

# A Multi-Perspective Software Visualization Environment

Jingwei Wu  
*Dept. of Computer Science*  
*University of Victoria*  
*Victoria BC, Canada V8W 3P6*  
*+1 250 472 4604*  
*jingwei@uvic.ca*

Margaret-Anne D. Storey  
*Dept. of Computer Science*  
*University of Victoria*  
*Victoria BC, Canada V8W 3P6*  
*+1 250 721 8796*  
*mstorey@uvic.ca*

## Abstract

This paper describes a multi-perspective software visualization environment, SHriMP, which combines single view and multi-view techniques to support software exploration at both the architectural and source code levels. SHriMP provides three different views: a primary nested view and two subsidiary views. The primary nested view employs fisheye views of nested graphs, provides contextual cues, and supports general exploration activities. In SHriMP, subsidiary views exist as a searching tool and a relation tracer. These views complement each other and allow programmers to examine a software system from multiple perspectives.

## Keywords

Software visualization, program understanding, multi-perspective, nested graphs, SHriMP views.

## 1 Introduction

Software maintenance accounts for over 70 percent of a software system's lifetime cost [1]. Enhancing the efficiency of software maintenance may bring huge economic benefits. Many software visualization tools were developed for reducing the time spent in program comprehension [2, 3, 4]. However, the majority of them are not very effective in practice. Part of the problem is that the collaborative use of various visualization techniques has not been addressed appropriately.

Quite a variety of software visualization tools choose to represent software architectures in a single view or multi-view style [5]. Either one can be used to create views at different levels of abstraction. However, both have limitations. Using multiple views, programmers need to conceptualize and trace implicit relationships among different individual windows. This kind of activity may cause programmers to become disoriented even if an overview window is provided. A single view technique often limits a user's capability to look at several disjoint parts of a software system at the same time.

Many software visualization tools are criticized for lacking appropriate support for source code browsing and symbol searching [6]. Some studies have demonstrated that code-related browsing and searching activities account for the majority of the time spent by programmers [6, 7].

Many software visualization tools are closed systems that integrate a variety of visualization techniques [8]. These systems may be powerful at solving a variety of software maintenance tasks, but their closed environments make it difficult to create a customized toolset to meet the end user's requirements. Additionally, some useful techniques in a closed environment may not be easily exported and integrated into other systems. However, an open system can not be built by simply glueing a set of tools together. Rather, complementary tools that are capable of solving a variety of tasks at different levels of abstraction should be carefully selected.

To remedy some of the problems with existing software visualization systems, we have developed SHriMP (Simple Hierarchical Multi-Perspective) views, a Java program for software visualization.

The prototype, described in this paper, utilizes knowledge gleaned from previous prototypes [5, 9] and empirical evaluations [10, 11]. SHriMP is designed to be a multi-perspective visualization environment, integrating three distinctive views. Each of these views is an independent software component that can be reused in other systems.

The rest of this paper is organized as follows. Section 2 describes the approach we followed for developing a software visualization tool with multiple perspectives. Section 3 discusses the primary view in SHriMP, which employs a zoom interface to support animated exploration over hierarchical structures. Section 4 describes two subsidiary views that demonstrate how to apply searching strategies. The final section presents our conclusions and outlines future work.

## 2 Multiple Perspective Views

The single view and multi-view techniques are two traditional ways for developing software visualization tools. Both approaches have benefits and limitations when dealing with large, complex software systems. In this section, we will describe our proposed approach for software visualization. It combines both the single view and multi-view techniques. Therefore, it is more suitable for large structured information spaces. We are currently experimenting with it in SHriMP.

### 2.1 Dealing with Views

Storey proposed a taxonomy for classifying program comprehension tools [12]. This taxonomy differentiates how views are organized and how context and detail of an information space can be coordinated. The taxonomy has four categories: *Multiple views without context*, *Multiple views plus context and detail*, *Single view without context*, and *Single view plus context and detail*.

Multi-view techniques use multiple windows to display a graph that is composed of software artifacts and relationships. Each window shows a distinct part of the graph or graph abstraction. Opened windows are usually arranged in a cascading style, thus making it easier for a user to examine a particular view. For example, Rigi is a multi-view system. It provides an overview window that displays contextual cues to guide users as they fetch partial views of an underlying system [14]. Details are made available in partial

views. One of the major limitations of multi-view techniques is that implicit relationships among individual views are difficult to conceptualize [5].

Single view techniques use a single window to display a graph either statically or dynamically. A static view usually focuses on some part of the graph, but it cannot be changed. In contrast, a dynamic view allows users to change their foci through interaction. In both cases, users rely on either auxiliary techniques, such as panning and zooming to explore the view, or they adjust the graph layouts to see both context and detail. A good example is the graphical fisheye view technique through which a user can change the view to see the details and maintain the contextual cues simultaneously [21]. Single view techniques also have limitations. For a large software system, with millions of lines of code, a single view may be overloaded with too much information and therefore fail to provide appropriate context. In addition, it is extremely difficult to provide distinct views, such as call graphs and class diagrams, within a single view simultaneously. Even if it were possible, it would be confusing and difficult to use.

