



^b
**UNIVERSITÄT
BERN**

Visualising Objects in Pharo

Bachelor Thesis

Eve Mendoza Quiros

from

Zürich ZH

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

June 19, 2018

Prof. Dr. Oscar Nierstrasz

Claudio Corrodi

Software Composition Group

Institut für Informatik

University of Bern, Switzerland

Abstract

Object inspection in the Pharo IDE¹ is currently focused on the individual object. The inspection of inter-object relationships is possible in a very limited way, making object set inspection difficult.

Understanding the relationship between objects and sets of objects is an important debugging aid and facilitates proper code analysis. In order to efficiently understand code, a visualization of data structures in an interactive graph helps programmers get a thorough conceptual overview. This can save time during debugging as well as code analysis and maintenance.

In this thesis a tool is presented that facilitates the visualization of object sets in a graph, in Pharo. The tool highlights the relationships between objects while also conveying important information about each individual object. The strengths of this framework are, first subgraphs persist over different graph renderings, making the comparison of similar sets easy and effectively presenting the set evolution. Second the interactive graph and ability to customize the visualization makes it more understandable and useful to the user.

By using this tool in Pharo interesting visualizations can be created since Pharo's mantra is *everything is an object*, therefore we can also make graphs containing classes as elements and show the relationships between different classes. The tool facilitates node customization, giving the user the possibility to mold the visualization to fit their needs. For each object an individual node representation can be created. In this thesis we present a node customization for linked lists and for abstract syntax trees. Overall the tool is very intuitive and supports program understanding and debugging.

¹<https://www.pharo.org>

Contents

1	Introduction	1
2	Related Work	4
3	Motivation	7
4	Visualization of Objects	12
4.1	Interactive graph interface	14
4.1.1	Camera Movement	15
4.1.2	Dragging	15
4.1.3	Popups	15
4.1.4	Minimize and Maximize Node	15
4.1.5	Minimized Information	16
4.1.6	Highlights	16
4.1.7	Removing Objects from the Visualization	16
4.1.8	Directly Adding Objects through the View	16
4.2	Persistent Subgraphs	17
4.3	Custom Node Visualization	20
4.3.1	Linked List	22
4.3.2	RB Program Node	24
5	Implementation	27
5.1	Design Patterns	27
5.2	Persistent Subgraphs	30
5.3	Custom Node Creation	30
6	Validation and Use Cases	32
6.1	General Object Set Inspection	32
6.2	Object Set Comparison	34
7	Conclusion and Future Work	43

A	Anleitung zu wissenschaftlichen Arbeiten	47
A.1	Installation of environment and tools	47
A.2	Basic usage	48
A.3	Implementing a custom node shape	51
A.3.1	Using pragmas	52
A.3.2	Heap customization	54
A.3.3	Sorted Heap label	55
A.3.4	Label with buttons to add/remove objects of the heap	55
A.3.5	Heap preview	57
A.3.6	Customizing CalendarMorph using pragmas	57

1

Introduction

Programmers spend a lot of time reading code to understand how a program works. This is a very tedious and time consuming task. Inspecting live objects and data structures used during run-time is a more feasible approach to understanding program structure and functionality. This examination and the modification of the run-time system is called structural reflection [10, p. 304]. In the Pharo IDE¹ there are various tools which support structural reflection, such as the inspector, the debugger or the system browser. These tools allow the inspection of individual run-time objects, but only offer limited capabilities to inspect multiple objects. The possibilities to inspect inter-object relationships are also very limited and the relationships are often presented as lists or tree-structures. This manner of inter-object relationship representation can quickly become complex and unclear with an increasing number of objects. Reducing these aspects helps programmers to use their time more efficiently. One way to make live object inspection more efficient would be to enable customized object representations. By having a customized object representation or so called view, the user can display information most relevant to them.

The Moldable Tools [4] presented by Andrei Chis offer an approach to create customized views for individual objects of a class. It is proposed that for individual objects there can be multiple views so that the process of finding the right view is optimized [4, p. 54]. This approach addresses one

¹<https://www.pharo.org>

limitation of inspecting run-time objects but does not take into consideration the limited inspection of multiple run-time objects.

To deal with that limitation, we propose an approach that uses graph visualizations to present objects sets. The justification for using visualizations is the following. Programmers create a model of their software in their mind, where objects have a manifestation, as stated in *Using visualizations to Foster Object-Oriented Program Understanding* [7]. Therefore the idea of visualizing object-oriented software follows naturally. As with all visualizations, their effectiveness comes from the amount of useful information the user is provided with. Programmers have different approaches to analyzing and debugging code, so a visualization tool should also be able to provide different visualizations to fit the programmer's needs.

The way programmers approach program optimization and understanding indicates that the user might not know in advance what exactly they are searching for. Therefore most visualization tools fall short, because they only offer a finite set of visualizations. The option of generating new visualizations in a easy manner is valuable to the user [9]. This implies that our solution of visualizing objects sets should be easily customizable and interactive, in order to generate a useful tool to fit the user's needs.

In this thesis we propose a visualization tool, that offers a graphical representation of object sets in the Pharo IDE². Our tool uses Roassal³ as a visualization engine, which provides the user with different visualization options to represent information valuable to the user. In our tool, object sets are visualized as graph structures, where the nodes represent the objects and the edges represent the inter-object relationships. The relationships between these objects, are either by variable reference or due to equality. By inspecting sets of objects, rather than individual objects, we offer an additional tool that can help programmers understand object relationships and recognize data structures. Instead of showing these relationships as trees or lists, which is commonly done in IDEs, we visualize them as a graph, which helps users conceptualize a program's structure. Since object-oriented languages are based on principles such as inheritance and polymorphism, there are various inter-object relationships and object aspects that can be visualized. In our tool we focus on visualizing variable references and visualizing smaller sets of objects rather than complete systems. We do this in order to provide a concise visualization of the individual object, while also presenting the incoming and outgoing inter-object relationships. This, on the one hand, limits the scope of the tool but also allows for a more detailed overview of

²<https://www.pharo.org>

³<http://agilevisualization.com>

the run-time objects that are visualized.

To address the previously mentioned customization, the tool enables the user to create node customizations for different object classes. This allows our tool to stand apart from existing visualization tools like *Heapviz* [1] for example. It allows the user to display individual content or interaction for different objects, providing the user with their desired information. This way the user has more freedom to generate new visualizations, tailored to their needs and thereby making program understanding more efficient. In the tool there are two exemplary node customizations for the class `LinkedList` and the class `RBProgramNode`, which is a class that represents an abstract syntax tree node. These classes serve as examples and can be used as inspiration for further node customizations.

Another feature that makes this tool more powerful, is that it implements subgraphs that persist across renderings. The user can interact with and navigate the visualization more intuitively. It allows the user to perform interactions such as zooming in on a specific area of the visualization and then adding new objects to the visualization. When new objects are added and the visualization is rendered again, the visualization is still zoomed in on the same area. This way the users do not have to reorientate themselves in the visualization, but can continue their program exploration seamlessly.

We provide the user with a basic visualization tool to inspect inter-object relationships and envision the structure of object sets. This should ease code understanding and maintenance, by offering a structural overview of the run-time objects while also providing detailed information about individual objects.

The tool serves as a useful visualization tool, providing intuitive visualization interactions and acting as a foundation to create object set visualizations. It can be used to mold visualizations to fit the user needs and to enhance structural inter-object relationship understanding. The tool can be used as an additional inspection tool along with the ones already provided by the Pharo IDE⁴.

⁴<https://www.pharo.org>

2

Related Work

In this chapter, we give an overview of graph visualization frameworks. In addition, we present related work on how visualizations help programmers understand object-oriented programs.

To visualize different software dependencies as a graph, Alexandre Bergel et al. [3] propose a domain-specific language that associates colors and size to software metrics. In their approach *GRAPH*, the authors take packages, classes, and methods as input and visualize different relationships between them. Their DSL also visualizes different software metrics of the input, such as the number of methods of a class. This has the advantage that it allows users to visualize multiple dependencies in an understandable manner, when more than one relationship at a time has to be represented. As in our work, they also use Roassal as the visualization engine, because it provides over 200 classes to implement shapes and layouts. Our tool does not focus on software dependencies of a larger system, but rather focuses on smaller object sets and highlights the inspection of the inter-object relationships and the individual objects.

Edward E. Aftandilian et al. present a tool that allows the visualization of a heap obtained from a running Java Program [1]. This gives the user a global overview of the program's state. Their tool provides various interactive capabilities for navigating the heap graph. Heapviz serves the purpose of making modern software understandable, by representing it in a graph, reducing its complexity and thereby making it easier to read. In our work, we also inspect the actual contents of our given object sets, so we have a

dynamic set analysis. In contrast to Heapviz, our focus is on analyzing the relationships between a few objects in detail, rather than a complete system.

A similar approach is presented by Thomas Zimmermann et al. [11]. In their work, they present memory graphs, a visual representation of a program's state. "*A memory graph gives a comprehensive view of all data structures of a program*" [11], helping program comprehension and debugging. Edges are drawn between value references, to show inter-object relationships, answering questions such as whether there are multiple pointers to the same object. Similarly, our tool is also used to inspect data structures and their inter-object relationships. We can also answer whether an object is referenced multiple times within a set of objects. Different than with the memory graphs obtained in that work, the user of our tool chooses the input which is to be visualized. Our tool can be used to only visualize certain parts of a program.

The paper James et al. [5] focus on supporting the understanding of data structures, by using a structure identifier to automatically find them in code. After the identification a visualization is rendered, which is based on the user's code. When the user steps through their program, the nodes appear in the visualization, meaning the visualization follows the sequence of the user's code. It is mentioned in their work that the degree of interaction with a visualization is vital for its effectiveness. Similar to this work, we also have multiple visualization interactions, thereby enhancing the effectiveness of the visualizations created with our tool. For our tool we implemented a node customization of a linked list data structure, as an example that our tool can also aid the understanding of data structures. But it is not the main focus of our tool. Our visualization tool does not categorize objects but it does integrate node customizations for classes with defined node customizations.

In *Moldable Tools* by Andrei Chiş [4], approaches to moldable inspection and debugging tools in Pharo are presented. During interviews with software developers he found that there is a need for object inspectors that support different high-level mechanisms to explore objects. The moldable object inspector that is proposed there focuses mainly on single objects. Our tool can be seen as an additional way to inspect objects, to explore not only single objects but object sets. The work also mentions the desire to create custom views for objects. This is also the inspiration for the possibility to create node customizations in our view. Following some basic needs found during exploratory investigation for *Moldable Tools*, our tool allows the developer to shape the visualization to fit their own contextual needs. The visualization created with our tool could even be used as an additional view of objects that could be integrated into the moldable object inspector.

Jerding and Stasko [7], use visualizations to support object-oriented program understanding. They identify ways in which visualizations can increase

program understanding and present a prototype visualization. Works such as this one validate our use of visualizations to aid structural program understanding. They classify different types of objects, such as classes, functions, and instances to encode relationships. In the case of Pharo, every entity is an instance of a class, which allows us to treat things like instances, methods, and classes in the same way.

3

Motivation

In this chapter, we look at how object sets can be inspected in Pharo and point out the IDE’s current limitations regarding object set inspection. For this work, we use Pharo because it is a modern live coding environment and IDE that allows fast prototyping. Furthermore, we can use Roassal and build on the *Moldable Tools* [4] approach. Also the previously mentioned concept “*everything is an object*” [10] makes for interesting object set inspections, since sets can contain different objects, such as classes, methods, and instances of classes.

As stated in Pharo by Example 5, the Pharo environment provides the programmer with different tools to inspect and debug code, which are different from many other programming environments [10, p. 128]. We focus on two main tools for the purpose of this paper, the inspector and the playground. Commonly the playground is used to run snippets of code or running given examples in classes. When the code is executed, an inspector opens up within the playground, allowing us to inspect the live object from the code, as seen in Figure 3.1.

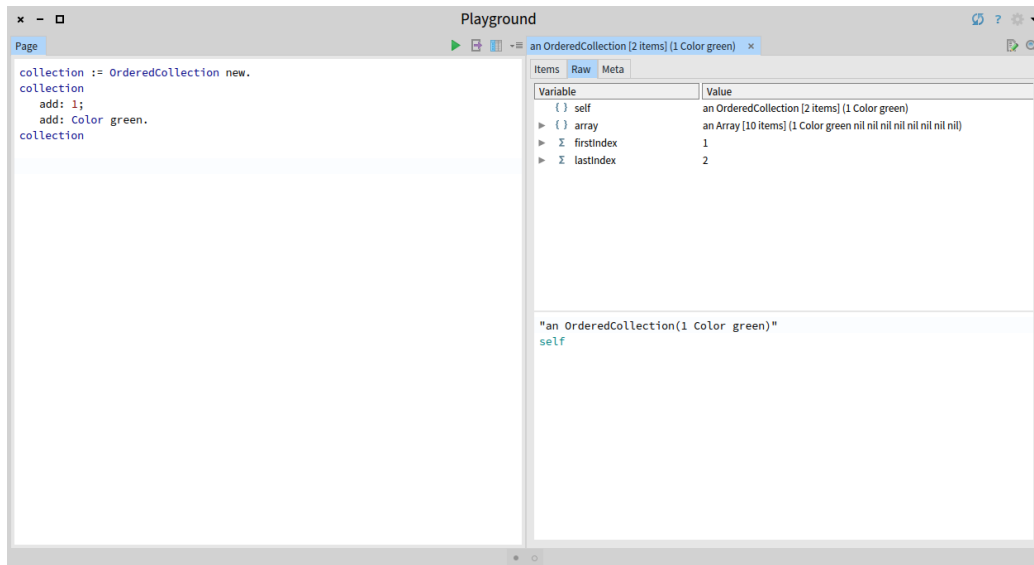


Figure 3.1: A playground containing an `OrderedCollection`, which after executing the code also displays an inspector on the `OrderedCollection`

The inspector provides information such as an object's variables and their values. If we want to inspect an object that is related to the currently inspected one (for example, a variable value), we can click that object in the inspector and a second inspector pane will be opened. This process can then be repeated leading to a multitude of inspector panes within the playground, as illustrated for a linked list in Figure 3.2.

