Implementing Mondrian in Glamorous Toolkit

Bachelor Thesis

Cyrill J. Rohrbach
from
Jegenstorf BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

September 2021

Prof. Dr. Oscar Nierstrasz
Prof. Dr. Alexandre Bergel

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Developers spend a lot of time reverse engineering software. To do this they often rely on reading the code, which is a slow and unscalable process. They rely on code reading because the environments used for the development are centered around the code editor and do not really offer other tools to help the developer understand the software.

The goal of moldable development is to change this. The environment should provide tools to help the user understand software. In order to prevent the tools from not being suitable for the application, the developer should adapt and develop the tools alongside the software.

Glamorous Toolkit is a new development environment based on Pharo and built around the philosophy of moldable development. One of the tools offered by Glamorous Toolkit to help understand a piece of software is a multipurpose visualization tool called GtMondrian.

GtMondrian takes scripts and turns them into interactive visualizations. These scripts allow for endless customizability, but to do this the user has to know how the graphical elements of Glamorous Toolkit work. Since it takes time to familiarize oneself with those elements, this could well be something that prevents developers from using it to adapt the development tools, and therefore sabotages the concept of moldable development.

We propose a tool similar to GtMondrian called CRMondrian. It has a lot of the same functionality with the major difference that the most commonly used customizations, such as changing the shape of the nodes within a graph, are done using builders. Therefore it just requires one simple keyword from the user and eliminates the need for the user to know how the graphical elements work.

Contents

1 Introduction **1**
1.1 Outline 2

2 Introducing Mondrian **3**
2.1 History 3
2.2 Key Features 6

3 Introducing Glamorous Toolkit **7**
3.1 Moldable Development 8
3.2 One Rendering Tree 9
3.3 GtMondrian 10

4 Introducing CRMondrian **13**
4.1 CRMondrian 14
4.2 Shape Builder 15
4.3 Edge Builder 17
4.4 Canvas 18
4.5 Node 18
4.6 Normalizer 19
4.7 Layout 19
4.8 CRGroup 20

5 Validation **21**
5.1 Glamorous Toolkit 21
5.1.1 The Good 21
5.1.2 The Bad 22
5.1.3 Conclusion 22
5.2 CRMondrian evaluation 22
5.2.1 Does CRMondrian have the main components of a Mondrian Implementation 23
5.2.2 Comparison to GtMondrian 24

6 Conclusion **28**

7 Anleitung zu wissenschaftlichen Arbeiten **29**
7.1 Installation 29
7.2 How do I use it? 30
7.2.1 Nodes 30
7.2.2 Edges 33
7.2.3 Layouts 34
7.2.4 Actions 36
7.2.5 Normalizer 37
7.2.6 System Complexity 39
7.2.7 Nested Graphs 40
7.2.8 Class Blueprint 41
7.2.9 Mind Map 43

1 Introduction

Software developers spend a considerable amount of time trying to understand how a piece of software works. This process is often done by reading code, which is a very slow and not a scalable way to understand software.
Visual representations of the software can help speed up this process. But the visualization tool needs to be easy to use and flexible to accommodate all sorts of different software. Existing tools often only create certain predefined visualizations on a specific domain. As a consequence, it is necessary for the developer to use multiple different tools to get a good understanding of the software. Since the different tools often need different types of input this leads to a lot of extra work, which in turn leads to the tools not being used.

Michael Meyer took this problem as motivation for his master thesis [3] and together with Dr. Tudor Gîrba he developed a tool that creates visualizations from simple scripts. They called their tool Mondrian.

A few years later Tudor Gîrba started developing Glamorous Toolkit, an entirely new Pharo environment, which is based on the idea of moldable development.

Moldable Development wants to enable the developer to create and adapt the tools used for development along side the software. This should not only help other developers to understand the code better, but also help debug the software, since objects are not just displayed as a list of attributes anymore, but in a way which makes sense for this specific object.

Since Moldable Development and Mondrian both have the goal to support the programmer in understanding code, Glamorous Toolkit has a capable Mondrian implementation, called GtMondrian, deeply integrated.

GtMondrian offers a lot of functions and works well. However, it requires the user to fully understand the graphic elements of Glamorous Toolkit. To change the shape of the nodes for example, the user will have to find out how they can create an element that matches his shape requirements, then pack this element within a stencil, which is then used by GtMondrian to create the nodes.

The goal of this thesis is to create a new Mondrian within Glamorous Toolkit that is easier to use than GtMondrian. The process of developing this new Mondrian should further answer the question whether it is easier to develop a visualization tool similar to Mondrian in Glamorous Toolkit, than it is to do so in the traditional Pharo environment.

1.1 Outline

In this thesis we first introduce Mondrian [chapter 2] and GlamorousToolkit [chapter 3] from there we gather the requirements for our new Mondrian. We then introduce CRMondrian and explain how it works in chapter 4. In chapter 5 we first discuss Glamorous Toolkit and then compare CRMondrian to GtMondrian. After that we draw a conclusion and describe some possible future improvements [chapter 6].

Chapter 7 consits of a manual on how to use CRMondrian.

2
Introducing Mondrian

Mondrian is a powerful mulitpurpose API to create interactive visualizations of all sorts by writing a simple script.

2.1 History

The idea of having small scripts to describe and create interactive visualizations was proposed in 2006 by Michael Meyer and Tudor Gîrba. Meyer's master thesis "Scripting Interactive Visualizations" [3] proposed a domain independent tool that takes such scripts and turns them into visualizations.

They noticed that existing programs to visualize software were very specialized and not at all flexible enough to adapt to the ever changing requirements for different software. This means the developer would need to be familiar with several visualization tools, all of them requiring a different input format and time to convert the software structure into their own graphical models. They decided to create a new tool called Mondrian.

The goal for their tool was to be flexible enough to adapt to whatever the user could need. Therefore they decided that the following requirements should be fulfilled [3, p. 2]:

- The visualization engine should be domain independent, meaning that the tool should be able to handle any data objects, no matter if it is a set of numbers, files or classes.
- The visualization should be composed of basic blocks that can be defined by the user.
- The visualization should have support for nesting and the shape of the nodes should be definable at any level.
- The tool should not create an internal copy of the data object but instead use the original object. This saves both memory and time and eliminates the necessity of updating the graph object when the original object is changed.
- The description of the visualization should only be a mapping between the data and the visualization model, and therefore be declarative.

The tool that was created from those requirements is an API that takes scripts and generates interactive visualizations from them. [Figure 2.1]

```
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.
```

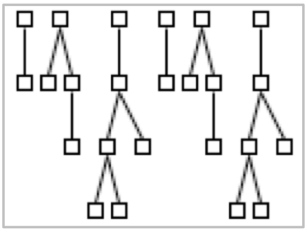

Figure 2.1: Original Mondrian script and resulting graph. [3, p. 4]

The scripts always start by creating a new ViewRenderer. After that the nodes are added. When adding the nodes the user can choose between a set of predefined shapes. All of them can be decorated with other shapes. For this they provide a set of methods with the most common decorations such as background color, size and border. These methods not only make the script easier to read and shorter, but since otherwise the

shapes would have to be created manually, it eliminates the need for developers to learn how to create these shapes.

Once the nodes and their shapes are defined, the user can add edges using an EdgeBuilder. The EdgeBuilder receives some information about which nodes need to be connected and the shape of the lines used. It then automatically connects the nodes.

At last the user can add a layout to arrange the nodes within the graph.

To make the graphs more interactive it is possible to add interactions, a popup text for example, to the nodes. [Figure 2.2]

```
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.
```

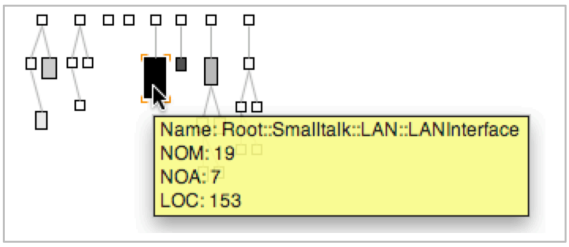

Figure 2.2: Original Mondrian script with interaction. [3, p. 25]

To accommodate more complex visualizations it is possible to have infinite nesting, meaning each graph can again be used as a node within another graph. This can for example be used to create a Class Blueprint like graph, as can be seen in figure 2.3.

```
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.
```

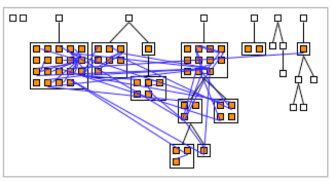

Figure 2.3: Original Mondrian script to create a Class Blueprint like graph. [3, p. 24]

Michael Meyer and Tudor Gîrba first implemented the tool in Pharo and later on proved with a more basic implementation in Java that their concept works platform and technology independent.

