

SHriMP Views: An Interactive Environment for Exploring Multiple Hierarchical Views of a Java Program

Margaret-Anne Storey

Department of Computer Science
University of Victoria, Victoria BC
email:mstorey@uvic.ca

<http://www.csr.uvic.ca/shrimpviews>

1 Introduction

This position paper describes the SHriMP visualization tool and how we are customizing it for browsing Java programs. Available visualizations include views of the Java source code, generated Java documentation, personalized annotations and several architectural views. The architectural views, displayed graphically as nested graphs, provide an interface through which the textual lower-level views can be accessed and explored.

The goal of combining graphical high-level views with textual lower-level views is to ease navigation within large and complex software programs. We conjecture that using an architectural view as the navigation structure will make it easier for a programmer to build a mental model of an unfamiliar program. We are currently implementing user studies to test this hypothesis in addition to actively seeking feedback from other researchers developing related tools and technologies.

SHriMP has been redesigned using Java beans. The new component based design facilitates data integration and control integration so that it can more easily interoperate with other tools. An active part of our research is exploring the integration of other views within SHriMP to investigate if these combinations can aid software maintenance.

The rest of the position paper is organized as follows. The next section provides some background on the SHriMP tool. Section 3 presents a scenario of how SHriMP can be used for browsing and exploring a Java program. Section 4 describes our current work and reviews the main research questions we are considering.

2 Background

The SHriMP (Simple Hierarchical Multi-Perspective) tool provides a customizable and interactive environment for navigating and browsing complex information spaces. The latest prototype, described more thoroughly in [1], utilizes knowledge gleaned from previous prototypes [2,3] and empirical evaluations [4,5].

The primary view in SHriMP uses a zoom interface to explore hierarchical software structures. The zoom interface provides advanced features to combine a hypertext-browsing metaphor with animated zooming motions over nested graphs [2]. Filtering, abstraction and graph layout algorithms are used to reveal complex structures in the software system under analysis.

SHriMP uses nested graphs to represent software hierarchies. For example, a Java program's architecture can be visualized using its package and class structure. A package may contain other packages, classes, and interfaces. Classes and interfaces may contain attributes and operations. This hierarchical structure is represented using a nested graph with the *parent-child* relationship showing subsystem containment (see Fig. 1). However, other relationships, such as inheritance could alternatively be used for the parent-child relationship. In this case parent nodes would represent superclasses, with their embedded

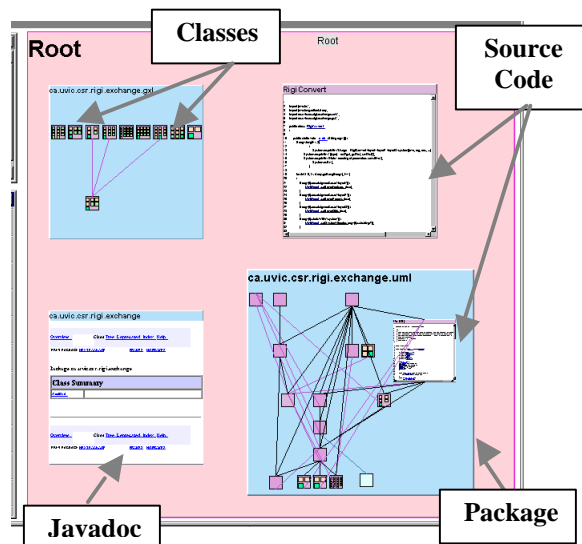


Figure 1. A SHriMP view of a Java program showing source code, Javadoc and architectural views.

children nodes representing subclasses.

Other relationships are directly visible as coloured arcs layered over the nested graph. In Fig. 1, the coloured arcs represent relationships such as *extends* (when one class extends another class using inheritance), *implements* (when a class implements an interface) and *hasType* (when a class uses an object of a type). Composite arcs are abstractions of multiple arcs at lower levels in the hierarchy.

SHriMP employs a fully zoomable interface for exploring software. This interface supports three zooming approaches: *geometric*, *semantic* and *fisheye* zooming [5]. Geometric zooming allows the user to scale a specific node in the nested graph while eliding information in the rest of the system. Fisheye zooming allows the user to zoom on a particular piece of the software, while simultaneously shrinking the rest of the graph to preserve contextual information. Semantic zooming will cause a particular view to be displayed inside a node depending on the task at hand. A node representing a Java package may display its children (packages, classes, and interfaces), its Javadoc or some other user defined view. A node representing a class or interface may display its children (attributes and operations) or it may display the corresponding source code. SHriMP determines which view to show according to the action that initiated the zoom action. For example, if a user clicks on a link within a Javadoc view, SHriMP will zoom to the appropriate destination node and display Javadoc within that node. The next section describes a scenario of how SHriMP may be used for browsing a Java program.

