SHriMP Views: An Interactive Environment for Exploring Java Programs

Margaret-Anne Storey Casey Best Jeff Michaud
Department of Computer Science
University of Victoria, Victoria BC
email: {mstorey,cbest,jmichaud}@csr.uvic.ca

Abstract

This paper describes a demonstration of the SHriMP visualization tool. SHriMP provides a flexible and customizable environment for exploring software programs. It supports the embedding of multiple views, both graphical and textual within a nested graph display of a program's software architecture. SHriMP has recently been redesigned and reimplemented using Java Bean components. These APIs allow SHriMP to be easily integrated with other software understanding tools. In this demonstration, SHriMP is used for exploring and browsing Java programs.

1. Introduction

There are many visualization tools that have been developed to help in understanding software. Some are powerful at solving a variety of software maintenance tasks, but their closed environments make it difficult to create a customized toolset to meet particular end users' requirements. To remedy some of the problems with existing tools, we have developed SHriMP (Simple Hierarchical Multi-Perspective) views, a Java program for software visualization. The latest prototype, described more thoroughly in [1], utilizes knowledge gleaned from previous prototypes [2,3] and empirical evaluations [4,5].

The Java Bean technology [6] provides an effective component-based approach for designing an extensible tool. Different views and features in SHriMP have been developed using Java Beans. Thus, 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.

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 and 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.

The next section in this short paper describes how SHriMP may be used for browsing Java programs.

2. Exploring Java Programs

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 relationships and is fully configurable by the end user. For example, parent nodes would represent superclasses, with their embedded children nodes representing subclasses.

Additional relationships are visualized using arcs layered over the nested graph. In Fig. 1, coloured arcs are used to 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). In SHriMP, composite arcs are higher level abstractions of arcs attached to nodes at lower levels. A composite arc can be expanded to show the constituent arcs it represents.

SHriMP employs a fully zoomable interface for exploring software. This interface supports three zooming approaches: *geometric*, *semantic* and *fisheye* zooming [5]. A user browsing a software hierarchy may combine these approaches to magnify nodes of interest. Geometric zooming is the simplest type of zooming. A part of the nested view is simply scaled around a specific point in the view. Geometric zooming elides information in the rest of the system. Fisheye zooming allows the user to zoom on a particular piece of the software, while preserving contextual information.

SHriMP also provides a semantic zooming method. When magnified, a selected node will display a particular view depending on the task at hand. For example, when zooming on a node representing a Java package, the node may display its children (packages, classes, and interfaces). Alternatively, it may show its Javadoc, if it

0-7695-1131-7/01 $10.00 © 2001 IEEE

exists. Other possible views may include annotation information, code editors or other graphical displays. A node representing a class or interface may be 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 node and display Javadoc within that node (see Fig. 1).

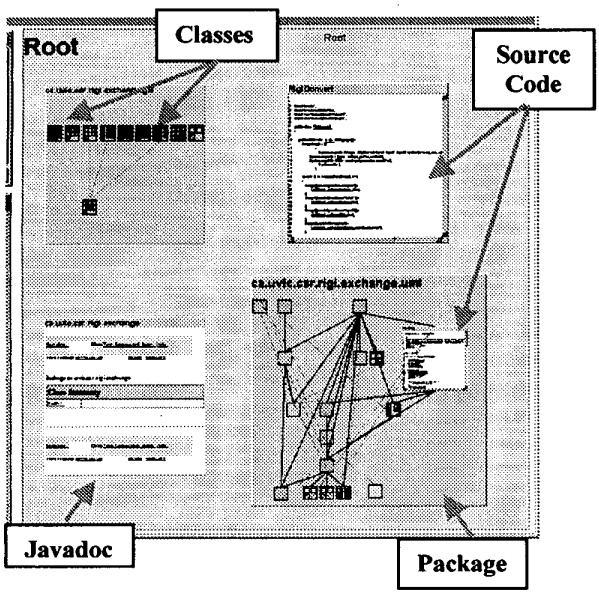

Figure 1. A SHriMP view of a Java program. Three of the displayed nodes (top left, bottom two) show packages in the program. The top left and bottom right nodes are opened to show the classes and interfaces in these packages. The bottom left node shows the Javadoc for that package. The top right node shows the source code for that class.

Searching is a critical activity for programmers trying to build mental models of a program's source code. Searching is used for tracing data accesses and method calls. Although users are able to browse source code and documentation via embedded hypertext links in SHriMP, a powerful searching tool is also provided to allow programmers to find software artifacts and symbols quickly and verify hypotheses easily. The searching tool in SHriMP supports three searching strategies: *General Search*, *Artifact Search* and *Relation Search* [5]. Regardless of the specified search category, all the search results are organized into a selectable list according to node names. For every selected result, a user can press a "**Browse**" button to magnify the corresponding node in the primary SHriMP view.

By embedding code and documentation views within graphical architecture level views, a user is allowed to examine both the software architectural views and source code or documentation simultaneously.

3. Current Research

This proposal describes the latest prototype of SHriMP. We are applying SHriMP to SHriMP's source code during its own development as a type of introspective case study. We intend to use SHriMP to document and capture the evolutionary design process that we are following. In addition, further user studies to evaluate the latest prototype are planned for Spring 2001.

There are many reverse engineering and reengineering tools in development. Closer collaborations between research groups will lead to better tools in shorter periods of time. To this end, we have reimplemented SHriMP using a component-based technology, thereby allowing other researchers to use one or more of the SHriMP views in their own tools. In addition, we can import other views and display them inside SHriMP's nodes as shown in Fig. 1. Part of SHriMP has already been successfully added to the Protégé-2000 tool [7] as a plug-in component for visualizing various ontologies in the health domain. Further details about the SHriMP projects are available at http://www.csr.uvic.ca/shrimpviews.

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. Fergerson, J. Gennari, S. Tu and M. Musen. Knowledge Modeling at the Millenium (The Design and Evolution of Protégé-2000), Stanford University.