseEnter: [ :entity | ToolTip show: (aBlock value: entity)]
```

The `#popupText:` method expects a block as argument that will be evaluated with the entity that is behind the figure. In the script we make use of a class called *ToolTip* and the method `#show:`. The `#show:` method expects a string as parameter and displays a popup window at the current cursor position.

Popup window. Another convenience method that the *EventHandler* provides is `#popupView:`. This method expects a block that will be evaluated with two parameters. The first parameter is the entity behind the figure and the second parameter is a *ViewRenderer* instance.

In the next script we use this utility method to show a *Class Blueprint* [Ducasse and Lanza, 2005] in a popup window while the mouse is over a class figure. The zoom factor of the popup window is set to 50% by default but it is possible to set a different zoom factor.

```
view := ViewRenderer new.
view interaction popupView: [:entity :popupView |
                                    entity viewBlueprintOn: popupView].
view nodes: model allClasses using: Shape forClass.
view edges: model allInheritances from: #superclass to: #subclass.
view layout: TreeLayout new.
view open.
```



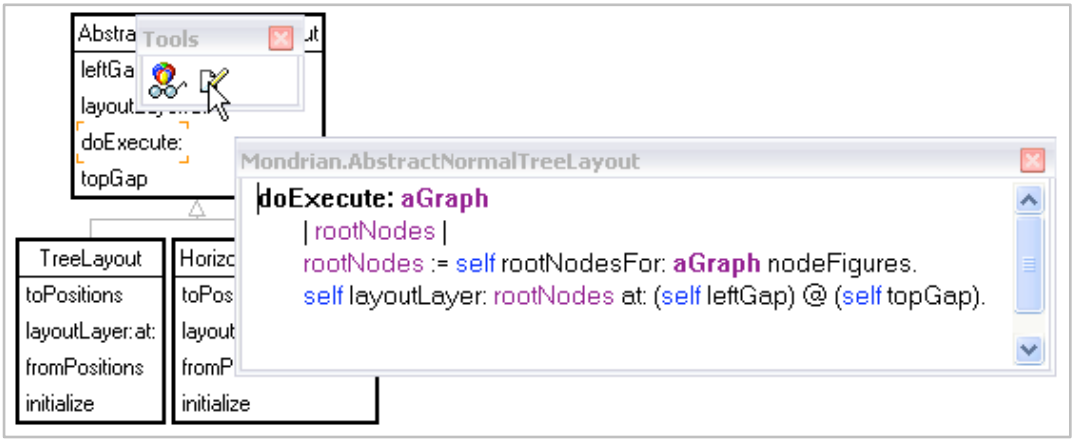
Since the `popupView` variable holds a completely functional *ViewRenderer* instance we can show any visualization we want in the popup window. This is also another example for reusing scripts. We do not need to script the *Class Blueprint* again. We can just reuse the existing script.

Toolbar. In the next example we will script a toolbar that opens when we click on a method. To build the toolbar we use the class *Toolbar*. The public interface of *Toolbar* consists of two methods. The method `#open` opens a toolbar at the current cursor position and the method `#icon:action:` is used for adding actions to the toolbar.

In the script we register the `#onClick` event of the method nodes. In the `onClick` block we create an instance of *Toolbar* and define two actions. The first action opens a *Refactoring Browser* on the selected method and the second action opens an editor that can be used to edit the code of the selected method directly. In the screenshot we opened an editor on the method `#doExecute:` of the class *AbstractNormalTreeLayout*.

We use a *TextEditorView* for editing the method. This is a class that is provided by the environment. In the `#loadBlock:` we specify what should be displayed in the editor and in the `#saveBlock:` we define into which class the method should be compiled.

```
view := ViewRenderer new.
view classShapeWithLabel: #name.
view nodes: self selectedClass withAllSubclasses forEach: [:eachClass |
    view interaction onClick: [:eachSelector |
        Toolbar new
            icon: ImageLib systemBrowser action: [
                RefactoringBrowser class: eachClass; selector: eachSelector; open];
            icon: ImageLib edit action: [
                | window |
                window := EventHandler windowHandle.
                window label: eachClass printString.
                window component: (
                            TextEditorView
                                loadBlock: [eachClass sourceCodeAt: eachSelector ]
                                saveBlock: [:aValue | eachClass compile: aValue ]).
                window open];
            open.
        ].
    view nodes: eachClass selectors using: (LabelShape label: #asString).
    view verticalLineLayout gapSize: 1.
].
view edgesFrom: #superclass using: UmlInheritance new.
view treeLayout.
view open.
```

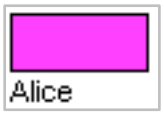


3.3 Composing shapes

In the previous scripts we used bordered rectangles to draw the nodes. To create the bordered rectangle we use the command `RectangleShape withBorder`. This is actually a short form for `RectangleShape new decoratedWith: BorderShape new`. The idea that a shape can be decorated with other shapes is a central part of our model. We provide basic shapes that can be composed by the user without requiring any knowledge of the underlying graphical framework.

Decorating shapes. The following script demonstrates the low-level way of composing shapes. The script creates a pink node with the text "Alice" attached to the bottom left corner. We use the method `#decoratedWith:` to decorate shapes and the method `#align:` to position shapes. In the example we use the keyword `#bottomLeft` to position the *LabelShape* at the bottom left corner.

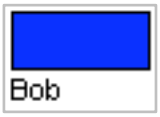
```
view := ViewRenderer new.
view node: 'Alice' using: (
     (RectangleShape width: 50 fillColor: Color magenta) decoratedWith:
     ((BorderShape new) decoratedWith:
     (LabelShape  align: #bottomLeft label: #yourself))
).
view open.
```



What is striking is the fact that the script is difficult to read although we only combined three shapes. The problem is that we need to make extensive use of parentheses which decreases the readability greatly. This is why we offer a number of high-level ways to compose shapes.

Shape. In the next script we create a blue node with the text "Bob" attached to the bottom left corner. But this time we make use of a utility class called *Shape*.

```
view := ViewRenderer new.
view node: 'Bob' using: (Shape new
           add: (RectangleShape width: 50 fillColor: Color blue);
           add: (BorderShape new);
           add: (LabelShape  align: #bottomLeft label: #yourself)).
view open.
```



The *Shape* class only offers the method `#add:`. Internally the class does nothing else than what we did in the previous script but reading and writing scripts becomes

easier.

ShapeBuilder. In the previous example we decorated a *RectangleShape* with a *BorderShape* and the *BorderShape* with a *LabelShape*. In this example we will decorate the *BorderShape* with *two LabelShapes*. Decorating the same shape several times is a more complicated task than only decorating the shape once and although it can still be done with the `#decorateWith:` method and clever use of brackets we do not advise this since the script becomes difficult to read and maintain.

We offer another utility class that is called *ShapeBuilder*. The *ShapeBuilder* implements the methods `#add:` and `#add:with:` that can be used to decorate shapes and the method `#asShape` that returns the finished shape.

```
builder := ShapeBuilder new.
builder add: (RectangleShape width: 250 height: 5 color: Color red) with: [
     builder add: (BorderShape lineWidth:  2) with: [
          builder add: (LabelShape align: #topLeft
                                    boldLabel: 'Align: Top - Left').
          builder add: (LabelShape align: #topRight
                                    boldLabel: 'Align: Top - Right').
     ]
].

view := ViewRenderer new.
view node: 'Align - Example' using: builder asShape.
view open.
```

In the script we compose the shapes with the *ShapeBuilder* before we start defining the visualization. The *BorderShape* is decorated with two *LabelShapes*. One is aligned to the top left corner and the other to the top right corner. To emphasize the labels we made use of the convenience constructor `#boldLabel:` that the *LabelShape* offers.

FormsBuilder. Thanks to the `#align:` method and utility classes like *Shape* and *ShapeBuilder* it is possible to create complex visual representations by merely composing shapes. In Mondrian the user can resize figures with the mouse. While this works without problems most of the time there are cases where the resize behaviour needs to be specified explicitly (*e.g.,* should a shape have a fixed width or height or should it occupy as much space as possible?).

For this we provide another utility class that is completely different from the previous approaches. The class is called *FormsBuilder* and has been influenced by the Form Layout that we introduced in Section 2.2.2 (p.16).

In the next script we start with an instance of *FormsBuilder*. Then we define a

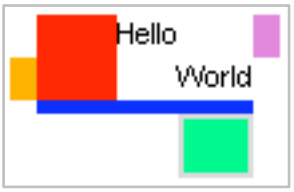
grid that consists of five columns and four rows.

In the definition we use the keywords `fill`, `pref` and `grow`. The keyword `fill` indicates that a shape should take up all available space in a cell even if the width or height of the cell is larger than the shapes preferred width or height, with `pref` we tell the layout manager that the shapes preferred size should be considered if possible and with `grow` we specify that a row or column should take up all the available space that is left at the end of the layouting process.

```
builder := FormsBuilder new.
builder columns: 'pref, fill:pref:grow, pref, pref, fill:pref:grow'.
builder rows: 'fill:pref:grow, fill:pref:grow, pref, pref'.

builder x: 1 y: 2 add: (RectangleShape width: 10 color: Color orange).
builder x: 2 y: 1 h: 2 add: (RectangleShape color: Color red).
builder x: 2 y: 3 w: 3 add: (RectangleShape color: Color blue).
builder x: 3 y: 1 add: (LabelShape label: [:entity | 'Hello']).
builder x: 4 y: 2 add: (LabelShape label: [:entity | 'World']).
builder x: 4 y: 4 add: (RectangleShape color: Color green) with: [
    builder add: (BorderShape lineWidth: 2 lineColor: Color lightGray).
].
builder x:5 y: 1 add: (RectangleShape height: 15 color: Color orchid).

view := ViewRenderer new.
view node: 'Forms Layout' using: builder asShape.
view open.
```



After defining the grid we use the `#x:y:add:` method to add the shapes to the slots. It is also possible to specify that a shape takes up more than slot by using one of the following methods: `#x:y:w:add:`, `#x:y:h:add:` or `#x:y:w:h:add:`.

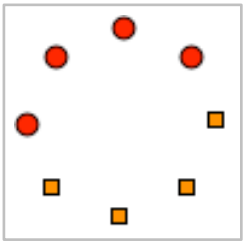
3.4 Instance based visualizations

Unlike many of the existing solutions our model is instance-based as opposed to type-based. This means that not all model objects of the same type need to have the same visual representation. In this example our model consists of all integers between one and eight.

The term `#(1 2 3 4)` is the *Smalltalk* notation for an array containing the numbers one, two, three and four. In the script below we add four numbers that should be painted as orange rectangles and four numbers that should be painted as red circles.

Although this example is fairly simple it shows nicely how our model works with any kind of data and how our model supports instance-based visualizations.