3 Exploring a Java program

In this scenario, we show how SHriMP may be used for exploring a Java program. The following views show the architecture, code and documentation of the SHriMP program itself. The SHriMP program is constantly evolving with new students joining the project every few months. Newcomers to our project are encouraged to use SHriMP to navigate the SHriMP code and architecture as a kind of introspective case study. Figure 2 shows an architectural diagram showing the main packages in SHriMP as well as the system packages in the standard Java libraries that are used by SHriMP.

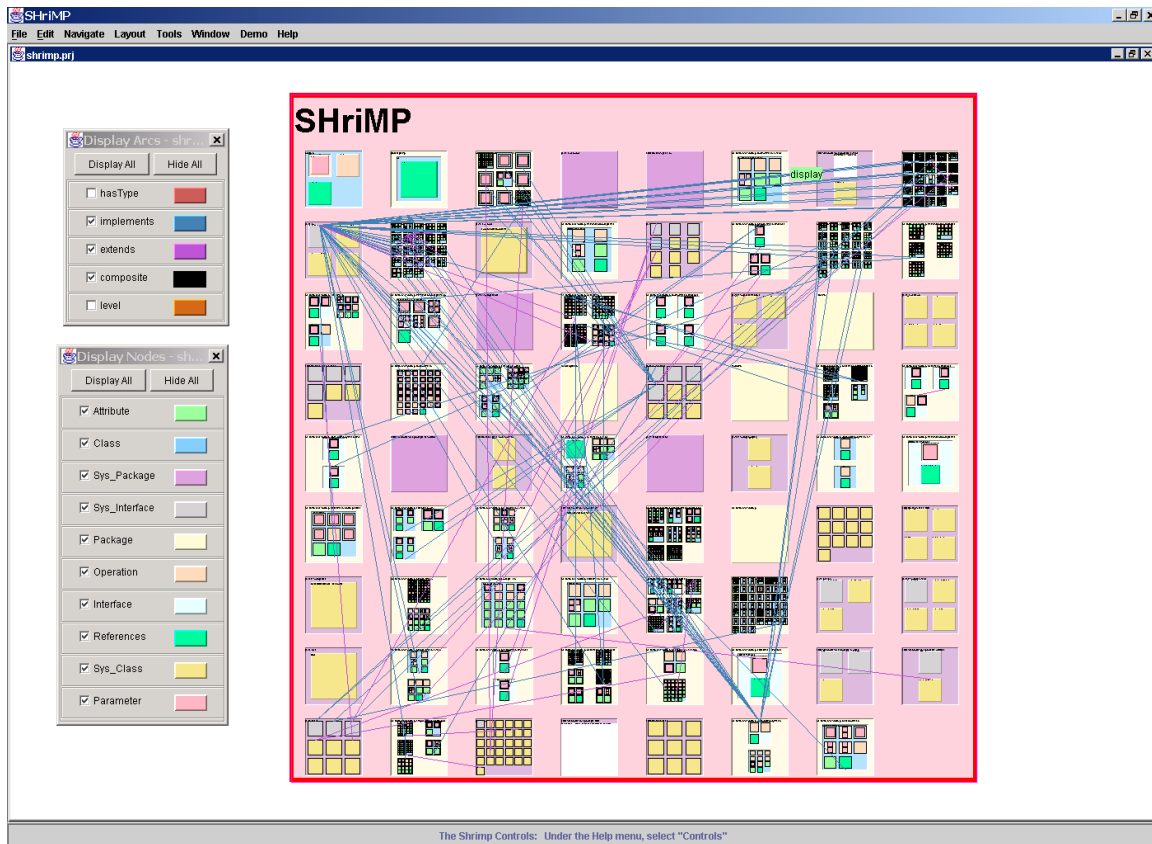


Figure 2. A SHriMP view showing the architecture of the SHriMP program itself. Only some relationships are layered over the nested graph (extends and implements relationships).