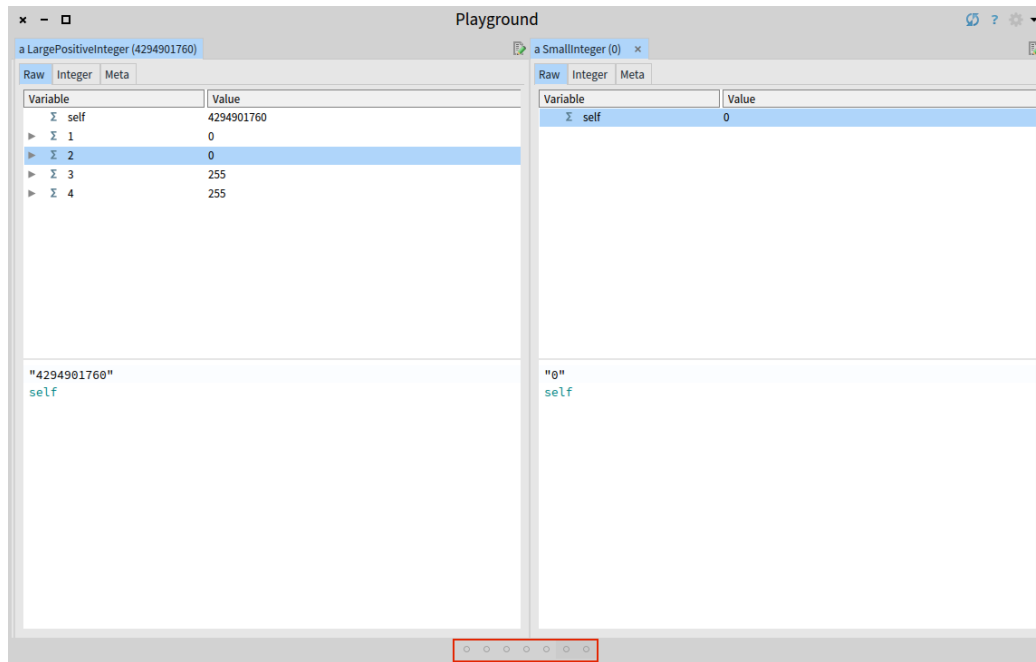


Figure 3.2: A playground with 7 inspector panes of which two can be seen, the first pane containing a linked list

This approach might give us a detailed overview for each object, but quickly leads to a time consuming code inspection. The user might lose track of all the objects and navigating through the different inspection windows can be tedious and time consuming. The relationships between the objects are not clearly shown and the user cannot quickly see whether there are multiple relationships to one specific object. The user gets a limited overview of inter-object relationships.

Another approach to inspecting objects is to navigate the instance variable values using the tree view in the *Raw* tab in an object's inspector, as shown in Figure 3.3.

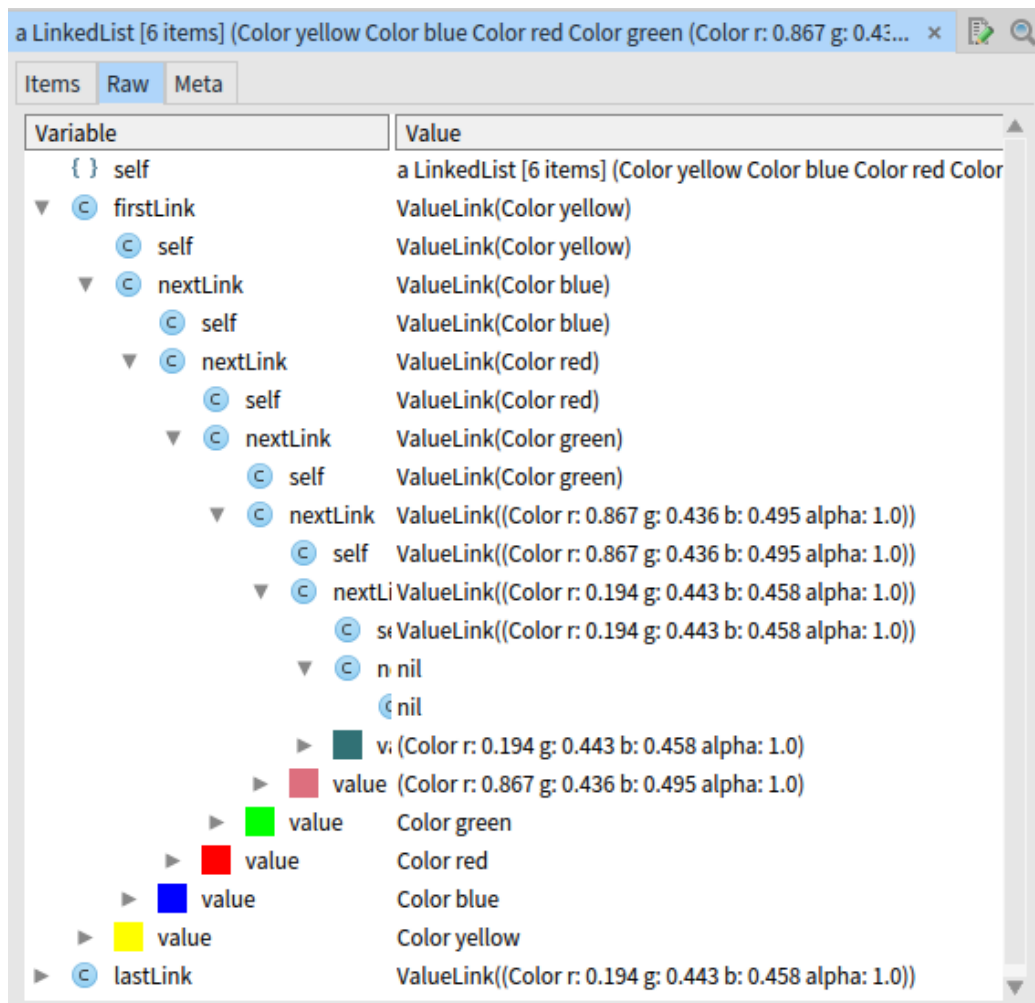


Figure 3.3: An inspector on a `LinkedList`, showing additional information about the `LinkedList`, displayed as a tree

With a large number of objects or instance variables, this again quickly leads to a complex and incomprehensible information load. The more a user goes into the object's depth, the harder it is to keep an overview of the coarse object structure, which in Figure 3.3 would be the `LinkedList`.

Both these ways to inspect objects mainly focus on the inspection of a single object. The user should be able to get information about multiple objects, while also having a simple overview of the individual objects and the relationships between the objects. To provide the user with exactly that, we created a tool which gives a graphical representation of a set of objects and the relationships between the objects. The tool should enhance inter-object

relationship inspection and reduce its complexity, which currently arises when we try to inspect these inter-object relationships. The tool in no way replaces the inspector but rather serves as a additional tool that provides an overview and highlights relationships between objects.

These are the limitations regarding object set inspection in Pharo, which we will focus on in this work. Our tool aims to improve the current situation by providing a different manner of object inspection, more precisely object set inspection, and by making the graph interaction more intuitive.

By using a graphical representation of a set of objects, the user can more efficiently comprehend different data structures, such as linked lists, heaps and ordered collections. Within our graph, the user can also interact with the objects and if desired still inspect them in the inspector by clicking on them. As the user cannot remove variables from the view in the inspector, this is a limitation since certain objects have many variables, making their inspection harder.

4

Visualization of Objects

In this chapter, we present our tool and its functionalities, explaining how the functionalities provide an approach to overcome the limitations in object set inspection in Pharo. Also, it will be explained how the user can mold the visualization through node customization.

Our tool enables the users to inspect sets of objects and the relationships between them. The complete object set is visualized as a structured graph. Each object in the set is represented as a node in the visualization. The basic node in our visualization displays the object name and class, its (inherited) variables and the values of the variables. In Figure 4.1 we see the node for the object `Color blue`. `Color blue` is the object name, `Color` is the object's class and `alpha`, `cachedBitPattern`, `cachedDepth`, `rgb` are its instance variables and along with those we have their values. In addition we have various icons, which are buttons, for node interactions, which will be explained in Section 4.1

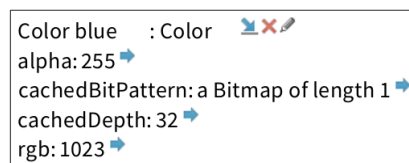
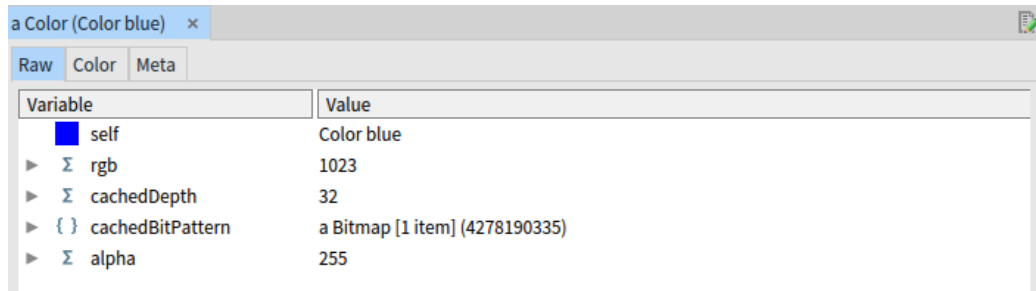


Figure 4.1: The object `Color blue` represented as a node in our visualization tool

This is the same information we get in the raw tab of the inspector as seen in Figure 4.2.



The screenshot shows the Pharo inspector window with the title 'a Color (Color blue)'. It has three tabs: 'Raw', 'Color', and 'Meta'. The 'Raw' tab is selected, displaying a table of instance variables and their values.

Variable	Value
self	Color blue
rgb	1023
cachedDepth	32
cachedBitPattern	a Bitmap [1 item] (4278190335)
alpha	255

Figure 4.2: The object `Color blue` shown in the Pharo inspector

The information displayed about the instance variables and their values is important to us, because it is also the basis for inter-object relationships in our graphs.

There are two different situations in which edges are drawn between two objects in our visualization. The first situation is if there is a variable reference between two objects, that is, if one of object A's variables has the value object B, then we have an edge from object A to object B. The edges are labeled with the variable which connects the two elements. In Figure 4.3, we can see a `LinkedList`, a `ValueLink` and a `SmallInteger`. From the `LinkedList` to the `ValueLink` we have a variable reference edge labeled `firstLink`. Another variable reference is illustrated from the `ValueLink` to the `SmallInteger`.

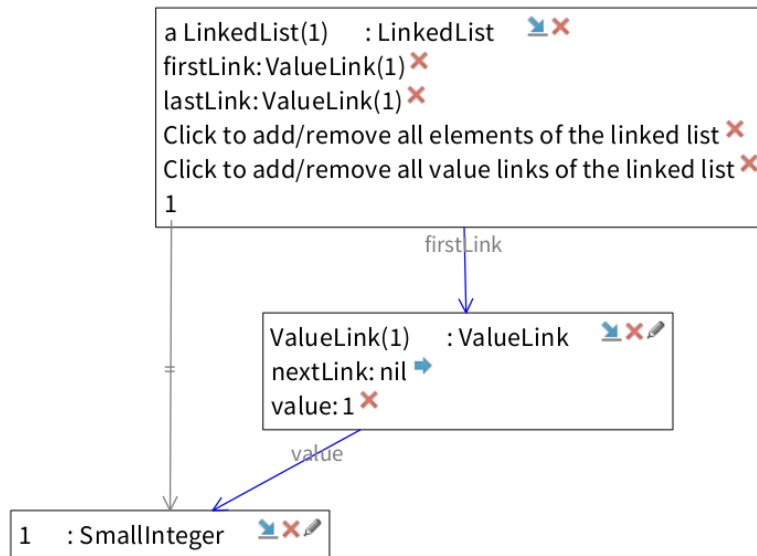


Figure 4.3: A visualization containing a `LinkedList`, a `ValueLink` and a `SmallInteger`. Both types of edges can be seen between the objects.

The second situation is if we have an equality connection between two elements. In Figure 4.3, the element preview in the `LinkedList` node (final line of the node text) is the same object as the integer node. Therefore, an equality edge is drawn between the two elements.

Once we have visually represented the objects and their relationships, the nodes are laid out, so that we have a graph showing our object set. We have chosen a basic `RTSugiyamaLayout`, which is provided by Roassal¹. The chosen layout assigns nodes to hierarchical layers, therefore nicely representing hierarchical structures and minimizing edge crossing. This is useful to us since we can use our tool on data structures and object sets with clear hierarchies and display the objects in an structured manner.

4.1 Interactive graph interface

To provide the user of our tool with intuitive navigation, we offer a variation of interactions. The first three interactions are provided by the Roassal¹ framework and the others have been added to enhance our tool and provide efficient object set inspection. The interactions listed here are general for all the visualizations created with our tool. Some additional interactions are specific to customized nodes; those will be presented in Section 4.3.

¹<http://agilevisualization.com>

4.1.1 Camera Movement

After the initial rendering, the camera is centered, so that the whole object set graph can be seen. The user can navigate the camera around the graph and zoom in and out of the visualization, to inspect individual objects in more detail, in case they are not readable when the complete graph is displayed. Once the user is done looking at specific graph sections, they can easily recenter the camera to the initial object set graph presentation. This is a standard feature of Roassal.



4.1.2 Dragging

If the full graph does not fit into the view, the user can drag around the complete graph and inspect different graph sections. Individual nodes can also be moved around the visualization freely. Users can rearrange nodes to design a layout that fits their needs. The rearranged nodes persist throughout renderings, so that even after adding new objects to the view, the previously moved nodes still remain in the same location. This is a standard feature of Roassal.

4.1.3 Popups

The nodes contain popups, which relay additional information. Some popups explain the interaction that can be had with the specific button or label. The popups serve as aids for comprehensive tool navigation. This is a standard feature of Roassal.