```
view := ViewRenderer new.
view nodes: #(1 2 3 4) using: RectangleShape orange.
view nodes: #(5 6 7 8) using: CircleShape red.
view layout: CircleLayout new.
view open.
```



3.5 Mondrian internals

Mondrian is the prototype that we implemented to validate our model. In this section we will take a look at the design of our framework. Figure 3.1 (p.32) shows the core structure of our framework.

The classes *Figure*, *NodeFigure* and *EdgeFigure* inherit from the underlying graphical framework. At the beginning we used HotDraw [Brandt and Schmidt, 1995] as the underlying drawing framework but as we added more and more features to our framework it got more and more difficult to adapt HotDraw to our needs. This is why at some point we abandoned HotDraw and started to inherit from the underlying graphical framework directly.

One of our assumptions is that there is a model object behind every figure. At this point we actually double the needed resources since we need a graphical object for every model object. Since this is a constraint that the graphical framework enforces on us we cannot avoid this.

Our entire model is graph-based. There is a graph behind every figure. This is how we provide "infinite" nesting. The graph provides methods to add, remove and find *NodeFigures* and *EdgeFigures* and a method to apply a layout. Since there is a graph behind every figure we decided to add all the graph-specific methods to *Figure*. Having a dedicated graph class would triple the number of needed objects since we would have the original model object, an instance of the figure and an instance of the graph.

The layouts are subclasses of *Layout*. The layout algorithm is implemented in the method `#doExecute:`. Since each figure is also a graph the `#doExecute:` method receives a figure as parameter. Like this the layout algorithm has access to a large amount of environmental information. Most layout algorithms only operate on

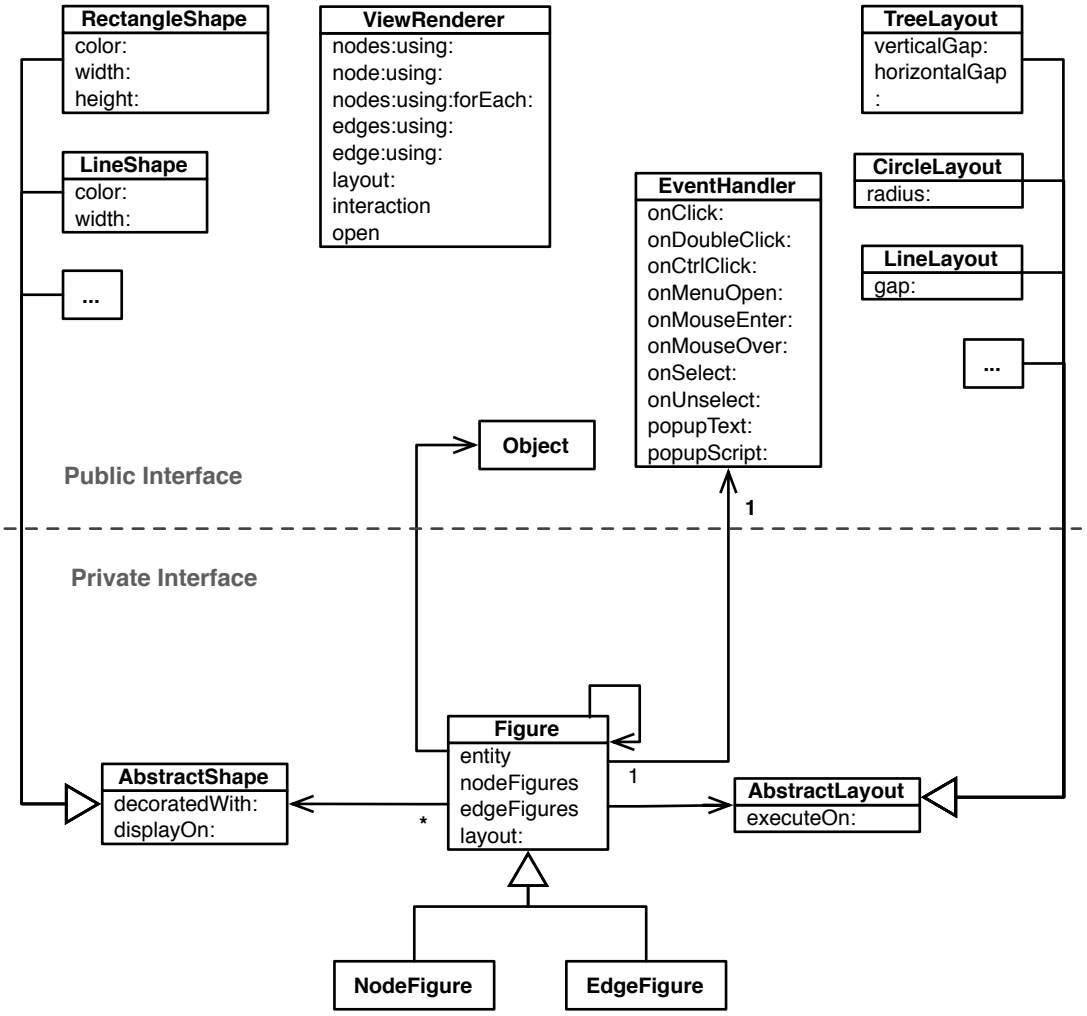


Figure 3.1: The internal model of Mondrian.

nodes but some more advanced layouts need additional information. The *ForceBasedLayout* for example computes the repelling force of a node based on the number of children that the node has.

In our model a figure does not know how to paint itself. Instead a figure forwards all calls to its `#displayOn:` method to its shapes. The shapes are subclasses of *AbstractShape*. We provide several basic shapes (*e.g.,RectangleShape*, *BorderShape*, *LineShape*, *CircleShape*, *etc.*) that can be composed by the users in various ways.

What is special about shapes is the fact that they hold no model-specific state as they are just specifications of how the original model should be read from the visualization point of view. Instead of doing a model transformation we use the shapes to do a meta-model transformation. This approach has several benefits:

- As a consequence of not duplicating the original model into an internal model we do not need to synchronize the two models when the original model changes.
- We can change the properties of a shape at runtime without having to recreate the visualization model. This is particularly interesting when working with an interactive editor.
- Since we can share the same instance of a shape among several figures, we can keep the memory overhead that the framework introduces at a reasonable size even for large models.

All the shapes provide a default constructor. Like this we can create a basic visualization quickly and can then start to refine the visualization. Most shapes offer specific properties that can be adjusted by the user. The *CircleShape* for example offers a radius property. The simplest thing that we offer is mapping a model method, a number or a color directly to a view property. It is also possible to provide a more complex mapping by using a closure.

Being able to use closures is useful when the model does not provide the needed information directly. A common use case is transforming numbers or booleans into colors since it is unusual to have hard-coded color values in the data model. This kind of code that is concerned with bridging incompatible interfaces and data types is known as glue code [Schneider and Nierstrasz, 1999]. The usual solution for a glue problem is adding a wrapper around the original component [Schneider, 1999]. The wrapper typically acts as an *adaptor* or as a *transformer*. By using closures to express the glue code we chose a lightweight approach that supports scripting of glue code. We argue that being able to script the glue code is important from a fast prototyping point of view.

Our model supports interaction at a fine-grained level. Each *Figure* has access to an instance of *EventHandler*. The *EventHandler*, like the shapes, holds no model-specific state. Like this several figures can share the same instance of the *EventHandler*. This again is for keeping the memory and time overhead that our framework introduces at a reasonable size. The *EventHandler* offers several events for which the user can register an action. There are simple events like `#onClick:` and `#onDoubleClick:` that get triggered when clicking or double clicking on a figure and also more sophisticated ones like `#popupView:` that can be used to show a complete visualization in a popup window while hovering over a figure.

The *ViewRenderer* is a facade that offers all methods needed to build a visualization. The *ViewRenderer* provides a concise way of writing scripts while hiding the internal details of the model. Internally the *ViewRenderer* uses a stack. This is useful to define nested visualizations since we can use the same *ViewRenderer* instance on all nesting levels. The design of the *ViewRenderer* was inspired by Seaside [Ducasse *et al.*, 2004b], a Smalltalk-based framework for creating dynamic web applications.

4 Implementing various visualizations using Mondrian

In this chapter we will validate our approach by implementing various visualizations that have been published. Each case study stresses another part of our model.

All case studies are presented in the same way. First we give an introduction to the visualization, then we present the script that we used for creating the visualization, and finally we show the visual result of the script. In some cases we add a small discussion, mainly to point out if a visualization was exceptionally easy or difficult to implement with our model.

4.1 System Complexity View

Our model supports polymetric views [Lanza and Ducasse, 2003]. In a polymetric view we can map model metrics to view properties. An important polymetric view is the System Complexity View. A view that shows an inheritance tree where the width, height and color of the class nodes depend on model metrics.

We already showed in Section 3 (p.19) how Mondrian can be used to build the System Complexity View. In the example we only applied the system complexity view to a small model (38 classes). In Figure 4.1 (p.36) we apply the System Complexity View to ArgoUML (1405 classes). In the classical System Complexity View the hierarchies are positioned next to each other. This is impractical for large models with a wide hierarchy since getting the general idea is difficult.

When using a screen, we would like to have an overview that uses the entire screen surface. That is why we designed a view that we call Screen Filling System Complexity. Figure 4.2 (p.37) shows the same data as Figure 4.1 (p.36) but this time we put each tree in a different node and then arrange the nodes in a *FlowLayout* to fill the screen surface. Besides the hierarchies, we also group the lonely classes in a single box to save more space.

```
view := ViewRenderer new.
view nodes: model classes
     using: (Rectangle withBorder width: #noa; height: #nom;
              linearColor: #loc within: model classes).
view edges: model inheritances
     from: #superclass to: #subclass
     using: LineShape new
view layout: TreeLayout new.
view open.
```

Figure 4.1: Classical System Complexity View of the sources of ArgoUML (1405 classes).

By grouping the nodes and applying a different layout we produced a radically different visualization.

4.2 Class Blueprint

The Class Blueprint [Ducasse and Lanza, 2005] is another polymetric view, designed for visualizing and understanding the internals of a class. It splits the class into five layers: the initialization methods, the public interface methods, the internal implementation methods, the accessor methods and the attributes. Figure 4.3 (p.38) shows the script that we use for creating the class blueprint and the visual result. In the script we start with a node that represents the class, inside we add five more nodes. Each node representing one of the layers. Then we apply a *HorizontalLineLayout* to the five layer nodes. Inside the layers we use a *VerticalLineLayout*. We also add some edges. The blue edges represent invocations and the cyan edges represent accesses.

In the script we use `Shape forMethod` and `Shape forAttribute`. These shapes define how we draw methods and attributes. Since we use these shapes in several places in the script we factored them out into a separate method. Apart from that they are normal shape definitions. Figure 4.4 (p.38) shows the definition of the `#forMethod` shape. We use a *RectangleShape* that is decorated with a *BorderShape*. The width depends on the number of invocations (`#ni`) and the height depends on the number of lines of code (`#loc`). To define the color we use a number of if-statements.

By moving the visual representation of a certain type of data to a separate method