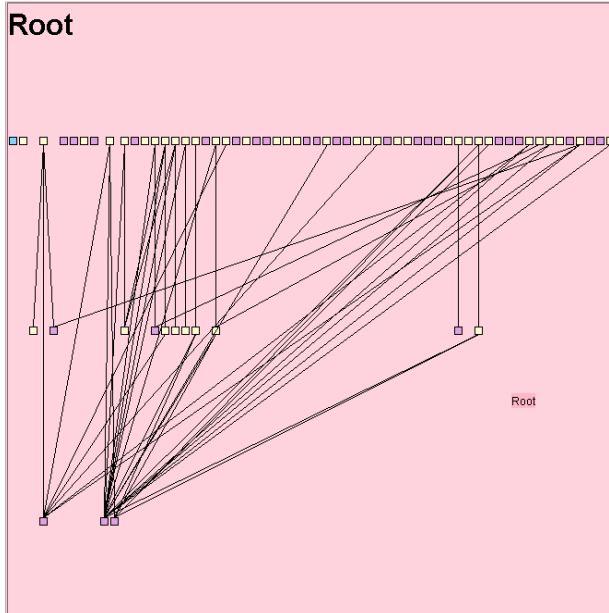


Figure 3: A tree layout showing the main inheritance and implements relationships between the packages, classes, and system packages in SHriMP.

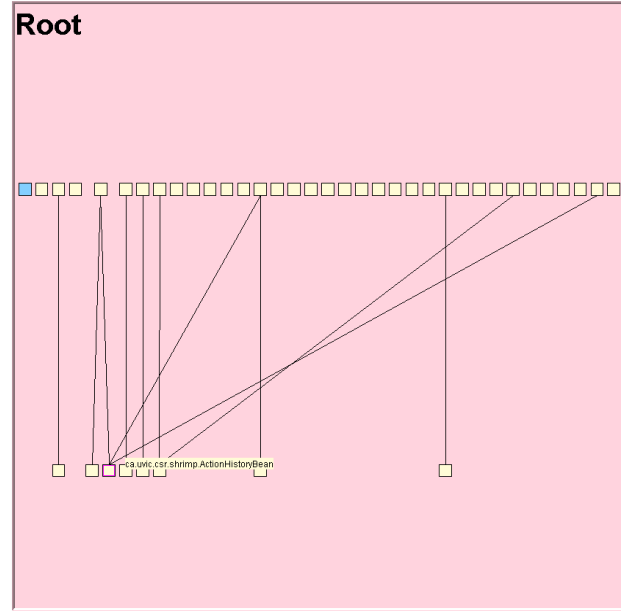


Figure 4: In this view, the system packages have been filtered, and the tree layout has been reissued.

Next (as shown above in Fig. 3), the programmer decides to elide the children of the top level nodes to focus on the high level architecture of the system. The “hastype” relationships were previously filtered, so a tree layout based on the “extends” and “implements” relationships is issued. Since a newcomer may be more interested in the SHriMP packages and classes (as opposed to dependencies with the Java libraries), the user may now choose to filter the system packages and explore only the “extends” and “implements” relationships between the SHriMP packages. The resulting view is shown in Fig. 4.

In the next view (see Fig. 5 opposite) the programmer has decided to explore more detail about certain key packages and classes in the program. First one package is zoomed to show the Javadoc and then next a class within another package is opened to show the source code. The user can zoom on any node to view it in full screen mode. The source code and Javadoc contains embedded hyperlinks which may be navigated. When a link to a chunk of code or Javadoc inside another node is activated, the view animates so that the destination node comes into focus. Back and Forward actions are supported enabling the architectural view within SHriMP to act as a simple web browser for exploring the HTML’ized source code and documentation. Although this interface may seem very different to standard approaches, we observed in our user studies that most users quickly adapted to it and liked navigating the program in this fashion. However, there are still many open questions yet to be answered. These are discussed in the next section.

4 Current Research

By embedding code and documentation views within graphical high-level views, a user is able to effortlessly switch navigating between the software architectural views and the source code and documentation textual views. However, it is not clear if such an

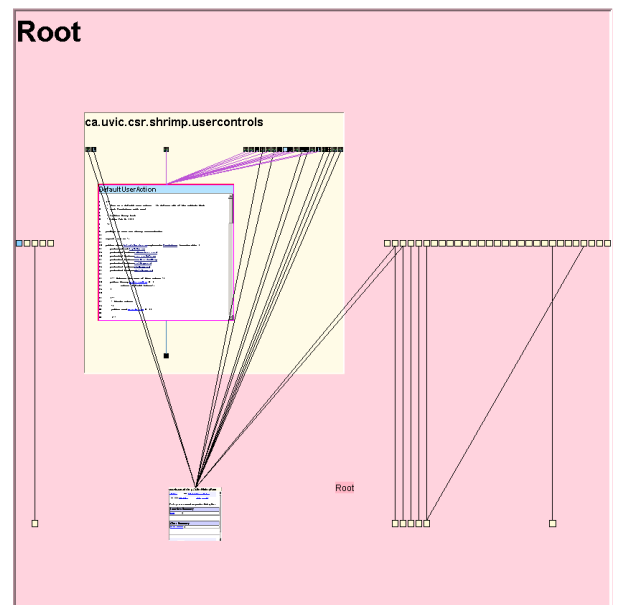


Figure 5: Here the user is exploring more detail in the SHriMP program by browsing the Javadoc of a package (shown bottom left) and the source code of a class.