4.1.4 Minimize and Maximize Node

Each node in the graph contains the object name, its class and a toolbar with buttons, as a header, followed by a list of instance variables and their values. For objects with a multitude of variables, the nodes in the visualization are rather large. Whenever that information is not needed, the user can click the minimize button (). The minimized node only displays the object title and the button toolbar, making the graphs less complex when there are a lot of objects. To show all the node information again, the user can maximize the node by clicking the maximize button (). When there are more than 20 objects in the visualization, all the nodes are automatically minimized.

4.1.5 Minimized Information

To manage the node sizes, strings that are longer than 70 characters are automatically cut and dots (.....) are added at the end. If users desire to view the complete string, they can click the dots and then the whole string will be displayed.

4.1.6 Highlights

When the user hovers over a node in the visualization, that node and all the nodes it is connected to are highlighted. The highlighting helps the user to get a quick overview of connected substructures. This is especially useful when we have highly connected elements or have a complex visualization.

4.1.7 Removing Objects from the Visualization

To make graph readability simple, nodes can be deleted directly from within the view. Each node has a remove button (✖) which deletes that node from the view. If there is a remove button next to the instance variables it indicates that the variable's value can be found as a node in the visualization. By clicking the remove button the instance variable value's node is removed from the visualization. In Figure 4.3, we see an example where next to the instance variables `firstLink` and `lastLink` we have a remove button, because the `ValueLink(1)` is an object in the visualization. Once the object is removed from the visualization it cannot be added back directly in the visualization window, with exceptions described below. To add the object again, one needs to do so via playground code.

4.1.8 Directly Adding Objects through the View

To allow intuitive inter-object relationship inspection, instance variable values can be added directly to the visualization, in the visualization window itself. The user can click the arrow button (➡), thereby directly adding the variable value object to the visualization. As with the remove button, the arrow button only appears if the variable value object is not yet in the visualization. Adding objects through the visualization makes for an interesting object inspection. Instead of having multiple windows open in the inspector, we have all the information in one graph. The arrow button next to the variables allows the user to add a single object with a click. With the node customizations in Section 4.3 the user will be able to add multiple objects with one click.

4.2 Persistent Subgraphs

To enhance graph readability, our tool implements subgraph persistency throughout renderings. We are not aware of any object graph visualizations currently in Pharo that persist subgraphs throughout renderings. Graph changes such as adding or removing an object cause the graph to be rendered again, thereby producing a new layout. This might be useful for balanced graph layouts, but to study the graph and object set evolution, it is rather hindering. When studying a graph's evolution, the user should be able to compare the differences between graphs. When subgraphs persist the differences can be spotted easily. Spotting changes such as node removals or additions is very time consuming, if each time a change in the graph occurs a new layout is rendered.

Persisting subgraphs not only saves time when searching for differences in graphs, but also during general graph inspection. If we zoom in on a certain section of our graph and a change to the graph is made, causing it to render afresh, a new layout would cause our zooming in to be undone. The whole graph would be displayed again and the position of the nodes would be changed. It would be more intuitive to remain zoomed in on the same graph section, so that we could continue our object inspection from that position. It would cost the user valuable time to re-orientate themselves within the view.

Implementing persisting subgraphs is important to our tool, because it aids the object set inspection even throughout changes within the object set. Other visualization tools such as *Mondrian* [9] do not implement persisting subgraphs, since the user has to decide on the objects to be presented in the visualization, before the visualization is rendered. If only certain parts of an object set should be displayed, the altered object set needs to be rendered and a completely new visualization is presented. We want to allow our users to be agile and decide which aspects of the view they want to focus on after the initial visualization has been created. The common objects of the initial visualization and the new visualization remain in their position, and the graph memorizes its previous layout.

We will now present an example of persisting subgraphs, showing how they can be used to inspect graph evolution.

We compare the methods of a class at two different points in time. To do this, we collect all methods of the class `OSVAddCustomizationButtonNode` in a `LinkedList`, which is our object set at time A. The object set is then added to the visualization and rendered, so that we obtain the visualization in Figure 4.4.

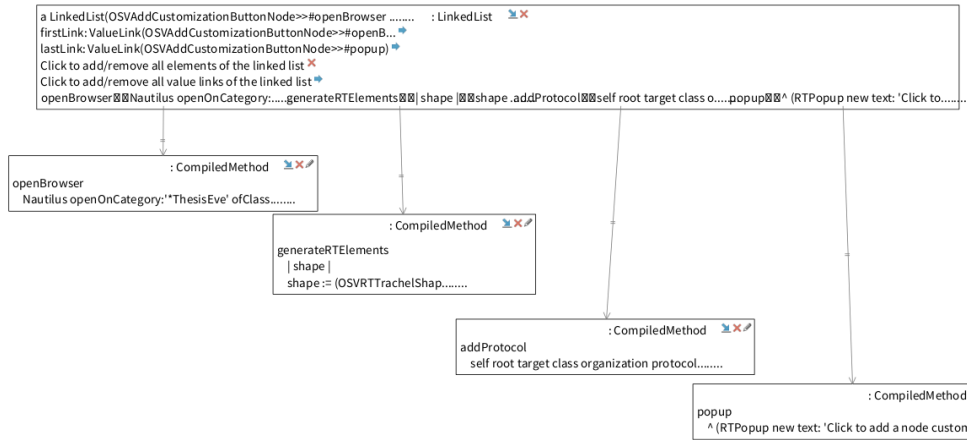


Figure 4.4: The methods of the class `OSVAddCustomizationButtonNode` and a `LinkedList` containing those methods at time A

We can see the class `OSVAddCustomizationButtonNode` has four methods at time A. We have moved around the nodes representing the methods to demonstrate that they will persist their position throughout renderings. In a second `LinkedList` we collect the methods of the class `OSVAddCustomizationButtonNode` at time B. We will now also add this `LinkedList` to the object set and render the visualization anew.

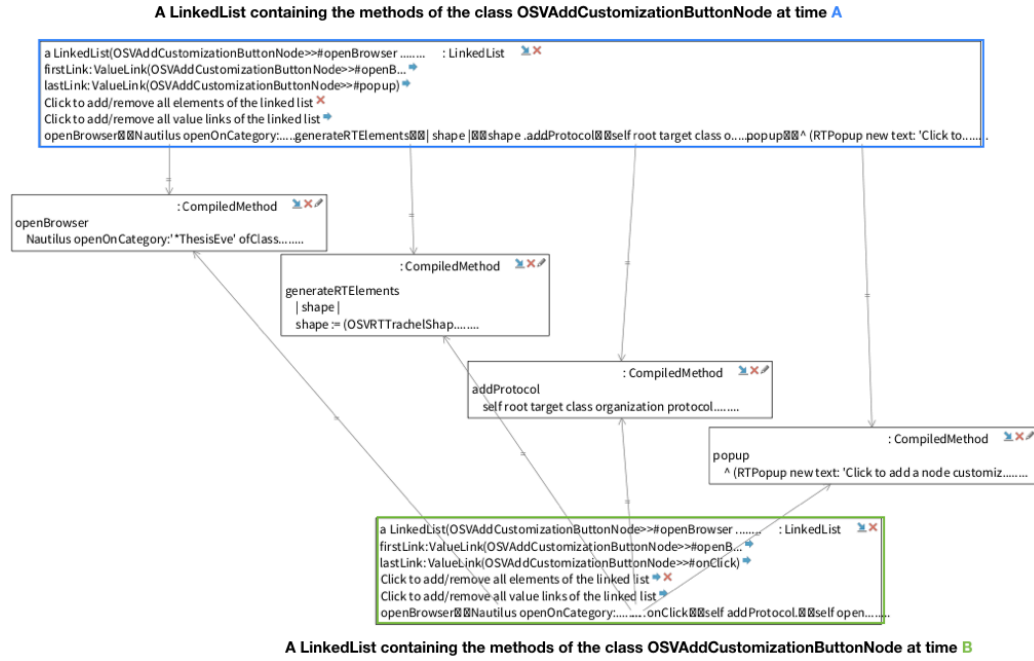


Figure 4.5: The methods of the class `OSVAddCustomizationButtonNode` at time A and two `LinkedList` containing those methods, at time A and time B. Blue and green node outlines and descriptions were added for emphasis and are not part of the original graph.

Right away we can see in Figure 4.5 that the nodes that were previously in the visualization have maintained their layout. The newly added object does not disturb the layout of the elements previously in the view, making graph readability more efficient. The class `OSVAddCustomizationButtonNode` at time B contains all the methods from time A. The arrow button next to the label “Click to add/remove all elements of the linked list” indicates there are more methods in the `LinkedList` at time B. By clicking the arrow button we add those methods to the visualization. This can be seen in Figure 4.6.

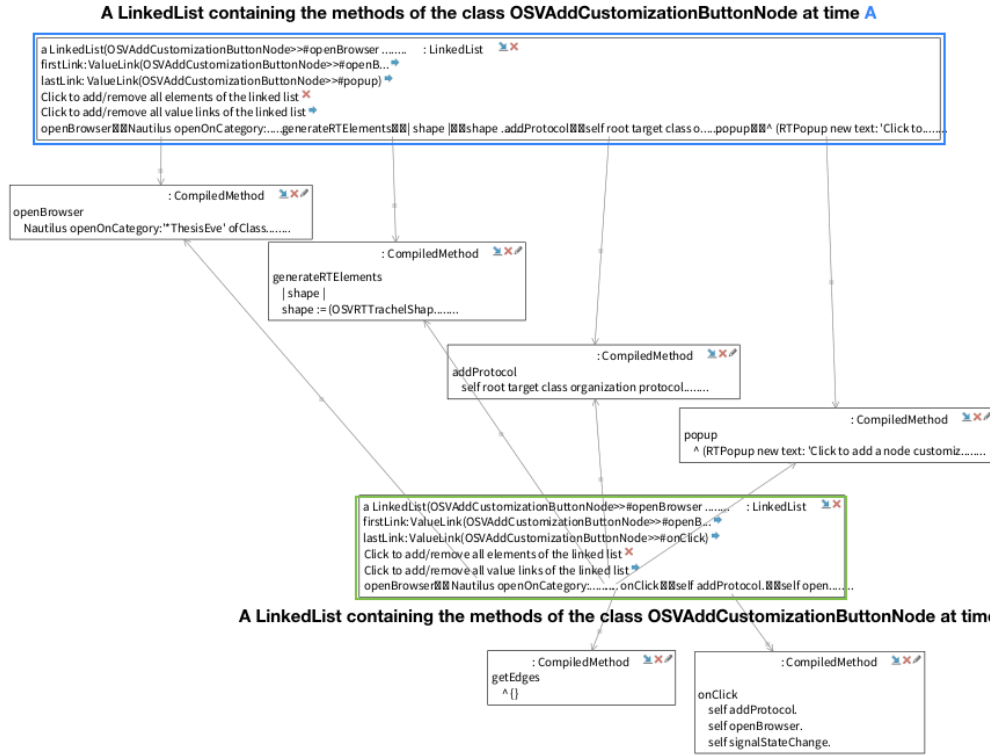


Figure 4.6: The methods of the class `OSVAddCustomizationButtonNode` at time A and B, and two `LinkedList` containing those methods, at time A and time B

We can directly read out of the graph that the two methods `getEdges` and `onClick` were added to the class `OSVAddCustomizationButtonNode` in the time span between time A and time B.

The program's evolution is visualized and the changes can be read out of the graph. Graph readability is an important factor to relay relevant information and make information displaying more efficient. Additionally the camera remaining in the same position makes the interaction with the visualization more intuitive.

4.3 Custom Node Visualization

Node shapes, sizes and colors can enrich a graph with additional information about the visualized objects. As described in "Agile Visualization" [2] by

Alexandre Bergel, the Roassal² framework allows users to create shapes based on the visualized object's metrics. This provides an efficient way of not only representing inter-object relationships, but also information about each individual object. Based on this idea, in our tool we took a different approach. Instead of having customized node shapes based on an object's metrics, we implement node customizations based on an objects's class.

Roassal provides the users with different kind of nodes thus making node customizations possible. Generally to customize nodes of different classes the user would have to define the shape for each object before each rendering. The basic node shape in our visualization tool is the node described in Figure 4.1 (Page 12). This node shape already provides the users with a lot of information, but in order to make our tool more useful, we use a simple process to create arbitrary node customizations. Our tool offers an easy way of creating node customizations by using pragmas. The exact implementation will be explained in Section 5.3. All the node customizations are inserted at the bottom of the basic node shape. An example of a simple node customization for the `Color` class can be seen in Figure 4.7.

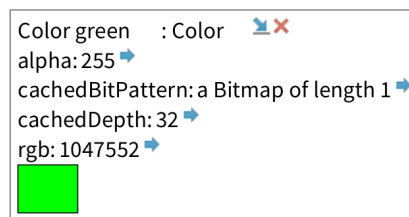


Figure 4.7: The object `Color green`, which contains a node customization at the bottom

The node has a preview of the color green at the bottom of the node, which is the node customization inserted for objects of the class `Color`.

Since each object has a different structure and distinct methods, we also want to allow distinct information display and interaction capabilities for the object visualizations in our graph. By having customized nodes we first have additional information about an object, or even offer special interactions with that object. This is very useful for objects that collect other objects or have subelements, because we can create interactions that allow us to add multiple objects to the visualization with one click. Second we can quickly differentiate objects in the graph, making it more readable.

Together with our tool we implemented two exemplary node customizations for the classes `LinkedList` and `RBProgramNode`. These two node cus-

²<http://agilevisualization.com>

tomizations will be presented in detail in the subsequent sections. These customizations can be used as inspiration for other node customizations. The exact process of node customization is explained in the appendix.

4.3.1 Linked List

In Pharo, a linked list is implemented using value links; the list object points to the head of the list, which is a value link. Each value link has a successor (pointing to the next element in the list, if any) and a value (pointing to the actual element). The class `LinkedList` is a nice example for our tool, because first a linked list is a data structure that serves as a container for other objects. A linked list in Pharo is made up of value links and the object's contained in the linked list, therefore we have two distinct object subgroups. Therefore we can create useful interactions for the linked list in our visualization.