2.2 Key Features

Based on Michael Meyer's master thesis one can create a list of features and requirements which an implementation of Mondrian should have to be able to create any visualization needed.

- *Interactive visualizations:* The resulting visualization should not just be a picture but allow for interactions. For example a click on the node should open up the object this node represents and the user should have the possiblity to add more interactions if needed.
- *Domain independence:* The tool should be able to create visualizations for any provided object.
- *Infinite nesting:* A graph should accept other graphs as nodes.
- *No meta Object:* The tool should not create meta objects representing the objects that should be displayed, but rather use those objects directly. This has the advantage that there is no need to update the meta objects once the original one has changed. Since the meta object creation would take time, it is also faster and less memory intensive.
- *Ability to easily change the node shape:* The visualization should be as easily customizable as possible and the user should be able to interact with a high level interface to do it.
- *Ability to create polymetric views:* It should be possible to map an attribute of the node (i.e. size or color) to a metric of the represented object.

3
Introducing Glamorous Toolkit

In software development IDEs or integrated development environments are used to help create software.

They are called integrated since they not only have tools to support development time activities, such as code editors, but also support run-time activities with debuggers and other tools, allowing one to develop software without leaving the IDE. [4]

Even though there are quite a few different IDEs on the market, they all look and work similar but none of them are really optimized to support the programmer in understanding a piece of software.

So what if integrated would not just mean a fixed set of features in a static environment to help write code, but instead an environment that adapts to the code written and can be molded to the things it should display?

Glamorous Toolkit does exactly this. It is a new Pharo environment that was developed with moldable development in mind, it therefore looks and works completely different from the normal Pharo and any other existing IDE.

3.1 Moldable Development

According to researchers 30 to 70 percent of the development and maintenance time is taken up by the comprehension of legacy code. [1] And even though there are existing tools which help in understanding a program, most developers still rely on the rather inefficient way of code reading. A reason for this is that most IDEs are centered around code reading and tools or plugins aiming to help the developer understand the software, are not integrated into the program and the workflow. Additionally the tools are built with a specific application in mind, leading to particular visualizations with little room for customization and poor support for the creation of visualizations outside their intended domain.

Taken all of this into account, and considering the time a developer would need to invest into studying these tools and customizing them as much to their application as possible, it is understandable why they are not used often.

Moldable Development offers a different approach: The developers should not only create their software but also adapt and evolve their development tools (inspector, code editor, debugger, *etc.*) alongside it. This has the benefit that the tools will be perfectly adapted to the actual application domain. These tailor-made tools then will not only support the current developers building the application, but also future developers to understand it. For this to work the tools and the ability to customize them have to be deeply integrated into their environment and their customization needs to be as low cost as possible, because if the tool development takes up too much time, the developers most likely will not develop them together with the software.

Glamorous Toolkit's very idea was to create a Pharo environment following the moldable development philosophy. This is visible throughout the environment, but probably the best example for it is the inspector. [Figure 3.1] When creating a new object type it is very easy to add multiple different views that will be shown when the object is inspected. In the inspector it then is possible to switch between those views to get the best visual representation depending on the application.

Especially compared to a traditional inspector where an object is often only represented by a table of its attributes, the different views allow for a better understanding of what the object represents.

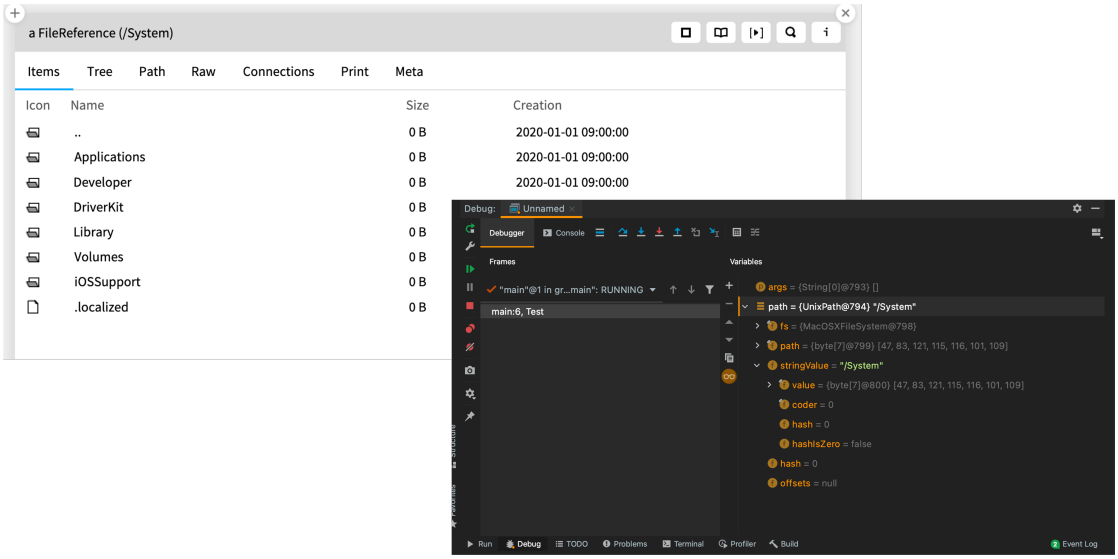

Figure 3.1: Inspector view in Glamorous Toolkit (left) and IntelliJ (right)

3.2 One Rendering Tree

One important part of Glamorous Toolkit is its all new graphical stack called Bloc. The special thing about Bloc is that every element shown to the user, be it some graphic, text or even the environment itself is rendered in one big tree.

In most development environments visualizations are confined to their own world, a world that is different from the one all the widgets and other parts of the environment are rendered in. This differentiation between the visualization and the environment world leads to poor integration of the visualizations within the environment and makes it hard to combine the two.

Especially for moldable development, where the environment should be adapted to and developed with the software, it is important that the visualizations, used to help understand the software, do not live in their own world, but are integrated into the environment. Exactly this is achieved with the one rendering tree, as all elements live in the same world and can interact with each other.

Since all elements can be combined and used in any context, it is necessary to have a way to mark objects according to what they represent, as graph or inspector for example. Normally this is done using higher level models that then interact with or even create the rendering tree. But since this is complicated and incompatible with a single rendering tree Glamorous Toolkit works with annotations instead. The annotations are

used to create tiny objects that know the element and extend their interface with specific functionality needed for its context. An element annotated as graph node for example can return all connected edges.

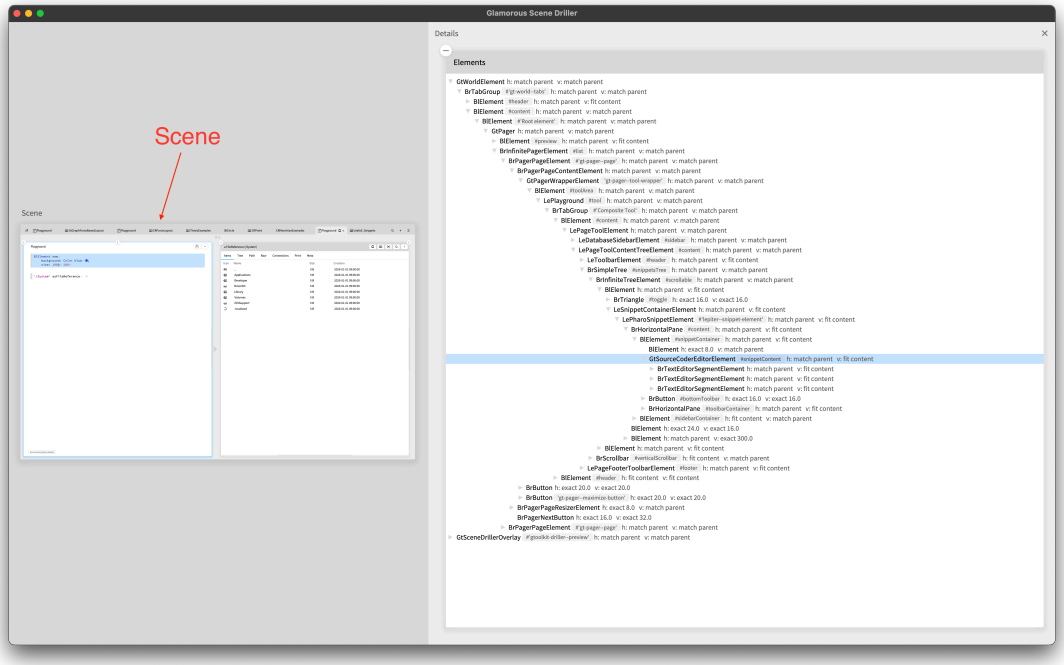

Figure 3.2: The rendering tree for all elements displayed in the scene

3.3 GtMondrian

To easily create visual representations of objects, Glamorous Toolkit has a deeply integrated Mondrian implementation called GtMondrian.

