Rapid Prototyping of Visualizations using Mondrian*

Adrian Lienhard, Adrian Kuhn and Orla Greevy
Software Composition Group
University of Bern, Switzerland
{lienhard, akuhn, greevy}@iam.unibe.ch

Abstract

Science requires tools, and computer science is no different. In a typical research context however, it is not known upfront how a tool should work. Researching the tool's design is part of the investigation process. Various designs have to be prototyped and experimented with. This paper focuses on the research process of interactive visualization tools. We present how to improve development, so that a novel tool can be tested and modified at (almost) the same time. We present the Mondrian framework, which supports on-the-fly prototyping of interactive visualizations. As an example, we present the research process of the Feature Dependency Browser, a visualization tool which we developed to allow software engineers inspect runtime dependencies between features.

1. Introduction

Research is an iterative process of exploration and hypothesis formulation, which makes it difficult to design research tools in advance, as they need to coevolve with the research process. In this paper we focus on researching visualizations for software comprehension. Building a novel visualization technique in one round-trip is difficult. It is easier to build it in iterations matching the research process. In each round-trip, the visualization technique typically needs to be adapted to accommodate new findings and hypotheses about the underlying information space.

To address this, researchers would like to have a visualization framework which allows them to change a visualization at runtime, while continuing the exploration of the information space in parallel. Unfortunately, most available frameworks do not facilitate an iterative development of visualizations. Long deployment cycles lead to long iterations between hypothesis formulation and verification. This is tedious as it slows down the research process.

In this paper we propose means to facilitate rapid prototyping of novel visualizations. We advocate to address this challenge by running the visualization engine within a dynamic container. A dynamic container is a scripting host that features changing the investigated visualization technique at runtime – or "viewtime", as might be more accurate in this context.

Another challenge addressed in this paper is how to facilitate interactivity and composition of visualizations. We present an extension of the model-driven Mondrian framework [6], which facilitates composition of interactive visualizations using the DOM event model to communicate between several visualization canvases of the same tool.

To illustrate the flexibility of our setup, we present the research development of our Feature Dependency Browser tool, which we developed to analyze runtime dependencies between features [4].

In the next section we briefly introduce the Mondrian framework. A sample research process using rapid visualization development is presented in Section 3 and Section 4 concludes.

2. On-the-fly prototyping

Using conventional development techniques, visualizations have to go through a complete compile round-trip each time changes need to be applied. Hence, to avoid this compile overhead, a key requirement for rapid or even on-the-fly prototyping is to host the visualization framework within a dynamic container. A dynamic container is a scripting host that features changing the investigated visualization technique at runtime. Within the hosting container, both IDE and application code run at the same time, allowing researcher to develop a visualization at runtime.

To the best of our knowledge, no visualization frameworks used in the research community supports *On-*

* In Proceedings of the IEEE International Workshop on Visualizing Software for Understanding (Vissoft'07), pp. 67–70

the-fly prototyping as we propose it in this paper. Note that we speak of the development of novel visualization techniques, which requires more than mere composition of predefined shapes. Furthermore, most visualization frameworks provide only limited interaction mechanisms.

Work on rapid prototyping of visualizations is still in its infancy. Other researchers are experimenting with rapid prototyping techniques for visualizations. A promising approach is presented by Bull *et al.*; they suggest Model Driven Engineering (MDE) for visualization building [1, 2]. They advocate the use of MDE to achieve customizable interfaces by composing small model-driven visualization components. Their approach differs from ours, as their focus on defining reusable visualization widgets, similar to UI widgets in user interface design. This allows researchers to build new visualization tools in the same way today's programmers build user interfaces.

For our research, we use Mondrian as engine and a Smalltalk image as hosting container. However, other dynamic environments (Ruby, Python, ECMAScript, etc...) might be used as hosting container as well.

The key features of Smalltalk that enable rapid prototyping are:

- Hot debugging support. The ability to hot-debug exceptions and use break points speeds-up productivity significantly.
- Hot recompilation. On-the-fly method recompilation, *i.e.*, recompiling a method while the application is running, is important since it supports updating code without having to restart the tool.

Mondrian has been successfully employed in the domain of rapid prototyping [5, 6]. Mondrian provides basic building blocks that allow declarative scripting to express visualizations based on an underlying model. It does not presuppose any structure on the data. The only requirement is that the data is described and queriable by the means of a simple metamodel.

A key aspect of Mondrian visualizations is that they are interactive. We can query the nodes to obtain more fine-grained details about the underlying entity which it represents and we can define actions to be executed when the user interacts with the visualization.