Second, with our implementation we support two types of edges, the ones for variable references and the ones for equality. Therefore, for a linked list there are inter-object relationships between the linked list and the value links, between the value links themselves and also between the value links and the values. It would be useful to also display the relationship between the linked list and its objects. For this we can use the second type of edge, the edge representing equality.

Third, it is a data structure commonly used for learning and teaching purposes, but also as an example in other works such as *Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches* [5]. A linked list offers multiple methods, such as adding an object at a certain position or removing an object, which are more comprehensible through visualization.

The customization of linked list nodes in our tool is illustrated in Figure 4.8

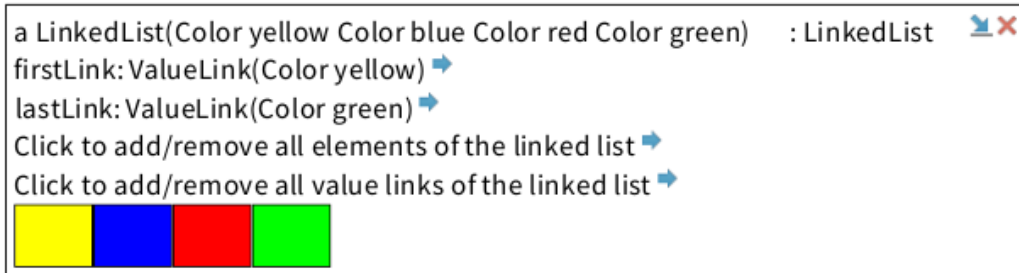


Figure 4.8: A linked list node. The node customization shows a preview of (some of) the elements (in this case, the colors).

The linked list node offers the option to add its value links and the values. In order to explore the linked list data structure, the user only needs to add the linked list to the visualization and can add the value links and objects directly through the visualization. This supports intuitive object inspection, since if desired all the value links and all the values can also be removed directly through the linked list node again. For the values we also have a preview, represented by the morphs of the values, if there are morphs. This is again to fit the aforementioned need to also have a representation of implicit object connections between relevant objects. To avoid having huge nodes, at most five elements are shown in the preview. Even when the preview is minimized there are still the equality connections from the preview to the corresponding objects in the view. In case the user prefers to have a preview of all the elements, they can do so by clicking the dots(. . .), then the complete preview will be displayed. In Figure 4.9 we can see an example of a minimized linked list preview.

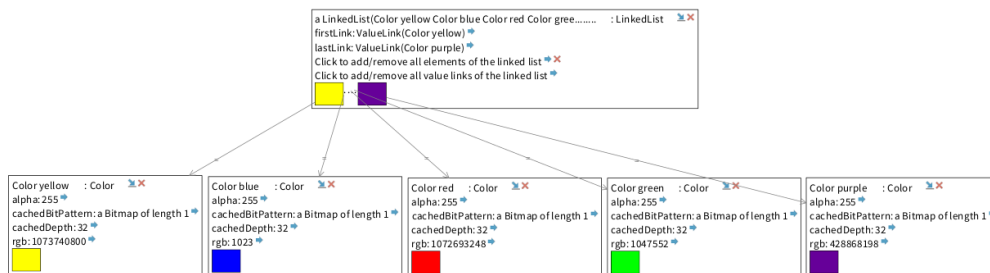


Figure 4.9: A linked list, where the preview is minimized, because there are more than 5 elements in the linked list. One is not displayed in the visualization

Apart from adding all the values at once we can also only add a single value, by clicking on its morph. These different interactions make the object set inspection extendible and enhance the inspection by having all of the elements in one visualization. Another speciality is that from the morphic value preview we have edges to the corresponding objects in the visualization. These edges represent an equality connection.

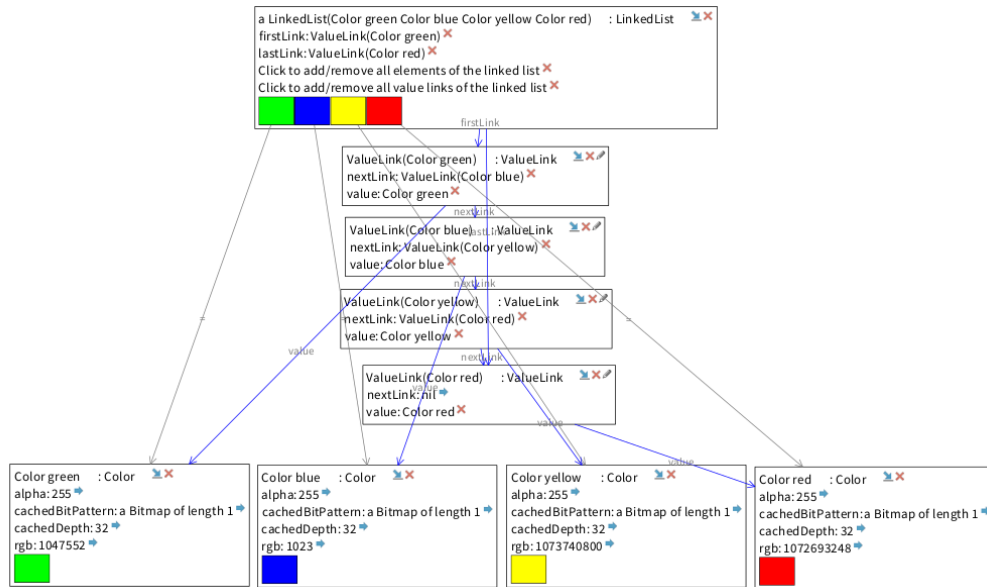


Figure 4.10: A linked list with all the value links and all the colors

This customization for linked lists was created to provide an example of how the framework can be used. Node customizations similar to this one also work well for other collections.

4.3.2 RB Program Node

The second structure we chose to make a node customization for is the `RBProgramNode` class in Pharo. `RBProgramNode` is a class that represents an abstract syntax tree node in a Pharo program. Instances of this class may have children of the same type, making up an abstract syntax tree (AST) of a compiled method or a program. The abstract syntax tree data structure is often used for program analysis. Its visualization can be used to check code correctness and structure.

Similar to the `LinkedList` for the `RBProgramNode` we customize the node by adding interactions. In Figure 4.11 we can see the customized `RBProgramNode`.

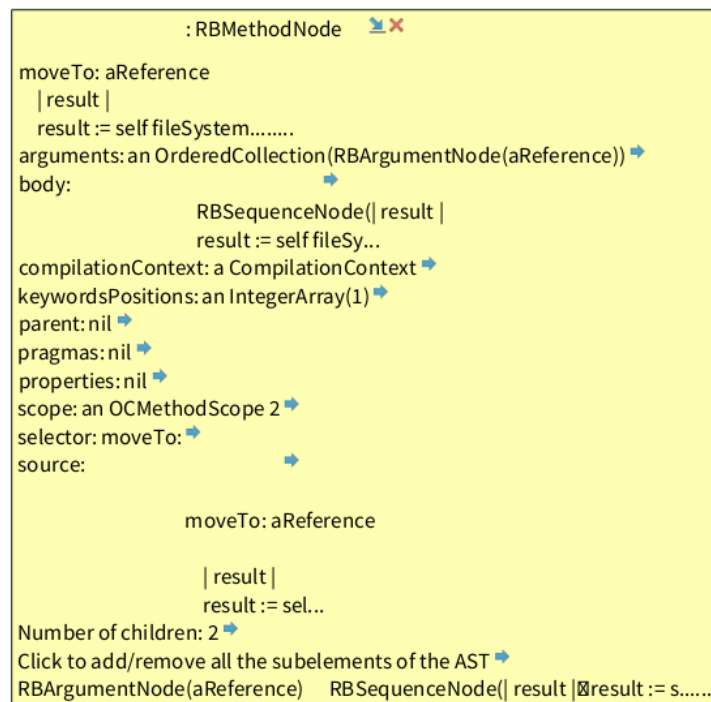


Figure 4.11: A `RBProgramNode` in Pharo which is the root of an abstract syntax tree

The first interaction (`Number of children: 2` with a blue arrow) enables the user to add only the direct children of the node. It also shows us how many children the node has. By having the possibility to add only the direct children we can build up the tree level by level and visualize the tree structure. The second interaction (`Click to add/remove all the subelements of the AST` with a blue arrow) allows the user to add the complete subtree of the node, so in the end we obtain the complete AST. The third interaction (`RBArgumentNode(aReference) RBSequenceNode...`) is similar to the one of the linked list, where we have a preview of the children and can add them individually, thereby allowing the user to only display the objects that are relevant to them.

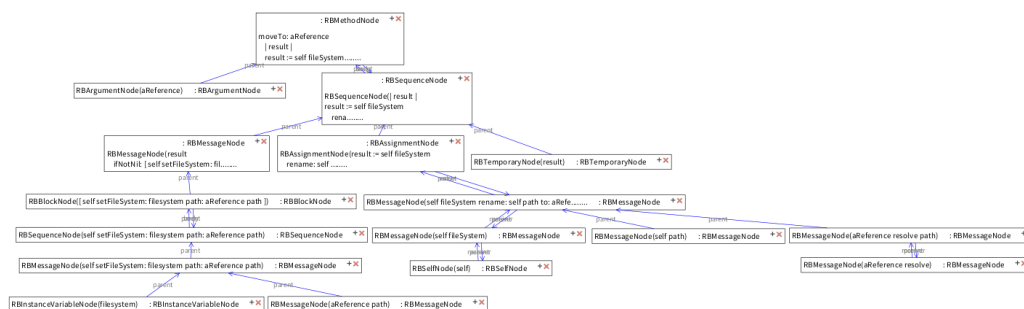


Figure 4.12: A complete abstract syntax tree for the compiled method `moveTo`; in the class `AbstractFileReference`

These interactions make the inspection of abstract syntax trees more efficient, since we can go down the tree layer by layer, making it easier to understand. A node customization similar to this one would also work well for trees in general, by making tree traversal easy.

5

Implementation

In this chapter, we will explain the object-oriented design principles and patterns that were used to implement our tool and present how the persistent subgraphs and node customizations are implemented.

For our tool, we followed the SOLID (single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion) design principles. These are a subset of principles presented by Robert C. Martin in *Agile Software Development: Principles, Patterns, and Practices* [8]. By following these principles we ensure easier code maintenance and make our tool extensible.

5.1 Design Patterns

We apply different design patterns in our tool to solve different issues or ease the use and maintenance of our tool.

To instantiate our visualization we apply the Builder design pattern, so the construction of our visualization is separated from its representation. When the method `render` of the class `ObjectSetViewer` is called, the positions of the nodes in the current visualization are saved first, and then all the elements in the current visualization are removed. After that the nodes and the edges are rendered. This happens in the classes `OSVEdge` and `OSVNode`. We then lay out the nodes and edges and restore the previously saved node positions. Finally the visualization is returned. The method implementation

looks as follows:

```

1  render
2      self saveRootNodePositions.
3      self view removeAllEdges; removeAllElements.
4      self renderNodes; renderEdges.
5      self doLayout.
6      self restoreRootNodePositions.
7      ^ view

```

We have a clear separation of the visualization construction which happens in multiple classes such as `OSVEdge` and `OSVNode` and the representation which is put together in the method `render` in the class `ObjectSetViewer`. The construction process is always the same no matter what the input is, but the visualizations that are returned are not always the same.

For the nodes in the visualization we also use the builder design pattern. When an object is added to the visualization we instantiate a new root node, which are the main nodes in the view. A root node is composed of multiple other nodes, which are instantiated and collected once the root node is instantiated. In Figure 5.1 we can see how a root node is put together.

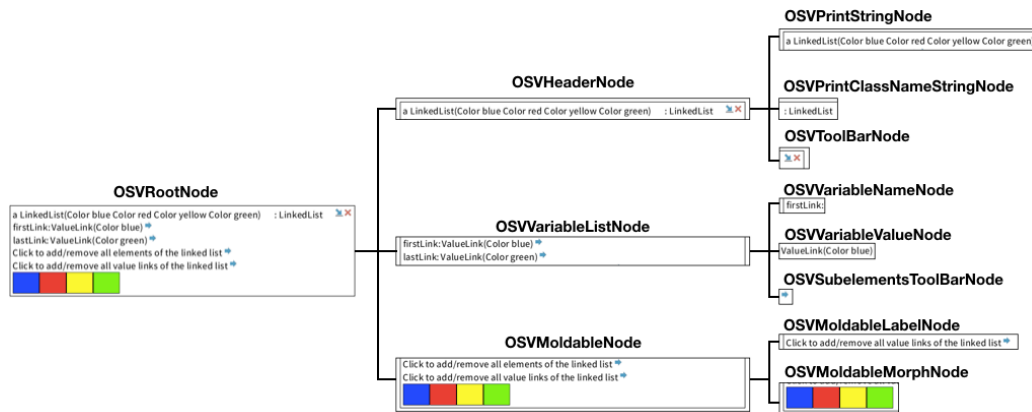


Figure 5.1: A diagram showing how a root node is composed in the visualization of a linked list node

The method `refresh` of the class `OSVRootNode` always follows the same process structure. It creates and collects instances of `OSVVariableListNode`, `OSVHeaderNode`, and `OSVMoldableNode`. However, the visual composition of the root nodes can differ. All nodes are constructed by following the same instruction pattern, but some may have customizations and others might not. Using the builder design pattern makes for a more complex instantiation but

also makes it more flexible. We can change our instantiation in only one single location. This makes code maintenance easier.

The root itself is instantiated when a new object is added to the view, but the visual elements of the node are instantiated once we call the method **render** of the class **ObjectSetViewer**. The root node always follows the same process of creating and collecting its children. For the instantiation of the visual elements the method **generateRTElements** is called in each of the root node's children's classes. Also for our nodes we at times use the factory method design pattern, depending on the state of the root node, subclasses decide which classes to instantiate. One example is the class **OSVToolBarNode**, depending on whether the root node is minimized it either instantiates the class **OSVMinimizeNode** or the class **OSVMaximizeNode**.