Since Tudor Gîrba helped create the Mondrian concept and is one of the key developers of Glamorous Toolkit it makes sense that GtMondrian is very similar to the Mondrian suggested in Michael Meyer's master thesis. [3]

```
view := GtMondrian new.
view nodes with: (1 to: 9).
view edges connectFrom: [:x | x // 2].
view layout tree.
view
```

Listing 1: Simple GtMondrian script

The scripts for GtMondrian, as seen in the example in listing 1, are very simple, as long as no customizations are needed.

They create a new view renderer, add the nodes and edges and set the layout. But they quickly get more complicated once the node shape should be changed. Different to the Mondrian proposed by Michael Meyer, GtMondrian does not have multiple predefined shapes to choose from, but instead gives the user the opportunity to provide a stencil according to which the nodes are then created. The stencil is a block closure that takes the object the node should represent and then defines the shape the node should have. [Listing 2] This approach has the advantage that the shape can be anything the user wants it to be. Furthermore it enables easy mapping of the object's attributes to the shape's attributes to create polymetric views. Since the stencil is nothing more than a block that is executed to create the nodes, it can also be used to add different interactions, in the form of event handlers, to the nodes.

```
view := GtMondrian new.
view nodes shape: [:obj |
      BlElement new
         geometry: (BlCircle new);
         background: Color blue.
   ];
   with: (1 to: 9).
view edges connectFrom: [:x | x // 2].
view layout tree.
view
```

Listing 2: GtMondrian script with point nodes

Overall the idea to provide a stencil for the shape gives the user unlimited customization, but exactly this customization is also the biggest drawback of GtMondrian. If the user does not want a lot of customization for the visualization, but just to change the node shape from boxes to points for example, they still have to learn how BlElements (the graphic elements used to create the nodes) work and how it is possible to create different shapes with them. It gets even more complicated if the nodes should for example have a popup text that appears when hovering over it. [Listing 3]

```
view := GtMondrian new.
view nodes shape: [:obj |
      BlElement new
         geometry: (BlCircle new);
         background: Color blue;
         aptitude: (BrGlamorousWithTooltipAptitude new
            showDelay: 0;
            hideDelay: 0;
            contentStencil: [BrLabel new aptitude:
               BrGlamorousLabelAptitude new
               glamorousRegularFontAndSize;
               alignCenter;
               text: obj])
   ];
   with: (1 to:9).
view edges connectFrom: [:x | x//2].
view layout tree.
view
```

Listing 3: GtMondrian script with point nodes and popup text

We propose a new and independent Mondrian implementation called CRMondrian within Glamorous Toolkit that besides the features listed in section 2.2 implements the following features:

- *Predefined shapes:* There are several ShapeBuilders implemented that allow for an easy change of the node shape.
- *Tooltip as standard:* The nodes always have a popup text when hovering over them.
- *Ability to add new shapes:* New ShapeBuilders to add shapes can easily be difined by the user.
- *Normalizer:* The tool provides a normalizer that allows for straightforward creation of polymetric graphs.

4
Introducing CRMondrian

CRMondrian is a new implementation of Mondrian within Glamorous Toolkit that is independent of GtMondrian.

CRMondrian works with a similar concept to what was suggested in Michael Meyer's master thesis. [3]

An instance consists of five main parts. CRMondrian, it controls everything and has instance variables to keep track of all the other parts. The EdgeBuilders and ShapeBuilders together with the Normalizers are used to create the elements according to the users specification. These elements are then added to the Canvas, which is then presented to the user.

4.1 CRMondrian

CRMondrian is the main object and the equivalent to the ViewRenderer in Meyer's Mondrian. It provides methods to create all the other objects needed for the visualization and keeps track of them using instance variables

Before an instance can be displayed, CRMondrian controls the actual building of the view. It first of all gathers all the nodes from the different ShapeBuilders, then passes those nodes to the EdgeBuilders to create the edges. Once all elements are created they are placed on to the canvas and arranged using a layout. [Figure 4.1]

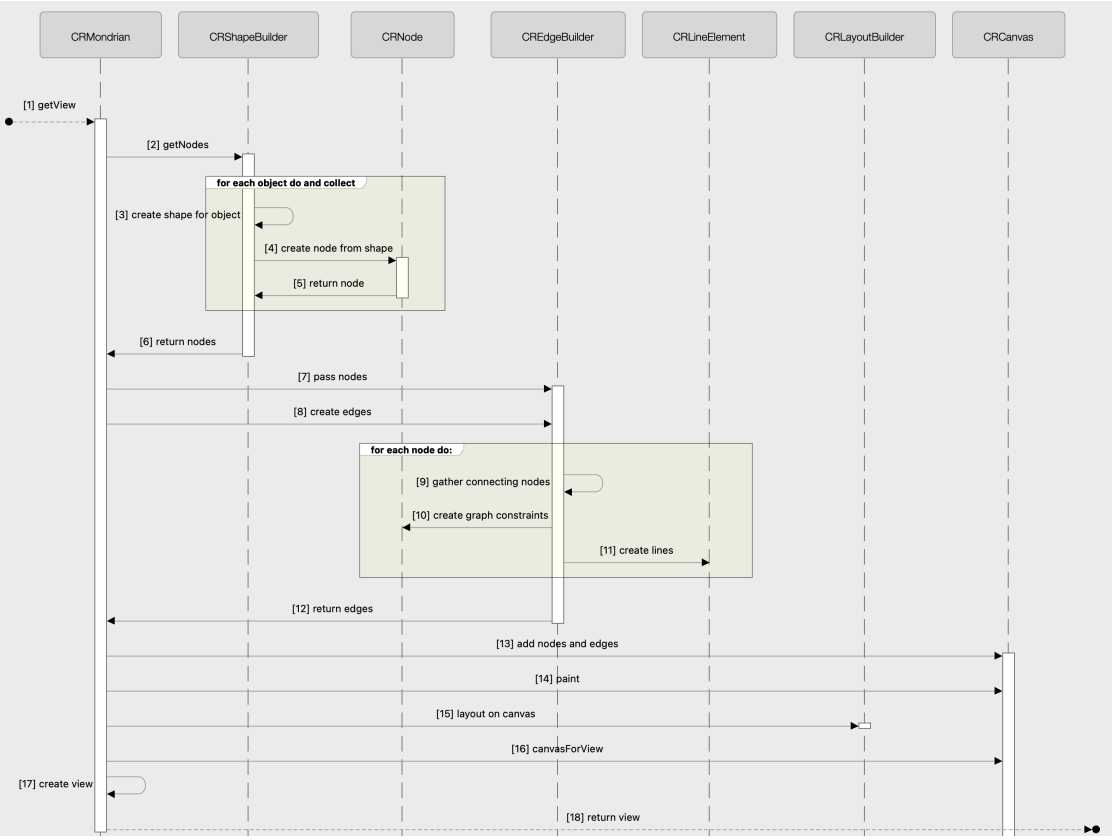

Figure 4.1: Sequence diagram describing the internal visualization creation process

4.2 Shape Builder

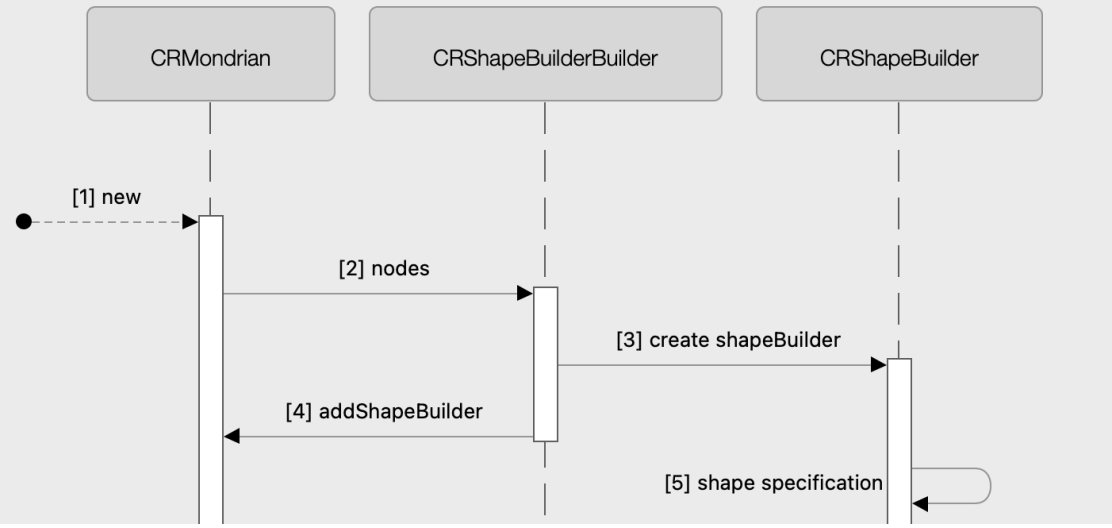

Figure 4.2: Sequence Diagram describing the ShapeBuilder creation and specification gathering process

With the method CRMondrian>>nodes a new CRShapeBuilderBuilder is created. It allows the user to specify which shape the nodes should have. The CRShapeBuilder-Builder then creates the actual ShapeBuilder and adds it to the shapeBuilderCollection of the Mondrian instance. (This process is visualized in figure 4.2)

An instance of CRMondrian can have as many ShapeBuilders as required to create the needed visualization.