When several Mondrian visualizations are integrated into a tool, they have to be coordinated. The selection of information in one visualization pane often filters the information or highlights an element in the other panes. In Mondrian, the coordination is achieved by sharing state between the visualizations and by updating this state through interaction handlers assigned to events. Mondrian provides a rich set of event handlers which can be conveniently assigned to nodes and edges of a visualization. The same mechanism is used to specify tooltips or to temporarily display an additional visualization when hovering over a node or edge.

To facilitate interaction in Mondrian, we implemented a subset of DOM event handlers [7]. Event handlers can be attached to the nodes of a visualization.

If an event is triggered its associated code, specified by the developer using block closures, is executed. To use interactions to coordinate multiple views, in this code, shared visualization state can be updated and the relevant visualization canvas be redrawn.

This is a very light-weight approach and may sound simple, but keep in mind that the more presumptions are made about interaction, the more limitations are introduced and as we do not know up front where our research will lead us, flexibility is to be preferred.

3. The Feature Dependency Browser

In this section we present the Feature Dependency Browser as a case study for rapid prototyping of visualizations. The Feature Dependency Browser is a tool for supporting a system engineer to understand how features in an object-oriented system depend on each other. In our previous work we proposed a new strategy to precisely detect runtime dependencies between features [4].

In the subsequent section we briefly introduce the concept of this strategy.

3.1. Detecting Runtime Dependencies

To detect runtime feature dependencies, we adopt a novel, fine-grained dynamic analysis approach, which we refer to as *Object Flow Analysis*. It captures details about how objects are referenced and how references are transferred at runtime [3].

To trace the flow of objects at runtime, we exercise an instrumented system and capture method calls, passed arguments, method returns, assignments and object instantiation. An *Alias* in our metamodel [3] represents an object reference. The aliases of an object record how it was passed through the system.

In our case studies we exercise different features of a system in a sequent row and ask how a feature depends on previous features, based on the alias dependencies between features. We detect for each object that is used in a feature whether it was passed into the feature from a previously exercised feature. For the formal definition of feature runtime dependencies and for a more

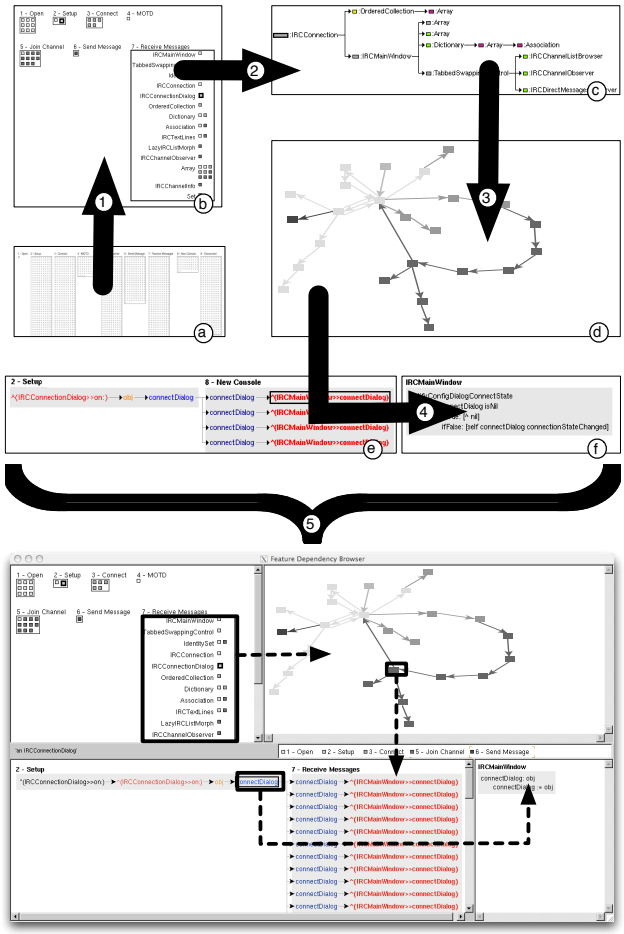

Figure 1. Above the evolution of visualizations (a-f) as happened during the research process of the final browser, each iteration cycle is depicted with a bold connector (1-5). Below the final browser, the interaction between the canvases is shown using dotted arrows.

in depth discussion we refer the reader to our previous work [4].

3.2. Iterative evolution of the browser

After completing the implementation of our research environment encompassing dynamic analysis, the metamodel and a detection strategy to identify feature runtime dependencies, we wanted to explore the results of our analysis. Our main goals were (i) to learn about the kind of dependencies we found and (ii) to investigate ways to make the resulting information accessible to a system engineer.

For our first experiment, we analyzed feature dependencies of an IRC Chat Client. We traced the execution of the program while exercising different features like Open, Setup, Connect, Join Channel, Send Message, Receive Message etc.

At the time when we investigated feature dependencies and started implementing the browser, we did not yet have a clear picture of how to meaningfully present the dependency information.