Another pattern that is implemented in our tool is the facade pattern. It is a structural pattern which simplifies the access to a complex subsystem. This pattern is at the top layer of our framework defining the interaction with the whole visualization. The user of our framework creates a visualization by using the class **ObjectSetViewer**, which is the facade class. It offers a simplified interface to the user and wraps together the node classes, the edge classes and makes the layout on them. We use the facade pattern, not because our subsystem is overly complex, but rather to provide the user with a single point of interaction with the visualization.

The classes creating the nodes in our visualization follow the Composite pattern. The composite pattern creates hierarchical tree structures of related objects. We use the abstract class **OSVNode** to specify the behaviour of all primitive and composite subclasses, which are all the node classes in our tool. Some of these child nodes have child nodes themselves and some are leaves in the hierarchical structure tree. In our visualization we use the composite root nodes anatomically, while at the same time we interact with the children of the composite. For example in our visualizations we also have edges from the morphs, which are children of the root nodes to root nodes. Additionally the component classes, such as the morph node class, do not have knowledge of the composite class (the root node class). Therefore, the component class is reusable and can be used in a context different from the composites, or for our tool it can also be used for node customizations.

In our tool we apply the observer pattern, which is when one object changes its state all dependant objects are changed automatically. If the state of a root node is changed then the dependant class **ObjectsetViewer** is notified automatically. As an example if a root node is minimized or maximized the **ObjectsetViewer** is notified and the visualization is newly rendered.

Once we call the **render** method of the class **ObjectSetViewer** the me-

mento pattern is applied and the object's internal state is restored; in our case each node's children are removed and then added again. Due to the object's state change the children might have changed as well. This is done by having a method refresh in every node class, which is called when the root node's state changes. The refresh method then updates elements that should be represented in the visualization. The observer pattern supports loose coupling between objects and allows for observers to be added or removed at any time.

5.2 Persistent Subgraphs

We implement persistent subgraphs by caching coordinates of individual nodes. Before a view is newly rendered the tool saves the coordinates of each node that is currently in the view. We save the coordinates before a new rendering and not directly after the laying out to make sure we still get the correct position coordinates, in case the node was interacted with and moved around. Then the layout is rendered on all the nodes including the nodes that have been newly added to the view. After laying out the objects the ones that were previously in the view are translated to their positions from the previous rendering. The objects that were newly added to the view, stay in the position the layout placed them. This can lead to overlapping elements. To avoid overlaps we check whether one of the new elements is covering one of the elements that were already in the view.

To deal with element overlap we look for a gap along the x-axis that fits the new element. If a gap that is big enough is found, then the element is placed directly to the right of the last element before the gap. If no gap is found, the element is placed to the right of the rightmost element. This solution makes sure that there is no element overlap no matter how many new objects are added at once.

5.3 Custom Node Creation

To make our tool extensible we provide the user with the possibility to create node customizations. The users can create node customizations for classes without having to modify any of the existing code in our tool. This is done by using pragmas. When the users create a node customization they do this directly in the class for which a node customization is being made. A method is written which contains the elements for the node customization and the method is then annotated with a pragma. We have defined two

different pragmas, one for customizations that are only text and one for morphs. This is done so that the text customizations come before the morph customizations in the customized node. In the class `OSVMoldableNodes`, all the returned elements of the methods which are annotated with either of the two pragmas are collected. The collected elements are then added to the visualization. The use of pragmas provides an easy solution to create node customizations without investing time to manipulate our tool's code.

6

Validation and Use Cases

Inspecting the run-time objects of a program helps one to understand how the program works. It also helps one to discover why a program might not be working. In this chapter we will present some use cases for which the use of our tool is helpful and aids problem solving.

6.1 General Object Set Inspection

Currently object set inspection of collections such as ordered collections, heaps, or linked lists can only be done by navigating through a textual tree of the objects in the collection. To enhance this experience one can inspect these collection objects in our tool, but the question is: when is it more useful to use our tool rather than the inspector? During an exploratory study [4] it was found that users would like specialized views for certain object types such as collections, dictionaries, trees and graphical elements. Users also mentioned that they want navigation which allows tracing back to previously inspected objects [4]. This problem would be solved by having all the inspected objects in the visualization. So the answer to this question is that the visualization provided by our tool can be seen as an additional view, and can therefore be used alongside the inspector. In Bragdon et al. [6] present a debugging tool that enables easy exploration of multiple objects. In user interviews they found that the tool's advantages are to debug complex and dynamic paths. It is therefore not a surprise that the tool is not so useful

when the paths do not have those factors [6]. Therefore the implication we can extract for our tool is that as soon as there are complex inter-object relationships between the objects in the inspected set, it is useful to use our tool. As a general rule as soon as one has two or more objects, or objects that are collections and one wants to inspect the inter-object relationships, it is useful to create a visualization with our tool.

If we inspect a set of different classes and would like to know the hierarchical structure of these classes commonly we would have to open up the inspector on each of those classes and then check whether one is a subclass or superclass of one of the other objects. Or one could use Roassal and create a visualization as described in Section 4 on <http://agilevisualization.com/AgileVisualization/Mondrian/0202-Mondrian.html>. Our tool enhances that visualization by providing additional information about the variables of the classes. For our example we added five different classes to our visualization. In Figure 6.1 we can read the hierarchical structure right out of the graph.

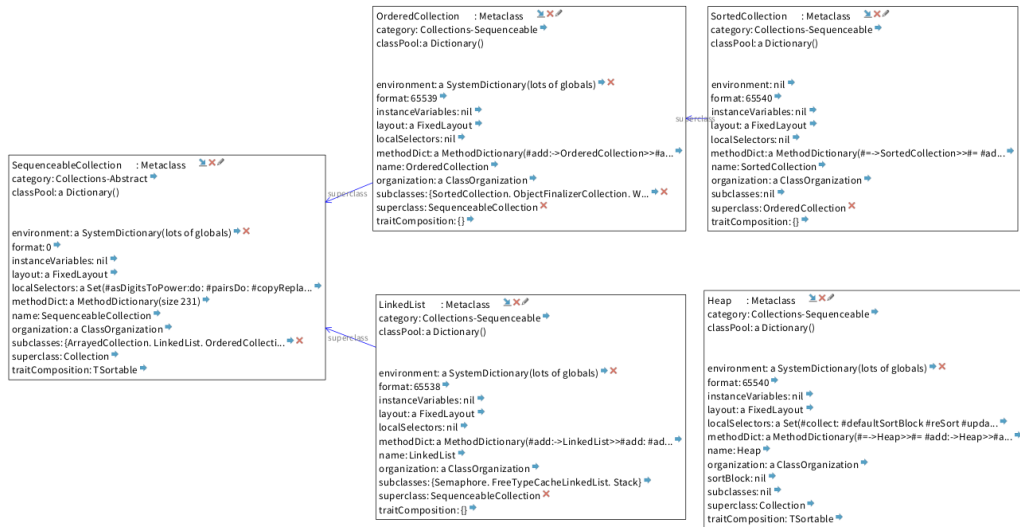


Figure 6.1: A visualization of the classes LinkedList, OrderedCollection, SequenceableCollection, Heap and SortedCollection

We can see that both classes `LinkedList` and `OrderedCollection` are subclasses of the class `SequenceableCollection` and that there is no relationship between the class `Heap` and all the other collections.

By using a visualization to represent the object set and inter-object relationships, it is easier to quickly grasp the structure of the object set. This

is an example consisting of a small set of classes, but one could also add all the classes of a package to inspect the hierarchy of these classes. For example the hierarchy of the package **SUnit-Core** in Pharo looks as illustrated in Figure 6.2.

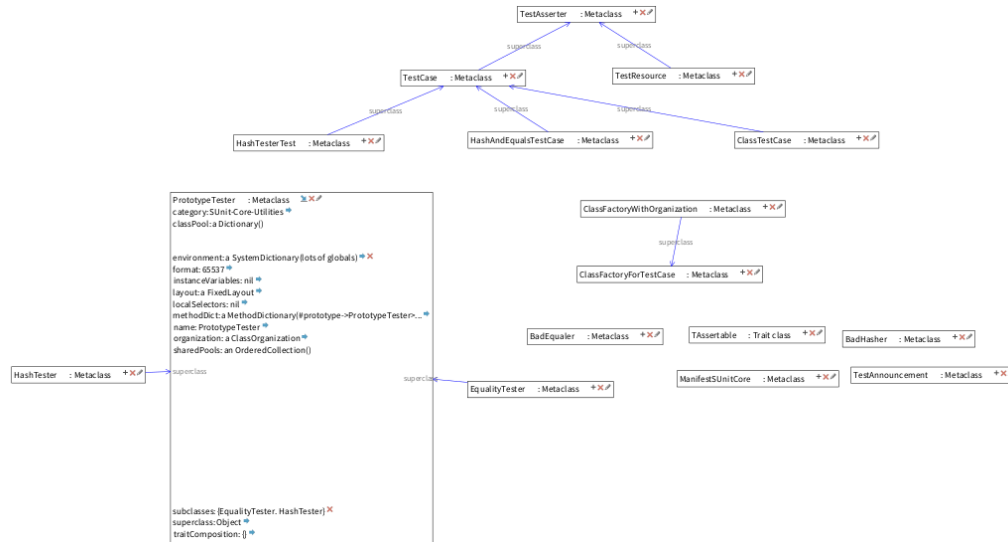


Figure 6.2: A visualization of 16 classes of the package SUnit-Core in Pharo, with all nodes minimized except for **PrototypeTester**

Like Bergel et al. [3], we show the dependencies between the different classes. Our tool allows the users to display additional information for desired nodes and hide information for other nodes.

6.2 Object Set Comparison

To compare object sets Pharo offers a limited amount of methods. We can check whether a set contains the same elements through equality, and with the method `Collection>>#intersection:` we can find the common elements of two collections. With our tool it is simpler to find objects, which the object sets do not have in common and we can do multiple assertions simultaneously. In addition you get a visual representation of the object sets. The answers to these different object set comparisons can help in debugging or creating test cases. With our visualization tool the software developer can make multiple object set comparisons simultaneously and inspect whether their given output is as expected. Aftandilian et al. [1] use a linked list as

a use case so we will also use linked lists to compare the evolution of two linked lists.

We compare the linked lists both in the inspector and in our tool, answering questions about the lists commonalities, disparities, and the ease of set comparison. The linked list `colorsOne`, at time A contains the objects: `Color yellow`, `Color red`, `Color orange`, `Color purple`, `Color black`, `(Color r: 0.658 g: 0.532 b: 0.424 alpha: 1.0)`, `Color white`. The second linked list `colorsTwo`, at time A, contains the objects: `Color green`, `Color white`, `Color red`, `Color yellow`, `Color purple`, `Color black`, `Color orange`, `(Color r: 0.433 g: 0.147 b: 0.07 alpha: 1.0)`. First we compare these two lists with the inspector. Since in the inspector we can only inspect one object we will need two inspectors to compare the two linked lists, as is displayed in Figure 6.3.

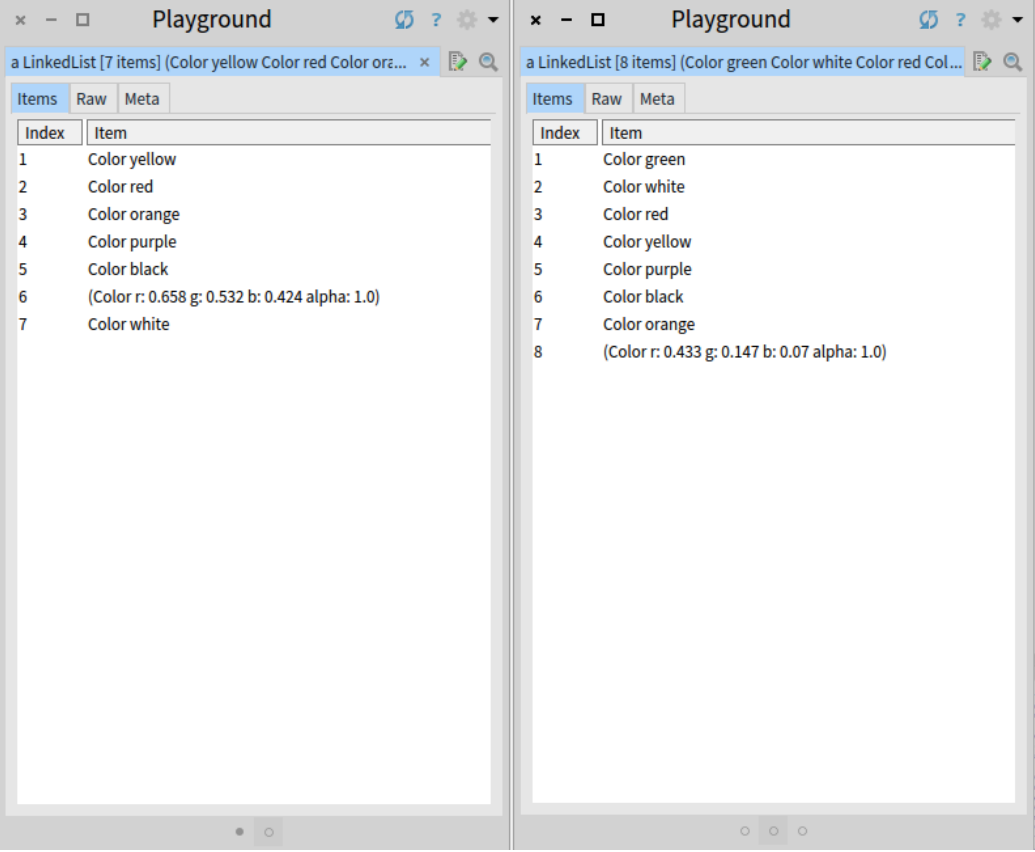


Figure 6.3: Two inspectors displaying the items of two distinct linked lists

To answer whether these two lists have objects in common the user has to

scan through each object of `colorsOne` and then check if is also in `colorsTwo`. Therefore the user basically has to do a linear search for each object in `colorsOne`, since the two lists are not sorted. This is a very tedious and inefficient process. The other option is to use the method `intersection:`, but this does not help us with the next question. To check which are the elements that the two lists do not have in common, the user has to search both lists and eliminate the previously found common objects. To answer the question, one would again need to write code to extract the elements that are not shared.