A ShapeBuilder is always a subclass of CRShapeBuilder. It has to implement the method createShapeFor: obj [Figure 4.3], which returns a new graphical element that is tailored to the given object and the specifications provided by the user.

To enter these specifications the ShapeBuilder provides methods specific to the shape it is going to build.

The class CRShapeBuilder already implements methods to add normalizers for the background color, border color, size, height and width as well as a method to set the toolTip content and the color of the different highlights, which are used to visualize which node is selected.

```
createShapeFor: obj
"
    I create a CRBox with the given object as model according to the stored specification.
"
    | shape |
    shape := CRBox new.


    size ifNotNil: [shape size: ((self evaluate: size on: obj) asPoint)].
    width ifNotNil: [shape width: (self evaluate: width on: obj)].
    height ifNotNil: [shape height: (self evaluate: height on: obj)].
    background ifNotNil: [shape background: (self evaluate: background on: obj)].

    (borderWidth notNil and: [borderColor notNil])
        ifTrue: [
            shape border: (BlBorder paint:(self evaluate: borderColor on: obj) width: (self evaluate: borderWidth on: obj))
        ]
        ifFalse: [
            borderWidth ifNotNil: [shape border: (BlBorder paint: Color black width: (self evaluate: borderWidth on: obj))].
            borderColor ifNotNil: [shape border: (BlBorder paint: (self evaluate: borderColor on: obj) width: 4)]
        ].


    ^ shape
```

Figure 4.3: Example for a createShapeFor method

To build the view CRMondrian will call all its ShapeBuilders and ask them to return nodes for the object collection provided by the user. The ShapeBuilder will then iterate through all objects, create the shapes for them using the createShapeFor: obj method, and create CRNode elements for each shape. All these nodes are then returned as a CRGroup. [Figure 4.4]

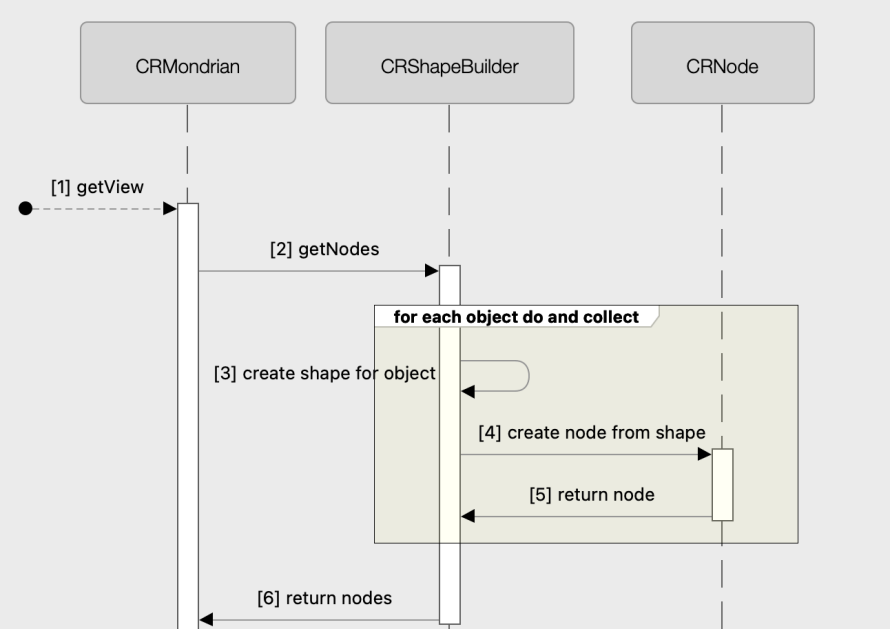

Figure 4.4: Sequence diagram for the node creation process

4.3 Edge Builder

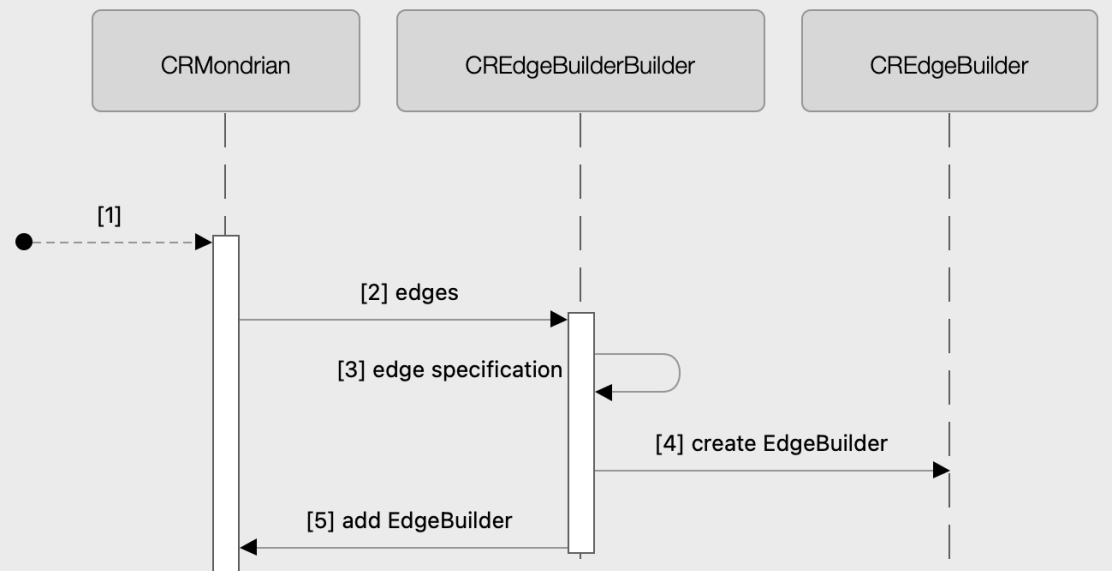

Figure 4.5: Sequence diagram describing the EdgeBuilder creation and edge specification gathering process

To add edges to the graph, the user calls the method CRMondrian>>edges. This will then create a new instance of CREdgeBuilderBuilder, which is then used to gather the specifications for the edges which then will be used to create the CREdgeBuilder, the actual object that will build all edges. This has to be done because edges can be built using four different builders for node:1, 1:node, node:m and m:node relationships.

The actual builder will then create the edges for a provided set of nodes. To only create edges between a certain subset of nodes it is possible to add a condition that has to be true to create an edge to this node.

When the view is built CREdgeBuilder will iterate through all nodes and for each node gather the nodes it should be connected to, then create the visible line and the graph constraint representing the edge.[1]

The lines created in this process are then returned to CRMondrian as a collection so they can be added to the canvas later on in the creation process. [Step 12 in Figure 4.1]

[1]Graph constraints can be added to any graphical object within Glamorous Toolkit and they are needed for the layouts to work correctly and for GT to properly treat the nodes and edges.

4.4 Canvas

The canvas is the top level element that will be displayed to the user. After all nodes and edges are created, they will be added to the canvas element. The elements are not directly added as child elements, but are stored as CRBindingGroup (see CRGroup) in an instance variable.

With the call of the CRCanvas>>paint method, all the elements will be added as child elements to the BlElement representing the canvas. This staging is done to allow the canvas to keep track of its elements.

CRMondrian can call two different methods to get the canvas element. If the canvas is used as a subgraph within a bigger visualization, the method CRCanvas>>canvasAsNode will return the canvas with a black border and create an event handler to open the Mondrian instance in a new inspector page when the user clicks on the canvas.

If the canvas is used by itself, the method CRCanvas>>canvasForView will return a zoomable and pannable element with scroll to zoom functionality added to it.

4.5 Node

The node is a wrapper element for any other BlElement. This element is then placed in the center of the node as child element. The wrapper element adapts its size to its child but is five points bigger on each side. This size difference is then used to add highlights to the node. The highlights are done by changing the background color of the wrapper element.