```
view = ViewRenderer new.
model rootClasses do: [:rootClass |
   view node: rootClass forIt: [
        view nodes: rootClass subclassHierarchy
             using: (Rectangle withBorder width: #noa; height: #nom;
                      linearColor: #loc within: model classes).
        view edges: model inheritances
             from: #superclass
             to: #subclass
             using: LineShape new.
      view layout: TreeLayout new.
   ].
].
view node: model lonelyClasses forIt: [
   view nodes: model lonelyClasses
        using: (Rectangle withBorder width: #noa; height: #nom;
                linearColor: #loc within: model classes).
   view layout: CheckerboardLayout new.
].
view layout: (FlowLayout withMaxWidth: 800).
view open.
```

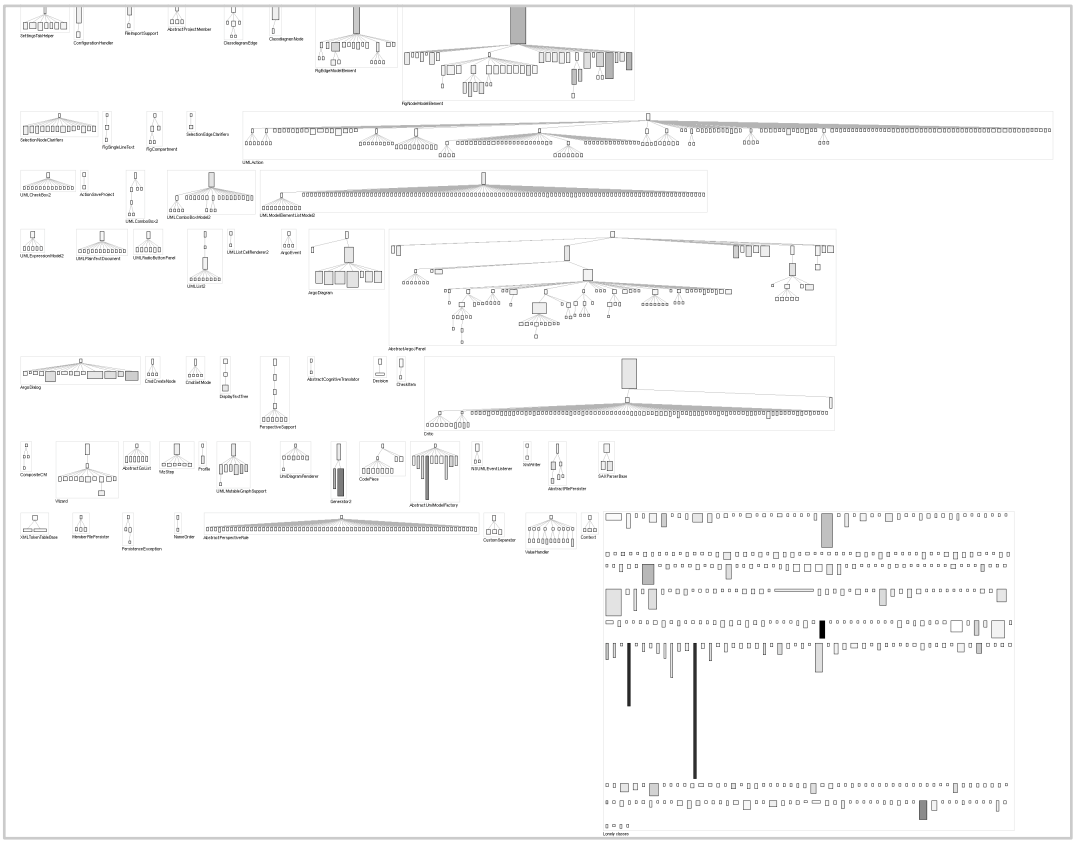


Figure 4.2: Screen Filling System Complexity View that makes better use of of the space that a computer screen offers.

```
view := ViewRenderer new.
view node: class using: (RectangleShape new) forIt: [

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

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

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

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

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

   view edges: class invocations
        from: #invokedBy
        to: #invoked
        using: LineShape blue.

   view edges: class accesses
        from: #accessedBy
        to: accessed
        using: LineShape cyan.
].
view open.
```

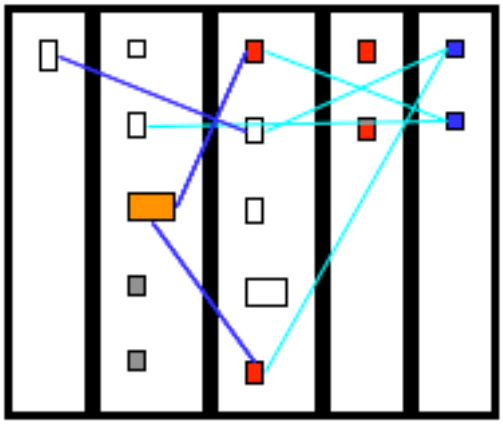


Figure 4.3: Example of a class blueprint and the script that was used to create the visualization.

```
Shape>>forMethod
    ^RectangleShape withBorder width: #ni height: #loc color: [:entity |
        entity isPureAccessor ifTrue: [^Color red].
        entity isPureWriter)  ifTrue: [^Color orange].
        entity isAbstract ifTrue: [^Color cyan].
        entity isOverriding ifTrue: [^Color brown].
        entity isCleanSuperSend) ifTrue: [^Color orange].
        entity isConstMethod) ifTrue: [^Color gray].
        entity isureSelfDelegatingReturn ifTrue: [^Color yellow].
        entity isPureSelfDelegatingNoReturn ifTrue: [^Color yellow].
        ^Color white
     ].
```

Figure 4.4: Definition of the shape that is being used to draw the methods.

we can reuse the definition in a different context.

4.3 Scatterplot

Our model can also be used for general information visualization. In Figure 4.5 (p.39) we show how we build a scatterplot showing the classes of Ant[1] (500 classes). For positioning and coloring the nodes we use metrics like in the polymetric views [Lanza and Ducasse, 2003]. We position the nodes according to the values of the `#loc` and `#nom` methods and we choose the color depending on the value of the `#tcc` method.

```
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: #loc
                    y: #nom
                    color: (LinearNormalizer context: model classes
                                             command: #tcc)).
    view layout: ScatterplotLayout new.
].
view open.
```

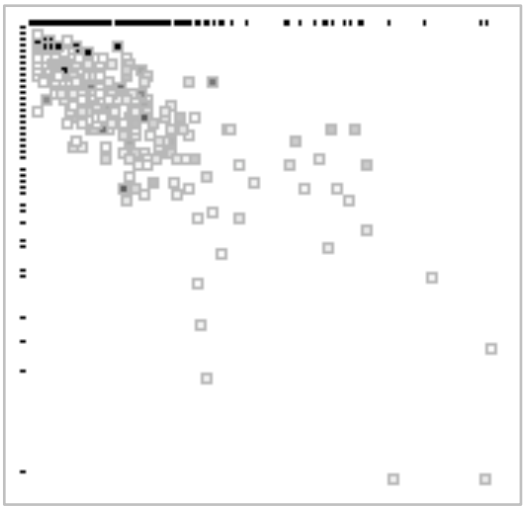


Figure 4.5: Example of a scatterplot visualization with "intelligent" decorations and the corresponding script.

[1] http://ant.apache.org/

The example shows how decorations can be used in a way that exposes information. The node that contains the scatterplot is decorated with coordinate lines (*i.e.,HorizontalCoordinateShape* and *VerticalCoordinateShape*). The coordinate lines show the density of the nested nodes based on their x and y position [Tufte, 2001].

4.4 Spectographs

Figure 4.6 (p.41) shows an example of a spectograph [Wu *et al.*, 2004]. We used the spectograph to visualize the files in the CVS repository of JBoss (2094 files). Each file is represented as a horizontal line. A dot on a line is red if there was a commit in the current month, yellow if there was a commit in the previous or the next month and green otherwise. In the script we iterate over the CVS files in the model and assign to each dot the appropriate color. We use the ScatterplotLayout to arrange the dots.

The script for this visualization is *not* as readable as the previous scripts. This is surprising considering that the visualization is rather simple. When looking at the script in detail we notice that we use several temporary variables (greenOk, yellowOk, *etc.*) and that the script is cluttered with if-statements. We are abusing the visualization script for extracting information from the model that the model does not provide directly.

We have encountered this situation several times while experimenting, and every time the script was too long and difficult to grasp, we came to the conclusion that the data model is not providing enough information and thus should be updated. Once we applied the enhancement, the visualization scripts became smaller and easier to read. In our example, a lot of lines of code are dedicated to the computation of the color of a dot (*i.e.,* red, green, yellow). If this information would be provided by the data model directly, the script would be more compact. From a prototyping point of view it is however a benefit that the user can *simulate* missing model information in the visualization script before deciding what information should be provided by the model directly.

Figure 4.7 (p.42) shows the script for creating the spectograph after updating the model to provide the necessary information. The model has been extended with the methods `#isHotIn:`, `#isWarmIn:` and `#isColdIn:` that return the state of a file in a given month.

This example also emphasizes something else: In our model there is an object behind every figure. While this is suited for graph like visualizations, for this

```
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: (RectangleShape x: xPos
                                          y: yPos
                                          color: #red).
            greenOk := false.
            yellowOk := true.
        ].
        ifFalse: [
            greenOk ifTrue: [
                view node: file
                     using: (RectangleShape x: xPos
                                              y: yPos
                                              color: #green)
            ].
            yellowOk ifTrue: [
                view node: file
                     using: (RectangleShape x: xPos
                                              y: yPos
                                              color: #yellow).
                yellowOk := false.
                greenOk := true.
            ]
        ].
        xPos := xPos + 5.
    ].
    yPos := yPos + 1.
].
view layout: Mondrian.ScatterplotLayout new.
view open.
```

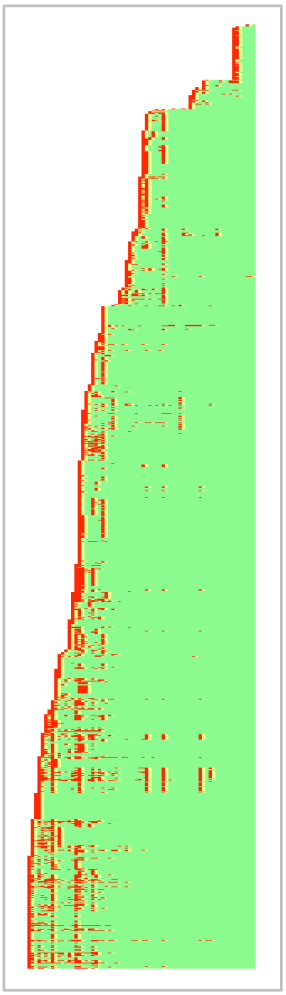


Figure 4.6: Example of a spectograph and the script that was used to create the visualization.