With our tool we can spot right away which objects the lists have in common and which not. We added both linked lists, their value links and the objects contained in the list to the visualization in our tool. In Figure 6.4 we can see the obtained visualization.

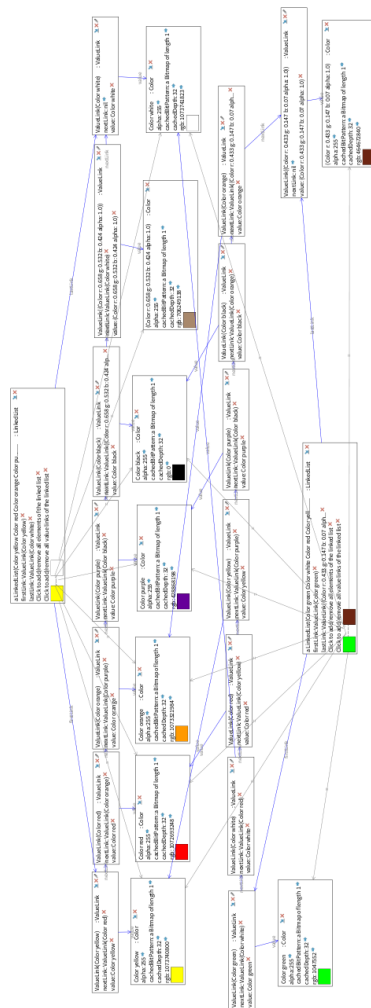


Figure 6.4: A visualization created with our tool, displaying the two linked lists `colorsOne` and `colorsTwo`

The user can spot the common objects and see which objects are only contained in one list, to enhance the inspection in this case we can remove the value links of both list, so there are less edges in the visualization. We then get the following visualization in Figure 6.5.

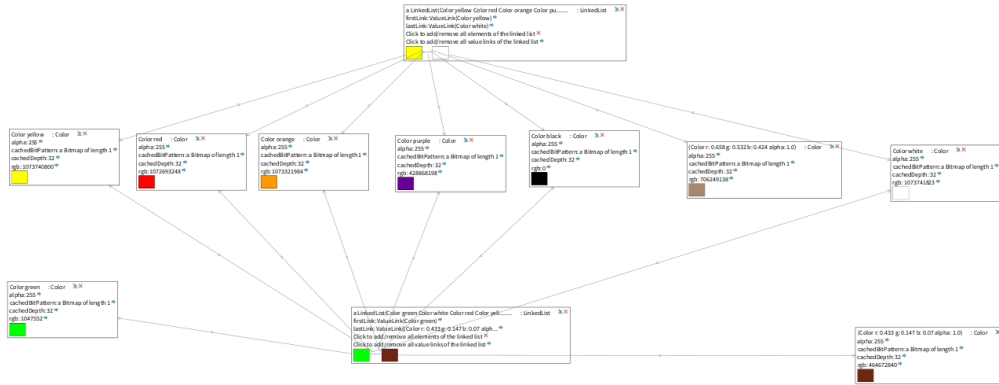


Figure 6.5: A visualization created with our tool, displaying the two linked lists `colorsOne` and `colorsTwo`, same as Figure 6.4 but without value links

The questions about common objects and objects only contained in one list can both be answered by using only one visualization. Now we would like to compare these same linked lists at a later point in time. At time B the list `colorsOne` contains the objects: Color yellow, Color red, Color orange, Color green, Color purple, Color black, Color white. We added the Color green in the middle of the list and removed (Color r: 0.658 g: 0.532 b: 0.424 alpha: 1.0). The list `colorsTwo` at time B contains the objects: Color green, Color white, Color red, Color yellow, Color black, Color orange, (Color r: 0.433 g: 0.147 b: 0.07 alpha: 1.0), Color blue. The colors Color purple and (Color r: 0.433 g: 0.147 b: 0.07 alpha: 1.0) were removed and the color blue was added at the end of the list. So now we want to answer the question: Which changes were made to the lists `colorsOne` and `colorsTwo`.

With the inspector we would again have to use two inspectors, this time for each linked list. In Figure 6.6 we have an inspector open on `colorsOne` at time A and time B.

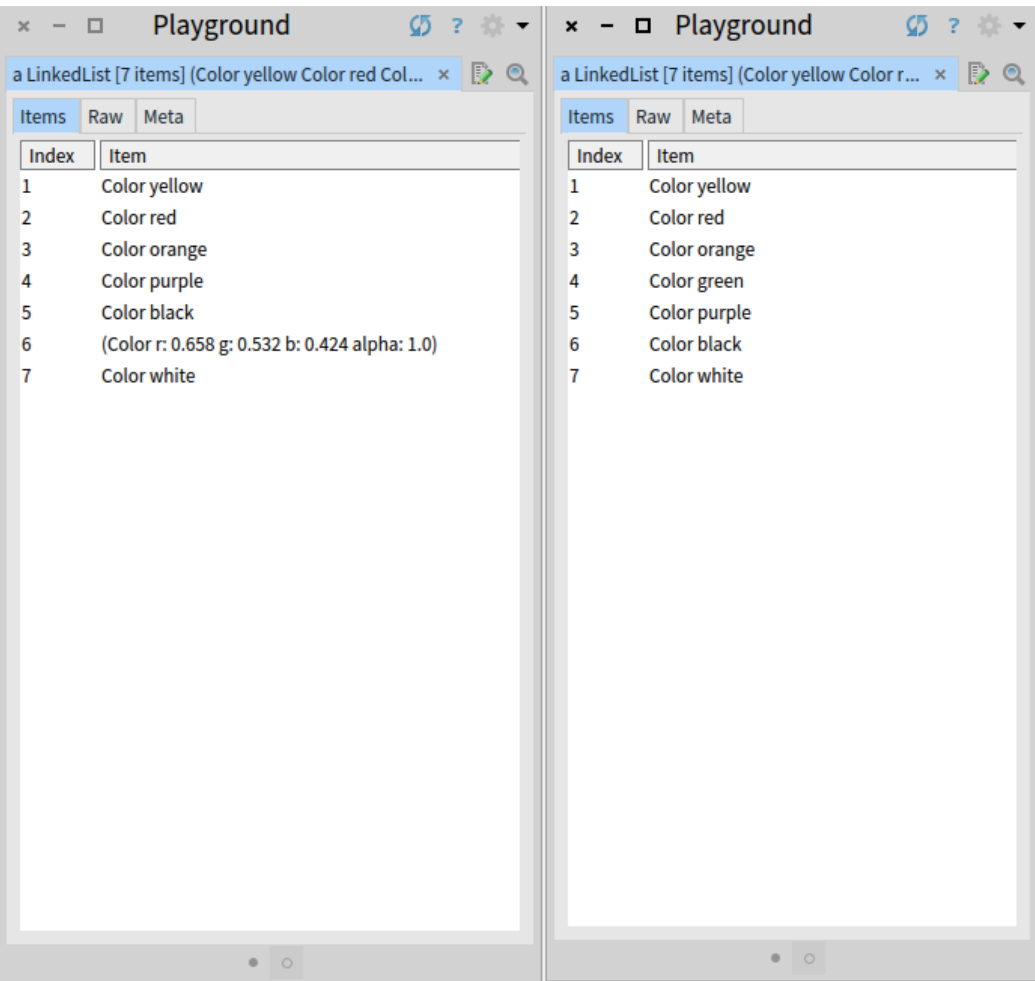


Figure 6.6: The linked list `colorsOne` at time A in the left inspector and at time B in the right inspector

As before the user would have to tediously compare all the objects to spot if any object were added or removed. The other approach would be to again use `intersection`: and then remove these objects from the list at time A to see which objects were removed and then remove them from the list at time B, to find the objects that were added. The same process has to be done for `colorsTwo`. Now if the users wants to compare the list again they would have to again open two inspectors and compare them. We can see that this process with the inspector is very tedious and the user has to do multiple steps and open multiple windows.

With our tool the user can do all the steps in the inspector using just

one visualization. With our visualization tool we can continue using the visualization in Figure 6.5. We keep the objects contained in the lists but remove both linked lists. Then to first inspect `colorsOne` we add it back to the visualization. We can see in Figure 6.7, that the linked list `colorsOne` does not have an edge to (Color r: 0.658 g: 0.532 b: 0.424 alpha: 1.0) but has one to Color green.

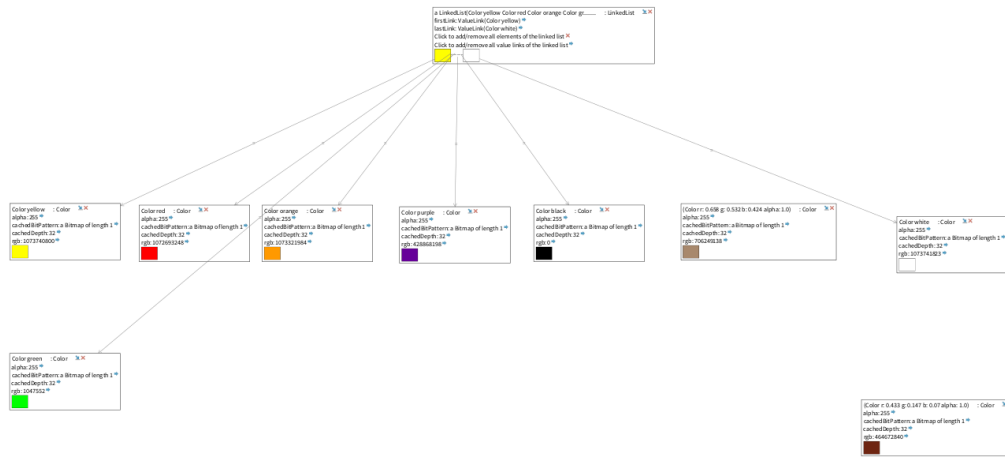


Figure 6.7: The linked list `colorsOne` and the objects from the visualization in Figure 6.5

We now also add `colorsTwo` to inspect the changes. We can easily spot in Figure 6.8 that the colors Color purple and (Color r: 0.433 g: 0.147 b: 0.07 alpha: 1.0) are not connected anymore to `colorsTwo`, therefore not in `colorsTwo` at time B.

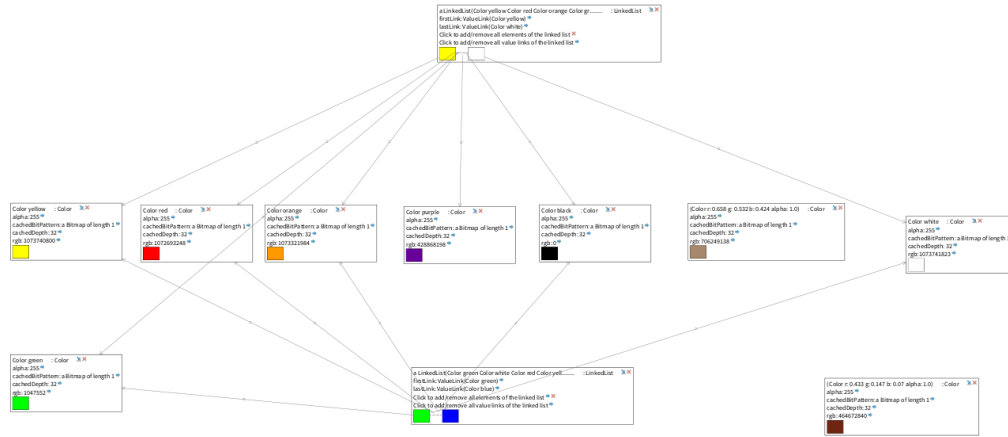


Figure 6.8: The linked lists `colorsOne`, `colorsTwo` and the objects from the visualization in Figure 6.5

In addition in the arrow button next to the label “Click to add/remove all elements of the linked list” of `colorsTwo` indicates there are objects in list `colorsTwo` that are not in the visualization yet. We click the button and can see in Figure 6.9 that the `Color blue` was added to the visualization, this color was added to `colorsTwo` between time A and time B.

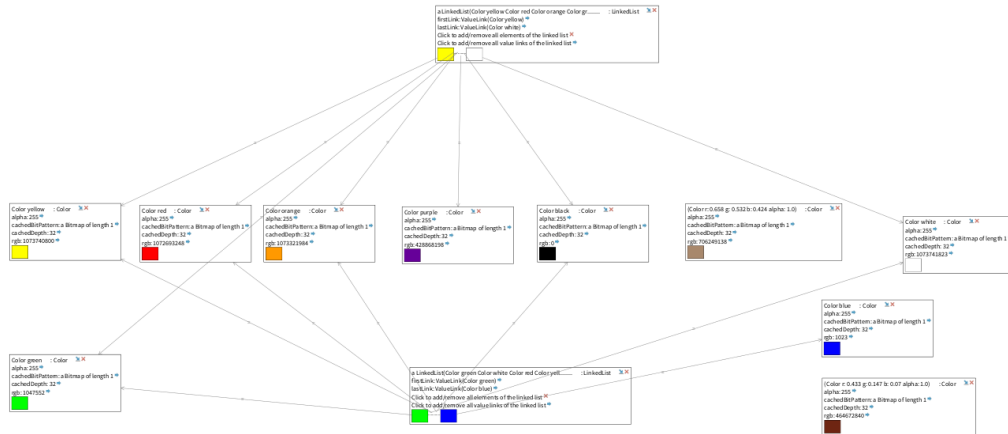


Figure 6.9: The linked lists `colorsOne`, `colorsTwo` and the objects from the visualization in Figure 6.5 and the `Color blue`

This example demonstrates a case for which it is more efficient to use our visualization tool instead of the object inspector. Our tool offers multiple

interactions that allow the users to answer multiple set comparison questions at once.

7

Conclusion and Future Work

In this thesis, we have presented a tool for visualizing sets of objects as graph structures as an alternative to traditional object inspectors. The visualization of object sets in our tool provides an alternative manner of object set comparison, which is otherwise not provided in the Pharo IDE. Inter-object relationships are easily detectable and object set structures are displayed in the graph. In addition to the base implementation, we provide two exemplary node customizations for two core classes. This library can be extended for any desired objects in Pharo by using pragmas and following the instructions in the appendix.