Besides the highlights the node object offers all the needed functionality to turn any regular graphical element into a node compatible with CRMondrian:

- It keeps track of the object it should represent using the model instance variable.
- CRNode will add a tooltip matching the description, provided with CRNode>>toolTipContent, to the center element.
- It keeps track of the CRBindingGroup it belongs to. This is used to make sure only one node at the time is marked as clicked within the same graph.
- It keeps track of the anchor position that should be used to attach edges.

4.6 Normalizer

The normalizer can be used to create polymetric views where for example the number of lines of code of a method is mapped to the height of a node. The resulting view will then have nodes with a height within a certain predefined range, and which are higher, the more lines of code the method has.

There is one normalizer for number values, for example the height, and another one for color. The CRColorNormalizer will blend two colors depending on the object the node should represent.

Normalizers can be added through the ShapeBuilder and will work within the object collection of the specific ShapeBuilder.

The value for a specific node will be evaluated using the CRNormalizer>>valueFor method pictured in Figure 4.6. It uses linear interpolation to calculate the value for a given object.

```
valueFor: aNode
"
    I return the value for the given node.
    Therefore I calculate the ratio for the given
    node and then blend the from and the to value.
"
    | nodeValue ratio value |
    nodeValue := key cull: (aNode model).

    ( (maxValue - minValue) <= 0 ) ifTrue:
        [^to].

    ratio := (nodeValue - minValue) / (maxValue - minValue).

    value := from + ((to - from) * ratio).

    ^value
```

Figure 4.6: CRNormalizer>>valueFor method

4.7 Layout

The layouts are used to arrange the nodes in a specific way.

The different layouts provided are either BlLayouts or GtGraphLayouts wrapped in a subclass that implements the trait TCRLayout that offers the on: aCanvas method. [Figure 4.7] This method wraps the layout in a BlOnceLayout and then applies it on the canvas. The reason for wrapping a layout in a BlOnceLayout is that this allows the nodes to be dragged by the user. If the layout is applied directly, the canvas will not allow the nodes to be moved.

```
on: aCanvas
"I apply the layout to the given Canvas. But first I create a BlOnceLayout from myself to allow draggable nodes"
    aCanvas layout: (BlOnceLayout new▸ layout:▸ self)
```

Figure 4.7: TCRLayout>>on method

4.8 CRGroup

The CRGroup is simply an OrderedCollection with the ability to add a name and more importantly to search for nodes representing a certain object. The search functionality is used for the edge creation process.

CRGroup also provides methods to manage the highlights of the nodes. This is necessary to be able to remove the highlight from all nodes, when an other node is selected.

Every node on the canvas therefore belongs to the same CRBindingGroup, a subclass of CRGroup with the only difference that it will add a reference to itself to all objects that are added to it. That way a node can call CRGroup>>highlight, and the group will take care of unhighlighting all other nodes.

5 Validation

5.1 Glamorous Toolkit

5.1.1 The Good

Glamorous Toolkit is based on the idea of moldable development, which is very dependent on good visualizations, and therefore offers a good integration of visualizations in the whole environment.

The visualization based inspector is a big step up from traditional inspectors and makes the debugging and the development of graphs easy.

Glamorous Toolkit already offers a very good graph structure that allows one to create graphs out of any BlElements. This especially made the basics of Mondrian easy to implement since there is already an existing structure and layouts that can be used.

The One Rendering Tree was especially useful to create nested graphs. The ability to add any BlElement to another BlElement is great. And the ability that elements of different levels can interact with each other was useful to create edges between nodes with different levels of nesting.

5.1.2 The Bad

Even though the One Rendering Tree, and how the BlElements behave in it, was very useful to create nested graphs, it also came with one big downside. BlElements cannot be added to more than one parent at a time and they cannot be copied.

This created a problem when opening a subgraph, a graph that is added to another graph as node, in a new inspector window. Since everything, also the environment, is rendered in the same graphical tree, the inspector is treated the same way any other element would be. And since the subgraph already has the graph as its parent it cannot be added to the inspector.

The only working solution we found is to recreate the whole graph every time it is added to an element. That works well, but is especially for big graphs with many nodes quite resource intensive and it takes a while to compute the visualization every time it is displayed.

The biggest downside to Glamorous Toolkit is definitely its shortage of documentation. Classes and methods do not have comments, examples are missing or contained in complicated structures making them unnecessarily difficult to understand. Since it is a new environment the usual way of consulting forums such as StackOverflow is also rather useless.

Since Glamorous Toolkit is still in its beta testing phase as of, September 2021, it is to be expected that the documentation will improve once the final release arrives.

5.1.3 Conclusion

Since this thesis focused on implementing Mondrian in Glamorous Toolkit and not in a normal Pharo environment, it is not possible to conclusively say whether it is better suited or not. But considering that its biggest downside right now is the missing documentation while providing an excellent ready-to-use graph structure, it does definitely not fall short in comparison with the traditional Pharo environment.

5.2 CRMondrian evaluation

The goal was to create a Mondrian within Glamorous Toolkit that offers the functionality proposed by Michael Meyer's master thesis and allows for simpler scripts than the already existing GtMondrian.

5.2.1 Does CRMondrian have the main components of a Mondrian Implementation

To check whether we have created a comparable implementation of Mondrian, we check if the key features described in 2.2 are available.

- *Visualizations have to be interactive*

  The visualizations created by CRMondrian offer the ability to click on nodes to open the objects, have draggable nodes and highlights when interacting with the nodes. Additionally the user can define actions to add further interactions to the nodes.

- *Domain independent*

  CRMondrian can create visualizations out of any objects within Glamorous Toolkit. Thanks to many different interfaces, which allows Pharo to interact with the outside world, the types of objects that can be visualized are almost unlimited.

- *Infinite nesting*

  Thanks to the way the one rendering tree works nesting is very easy, and since CRNode can take any element and turn it into a node within a CRMondrian visualization, it is possible to take the canvas created with an instance of CRMondrian and use it in an other instance as node.

- *No meta Object*

  CRMondrian only wraps the objects in a CRNode, but it does not create a copy of the object. Therefore the overhead created is as small as possible.

- *Ability to easily change the node shape*

  Thanks to the different ShapeBuilders, changing the shape of the nodes is as easy as adding a keyword. Furthermore it is possible to add a new ShapeBuilder to make scripts with complicated shapes, such as a Class Blueprint for example, as simple and readable as scripts with simple shapes.

- *Ability to create polymetric views*

  The Normalizer allows one to easily create polymetric views without making the script much longer or the visualization less pleasing.

Overall CRMondrian fulfills all previously mentioned requirements and is therefore comparable to GtMondrian.

5.2.2 Comparison to GtMondrian

Since the goal of this thesis was to create an easier to use visualization tool than GtMondrian, we will focus on comparing the scripts needed to create the same visualizations in GtMondrian and in CRMondrian.

The scripts of CRMondrian and GtMondrian have pretty much the same basic structure. They start by creating a new view renderer, then the nodes and edges are added, and lastly the layout is set.

Probably the biggest difference of CRMondrian compared to GtMondrian is the implementation of different ShapeBuilders for different shaped nodes.

This allows the user to choose the node shape with a simple keyword, while GtMondrian requires the user to add a stencil describing the shape. [Listing 4 and 5]

```
view := CRMondrian new.
view nodes point
         with: (1 to: 10).
view
```

Listing 4: CRMondrian

```
view := GtMondrian new.
view nodes shape: [:obj |
     BlElement new
       geometry: (BlCircle new);
       background: Color black.
   ];
   with: (1 to: 10).
view
```

Listing 5: GtMondrian

If the user desires to add a tooltip showing the name of the represented object to the nodes, it will get even more complicated in GtMondrian. [Listing 6] In CRMondrian on the other hand this is a standard and every node has a tooltip.

```
   view := GtMondrian new.
   view nodes shape: [:obj |
      BlElement new
         geometry: (BlCircle new);
         background: Color black;
         aptitude: (BrGlamorousWithTooltipAptitude new
            showDelay: 0; hideDelay: 0;
            contentStencil: [BrLabel new aptitude:
               BrGlamorousLabelAptitude new
               glamorousRegularFontAndSize;
               alignCenter;
               text: obj])
      ];
      with: (1 to: 10).
```

Listing 6: GtMondrian code for point nodes with a tooltip

Similar to how one can add a tooltips to all nodes it is possible to add any EventHandler to nodes with GtMondrian. [Listing 7]

In CRMondrian event handlers can be added to all nodes using Actions. [Listing 8]

```
view := CRMondrian new.
view nodes point with: (1 to:9).
view action randomBackground.
view
```

Listing 7: Random background color action in CRMondrian

```
view := GtMondrian new.
view nodes shape: [:obj | | el |
   el := BlElement new
      geometry: (BlCircle new);
      background: Color blue;
      addEventHandlerOn:
         BlClickEvent
      do: [el background:
                     Color random]
  ]; with: (1 to:9).
view
```

Listing 8: Random background color action in GtMondrian

One important feature of Mondrian is the ability to create nested graphs. One application for this would be to have a graph showing different classes with each node, representing a class, containing a graph with the methods of this specific class.