```
view := ViewRenderer new.
yPos := 5.
model files do: [:file |
    | xPos |
    xPos := 5.
    model months do: [:month |
       (file isHotIn: month) ifTrue: [
           view node: file using: (RectangleShape x: xPos y: yPos color: #red)].
       (file isColdIn: month) ifTrue: [
           view node: file using: (RectangleShape x: xPos y: yPos color: #green)].
       (file isWarmIn: month) ifTrue: [
           view node: file using: (RectangleShape x: xPos y: yPos color: #yellow)].
        xPos := xPos + 5.
     ].
    yPos := yPos + 1.
].
view scatterplotLayout.
view open.
```

Figure 4.7: Script for creating the spectograph after updating the model to provide the necessary information.

particular one, it generates a significant overhead in object creation since we create a figure for every dot. Definitely, the spectograph could be implemented in a much more concise way using either a more intelligent shape for a line, or by just creating one single shape for the entire view. However, the example does show that it is possible to prototype even such a visualization using our framework.

4.5 UML class diagram

Our model can also be used to generate UML class diagrams [Fowler, 1997]. The UML representation of a class consists of three sections. The first section contains the name, the second section contains all the attributes and the third section contains all the methods of the class. Figure 4.8 (p.43) shows part of the Mondrian layout hierarchy.

Considering the popularity of UML we decided to introduce specialized shapes to represent classes and inheritance edges. This is why the script in Figure 4.8 (p.43) gets by with only five lines of code. The shape that we use to draw classes is called *UmlClass*. In our data model a class provides the methods `#name`, `#attributes` and `#methods`. The `#name` method returns the name of the class, the `#attributes` method returns a collection containing all the attributes and the `#methods` method returns a collection containing all the methods. In the script we map the *UmlClass* to these methods.

The UML case study was interesting since we had to decide between introducing

```
view := ViewRenderer new.
view nodes: model allClasses
            using: (UmlClass name: #name
                             attributes: #attributes
                             methods: #methods).
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: UmlInheritance new.
view layout: TreeLayout new.
view open.
```

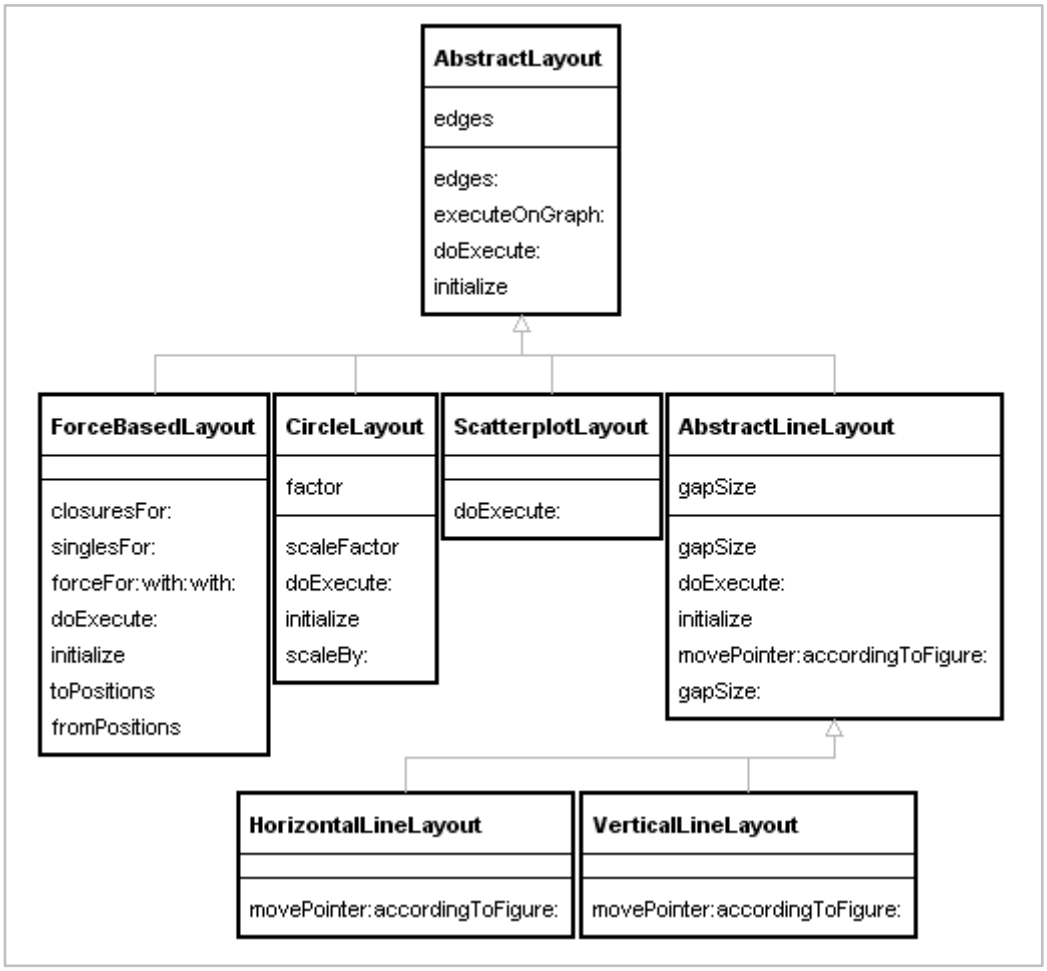


Figure 4.8: A UML class diagram showing part of the Mondrian layout hierarchy.

a specialized shape for drawing classes and using the ordinary approach of composing simple shapes and nested figures. We ended up trying both approaches. Figure 4.9 (p.44) shows a Mondrian script that can be used to create a UML class representation. Since this script needs to be run for each class we moved the script to a utility class called *UmlLib*. The utility class has a method called `#class:attributes:methods:view:` that takes the class, the attributes, the methods and an instance of *ViewRenderere* as parameter. The method produces exactly the same visual result as the dedicated *UmlClass* shape.

```
UmlLib >> class: class attributes: attributes methods: methods view: view

    view node: class using: RectangleShape withThickBorder forIt: [

        view node: class name using: (LabelShape withBottomLine boldLabel: #name).

        view node: attributes using: RectangleShape withBottomLine forIt: [
            view nodes: attributes using: (LabelShape label: #name).
            view layout: VerticalLineLayout new.
        ].

        view node: methods using: RectangleShape new forIt: [
            view nodes: methods using: (LabelShape label: #name).
            view layout: VerticalLineLayout new.
        ].

        view layout: VerticalLineLayout new.
    ].
```

Figure 4.9: A script for creating an UML class representation.

The script is straightforward. We start by adding a node for the class that is surrounded by a thick line. Then we add a node for the class name and decorate the node with a horizontal line along the bottom. We add a node that contains the attributes, again decorated with a horizontal line and a node that contains the methods. To these three nodes we apply a *VerticalLineLayout*.

The downside of the script approach is that we need to execute the script for each class that we want to visualize. This can be a problem for large models since we need to create a lot of figures for each UML class. Since the dedicated *UmlClass* shape is stateless we only need one instance of the shape to draw all UML classes which is an advantage in terms of memory usage.

A dedicated shape is almost four times faster than a script that we need to execute for each model object. We did some time profiling on a computer with a 2GHz CPU and 1GByte of memory. Rendering a UML diagram with 1797 classes takes 2.892 seconds when using the *UmlClass* shape and 10.328 seconds when using the script.

In our model we assume that there is only one entity behind every figure. From this point of view the script approach is better since it would be possible to add invocation edges to the methods of the UML class. This is not possible with the dedicated *UmlClass* shape since the only entity behind the shape is the actual class object.

The decision to use a script or a dedicated shape is a trade-off between memory usage and having the possibility of adding edges.

4.6 UML sequence diagram

UML sequence diagrams are used to visualize the interaction between collaborating objects [Fowler, 1997]. In a sequence diagram an object is usually represented as a box on top of a dashed line. The dashed line is called *life line*. On the life line we have boxes that represent methods. The boxes are placed on the life line in chronological order with the height of the box depending on the execution time of the method. Message sends between methods are drawn as arrows. Usually the name of the message is displayed above the arrow.

Figure 4.10 (p.46) shows an UML sequence diagram that has been scripted with Mondrian. The presented use case starts with an instance of *PersonTest*. *PersonTest* creates an instance of *Professor*, an instance of *University* and an instance of *Person*. Finally *PersonTest* sends the message `#name:` to the *Person* instance.

In the diagram we have self message sends. The *Professor's* `#initialize` method calls the `#initialize` method of its two superclasses. To indicate which methods have a return value we added a dashed arrow to the end of the methods that return a value.

Figure 4.11 (p.47) shows the script that we used to create the UML sequence diagram. We start by adding a node for each object. To get the objects we call the model's `#allInstances` method. We moved the shape definition of the objects to the utility class *UmlLib* since it would just be another example of the *FormsBuilder*.

Inside the objects we add a node for each message. To arrange the nodes we use a *Scatterplot* layout. We distinguish between normal messages and self messages since the self messages need to be shifted to the right. For the x coordinate of the messages we use the keyword `#centered`. This is a special keyword that the *Scatterplot* layout provides. All figures with x coordinate set to `#centered` will be

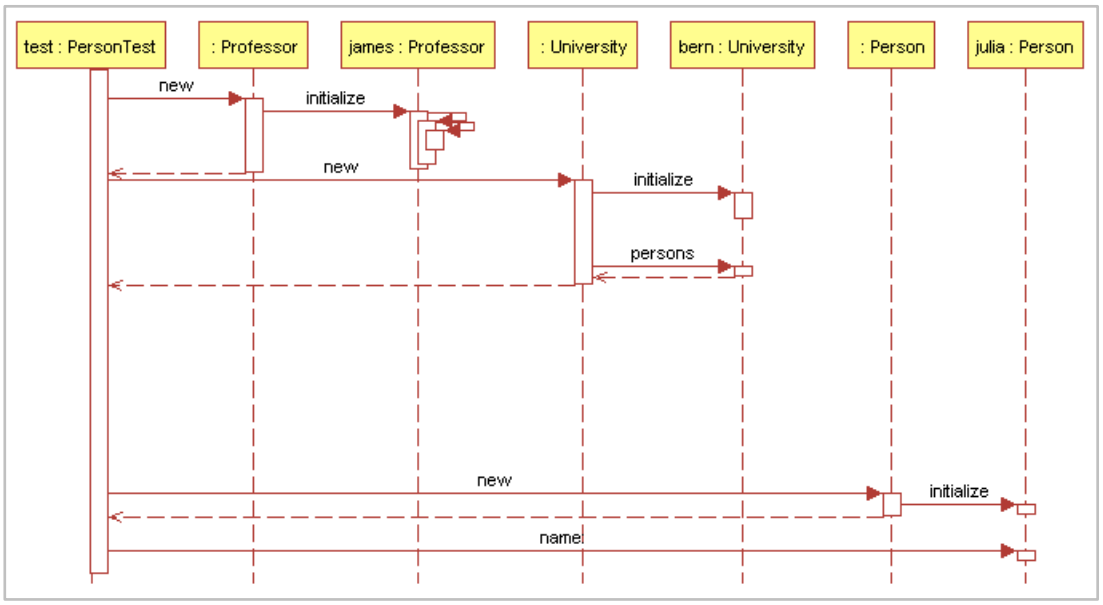


Figure 4.10: UML sequence diagram.

centered within the parent bounds.

Next we add the edges. In the sequence diagram we have three different kinds of edges. We obtain the message send and message return objects by calling the model's `#allMessageSends` and `#allMessageReturns` methods. The shape definition of these edges is in the utility class *UmlLib*. For drawing the self messages (`#allSelfMessageSends`) we use a dedicated shape called *UmlSelfSend*.

The object nodes need to be aligned horizontally. Additionally all nodes should have the same height. This cannot be done with the *HorizontalLineLayout*. Instead we use the *FormsBuilder* to define the layout. Until now we only used the *FormsBuilder* to compose shapes but the *FormsBuilder* can also be used to set the size and position of figures.

In the script we define a grid that consists of one row and several columns. Since we do not know up front how many nodes we have, we define the layout dynamically while looping over the object nodes. We also add a 20 pixel gap between successive nodes since the nodes would be to close to each other otherwise. To obtain the layout from the *FormsBuilder* we call the `#asLayout` method.

Figure 4.12 (p.48) shows the definition of the shape that we use for drawing the message send objects. We start by defining the blocks `startPoint` and `endPoint`. These blocks contain the strategies that the *LineShape* should use to calculate its start and end point. We need specialized strategies since the default edge strate-