The following features would be interesting and useful for future work:

- Add custom nodes for set and data structure classes, such as the subclasses of the class `Collection`.
- Class specific layouts. The user should be able to chose a custom layout for an object and the related objects, allowing for individual layouts for substructures. As an example a visualization containing a linked list and a tree could have a circular layout for the linked list and its objects, and the tree could have a tree layout for its objects. Having a single layout on a set containing multiple different data structures, limits the graphs readability.
- Proper inspector integration, so that when an object is selected in the inspector it would also be selected in the graph and vice versa. It

would be a useful feature to make object modification possible directly through the graph.

- The current node representation could be replaced by displaying the inspector window of an object as a node in the graph. Then objects could already contain customizations by having multiple views in the inspector node in the graph.

These limitations show that the tool is not to be used as a stand alone inspection tool, but rather as an additional help. The tool provides a more thorough overview of object structures.

Bibliography

- [1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. “Heapviz: interactive heap visualization for program understanding and debugging.” In: *Proceedings of the 5th international symposium on Software visualization*. SOFTVIS '10. Salt Lake City, Utah, USA: ACM, 2010, pp. 53–62. ISBN: 978-1-4503-0028-5. DOI: 10.1145/1879211.1879222. URL: <http://doi.acm.org/10.1145/1879211.1879222>.
- [2] Alexandre Bergel. *Agile Visualization*. LULU Press, 2016. ISBN: 9781365314094. URL: <http://AgileVisualization.com>.
- [3] Alexandre Bergel, Sergio Maass, Stéphane Ducasse, and Tudor Gîrba. “A Domain-Specific Language For Visualizing Software Dependencies as a Graph.” In: *Proceedings of 2nd IEEE Working Conference on Software Visualization (VISSOFT NIER)*. 2014. URL: <https://dl.dropboxusercontent.com/u/31543901/MyPapers/Berg14c-Graph.pdf>.
- [4] Andrei Chiş. “Moldable Tools.” PhD thesis. University of Bern, Sept. 2016. URL: <http://scg.unibe.ch/archive/phd/chis-phd.pdf>.
- [5] James H. Cross II, T. Dean Hendrix, David A. Umphress, Larry A. Barowski, Jhilmil Jain, and Lacey N. Montgomery. “Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches.” In: *Trans. Comput. Educ.* 9.2 (June 2009), 13:1–13:32. ISSN: 1946-6226. DOI: 10.1145/1538234.1538240. URL: <http://doi.acm.org/10.1145/1538234.1538240>.
- [6] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. “Debugger Canvas: Industrial experience with the code bubbles paradigm.” In: *2012 34th International Conference on Software Engineering (ICSE)* (2012), pp. 1064–1073.
- [7] Dean Frederick Jerding and John Stasko. “Using Visualization to Foster Object-Oriented Program Understanding.” In: (Jan. 1994).

- [8] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445.
- [9] Michael Meyer, Tudor Gîrba, and Mircea Lungu. “Mondrian.” In: *Proceedings of the 2006 ACM symposium on Software visualization - SoftVis '06*. ACM Press, 2006. DOI: 10.1145/1148493.1148513.
- [10] Nicolai Hess Stéphane Ducasse Dimitris Chloupis and Dmitri Zagidulin. *Pharo by example 5*. Second. 2017.
- [11] Thomas Zimmermann and Andreas Zeller. “Visualizing Memory Graphs.” In: *Revised Lectures on Software Visualization, International Seminar*. London, UK, UK: Springer-Verlag, 2002, pp. 191–204. ISBN: 3-540-43323-6. URL: <http://dl.acm.org/citation.cfm?id=647382.724787>.



Anleitung zu wissenschaftlichen Arbeiten

This chapter provides a tutorial for using the presented tool. We explain how to obtain it, use the basic features, and create a simple custom visualization. For this tutorial, we assume that the reader has basic knowledge of Pharo and Moose. For an introduction to Pharo, we refer the interested reader to *Pharo by example 5* [10].

A.1 Installation of environment and tools

- Obtain a fresh Moose 6.1 image (and VM if necessary) and open it.
- In the World Menu open an Iceberg Browser
 1. Click +Clone repository
 2. Enter the remote URL from github: `https://github.com/ccorrodi/bachelorarbeit-eve.git`
 3. Go to the packages tab for bachelorarbeit-eve
 4. Right click and load the Thesis Eve package

A.2 Basic usage

Using the visualization tool is straightforward. After obtaining the target objects, all a user has to do is add them to a new visualization, which is represented by an instance of `ObjectSetViewer`. Here, we illustrate this using a linked list. We will explain how to visualize and interact with an object such as a `LinkedList`. For the class `LinkedList` we have a node customization. Therefore we have two ways of adding the `ValueLinks` and elements to the visualization.

To start, we open up a Playground in Pharo. In order to obtain a visualization with our tool we instantiate a new object of the class `ObjectSetViewer`. In our example we will use a `LinkedList` containing the colors yellow, blue, red and green. In order to add our given `LinkedList` to the instantiated visualization we send the message `addObject:` to the visualization. The parameter we give to the method is the `LinkedList` itself. Lastly we send the message `render` to the `ObjectSetViewer`. We then have the following code in the playground:

```
1 view := ObjectSetViewer new.  
2  
3 list := LinkedList new.  
4 list  
5     add: Color yellow;  
6     add: Color blue;  
7     add: Color red;  
8     add: Color green.  
9 view  
10    add: list;  
11    render.
```

After running the code we obtain the visualization in Figure A.1

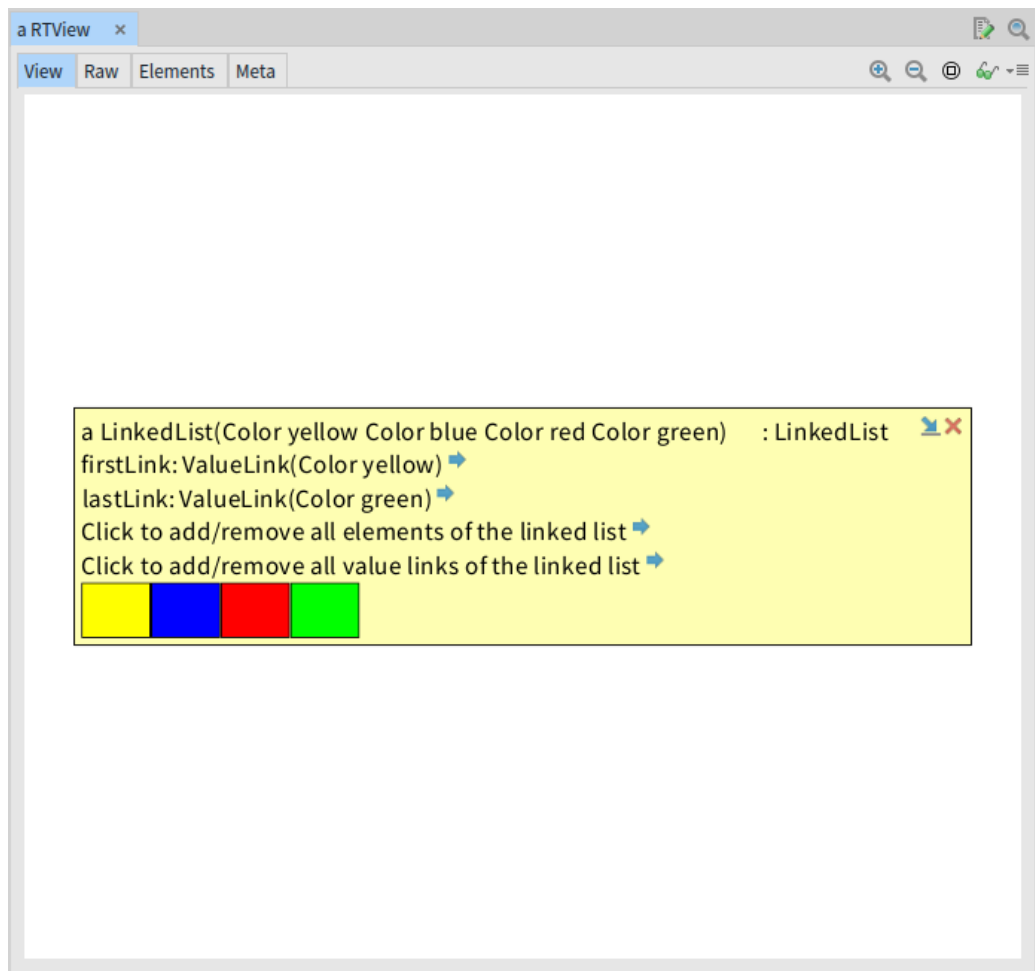


Figure A.1: A visualization of a linked list, containing four colors, in our tool

Now to enhance the visualization we will add the **ValueLinks** and colors of the **LinkedList**. We can do so by directly interacting from within the visualization. In Figure A.2, we can see the different interactions that are possible for our **LinkedList**. These different interactions enable the users to create an individual visualization that only represents information relevant to the user.

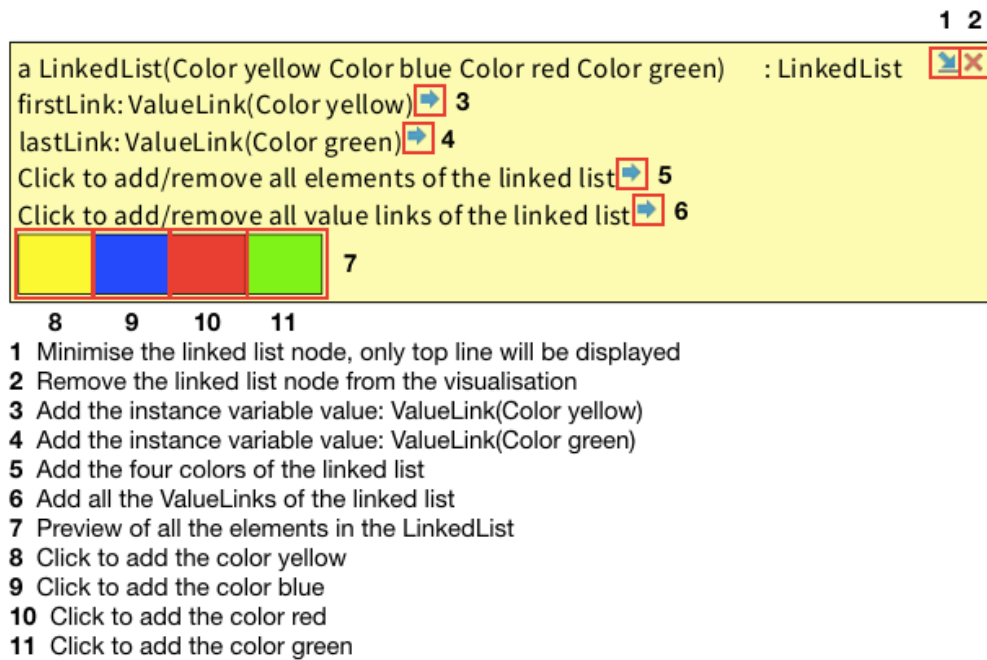


Figure A.2: The interactions of the LinkedList containing the colors yellow, blue, red and green

There are two buttons which can be found under number 5 and 6 in the legend in Figure A.2. With these we can once add all the **ValueLinks** and with the other add all the colors of our **LinkedList**. We can then inspect the single elements while also inspecting the structure and relationships between them. All the elements in the visualization can be moved around individually and removed if desired.

Adding the elements to the visualization directly through the visualization is one option. The other option is to add elements through the playground. Going back to the initial playground we can then for our **LinkedList** add the set of **ValueLink** objects and the set of colors through the code. The code for that looks as follows:

```
1 view := ObjectSetViewer new.
2
3 list := LinkedList new.
4 list
5     add: Color yellow;
6     add: Color blue;
7     add: Color red;
8     add: Color green.
9
```

```

10 links := OrderedCollection new.
11 list linksDo: [ :each | links add: each ].
12
13 view
14     addObject: list;
15     addObjects: list, links;
16     render.

```

After adding the `ValueLinks` and the elements, by using any of the two described options, we obtain a visualization as the one shown in Figure A.3.



Figure A.3: Visualization of a LinkedList and its ValueLinks and colors

All the important interactions such as removing an element or removing a set of elements can also be done with the following functions:

- `ObjectSetViewer>> removeObject`: function to remove one single object from the visualization
- `ObjectSetViewer>> removeObject`s: function to remove a set of objects from the visualization

A.3 Implementing a custom node shape

In order to customize individual object nodes, pragmas are used. Pragmas are method annotations. The annotated methods are then collected dur-

ing run time and we can then interact with their outputs. This allows for class independent customizations. The class `OSVMoldableNode` and its subclasses provide methods which collect these pragmas and allow interaction with them. Basic node shapes in the `*ThesisEve` package can be used to compose more complex shapes. For example, a `OSVPrintStringNode` is typically used to display an arbitrary string, whereas a `OSVVariableNode` is used for instance variables that automatically link to the targets if they are in the view. This is useful to highlight equality between objects, even if there is no direct reference.

In this section, we show how nodes can be customized, using the class `Heap` as an example.

A.3.1 Using pragmas

We have defined two pragmas for two different return types. Methods with the pragma `<OSVAsLabel>` must return a Roassal label. While methods annotated with the pragma `<OSVAsMorph>` must return a morph. The internal process when an object is added to the visualization is the following: 1. Our visualization tool looks for methods with the `<...>` annotation in the target's class hierarchy; 2. a generic root node is created; 3. for each method found in step 1, a custom subvisualization is rendered using that methods output; 4. the whole visualization is returned and displayed.

There are two main ways the user can insert the pragma annotation into the methods used for node customization.

First, the user can directly go to the class for which a customization should be made, then create a method with the desired output and annotate the method with the correct pragma.

Second, the user can have a visualization (`ObjectSetViewer`) containing an instance of the class to be customized. In the node representation of that object we have a button, which allows us to directly make a node customization. The button directly takes the user into the class to be customized. The user can then again create a method there and add the correct pragma to the method.