There is no single panacea [14]. Both approaches can be used to complement each other and to support a wider variety of tasks. Multiple views at higher levels of abstraction can reduce information overload in each individual window. However, a single view technique may be more applicable when viewing details at lower levels of abstraction. Single view techniques can also be augmented to support *context+detail* exploration. A single technique can not meet all possible requirements and a tradeoff has to be made to integrate multiple techniques.

### 2.2 The Rigi Approach

The Rigi system nicely illustrates how a multi-view system works [13]. Rigi extracts software artifacts and organizes them into a layered graph. Every piece of information at a specific layer can be graphically displayed in a separate window. Users can continually open new windows to examine more detailed information until an empty window is reached and no further exploration is possible.

This multi-view approach relies on a single visualization technique to manage each separate view. Although distinct views can be generated in

Rigi, they are difficult to relate to one another. In addition, views are spread all over the screen and difficult to locate if too many views exist. Rigi examines software systems with *a priori* emphasis on distinct parts although it does support multiple perspectives.

Figure 1 describes a multi-view visualization system. The underlying data model stores extracted software artifacts and relationships. For a specific task, a user can extract the required information and create a separate view through the extraction and creation layer. For each separate view, a new window is opened for further exploration.

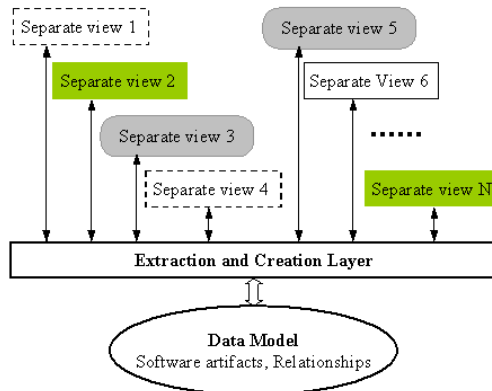


Figure 1: A conceptual architecture of multi-view visualization systems (Shapes and colors are used to indicate distinct kinds of views.)

### 2.3 A Multi-Perspective Approach

We believe that a software visualization system should make it possible for users to create intuitive views from multiple **task-relevant** perspectives. Here, a *perspective* is a mental view of some kind of task, software artifact or relationship. It relies on a specific kind of visualization technique that intuitively reveals how software artifacts relate to one another. Furthermore, a perspective should address and facilitate a series of tasks, not just one in particular. General perspectives are reflected in widely used diagrams of data flow, call sequence, class inheritance, etc. These diagrams have their own specification techniques and appearances.

To address a particular perspective, a well-defined module component or can be developed independently and then integrated into an open visualization system. Each component functions

as a tool and has adequate expressiveness. Each tool can manage multiple views of the same type with one or more views in the foreground of the display. Views in the background can be switched to the foreground at the request of a user. With a perspective in mind, a user can navigate through different types of views easily. Figure 2 describes a simple conceptual architecture of visualization systems that support multiple perspective views.

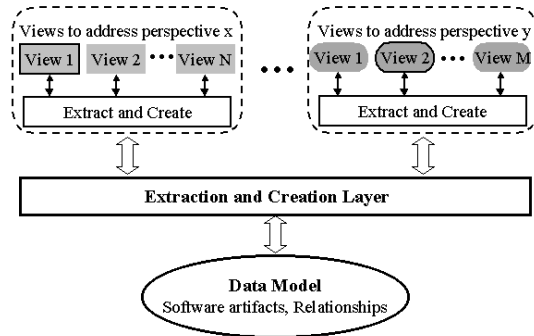


Figure 2: A conceptual architecture of multi-perspective visualization systems (The views with black borders are in the foreground.)

Each perspective view has its own set of specifications, requires special kinds of software artifacts, exists under certain conditions, and solves part of the task of program comprehension. No perspective view is capable of supporting software exploration all by itself. A variety of perspective views are necessary for program comprehension. For the purpose of coordination there should exist a primary view that provides programmers with guidance and contextual cues for exploring subsidiary perspective views. The primary view is of central importance to the entire system as it supports general exploration activities.

In a multi-perspective visualization system, a programmer can rely on a primary view to apply general comprehension strategies such as a top-down strategy [22]. When reaching a particular software artifact or handling a specific task, the programmer can open subsidiary views for further exploration. Using various views effectively may allow easy switching of program comprehension strategies [26]. The primary view in SHriMP is a nested graphical view, and subsidiary views exist as additional tools such as a relation tracer and a searching tool.

### 3 The Primary View in SHriMP

The primary view in SHriMP uses a zoom interface to explore hierarchical structures. The zoom interface provides advanced features to combine the hypertext-browsing metaphor and animated zooming motions over nested graphs [5]. Other features such as filtering, abstraction and graph layout algorithms are provided to reveal complex structures. Figure 3 shows a nested view of the `gui` (graphical user interface) package in SHriMP.