```
view := ViewRenderer new.
nodes := view nodes: model allInstances using: UmlLib instanceShape
             forEach: [:instance |
                 view nodes: (model messageSendsFor: instance)
                      using: (RectangleShape withBrownBorder
                                   width: 10 height: #duration
                                   x: #centered y: #timestamp).
                 view nodes: (model selfMessageSendsFor: instance)
                      using: (RectangleShape withBrownBorder
                                   width: 10 height: #duration
                                   x: #centered y: #timestamp
                                   xOffset: [:event | 5 * event level]).
                  view layout: ScatterplotLayout new].

view edges: model allMessageSends
     from: #sender to: #receiver
     using: UmlLib messageSendShape.

view edges: model allMessageReturns
     from: #sender to: #receiver
     using: UmlLib messageReturnShape.

view edges: model allSelfMessageSends
     from: #sender to: #receiver
     using: UmlSelfSend new.

builder := FormsBuilder new.
builder rows: 'fill:pref'.
nodes do: [:node |
     builder column; size: 20; column; pref.
     builder x: builder lastColumn y: 1 add: node.
].
view layout: builder asLayout.
view open.
```

Figure 4.11: Script for creating a UML sequence diagram with Mondrian. To arrange the top-level nodes we make use of the FormsBuilder.

gies do not produce the desired result. The rest of the script is straightforward. We use a *ShapeBuilder* to decorate the *LineShape* with an *ArrowShape* and a *LabelShape*.

```
UmlLib>>messageSendShape
   | builder startPoint endPoint line |

   startPoint :=  [:anEdge |
      | base delta |
      base := (anEdge fromFigure right < anEdge toFigure left)
                  ifTrue: [anEdge fromFigure topRight]
                  ifFalse: [anEdge fromFigure topLeft].
      delta := anEdge toFigure preferredY - anEdge fromFigure preferredY.
      base + (0 @ delta)].

   endPoint := [:anEdge |
      (anEdge toFigure left > anEdge fromFigure right)
         ifTrue: [anEdge toFigure topLeft]
         ifFalse: [anEdge toFigure topRight]].

   line := LineShape color: Color brown start: startPoint end: endPoint.
   builder := ShapeBuilder new.
   builder add: line with: [
      builder add: (ArrowShape width: 5 height: 10).
      builder add: (LabelShape align: #topCenter label: #asString).
   ].

   ^builder asShape
```

Figure 4.12: The shape definition that we use for the message sends in the UML sequence diagram. The strategy for the start and end point of a *LineShape* can be scripted.

An edge can be connected to a node in many different ways. It is almost impossible to provide dedicated shapes for all possible connection positions. This is why we decided to use strategies instead of hard-coded algorithms. By allowing the user to script the start and end point of the *LineShape* we give him the possibility to experiment with different connection positions without having to resort to subclassing.

4.7 Scripting tools

In this case study we use Mondrian to script a simplified version of Softwarenaut (Section 2.1.9 (p.13)). Softwarenaut is an interactive reverse engineering tool. Instead of analyzing low-level artifacts (*e.g.,* classes, files) the tool focuses on high-level abstractions (*e.g.,* packages) and the relationships between them.

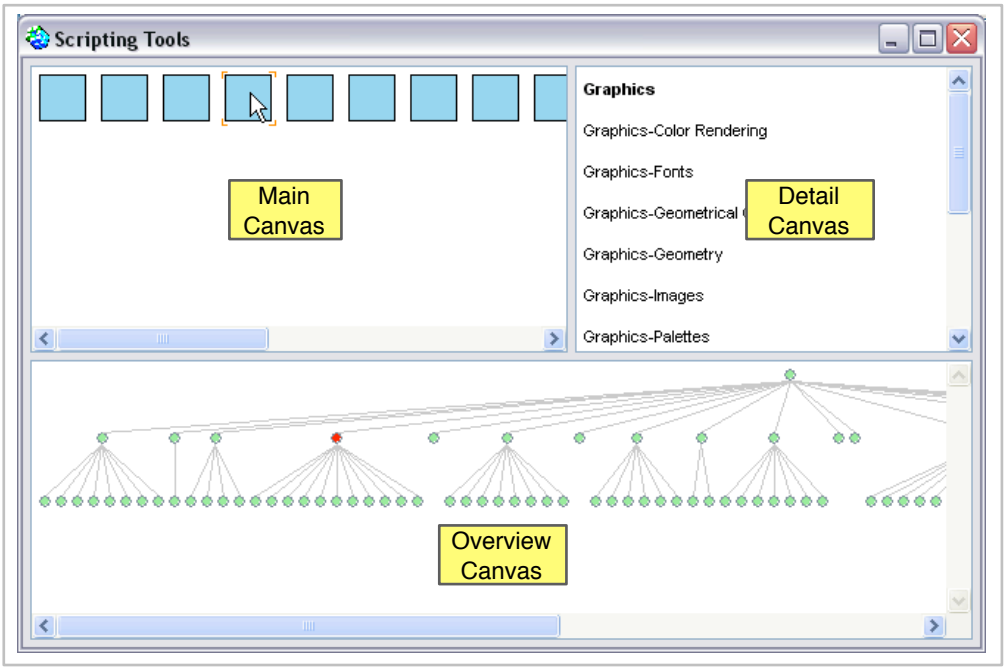


Figure 4.13: A simple tool for navigating through packages.

Figure 4.13 (p.49) shows a screenshot of our application. The user interface consists of three areas. The *Main Canvas* displays packages as blue rectangles. The *Detail Canvas* shows the name of the selected package in bold letters and the names of the child packages in normal letters. The *Overview Canvas* shows the complete package tree. The currently selected package is highlighted using a red background.

In Figure 4.13 (p.49) we are browsing the Cincom Smalltalk package hierarchy. The currently selected package is called *Graphics*. *Graphics* is a subpackage of *Base VisualWorks*.

Figure 4.14 (p.50) shows the script that we use to create the application. The complete business logic is defined in the script. The *ScriptingToolsUI* class is only responsible for opening the application window and to position the three Mondrian windows. The *ScriptingToolsUI* class implements the three methods `#mainCanvas`, `#detailCanvas` and `#overviewCanvas` which we use in the script to access the Mondrian windows.

The script can be split into three parts. In the script we define the variables `mainScript`, `detailScript` and `overviewScript`. Each of these variables contains a script that is responsible for one of three Mondrian windows that *ScriptingToolsUI*

```
window := ScriptingToolsUI new.
window open.

currentPackage := aPackage.
currentPackages := OrderedCollection with: aPackage.

mainScript := [
    mainView := ViewRenderer new.
    mainView interaction
                onClick: [:package |
                          currentPackage := package.
                          detailScript evaluate.
                          window oveviewCanvas repaint];
                onDoubleClick: [ :package |
                           currentPackages := package childPackages.
                           mainScript evaluate].
    mainView nodes: currentPackages using: Shape forPackage.
    mainView installOn: window mainCanvas].

detailScript := [
    detailView := ViewRenderer new.
    detailView node: currentPackage using: (LabelShape boldLabel: #name).
    detailView nodes: currentPackage childPackages
              using: (LabelShape label: #name).
    detailView verticalLineLayout.
    detailView installOn: window detailCanvas].

overviewScript := [
    overviewView := ViewRenderer new.
    overviewView tree: self to: #childPackages using: (CircleShape
                color: [:package | (package = currentPackage)
                        ifTrue: [Color red]
                        ifFalse: [Color lightGreen] ]).
    overviewView treeLayout.
    overviewView installOn: window oveviewCanvas].

mainScript evaluate.
detailScript evaluate.
overviewScript evaluate
```

Figure 4.14: We define the complete business logic in this script. *ScriptingToolsUI* is only responsible for opening the application window.

provides.

mainScript. In this script we specify that the packages should be painted as blue rectangles and we provide code for the `#onClick:` and `#onDoubleClick:` events. When a user clicks on a package we update the variable `currentPackage`, repaint the *Overview canvas* and reevaluate the `detailScript`. When a user double-clicks on a package we update the variable `currentPackages` and reevaluate the complete `mainScript`. `mainScript` is actually a recursive definition.

detailScript. This script is responsible for the *Detail Canvas*. The script creates a bold label for the selected package and normal labels for the subpackages. The labels get aligned vertically. This script is reevaluated each time the user clicks on a package in the *Main Canvas*.

overviewScript. In this script we create the package tree. All packages get painted as green circles except for the package that is currently selected. That package gets painted as a red circle. To build the tree we make use of the convenience method `#tree:to:using:` that the *ViewRenderer* provides.

This is an example of how the scripting approach of our model can be extended to not only script visualizations but to also script complete tools. It also shows one possible way of building a specialized application on top of our framework.

4.8 MetricView

MetricView (Section 2.1.8 (p.13)) is a tool that extends UML diagrams with polymetric features. Figure 4.15 (p.52) shows how a MetricView visualization can be scripted using Mondrian. In the script we make use of the *FormsBuilder* to compose the shapes. We start by defining a grid with two rows and two columns. Then we add a *UmlClass* shape that covers the entire grid. Now, like in the original tool, we start mapping metrics to cells. The cells (1, 1), (1, 2) and (2, 2) contain an icon. If the value of the metric is within the specified bounds the icon will be a green tick otherwise we display a red cross. In cell (2, 1) we display a green rectangle. The height of the rectangle is based on the length of the class name.

Combining the shapes in this way is not possible with the `#decoratedWith:` approach since the individual shapes have only got limit knowledge of the environment. The *FormsBuilder* is a way around this limitation. With the *FormsBuilder* we have a high-level view and can freely position the shapes.