GtMondrian offers a forEach function. This is how it was imagined by Michael Meyer. [3, p. 21] One creates the nodes and then specifies the graphs within each node. [Listing 10]

In CRMondrian this feature is again implemented using a ShapeBuilder called mondrianNodes. The idea behind it is that in the end it is not really something different if the shape of the node should be a box or another graph. It allows the user to add a stencil according to which the subgraphs should be built. [Listing 9]

```
view := CRMondrian new.
view nodes
   mondrianNodes
   stencil: [:c | | mon |
      mon := CRMondrian new.
      mon nodes with: c methods.
      mon
   ];
   with: (CRMondrian package
                        classes).
view
```

Listing 9: Nested graphs in CRMondrian

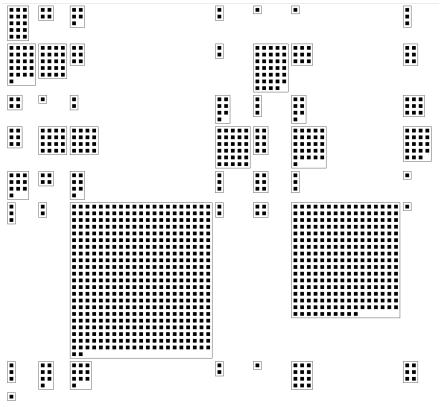

Figure 5.1: Graph created by Listing 9

```
view := GtMondrian new.
view nodes
      shape: [ :x |
         BlElement new
            border: (BlBorder paint: (Color black) width: 2);
            constraintsDo: [ :c |
               c vertical fitContent.
               c horizontal fitContent.
            ];
            margin: (BlInsets all: 20) ];
      with: (CRMondrian package classes)
      forEach: [ :each |
         view nodes
            shape: [ :x | BlElement new
               background: Color black;
               margin: (BlInsets all: 5).
             ];
            with: (each methods).
            view layout grid.
         ].

view
```

Listing 10: Nested graphs in GtMondrian

Additionally to the already implemented ShapeBuilders CRMondrian offers the ability to package a new complicated shape in a new ShapeBuilder. To do this one simply needs to create a new subclass of CRShapeBuilder and implement a createShapeFor: obj method. This allows for straightforward creation of difficult visualizations.

To sum up, the scripts in CRMondrian are a bit cleaner and easier to write without having to know how BlElements work. Essentially GtMondrian offers the same functionality, but it is a bit more complicated to use.

6 Conclusion

Overall Glamorous Toolkit and the concept of moldable development it is based on, could well be the future of programming.

The way visualizations are integrated into the environment is inspiring, but Glamorous Toolkit also needs the tools to easily create these visualizations. Mondrian certainly is very capable of that, but needs to improve its user friendliness.

We have proven that Mondrian scripts can be easier than what Glamorous Toolkit currently offers, but there is still quite a lot to do if CRMondrian should be part of this new moldable future.

The performance of CRMondrian definitely has to be improved to make it more usable. Currently it works fine for simple graphs, but if the graph has a few hundred nodes or many edges it will be noticeably slower than GtMondrian. The reason behind this performance deficit was not examined in this thesis, but it likely has to do with the complicated creation process of the graphs. To solve the problem one would have to build functionality to keep track of how many times the shapes are created, and how often there is an iteration through all nodes and edges. This information could then be used to improve the building process to make it faster. Since Glamorous Toolkit itself is still being developed, it most likely will get faster and therefore make CRMondrian faster as well.

Besides the performance issue there is a never ending list of features that could be desirable. Such as the ability to create different diagrams, being able to search for nodes within a graph and having more layouts.

7 Anleitung zu wissenschaftlichen Arbeiten

This is a detailed explanation on how to use CRMondrian.

7.1 Installation

To install CRMondrian in Glamorous Toolkit you will have to clone it from the following repository on Github: https://github.com/cjrohrbach/BA-GT-Roassal

To do this you can simply execute the following code in the Playground of Glamorous Toolkit:

```
Metacello new
   baseline: 'BAGTRoassal';
   repository: 'github://cjrohrbach/BA-GT-Roassal/src';
   load.
```

Listing 11: Code used to load CRMondrian

7.2 How do I use it?

7.2.1 Nodes

The ShapeBuilder [Listing 12 line 2] is a very central element of CRMondrian since it creates all nodes. Therefore it is the only element that is absolutely necessary for an instance to work.

```
1 view := CRMondrian new.
2 view nodes with: ( 1 to: 10).
3 view
```

Listing 12: Simple CRMondrian script

The basic shape of the nodes can be changed by choosing between the following ShapeBuilders:

- Box: The box is the default shape used for CRMondrian. If no shape is specified a black box will be used. A box offers the ability to customize the background, the border and the size.

```
view nodes box
   size: 20@20;
   background: Color green;
   borderColor: Color black;
   borderWidth: 1;
```

Listing 13: ShapeBuilder box with customization

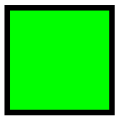

Figure 7.1: Shape created by Listing 13

- Circle: The size, border color and border width of a circle can be customized.

```
view nodes circle
   size: 20@20;
   borderColor: Color blue;
   borderWidth: 1;
```

Listing 14: ShapeBuilder circle with customization

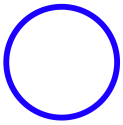

Figure 7.2: Shape created by Listing 14

- Point: A point allows for customization of the size, border and background

```
view nodes point
   size: 20@20;
   background: Color gray;
   borderColor: Color blue;
   borderWidth: 1;
```

Listing 15: ShapeBuilder point with customization

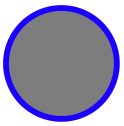

Figure 7.3: Shape created by Listing 15

- Label: Labels are based on BlTextElements. The background color, as well as the text can be customized

```
view nodes label
   text: 'some Text';
   background: Color lightBlue;
```

Listing 16: ShapeBuilder label with customization

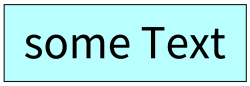

Figure 7.4: Shape created by Listing 16

All ShapeBuilders except for the label have the ability to customize the text shown in the tooltip. You can do this using the following code:

```
view nodes box toolTip: #name;
```

Listing 17: Change the tooltip content

The ShapeBuilders accept customizations in the form of absolute values [line 4], symbols [line 6] or blocks [line 8]. [Listing 18]

```
1  view := CRMondrian new.
2  view nodes
3     box
4     width: 10; "<- Absolute value"
5
6     height: #linesOfCode; "<- Symbol"
7
8     background: [:x | (x hasComment)
9                     ifTrue: [Color green]
10                    ifFalse: [Color red] ]; "<- block clousure"
11    with: (CRMondrian methods).
12 view
```

Listing 18: Example for customizations using absolute value, symbol and block

Besides the above listed builders there are two other more special ones:

- PreBuilt: The pre-built node ShapeBuilder allows the creation of visualizations using already existing nodes. [Listing 19]

  Important: since graphical elements and therefore also CRNode elements cannot be added to more than one parent element, the prebuilt node shapeBuilder will not work for nested graphs, or if the nodes are already added to another element at some point.

```
node := CRNode new
   shape: (CRBox new
         size: 50@100;
         background: Color blue;
         border: (BlBorder paint: (Color gray) width: 5));
      model: 'test Node'.

mondrian := CRMondrian new.
mondrian nodes preBuiltNode addNode: (node).
mondrian
```

Listing 19: Example for CRMondrian using the CRPrebuiltNode ShapeBuilder

- Custom: The custom ShapeBuilder allows the creation of nodes according to a provided stencil. This way it is possible to create any shape without having to create a new ShapeBuilder. [Listing 20]

```
view := CRMondrian new.
view nodes custom
   stencil:  [ :x |
      BlElement new
         size: 20@50;
         background: Color gray.
      ];
      with: (1 to: 9).
view
```

Listing 20: Example for CRMondrian with custom node shape

It is also possible to create a visualization using multiple different node shapes. To do this simply add another ShapeBuilder by calling the CRMondrian>>nodes method on a Mondrian instance. [Listing 21]

```
view := CRMondrian new.
view nodes
   point
   background: (Color gray);
   with: (1 to: 10).

view nodes
   label
   background: (Color white);
   with: (1 to: 10).
view
```