Step I. We started with a simple visualization script (listed below). It generates a labelled box for each feature. Inside the box of a feature we used small rectangles to represent its dependencies (see Figure 1-a).

```
view labelShape.
view
  nodes: self features
  forEach: [ :each |
    view rectangleShape.
    view nodes: each dependencies.
    view flowLayout gapSize: 2 ].
view horizontalLineLayout
```

Based on this initial script we evolved the Feature Dependency Browser. The evolution was an iterative process driven by new insights we gathered during experimentation phase.

Step II. The view we obtained from the script above gives us a first impression of the number and the distribution of the dependencies among the features. In a second step we extended the script to present a view which (i) focuses on exactly one selected feature at a time and (ii) aggregates the dependencies by objects and classes (Figure 1-b).

Our hypothesis was that, with only few exceptions, each object a feature depends on is referenced by another object the feature depends on. As a first attempt to verify this, we implemented Figure 1-c, a tree visualization showing the owner relationship of objects. Ownership means that all access paths of an object go through its owner. The nodes in the tree are the objects and the edges from left to right denote an owner relationship.

Although this approach is quite sophisticated, we experimented with alternatives due to the feedback we got from the software engineers we showed the tool to. They had difficulties to understand the meaning of our visualization. Going through various iterations, we eventually came up with a completely different visualization (see Figure 1-d). It shows how objects reference each other in the context of the selected feature.

This step also marks the point in the evolution of our tool where we moved from the approach of single scripts to a tool approach, which composes the views and provides interaction to change a selected feature.

The script below shows the method implementing the top left view (Figure 1-b). Basically, it is the initial

script extended with interaction (and having factored out the implementation displaying the content of a feature). Now, when clicking on a node, the block passed to onClick: is evaluated with the clicked feature as argument. In this case we update the field of the application window instance which points to the currently selected feature and then we trigger a redraw of the visualizations.

```
view labelShape.
view interaction onClick: [ :feature |
  self selectFeature: feature.
  self refresh ].
view
  nodes: self features
  forEach: [ :each |
    self displayFeature: each on: view ].
....
```

At this stage, what is missing is a way to know where the dependencies occur in the source code. What we require is a way to drill down from an object to the source code where it is used.

Since we are looking at objects which have been passed around at runtime it is not obvious how to link them to the source code. The solution we found comprises two new panes at the bottom of the browser (Figure 1-e and Figure 1-f). The left pane shows the flow of an object between and within features. The right pane shows the source code of the method in which one selected object reference (alias) in the tree is created. These two views allow the developer to investigate the exact flow of an object and see from where to where the object is passed at runtime.

As a last step we added contextual information through tooltips to various elements. For example, when moving the mouse over a node in the object dependency graph, a tooltip displays the class name of the object. Another example is the object references which show the messages that were sent through it. In Mondrian, tooltips are convenient to add as illustrated below.

```
view interaction popupText: [ :alias |
  alias receivedMessages printString ].
```

The final browser is shown on Figure 1 at the bottom, selection and composition interactions are indicated using dotted rectangles and arrows. Also, the entire evolution of the tool is summarized on Figure 1.

4. Conclusion

In this paper, rapid prototyping has been applied in the context of a software visualization tool. However, the technique presented here is not only applicable to software, but can be generalized to any research in any information space. On-the-fly prototyping is especially useful at exploring large datasets with complex structures prior unknown to the researcher. This paper shows how the researcher can adapt a visualization on-the-fly to reflect new findings and hypotheses, without the need to stop and restart the research tool. Because Mondrian provides a lightweight mechanism for generating and modifying visualizations at "view-time" –, we were able to adapt our visualization during the experimentation phase.

The prototyping techniques described in this paper reduce the need for feedback round-trips when experimenting with the development of novel visualizations. It is our conviction, that this will result in a more efficient research process. As future work, we suggest to investigate this hypothesis using empirical studies.

Acknowledgments: We want to thank Michael Meyer and Tudor Gîrba for designing and realizing the Mondrian framework. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] R. I. Bull and J.-M. Favre. Visualization in the context of model driven engineering. In A. Pleuss, J. V. den Bergh, H. Hussmann, and S. Sauer, editors, *MDDAUI*, volume 159 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[2] R. I. Bull, M.-A. Storey, J.-M. Favre, and M. Litoiu. An architecture to support model driven software visualization. In *International Conference on Program Comprehension*, pages 100–106, Athens, Greece, 2006. IEEE Computer Society.

[3] A. Lienhard, S. Ducasse, T. Gîrba, and O. Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*, pages 39–43, 2006.

[4] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC'07)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[5] M. Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, Nov. 2006.

[6] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[7] World Wide Web Consortium. *Document Object Model DOM Level 2 Events Specification*, 1998.