```
builder := FormsBuilder new.
builder columns: 'center:pref:grow, center:pref:grow'.
builder rows: 'center:pref:grow, center:pref:grow'.

builder  x: 1 y: 1 w: 2 h: 2 add: (
    UmlClass name: #name attributes: #attributes methods: #methods).

builder  x:1 y: 1 add: (
    ImageShape image: [:entity | entity wloc < 100
        ifTrue: [ImageLib ok] ifFalse: [ImageLib error]]).

builder  x:1 y: 2 add: (
    ImageShape image: [:entity | entity nom < 20
        ifTrue: [ImageLib ok] ifFalse: [ImageLib error]]).

builder x: 2 y: 1 add: (
    RectangleShape withBorder width: 15 height: #nl color: Color lawnGreen ).

builder  x:2 y: 2 add: (
    ImageShape image: [:entity | entity noa < 10
        ifTrue: [ImageLib ok] ifFalse: [ImageLib error]]).

view := ViewRenderer new.
view nodes: model allClasses using: builder asShape.
view edges: model allInheritances
     from: #superclass
     to: #subclass
     using: UmlInheritance new.
view layout: TreeLayout new.
view open.
```

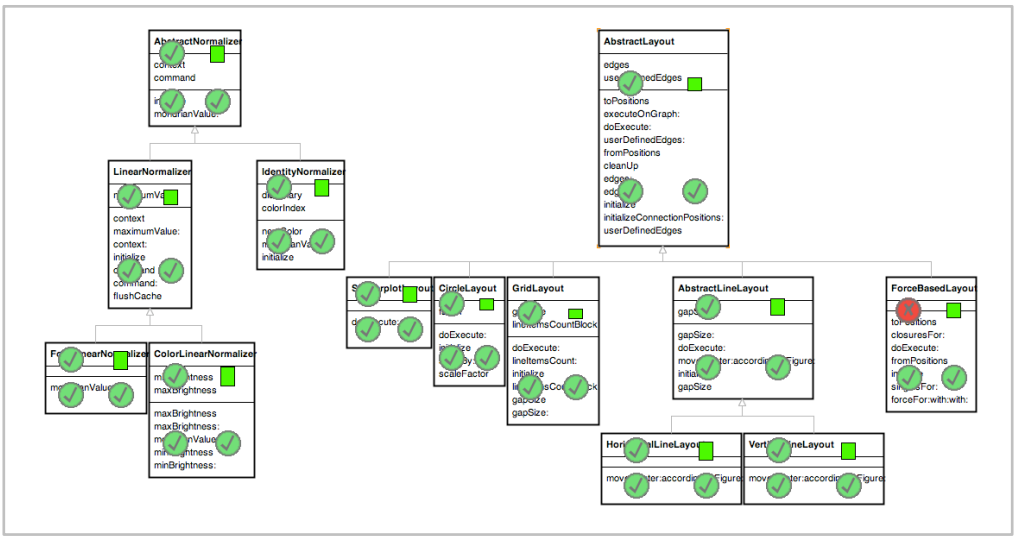


Figure 4.15: Script for creating a MetricView-like visualization and the visual result that the script produces.

5 Discussion

While developing our model we had to find solutions to a number of problems. In this section we will present some of these problems and the solution that we provide. We also discuss the details of implementing our model in other programming languages.

5.1 Composing shapes

In Section 3.3 (p.28) we presented the various ways that Mondrian offers to compose shapes. The development of these interfaces was driven by the increasingly complex visualizations that the users were trying to do. By adding the *FormsBuilder* we supplemented our model with a completely new paradigm for composing shapes. We moved from the *decorator pattern* where each shape is responsible to calculate its size and position based on the decorated shape to a *global layout algorithm* that is responsible for position and size of all shapes. This new paradigm was needed for the reasons that we describe in the next two sections.

5.1.1 Human speech is ambiguous

To align shapes we offer keywords like `#bottomLeft` and `#topRight`. Figure 5.1 (p.53) shows a *LabelShape* that is aligned to the bottom left corner of a *BorderShape*. This is where most people expect the label to be but this is not the only possible position.

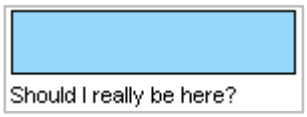


Figure 5.1: LabelShape that is aligned to the bottom left.

There are actually four positions that could be associated with `#bottomLeft`. Figure 5.2 (p.54) shows the same visualization as Figure 5.1 (p.53) but this time we

added the letters A to D to show all the possible positions that `#bottomLeft` could mean in human speech.

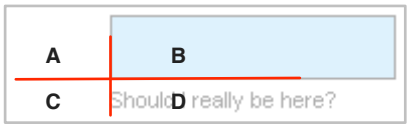


Figure 5.2: There are four positions that could be associated with `#bottomLeft`.

This problem can be solved by introducing more accurate terms for defining the position. As an example we could introduce the keyword `#bottomLeftSouthWest` to reference position C. We decided otherwise since we came to the conclusion that these terms would be too artificial. Only for the four corners we would need 16 keywords. So we decided to use a small number of keywords and to implement them in a way that allows the user to guess how `#topLeft` will behave if he already knows `#bottomLeft`. This makes the align mechanism easier to use at the cost of flexibility. We believe that this is an acceptable trade-off since we provide the *FormsBuilder* that offers plenty of flexibility while still being easy to use.

5.1.2 Resize behaviour of shapes

Shapes have limited knowledge of their environment. They only know the size of the shape that they decorate. They do not know their position in the "big picture". They do not even know their decorators. When creating the initial visualization this is not an issue. It becomes an issue when we start resizing figures.

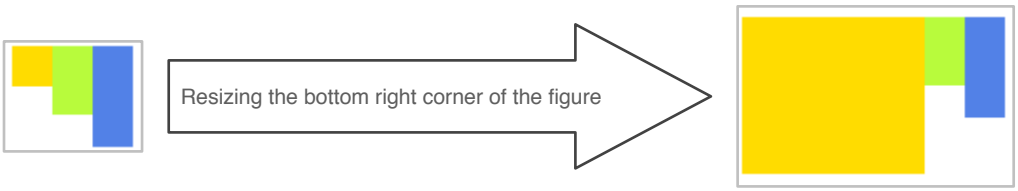


Figure 5.3: On the left we see three RectangleShapes with alignment `#rightTop`. On the right we see the same three *RectangleShapes* after resizing the bottom-right corner of the figure.

For Figure 5.3 (p.54) we used a single node. The visual representation of the node is given by three *RectangleShapes* that have alignment `#rightTop`. On the left we see how the node looked initially and on the right how it looks after resizing the bottom-right corner of the figure with the mouse.

Did we expect the node to look like this after resizing the bottom-right corner or did we expect that the three RectangleShapes distribute the additional space equally amongst themselves?

As we mentioned in the first paragraph of this section a shape does not know its decorators. This means that the orange shape does not know anything about the green and the blue shape and the green shape does not know anything about the blue shape. The result of this is that it is technically impossible to distribute the space equally among the three shapes by only using the decorator mechanism.

Another thing that is not possible with the decorator mechanism is telling the shapes that they should all have the same height. The preferred height of the orange shape is 20, of the green shape is 35 and of the blue shape is 50. Since the orange shape does not know the other shapes it cannot ask them for their heights and can therefore not find out that the maximum height of all three shapes is 50.

The *FormsBuilder* provides a solution for both of these problems. It allows a fine-grained definition of the resize behaviour and we can define if a shape should occupy additional space. At the moment we distribute additional available space equally among the "greedy" shapes but it is planned to make this process more fine-grained by allowing the user to specify the "greediness" with a percentage value.

5.2 Positioning the nested visualization explicitly

When we first started to work on our framework an important decision was taken somewhat implicitly. Namely the way how a nested visualization interacts with the decorators (*i.e.,* shapes) of the parent figure.

All shapes implement the method `#outerBoundsForFigure:forBounds:`. This is the method that is responsible for calculating the size and position of the shape. We pass the figure as parameter since shapes are stateless. In the `forBounds:` part of the method we pass the outer bounds of the decorated shape. Like this a shape can calculate its bounds based on the bounds of the shape that it decorates. The first shape of the entire shape composition is special since it *decorates* a figure and not a shape. In this special case we pass the bounds of the nested visualization as parameter to the `#outerBoundsForFigure:forBounds:` method.

Figure 5.4 (p.56) shows a simple example of a nested visualization. We have a

single node that contains three other nodes. When rendering the script we start by calculating the size and the position of the nested nodes. Then we calculate the compound bounds of the nested nodes and use these bounds as the input for the Rectangle- and BorderShape of the parent node.

```
view := ViewRenderer new.
view node: #a using: RectangleShape withBorder forIt: [
     view nodes: #(1 2 3)
          using: (RectangleShape withBorder width: 20 height: 10).
].
view open.
```

Figure 5.4: The shapes that decorate the parent node take the outer bounds of the nested visualization as input.

This implies that we always need to start decorating the outer bounds of the nested visualization. In this simple example this fact is well hidden from the user. When we want to apply more complicated decorations to the parent figure this constraint becomes a nuisance though. Often it is difficult to compose shapes while keeping in mind that they need to be arranged around the nested visualization.

This led to the insight that we somehow need to give the user the possibility to position the nested visualization explicitly during the shape composition process.

The solution that we came up with was to introduce a new shape. The shape is called *ChildrenShape* and it can be used like any other shape. Like this the user can explicitly define the position of the nested visualization while still using his favorite way of composing shapes (*i.e.,*`#decoratedWith:`, *Shape*, *ShapeBuilder* or *FormsBuilder*).

Figure 5.5 (p.57) shows a simplified UML class visualization. We show the methods but not the attributes. We used the *FormsBuilder* to compose the shapes and the *ChildrenShape* to position the nested visualization (*i.e.,* the methods). The yellow box indicates the position and size of the *ChildrenShape*.

In our current implementation we only support one *ChildrenShape* per shape composition. This is why Figure 5.5 (p.57) only shows a simplified UML class and not a proper one. Being able to place two *ChildrenShapes*, one for the methods and one for the attributes, would be an elegant way to build a UML class diagram.

The reason why we do not support this currently is that we have not found an intuitive way yet to script the mapping between a specific *ChildrenShape* and

a subset of the nested visualization. At the moment the single *ChildrenShape* instance is responsible for the complete nested visualization.

```
builder := FormsBuilder new.
builder columns: '2px, 2px, fill:pref:grow, 2px, 2px'.
builder rows: '2px, pref, 1px, fill:pref:grow, 2px'.
builder x: 1 y: 1 w: 5 h: 5 add: RectangleShape new with: [
    builder add: (BorderShape lineWidth: 2)
].
builder x: 3 y:2 add: (LabelShape centeredLabel: #name).
builder x: 2 y: 3 w: 3 add: SpacerShape new.
builder x: 3 y: 4 add: ChildrenShape new.