Listing 21: Example for CRMondrian with multiple ShapeBuilders

7.2.2 Edges

The EdgeBuilder is used to create edges between the nodes. A new EdgeBuilder is created by calling the CRMondrian>>edges method on a Mondrian instance. [Listing 22 line 3]

```
1 view := CRMondrian new.
2 view nodes with: (1 to: 9).
3 view edges connectFrom: [:x | x//2].
4 view
```

Listing 22: Simple CRMondrian script with edges

There are four different EdgeBuilders to choose from:

- connectTo: Creates an edge from one node to exactly one other node
- connectToAll: Creates edges from one node to all nodes matching the connection criteria
- connectFrom: Creates an edge to one node from exactly one other node
- connectFromAll: Creates edges to one node from all nodes matching the connection criteria

The EdgeBuilder provides the ability to customize the width and color of the lines as well as to add an arrowhead to the line. [Listing 23]

```
{ ... }
view edges
   arrow;
   color: Color blue;
   width: 3;
   connectFrom: [:x | x//2].
{ ... }
```

Listing 23: Mondrian script with customized edges

Furthermore you can add a filter to the EdgeBuilder to only create edges between nodes that fulfill a certain criterion. [Listing 24]

```
view := CRMondrian new.
view nodes with: (1 to: 20).
view edges forAll: [:x | x%2 = 0]; connectTo: [:x | x*2].
view
```

Listing 24: Mondrian example with edges on a subset of the nodes

7.2.3 Layouts

The layouts are used to arrange the nodes in a certain way. The following layouts are already implemented and ready for use. Since they extend the existing layouts within Glamorous Toolkit they allow for the same customizations the original layouts offer.

- Grid: The grid layout is set as the default. To make it square, the number of columns needed is calculated depending on the number of nodes in the visualization. This way the grid will always have about the same number of columns as rows. If you would like to create a grid without visible spacing between the nodes you will have to set cellSpacing to minus ten. This is due to the node being a bit bigger than its shape to allow for visible highlights.

```
view := CRMondrian new.
view nodes with: (1 to: 9).
view layout grid.
view
```

Listing 25: Script with grid layout

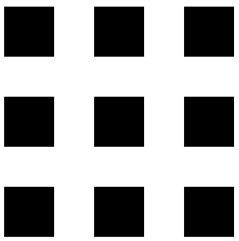

Figure 7.5: Mondrian with grid layout

- Tree: The tree layout is often used to visualize a hierarchy of some sort.

```
view := CRMondrian new.
view nodes with: (1 to: 15).
view edges connectFrom: [:x| x//2].
view layout tree.
view
```

Listing 26: Script with tree layout

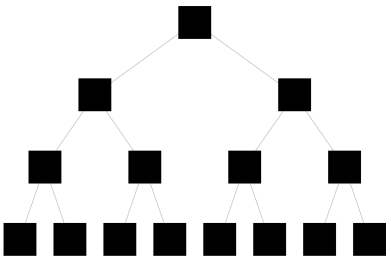

Figure 7.6: Mondrian with tree layout

- Flow: The flow layout extends BlFlowLayout and can be used to arrange the nodes with a flow layout.

```
view := CRMondrian new.
view nodes with: (1 to: 10).
view edges connectFrom: [:x| x//2].
view layout flow.
view
```

Listing 27: Script with flow layout

Figure 7.7: Mondrian with flow layout

- Force: A force layout can be used to create visualizations where the connected nodes repel each other.

```
view := CRMondrian new.
view nodes with: (1 to: 10).
view edges connectFrom: [:x| x//2].
view layout force.
view
```

Listing 28: Script with force layout

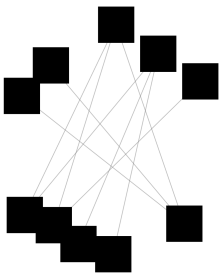

Figure 7.8: Mondrian with force layout

- Circle: The Circle layout arranges the nodes in a circle.

```
view := CRMondrian new.
view nodes with: (1 to: 10).
view layout circle.
view
```

Listing 29: Script with circle layout

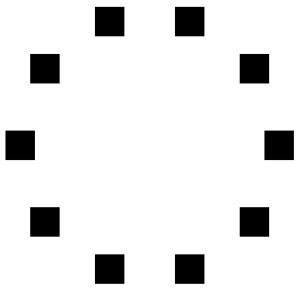

Figure 7.9: Mondrian with circle layout

- Custom: The custom layout can be used to apply any other layout to the graph. The provided layout is then packaged within a BlOnceLayout before being applied. This is done to keep the ability to move the nodes by dragging them.

```
view := CRMondrian new.
view nodes with: (1 to: 10).
view edges connectFrom: [:x| x//2].
view layout custom:
 (GtGraphHorizontalTreeLayout new).
view
```

Listing 30: Script with custom layout

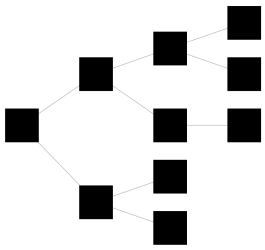

Figure 7.10: Mondrian with horizontal tree layout created by Listing 30

To add a new layout, simply create a subclass of the layout you would like to add and add the TCRLayout Trait to it. This will create a method that is used to apply the layout on a canvas.

7.2.4 Actions

Actions allow you to add an event handler to all nodes within the visualization. This can be used to make the graphs as interactive as possible.

There are 2 predefined actions available:

- PrimeSieve: This action can be applied to a set of nodes representing integers. Initially all nodes will have the same color. If you click on a node that still has the default background color, it will be marked as a prime number and all multiples of this number will be marked as non prime number. In the end you will have a set of numbers with the prime numbers in blue and the non prime numbers in gray.
- RandomBackground: This action is mostly used to demonstrate how actions work. It adds an event handler to all nodes that changes the background of the node to a random color when the user clicks on it.

More important than the predefined actions is the ability to add custom actions consisting of a trigger and a block describing what should happen when the action is triggered. [Listing 31]

```
view := CRMondrian new.
view nodes box with: (1 to: 9).
view action
   custom
   on: BlClickEvent do: [:x | x target background: Color random].
view
```

Listing 31: CRMondrian script to create a graph with custom Actions

7.2.5 Normalizer

For a lot of visualizations it is useful to map a certain object attribute to a node, for example the number of lines of code to the size. But if you map it directly it could happen that some nodes are not displayed at all while others are way too big.

Normalizers solve this problem, by taking an upper and a lower bound and assigning to the nodes a value within this bound according to the object they represent.

```
1 CRNormalizer new
2       key: #linesOfCode;
3       attribute: [:node :value | node size: value];
4       from: 10;
5       to: 50;
6       condition: [:node | node class = CompiledMethod];
7       nodes: someNodes
```

Listing 32: Script used to create a CRNormalizer

A normalizer requires at least the following input:

- From and to values to set the bounds [Listing 32 line 4 and 5]
- The collection of nodes that should be normalized [Listing 32 line 7]
- The object attribute that should be used to determine the value, as key [Listing 32 line 2]
- A block with a node and a value as input to change the shape of the node [Listing 32 line 3]

If the normalizer should only be applied to a subset of the provided nodes it is possible to add a condition, a block that takes a node and returns true if the normalizer should be applied on this node. (See listing 32 line 6)

All basic shapeBuilders, except the label, allow you to normalize

- the size — height and width independently or together
- the color of the border
- the width of the border
- the background color

You can add a normalizer by either creating it yourself and adding it to the ShapeBuilder manually using the method CRShapeBuilder>>addNormalizer or by using the methods provided by the ShapeBuilder [Listing 33], which will create the normalizer automatically.

```
view := CRMondrian new.
view nodes
   box
      normalizeHeight: #numberOfMethods;
      normalizeWidth: [:x | x slots size];
      normalizeBackground: #linesOfCode;
      with: Collection withAllSubclasses.
view
```

Listing 33: CRMondrian script with normalized nodes

7.2.6 System Complexity

Using all these different normalizers it is pretty easy to create a System Complexity view: a view where the nodes are classes with edges to their superclass.