As mentioned earlier we will make a customization for the class `Heap`. The basic default node visualization for the class `Heap` can be seen in Figure A.4

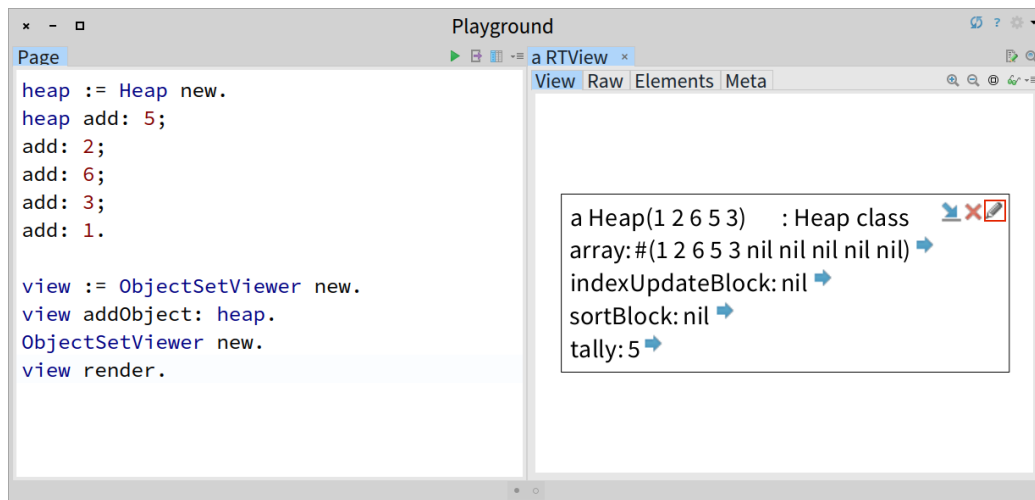


Figure A.4: An ObjectSetViewer containing a heap object

The code in the playground is the following:

```

1 heap := Heap new.
2 heap add: 5;
3 add: 2;
4 add: 6;
5 add: 3;
6 add: 1.
7
8 view := ObjectSetViewer new.
9 view addObject: heap.
10 view render.

```

The highlighted pencil button at the top right of the node only appears when there is currently no node customization for the object class. When the button is clicked, the protocol `*ThesisEve` is added to the object class and a browser is opened on the object class.

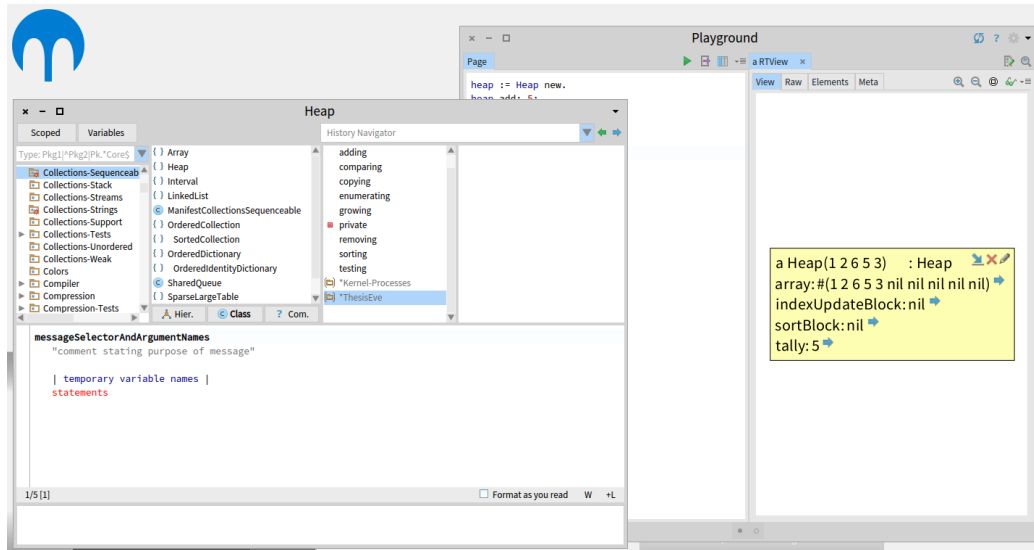


Figure A.5: Protocol `*ThesisEve` is added to the class `Heap` and a browser is opened on the `Heap` class

Within this protocol we can then write our methods which contain a pragma. Currently, the following two pragmas can be used to define custom node shapes:

- `<OSVAsLabel>`: this pragma expects the method to give back a Roassal label.
- `<OSVAsMorph>`: this pragma expects a Pharo morph to be returned

A.3.2 Heap customization

For our example of a node customization for the class `Heap`, we would like to first add a label containing some text, such as “Sorted Heap:”. Followed by a label containing the sorted `Heap`. Second we would like to add another label containing the text, “Click here to add/remove the elements of the heap”, followed by buttons which allow the addition and removal of those elements. The add button should only be displayed when not all of the elements are in the view. Similarly, the remove button is shown only if at least one of the elements is in the view. Third, we would like to have a preview of the objects in the heap in the main heap node itself.

We will now create the methods used for our node customization.

A.3.3 Sorted Heap label

For our label we will use the pragma `<OSVAsLabel>`. We write a new method called `asOSVSortedLabelNode` in the class `Heap`.

Now since this method will only be displaying a string, we can use the class `OSVPrintStringNode` to obtain a node that will print the given input. Our code in the method `asOSVSortedLabelNode` will look as follows:

```
1 asOSVSortedLabelNode
2     <OSVAsSubelementsLabel>
3     ^ OSVPrintStringNode new
4         target: 'Sorted_Heap:', (self deepCopy fullySort
                                   ) asString
```

We make a `deepCopy` of our heap because we do not want to sort our original heap, to highlight the difference between our heap and the sorted one. For our `OSVPrintStringNode` we set the desired label text as `target`.

We now render our original heap node again and our heap node will now look as illustrated in Figure A.6.

```
a Heap(1 2 6 5 3) : Heap
array: #(1 2 6 5 3 nil nil nil nil nil)
indexUpdateBlock: nil
sortBlock: nil
tally: 5
Sorted Heap: a Heap(1 2 3 5 6)
```

Figure A.6: A heap node after adding a label with the sorted heap. The highlighted section is the added node customization.

A.3.4 Label with buttons to add/remove objects of the heap

As with the previously created label, we extend the class `Heap` with a new method `asOSVObjectsLabelNode`. For this label we will also use the pragma `<OSVAsSubelementsLabel>`, because we will be returning a text. Fortunately, there is already a class implementing displaying sub elements (which was used in the `LinkedList` customization), namely `OSVSubelementsLabelNode`.

The implemented method `asOSVObjectsLabelNode` then looks as follows:

```
1 asOSVObjectsLabelNode
2     <OSVAsSubelementsLabel>
3     ^ OSVSubelementsLabelNode new
4         target: self;
```

```

5      label: 'Click to add/remove the objects of the
        heap'

```

Here we can see that for the class `OSVSubelementsLabelNode` we do not set the output string as the target, but we set it as the variable `label`. The target we give is the set of objects we want to be able to add or remove from the visualization, in our case the objects in the heap. The class `OSVSubelementsLabelNode` automatically adds buttons to the label. It also provides the logic for adding and removing the subelements (i.e., all elements in the given collection) when buttons are pressed. If we render the node again we receive the customized node displayed in Figure A.7

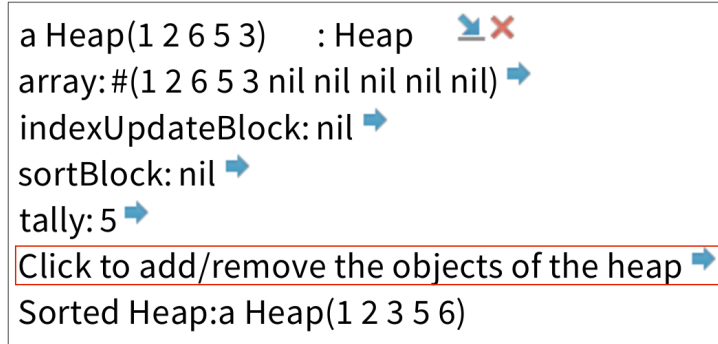


Figure A.7: The heap node after adding two node customizations, again the highlighted area is the newly added node customization.

When the user clicks the arrow button the objects of the heap will be added to this visualization as nodes, as illustrated in Figure A.8.



Figure A.8: A visualization displaying the customized heap node and its objects

This visualization is lacking the edges for equality. For that purpose we will add a heap preview which will also show the equality edges to the heap elements.

A.3.5 Heap preview

We follow the same procedure as before. In the class `Heap`, we create a new method `asOSVMorphNode`. For the `LinkedList` customization we also have a preview, so we can again use the same node to obtain a preview of our heap. The code in the method `asOSVPreviewNode` is the following:

```

1 asOSVMorphNode
2     <OSVAsLabel>
3     ^ OSVLinkedListNode new
4         target: self

```

As a target we set the objects of which we want a preview. In Figure A.9 we can see our final heap node after adding three different node customizations.

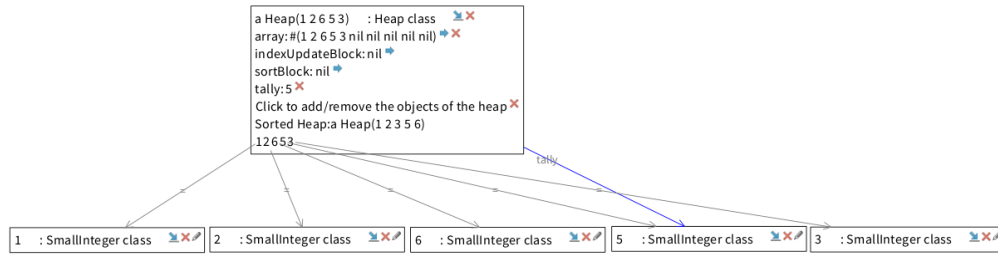


Figure A.9: A visualization displaying the customized heap node and the elements of the node, so to highlight the equality edges.

A.3.6 Customizing CalendarMorph using pragmas

To present the use of `<asOSVMorphNode>` pragma we will create a node customization for the class `CalendarMorph`, with our code looking as follows in the playground:

```

1 view := ObjectSetViewer new.
2 view addObject: (CalendarMorph on: Date today).
3 view render

```

We obtain the default node visualization in Figure A.10

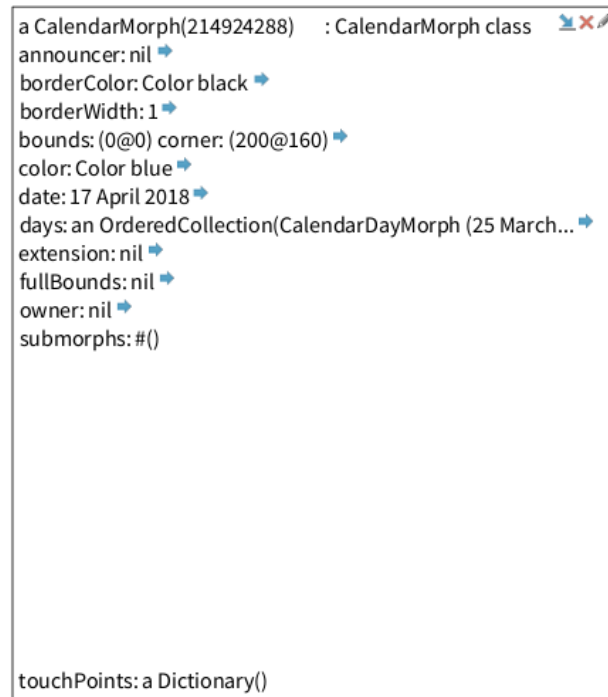


Figure A.10: A visualization displaying an object of the class `CalendarMorph` with the date of today.

We will now create a method `asOSVMorphNode` in the class `CalendarMorph` and annotate it with the pragma `<asOSVMorphNode>`. It returns a `OSVMorphNode` with the calendar as a target. The code look as follows:

```

1  asOSVMorphNode
2      <OSVAsMorph>
3      ^ OSVMorphNode new
4          target: self

```

The visualization can be seen in Figure A.11

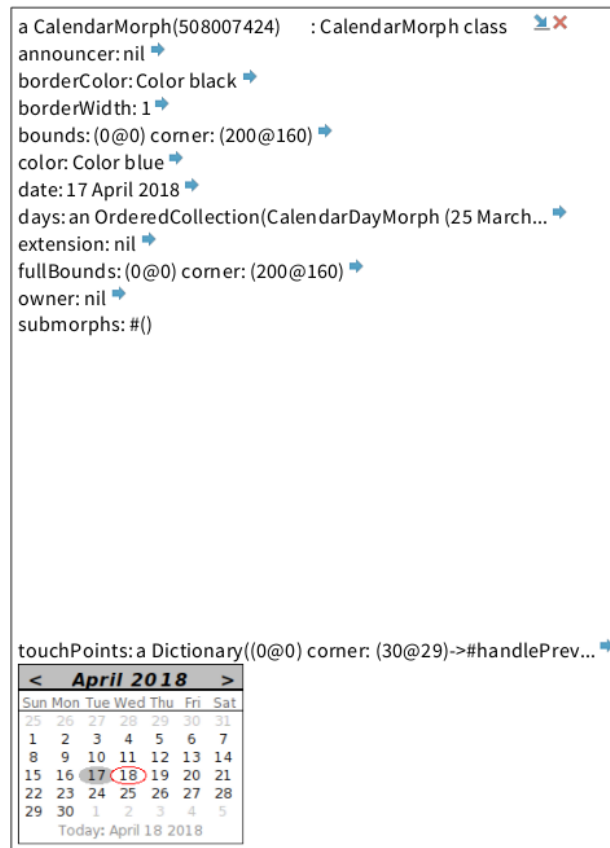


Figure A.11: A visualization displaying an object of the class `CalendarMorph` with the date of today. The node shows the morph of this class as a node customization.

This step by step instruction has guided the user through the creation of a node customization for the classes `Heap` and `CalendarMorph`. This is just one example of the use of the methods containing pragmas. The user can create other pragma methods to collect objects for node customization.