view := ViewRenderer new.
view node: aClass using: builder asShape withBorder forIt: [
     view nodes: aClass selectors using: (LabelShape label: #asString).
     view verticalLineLayout.
].
view open.
```

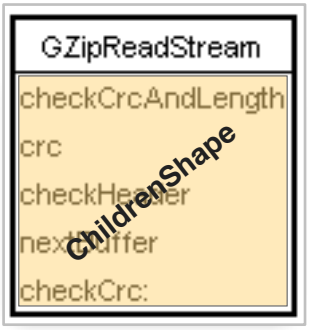


Figure 5.5: Class visualization in an UML kind of way. The ChildrenShape has been used to position the nested visualization.

5.3 Implementing our model in other languages

We implemented our model in Cincom[1] Smalltalk. Before looking at other implementations of our model we would like to mention a couple of reasons why we chose Cincom Smalltalk for our implementation:

- Smalltalk can be used for writing scripts. With no additional work one has access to basic constructs like loops and conditions and to all the classes that are in the image.
- Closures are a central part of Smalltalk. We rely heavily on closures for defining the mapping between the data model and the visualization model.

[1] http://smalltalk.cincom.com

- Moose is only available for Cincom Smalltalk and since many of our potential customers are working with Moose we thought it ideal to make our framework available on the same platform, although our framework is not coupled to Moose in any way.

We also implemented our model in two other languages. We did a Squeak implementation to verify that our model works with a *different graphical framework* and a Java implementation to verify that our model works with a *different language*.

5.3.1 Squeak

We did a basic implementation of our model with Squeak[2]. Squeak is another Smalltalk dialect. At first sight a port to another Smalltalk flavor may not look that interesting since there are many similarities among the different Smalltalk dialects. The reason why we did it is the fact that our framework relies heavily on the underlying graphical framework. This is an area where the different Smalltalk implementations differ. Cincom Smalltalk offers a basic but stable graphical framework. The core of the framework is the class *GraphicsContext* that provides low-level drawing commands like `#drawLineFrom:To:` or `#drawRectangle:`. Two things that the framework does not provide are transparency and antialiasing.

Squeak has a sophisticated graphical framework that is called *Morphic*. *Morphic* figures are interactive by nature. After creating a *Morph* it can be selected, moved and resized. All by only using the mouse. *Morphic* even supports connecting figures out of the box. In Cincom Smalltalk this functionality had to be implemented from scratch.

5.3.2 Java - Groovy

We also wanted to test our model with a completely different programming language. This is why we did a basic Java implementation. The main problem we had was to come up with a replacement for closures. The closest to a closure that Java offers are "anonymous inner classes". This is a poor solution for two reasons[3]. Firstly a closure should make things easier. An "anonymous inner class" only makes things more complicated because of the verbose syntax. And secondly a "anonymous inner class" is no proper replacement for a closure since a closure

[2] http://www.squeak.org
[3] http://fishbowl.pastiche.org/2003/05/16/closures_and_java_a_tutorial

is a proper object that can be passed around in the system. "Anonymous inner classes" only have a limited scope. Another issue was that Java does not support scripting out of the box.

This is why we decided to use Groovy [Koenig *et al.*, 2006] for our implementation. Groovy is an agile dynamic language for the Java Platform that has been inspired by other languages like Ruby, Pyhton and Smalltalk. Groovy extends Java with an easy to use scripting language and also introduces the notion of closures to Java. Behind the scenes Groovy scripts get compiled into proper Java objects. Groovy can be embedded in a Java application but it also comes with a small standalone Swing application that can be used for writing and running scripts. Groovy has access to all Java classes and all custom classes that are in the classpath.

Figure 5.6 (p.59) shows a screenshot of the Groovy Console. The console consists of two input fields. The top one is for writing the script and the bottom one shows log and error messages.

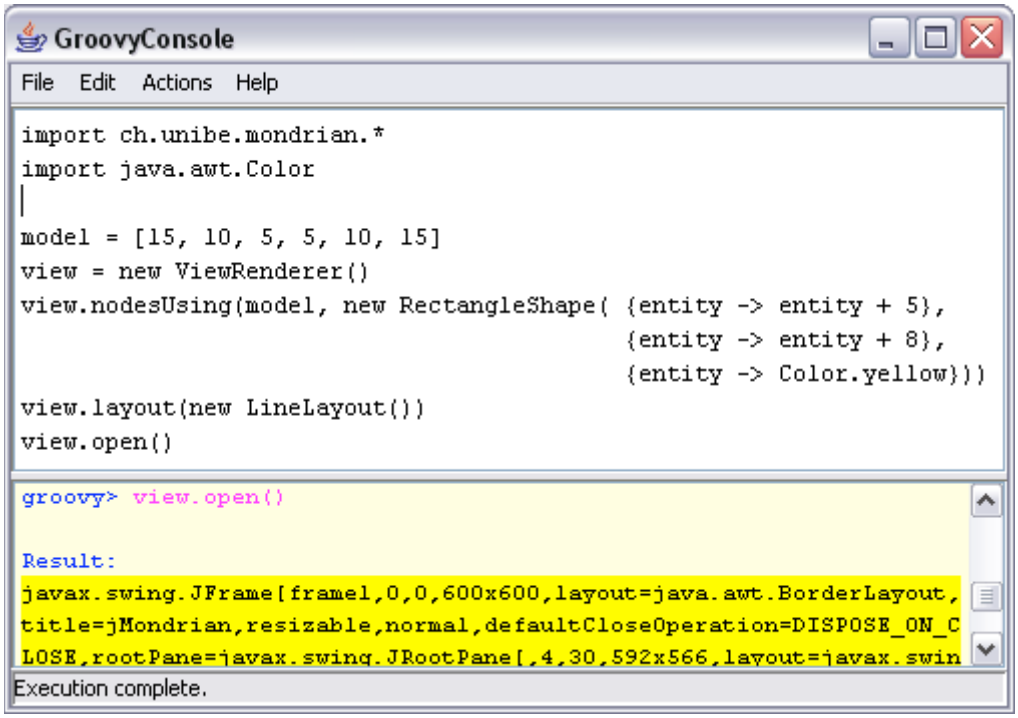


Figure 5.6: The Groovy console adds scripting functionality to Java.

Figure 5.7 (p.60) shows a simple visualization that we produced with the Java port of our model and the script that we used to create the visualization. In the first line of the script we define a simple model. In this example we use a *Collection* that contains six *Integers*. In Smalltalk the *ViewRenderer* has the

method `#nodes:using:`. In the Java port we use a similar syntax. The Java version of the command is called `#nodesUsing( , )` and it takes two parameters. The first parameter is expected to be a *Collection* of model objects and the second parameter needs to be a *Shape*. The *RectangleShape* has a constructor that takes three parameters. A closure for the width, a closure for the height and a closure for the color. Like in the Smalltalk version the closure expects one parameter, the model entity behind the figure.


```
model = [15, 10, 5, 5, 10, 15]

view = new ViewRenderer()
view.nodesUsing(model, new RectangleShape( {entity -> entity + 5},
                                           {entity -> entity + 8},
                                           {entity -> Color.yellow}))

view.layout(new LineLayout())
view.open()
```

Figure 5.7: A simple visualization that has been done with the Java based implementation of our model.

The Groovy syntax for a closure is straightforward. For the width we use the closure {*entity -> entity + 5*}. On the left hand side of the arrow we tell the closure that it will be executed with one parameter and that the parameter should be known as `entity` inside the closure. On the right hand side of the arrow we have the code that should be executed when evaluating the closure. In this case we return the value of the model entity increased by five. For the fill color we use the closure {*entity -> Color.yellow*}. This closure always returns the color yellow regardless of the value of the model entity.

In Figure 5.8 (p.61) we define a nested visualization. We take the same model as in the previous example but this time we add three nested nodes to each node. To define the nesting we make use of the method `#nodesUsingForEach( , , )`. This method expects three parameters. Like in the previous example the first parameter is a *Collection* of model objects and the second parameter is the shape that is used for painting the nodes. As third parameter the method expects a closure that will be executed for each model object.

Thanks to Groovy a Java version is almost as easy to use as the Smalltalk version. The major difference is the fact that the command `#nodesUsingForEach( , , )` with the third parameter being a rather large closure is harder to read. It is likely