The height of the nodes is normalized using the number of methods within the class, the width uses the number of instance variables as key, and the background color is darker the more lines of code a class has.

```
"Setup the collection of classes used for the graph:"
col := CRMondrian package classes.
col remove: BlElement.
col remove: CompiledMethod.

view := CRMondrian new.
view nodes
   box
     normalizeHeight: #numberOfMethods
         from: 5 to: 25;
     normalizeWidth: [:x | x slots size]
         from: 5 to: 25;
     normalizeBackground: #linesOfCode
         from: Color lightGray to: Color black;
     with: col.
view edges connectFrom: #superclass.
view layout tree.
view
```

Listing 34: Script used to create System Complexity view

Figure 7.11: System Complexity view as it is created by Listing 34

7.2.7 Nested Graphs

Nested graphs can be created using the CRMondrianAsNode ShapeBuilder. It allows you to create a stencil according to which the inner graphs should be built. [Listing 35]

```
view := CRMondrian new.
view nodes
   mondrianNodes
   toolTip: #name;
   stencil:
   [ :p | | m |
      m := CRMondrian new .
      m nodes
         box
         toolTip: #name;
         with: (p methods ).
      m];
   with: ( (RPackageOrganizer default packageNamed: 'CR-Graph')
                                                        classes ).
view
```

Listing 35: Example for nested graphs

7.2.8 Class Blueprint

Using nested nodes it is possible to create a complete class blueprint modeled after what was proposed by Stéphane Ducasse and Michele Lanza in their paper “The class blueprint: Visually supporting the understanding of classes”. [2]

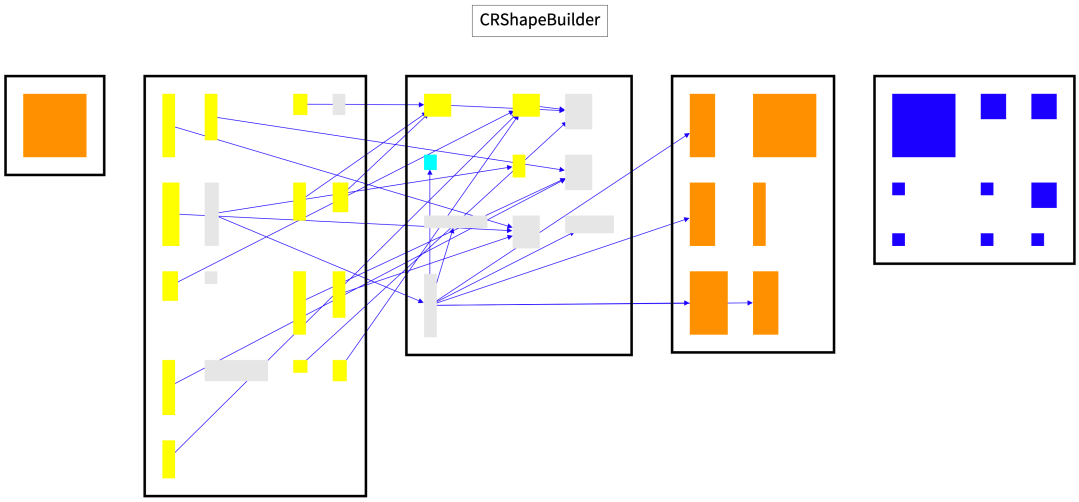

Figure 7.12: Class Blueprint for a single class

The Class Blueprint consists of five containers, four of them being used to sort methods into the following categories:

- Initialization: Methods used to create and initialize a new object.
- Interface: Methods used for interaction from outside the class. These methods are not invoked by other methods of the class, except the once used for initialization.
- Implementor: Private methods used by other methods within the class.
- Accessor: Getter and Setter methods.

The fifth container is filled with the instance variables.

The node’s size is determined by:

- The size of the method nodes is normalized using the number of lines of code for the height and the number of invocations for the width.
- The size of the instance variable nodes is normalized using the number of accesses.

The nodes are colored depending on their method type:

- Blue: Instance variable nodes
- Cyan: Abstract methods (methods with subclass responsibility)
- Orange: Extending methods with a super invocation
- Brown: Overriding methods with no super invocation
- Yellow: Delegating method
- Grey: Methods returning a constant value

If none of the above colors are applied the node is colored depending on the category it is in:

- White: Methods within the interface or implementor category
- Red: Getter methods
- Orange: Setter methods

To represent the invocations blue edges are added from a method node to all nodes representing methods and variables called within this method.

If there is more than one class provided, the class hierarchy is represented by grey edges and the Class Blueprints are arranged with a tree layout.

To create a Class Blueprint for a set of classes you can simply create a new Mondrian instance and call the CRMondrian>>createClassBlueprint method with the set of classes you would like to create a Class Blueprint for.

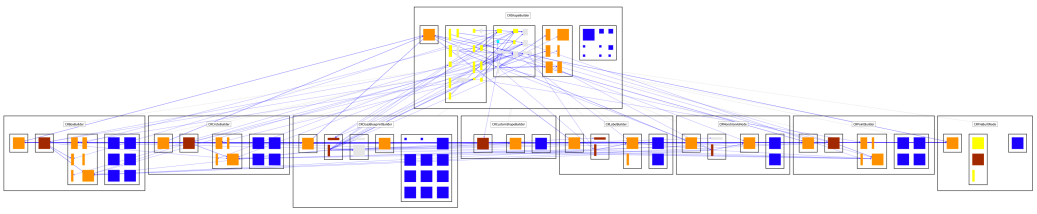

Figure 7.13: Class Blueprint example

7.2.9 Mind Map

Mindmaps can be a way to visualize connections between different objects.

To create a mindmap in CRMondrian you will first need to create the mindmap structure. A mindmap-structure is a set of nodes connected to each other with a parent child relationship, where every node, except the root, has exactly one parent.

```
root := MindMapNode new
   model: CRMondrian;
   addChildren: {
      MindMapNode new
         model: CRCanvas;
         addChildren: {}.
      MindMapNode new
         model: CRNode;
         addChildren: {
            MindMapNode new
               model: 'Node-Shape';
               addChildrenFromModels: {
                  CRBox.
                  CRCircle}.
            MindMapNode new
               model: CRShapeBuilderBuilder;
               addChildren: {
                  MindMapNode new
                     model: CRShapeBuilder;
                     addChildrenFromModels: {
                        CRBoxBuilder.
                        CRCircleBuilder. }
               }
         }.
   };
   isRoot.

s := MindMapStructure new
setRootNode: root.
```

Listing 36: Script to create a Mindmap Structure

This structure can then be visualized using CRMondrian>>createMindMap. This will create a tree where a node is connected to all its children.

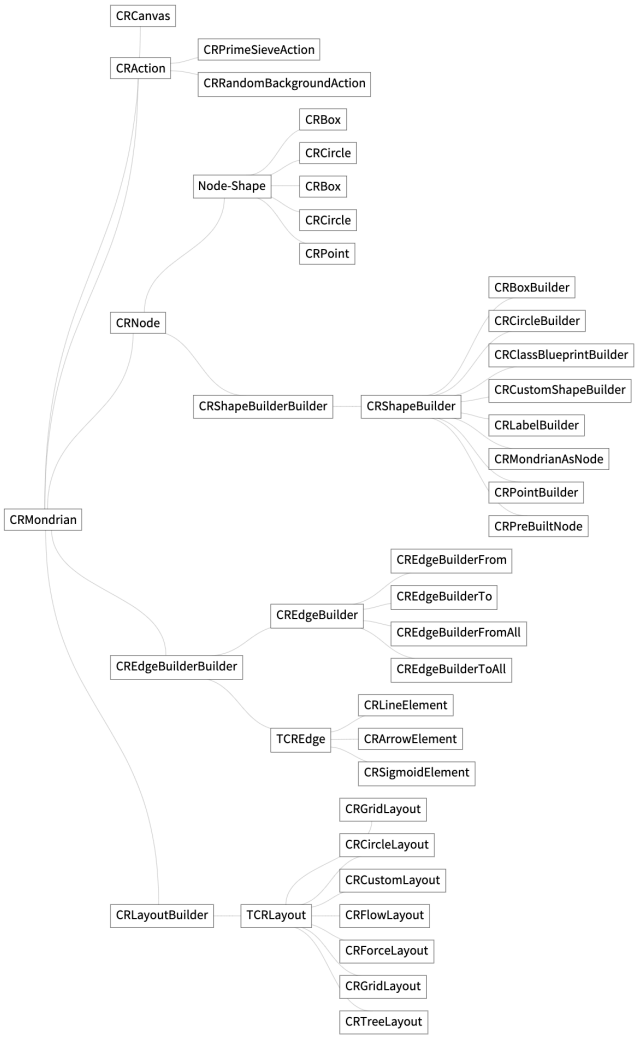

Figure 7.14: Screenshot of a Mindmap representing parts of CRMondrian

Bibliography

[1] Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. Exemplifying moldable development. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop*, PX/16, page 3342, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Stphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *Software Engineering, IEEE Transactions on*, 31:75 – 90, 02 2005.

[3] Michael Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, 2006.

[4] Hossein Tajalli and Nenad Medvidovi. A reference architecture for integrated development and run-time environments. In *2012 Second International Workshop on Developing Tools as Plug-Ins (TOPI)*, pages 19–24, 2012.