interface will be of use to programmers throughout the entire development life cycle or if the technique will scale to larger systems. We (as well as other researchers) have observed that expert programmers build mental models of the programs they are developing and quickly internalize these diagrams. They seem to have little need to have an external image of these models. However, newcomers to projects have to rely on conversations with the experts to learn these models, or they have to painstakingly browse the code building abstractions in their head as they go. Accurate documentation is generally not available. We conjecture that the SHriMP interface would be very useful for individuals new to a project. We have also observed that these visualizations are useful for managers wishing to gain an overview of the complexity and size of a system. From our demonstrations at conferences, we have had many requests to use SHriMP as a general interface for a software development environment. That is, to allow programmers to edit code directly instead the nodes and to recompile from within SHriMP. Other requests have been made to incorporate dynamic analysis and incorporate a debugger within SHriMP. Apart from these requests and anecdotal comments, we have no concrete evidence to indicate that SHriMP would be useful as an interface for a software development environment. We intend to explore these possibilities further in the near future.

The performance of our prototype is still a concern. Although, we have not expended much effort in improving the efficiency of our tool to date, it does scale to our own project (about 30,000 LOC) and is very interactive for this size and smaller sized projects. If the results from our current user studies are favourable, we will turn our attention to improving the performance of our tool.

With any visualization system, there are many human computer interface issues at stake. We are currently striving to improve the usability of our tool and so consequently we are evaluating it in user studies. However, the more visualization features we add, the more complex the user interface becomes. We are exploring ways of keeping the user interface simple to use and easy to learn for novice users, and yet powerful and flexible for more expert users.

The Java Bean technology [6] has so far proved to be an effective component-based approach for developing an extensible and flexible tool. Different views and features in SHriMP have been developed using Java Beans. A variety of Java components can be composed into customized applications by other tool designers. Moreover, this approach allows other researchers to integrate single components from SHriMP within their own environments. For example, SHriMP has already been successfully added to the Protégé-2000 tool [7] as a plug-in component for visualizing ontologies in the health domain. As mentioned above, we have had numerous requests to embed editors within the SHriMP nodes as well as integrate dynamic information such as debuggers within SHriMP. The component based design will allow us to rapidly incorporate new functionalities and tools within SHriMP. However, it is not clear at this point whether such functionalities would be useful if deployed within the SHriMP environment. These new features in addition to other issues will be explored over the coming months. In the meantime we are very interested in feedback from researchers working on related problems and tools.

Further details about SHriMP and related projects are described at <http://www.csr.uvic.ca/shrimpviews>.

Acknowledgements

This research was supported in part by B.C. ASI, CSER, NSERC, IBM, Darpa, Stanford University and the University of Victoria. SHriMP uses the Jazz zooming library developed at the University of Maryland.

References

1. J. Michaud, M.-A. Storey and H. Muller. Integrating Information Sources for Visualizing Java Programs. Submitted to the *IEEE Conference on Software Maintenance*, 2001.
2. M.-A. Storey, H. Müller and K. Wong. Manipulating and Documenting Software Structures. In *Software Visualization*, pages 244-263. World Scientific Publishing Co., 1996.
3. J. Wu and M.-A. Storey. A Multi-Perspective Software Visualization Environment.. In *Proc. of CASCON'2000*, November 2000.
4. M.-A. Storey et al. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proc. of WCRE'96*, pages 31-40, Monterey, USA, Nov 1996.
5. M.-A. Storey, K. Wong and H. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proc. of WCRE'97*, pages 12-21, Amsterdam, October 1997.
6. Sun Microsystems. *JavaBeans API specification*, Version 1.01, <http://java.sun.com/beans>, 1997.
7. W. Grosso, H. Eriksson, R. Ferguson, J. Gennari, S. Tu and M. Musen. Knowledge Modeling at the Millenium (The Design and Evolution of Protégé-2000), Stanford University.