```
view = new ViewRenderer();
view.nodesUsingForEach(
     [15, 10, 5, 5, 10, 15],
     new RectangleShape( {entity -> Color.yellow}),
     {enity ->
         view.nodesUsing([1, 2, 3], new RectangleShape(
                                              {entity -> 15 },
                                              {entity -> 8 },
                                              {entity -> Color.green }));
      }
);
view.layout(new LineLayout());
view.open();
```

Figure 5.8: A possible way tor define nested visualizations with the Java port of Mondrian.

that an interface could be found that fits better with the Java language syntax but this was not a goal of this work.

5.4 Improving the workflow

When we started to work on *Mondrian* most of the scripts were written in the workspace or in the evaluation pane of an inspector. Since writing *Mondrian* scripts is an iterative process the script needs to be selected and evaluated several times. Each time when we evaluate the script a *Mondrian* window opens that needs to be closed again after checking the visual result. Once the script works as expected it needs to be saved as a method. Otherwise the script will be lost when switching to a new image.

What we did not like about this workflow is the fact that we constantly need to change between the keyboard and the mouse. We write the script with the keyboard but we need to switch to the mouse for selecting the script and closing the *Mondrian* window.

This is why we developed the *Mondrian Easel*. The *Mondrian Easel* is an IDE for writing *Mondrian* scripts. Figure 5.9 (p.62) shows a screenshot of the *Mondrian Easel*. In the upper part of the editor we see the visualization that the script in the lower part of the editor creates. The visualization can either be created by clicking

the "Generate View" button or by using the platform-specific *Do-It* shortkey. Like this we can write the script and update the visualization without having to use the mouse once.

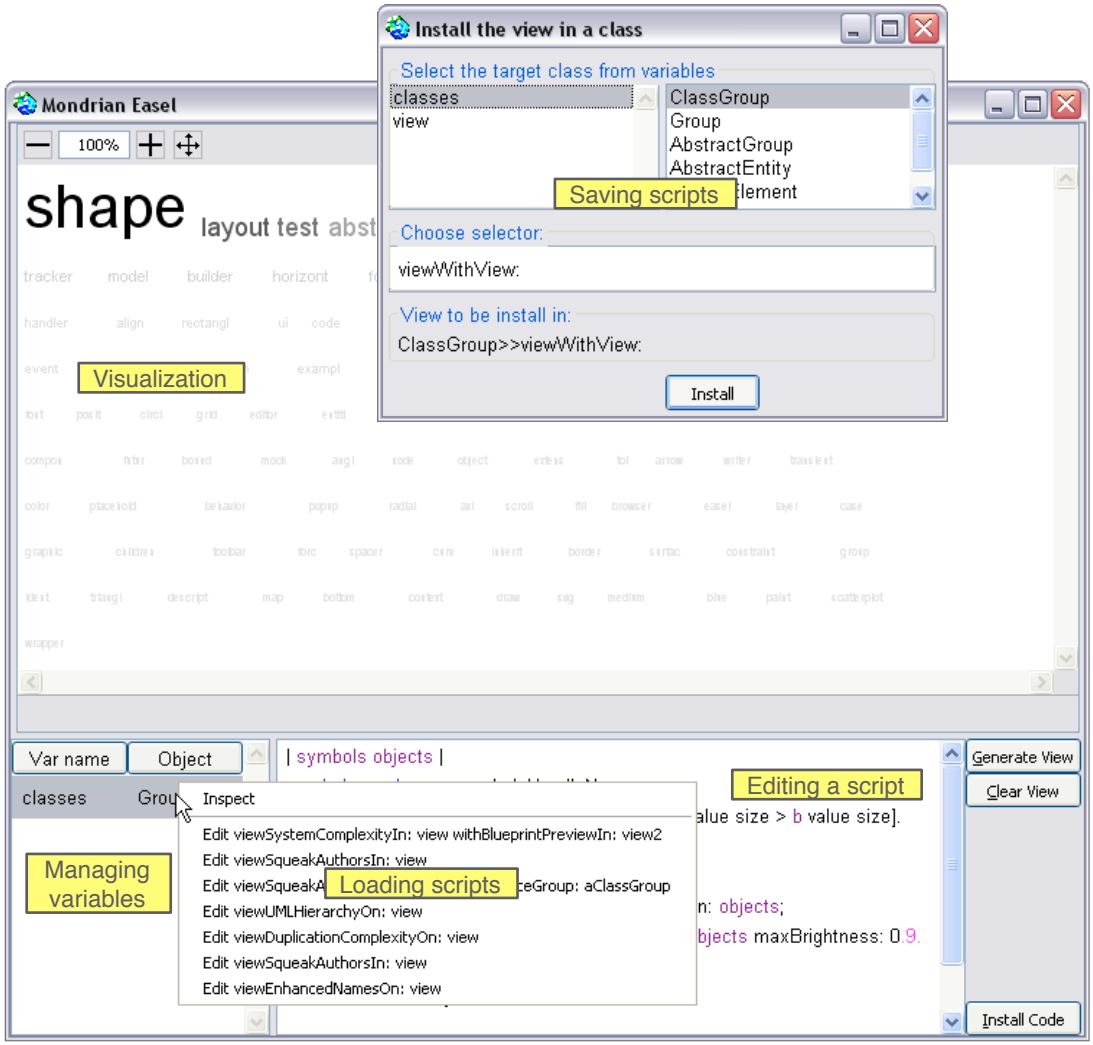


Figure 5.9: The *Mondrian Easel* is an IDE for loading, writing, viewing and saving scripts.

In the left bottom corner of the editor we can manage the data that is available to the editor. In Figure 5.9 (p.62) we only have one variable called "classes". This variable can be accessed from the script. Additional data can be added by dragging the data into the editor. We can also freely choose the variable names.

When right-clicking on any object in the variables panel we get a list with all

scripts that have already been defined for that particular object and we can load the script into the editor by just selecting a script from the list. Saving a script is just as easy as loading a script. To save a script we can click on the "Install code" button. This will open the "Install the view in a class" window. In that window we have access to all the variables that are defined in the editor. And for each variable we see the complete class hierarchy. Like this we can freely choose where to install the script. We can also define the name of the method that we would like to create. As a *safety measure* we add the method to a class extension in the "none" - package. Like this there is no danger of unintentionally overwriting an existing method.

The *Mondrian Easel* makes working with Mondrian a lot easier since we have all important tasks assembled into one IDE.

5.5 Custom tool example: A Moose property editor

Figure 5.10 (p.63) shows an example of a specialized tool. The editor can be used for fine-tuning visualizations that have a Moose model [Ducasse *et al.*, 2005] as data model. On the left hand side of Figure 5.10 (p.63) we have the visualization and on the right hand side we have the *Property Editor*.

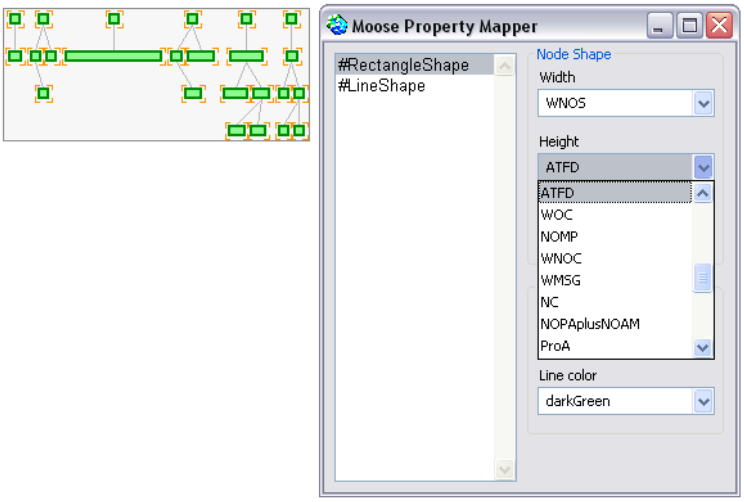


Figure 5.10: A visual editor for Moose entities

The *Property Editor* shows the shapes that are used in the visualization. In this example the list contains a *RectangleShape* and a *LineShape*. When clicking on

a shape in the list all the figures that are using that shape get highlighted in the visualization and the *Property Editor* shows all editable properties and the possible values. In the visualization we use a bordered rectangle. The *BorderShape* does not show up in the list since it is a decoration of the *RectangleShape*. When we click on the *RectangleShape* we get the properties of the *RectangleShape* and of the *BorderShape*.

When we change a property in the *Property Editor* the affected figures get updated instantly. This is possible because we work directly on the data model and because all "affected" figures share the same instance of the shape.

6 Conclusions

Visualizations are an important aid for data analysis and problem detection [Lee *et al.*, 2003]. By presenting information in a graphical way we make use of the massively parallel architecture of the human visual system for interpreting the data [Larkin and Simon, 1987].

A lot of tools only focus on a limit set of visualizations. These specialized tools are not flexible enough to support the user when a slightly or even completely different visualization is needed.

The goal of this thesis was to develop a flexible visualization model that can be used for fast prototyping of visualizations. We identified a number of factors that such a model needs to address:

- The visualization engine should be domain-independent.
- Visualizations should be easily composed from simpler parts.
- The visualization should be definable at a fine-grained level.
- Object creation overhead should be kept to a minimum.
- The visualization description should be declarative.

The visualization model that we developed works directly on the underlying data. Instead of making the data model compliant with the internal model of our visualization model we use *shapes* to perform a meta-model transformation. Like this we avoid duplicating the original data and we do not need to recreate the internal model when the original data changes.

Since *shapes* hold no model-specific state we can use the same instance of a *shape* to paint several model objects. This is an optimization in terms of memory usage and time.

We developed a scripting language for writing visualizations. The *ViewRenderer* is a facade that can be used for writing scripts in a concise way while hiding the internals of our model.

Nesting is a central part of our model. With *nesting* it is possible to prototype all sorts of different visualizations. We also support *instance-based representations* and *interaction*. Combining these three features makes it possible to explore the underlying data in an interactive way. Since we support *interaction* it is also possible to build complete tools on top of our model.

As a validation of our model we implemented Mondrian and used it to express various different visualizations.

List of Figures

Bibliography

[Arévalo, 2005] Gabriela Arévalo. *High Level Views in Object-Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, Berne, January 2005.

[Battista *et al.*, 1999] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tolls. *Graph Drawing — Algorithms for the visualization of graphs*. Prentice-Hall, 1999.

[Bederson and Hollan, 1994] Benjamin B. Bederson and James D. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, Marina del Rey, California, 1994.

[Bellin and Simone, 1997] David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison Wesley, 1997.

[Bosch *et al.*, 2000] Robert Bosch, Chris Stolte, Diane Tang, John Gerth, Mendel Rosenblum, and Pat Hanrahan. Rivet: a flexible environment for computer systems visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73, 2000.

[Brandt and Schmidt, 1995] Soren Brandt and René W. Schmidt. The design of a meta-level architecture for the BETA language. In *Proceedings of META '95: Workshop on Advances in Metaobject Protocols and Reflection at ECOOP '95*, August 1995.

[Crapo *et al.*, 2000] Andrew W. Crapo, Laurie B. Waisel, William A. Wallace, and Thomas R. Willemain. Visualization and the process of modeling: a cognitive-theoretic view. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 218–226, New York, NY, USA, 2000. ACM Press.

[D'Ambros and Lanza, 2006a] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, page to be published, 2006.

[D'Ambros and Lanza, 2006b] Marco D'Ambros and Michele Lanza. Software

bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering)*, pages 227 – 236. IEEE Computer Society Press, 2006.

[Ducasse and Lanza, 2005] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, January 2005.

[Ducasse *et al.*, 2004a] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Ducasse *et al.*, 2004b] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of ESUG International Smalltalk Conference 2004*, pages 231–257, September 2004.

[Ducasse *et al.*, 2005] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[Eick *et al.*, 1992] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[Fowler, 1997] Martin Fowler. *UML Distilled*. Addison Wesley, 1997.

[Gansner and North, 2000] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[Gansner *et al.*, 1993] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[Gîrba *et al.*, 2005] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[Gîrba, 2005] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[Greevy *et al.*, 2006] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3d. In *Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization)*, September 2006. to appear.

[Koenig *et al.*, 2006] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications, 2006.

[Koschke, 2003] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[Lange, 2003] Christian F. J. Lange. Empirical investigations in software architecture completeness. Master's thesis, University of Eindhoven, 2003.

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

[Lanza and Ducasse, 2005] Michele Lanza and Stéphane Ducasse. Codecrawler– an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, Milano, 2005.

[Lanza, 2003a] Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

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

[Larkin and Simon, 1987] Jill Larkin and Herbert Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, pages 65–99, 1987.

[Lee *et al.*, 2003] Michael D. Lee, Rachel E. Reilly, and Marcus E. Butavicius. An empirical evaluation of chernoff faces, star glyphs, and spatial visualizations for binary data. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 1–10, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[Lentzsch, 2004] Karsten Lentzsch. The jgoodies forms framework, 2004. The whitepaper can be downloaded together with the jar file and the source code.

[Lungu *et al.*, 2005] Mircea Lungu, Adrian Kuhn, Tudor Gîrba, and Michele Lanza. Interactive exploration of semantic clusters. In *3rd International Work-*

shop on Visualizing Software for Understanding and Analysis (VISSOFT 2005), pages 95–100, 2005.

[Lungu *et al.*, 2006] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 183–192, Los Alamitos CA, 2006. IEEE Computer Society Press.

[M.-A. D. Storey and Michaud, 2001] C. Best M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[Meyer *et al.*, 2006] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006)*, pages 135–144, New York, NY, USA, 2006. ACM Press. To appear.

[Moore *et al.*, 2004] William Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. IBM International Technical Support Organizat, 2004.

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

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

[Muskens, 2002] Johan Muskens. Software architecture analysis tool. Master's thesis, University of Eindhoven, 2002.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[Panas *et al.*, 2005] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualization with Vizz3D. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS 2005)*, pages 173–182, 2005.

[Reiss, 2001] Steven P. Reiss. An overview of bloom. In *PASTE '01: Proceedings of*

the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 2–5, New York, NY, USA, 2001. ACM Press.

[Reiss, 2006] Steven P. Reiss. Visualizing program execution using user abstractions. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 125–134, New York, NY, USA, 2006. ACM Press.

[Schneider and Nierstrasz, 1999] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures — Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[Schneider, 1999] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[Snyder, 2003] Carolyn Snyder. *Paper Prototyping*. Morgan Kaufmann, 2003.

[Storey and Müller, 1995] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[Termeer *et al.*, 2005] Maurice Termeer, Christian F. J. Lange, Alexandru Telea, and Michel R. V. Chaudron. Visual exploration of combined architectural and metric information. In *VISSOFT*, pages 21–26, 2005.

[Tufte, 1990] Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[Tufte, 2001] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[Viégas *et al.*, 2006] Fernanda B. Viégas, Scott Golder, and Judith Donath. Visualizing email content: portraying relationships from conversational histories. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 979–988, New York, NY, USA, 2006. ACM Press.

[Wade and Swanston, 2001] Nicholas Wade and Michael Swanston. *Visual Perception: An Introduction*. Psychology Press, 2001.

[Wu *et al.*, 2004] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, November 2004. IEEE Computer Society Press.