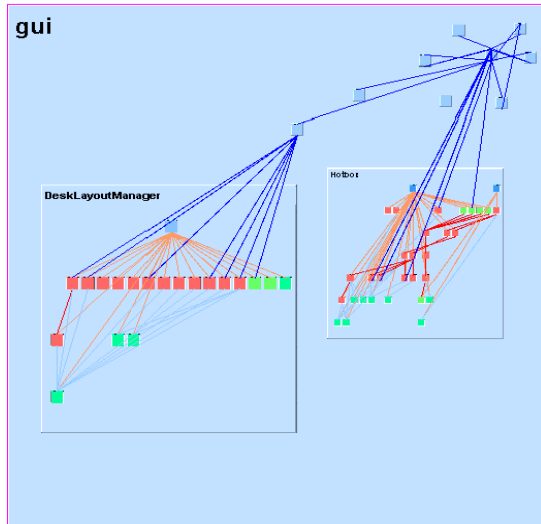


Figure 3: A nested view of the `gui` package in SHriMP. A user issued a spring layout [23]. Two classes (represented by intermediate nodes) are magnified using the fisheye-view algorithm [15]. The contents of the two nodes were drawn using the Sugiyama layout algorithm [24].

#### 3.1 Nested Graphs

One of the important characteristics of software is that a system can be decomposed into subsidiary systems recursively. This process is often referred to as system decomposition, and it finally results in a software hierarchy with a nesting relationship of subsystem containment. SHriMP uses nested graphs to represent software hierarchies.

For example, a software system can be divided into modules, modules into files, and files into variables, functions and blocks. This hierarchical structure can be represented using a nested graph with the *parent-child* relationship of subsystem containment. Other relationships, such as function

calls, can be treated as parent-child relationships. For example, in a nested graph, the parent node represents the calling method, with its children nodes representing the called methods.

The relationships that are not relevant to a nested graph's hierarchy are represented as simple arcs in that graph. In SHriMP, composite arcs are used to represent multiple simple arcs. Actually, a composite arc is a higher level abstraction of the arcs attached to nodes at lower levels, implying how software artifacts relate to one another in a nested hierarchy. A composite arc can be opened to show the constituent arcs it represents.

#### 3.2 A Zoom Interface

SHriMP employs a fully zoomable interface for exploring software. This interface supports three zooming approaches: *geometric*, *semantic* and *fish-eye* [25]. A user browsing a software hierarchy may combine these approaches to magnify nodes of interest.

Geometric zooming is the simplest method of zooming. A nested view is simply scaled around a specific point in the view. This zooming method does not convey any semantic meaning and may cause frustrations in an interactive environment. When inspecting a point not visible in the current view, a user has to zoom out to locate that point and then zoom in for examination. This zooming method is very slow for inspecting two separate points far away from each other.

SHriMP provides a semantic zooming method. When being magnified, a selected non-leaf node will open to display its children nodes if they are not already visible; a selected leaf node will open to show the related code segments or annotations automatically. In addition, the magnified node will fit within the current view window via animated motions.

Fisheye zooming is a *context+detail* approach for magnifying nodes of interest. It is based on the SHriMP fisheye-view algorithm [15], in which a *focal* node grows by forcing its *sibling* nodes to decrease in size accordingly. This algorithm can preserve constraints such as orthogonality and topology as nodes are resized. Fisheye zooming has the advantage of showing both context and detail, but depending on the given task and the required information, contextual cues may not always be needed.

## 4 Subsidiary Perspective Views

In SHriMP, two subsidiary views, a searching tool and a relation tracer, are provided to complement the primary view [25]. A programmer can apply different comprehension strategies through using subsidiary views collaboratively. For example, call graphs and data flow diagrams are helpful for applying goal-directed, hypothesis-driven program comprehension strategies.

### 4.1 Searching

Searching provides a very useful perspective for programmers. Searching activities can reflect how a programmer traces software artifacts and finds implicit clues to form mental models. One of the classical uses of searching is to trace data accesses and method calls.

The primary nested view in SHriMP is by itself insufficient to support bottom-up comprehension strategies [19], even though programmers are able to browse source code and documentation via embedded hypertext links. The programmer needs a searching tool to search for software artifacts and symbols and to verify hypotheses quickly.

#### 4.1.1 Review of Searching Tools

Since searching is one of the major activities in software exploration, many searching tools have been developed to help programmers. Searching tools can be classified into four categories:

**Grep:** The *grep* tool [16] is one of the standard tools supplied with the UNIX operating system. It is perhaps the most widely used searching tool for programmers. It provides a command line interface to search through files for arbitrary strings and regular expressions without considering their semantic meaning. Normally, each line found is copied to standard output. Interpreting the search results may be very time consuming for some programmers. It is also difficult to use *grep* tools to trace relationships among software artifacts.

**Searching facilities of text editors:** Most text editors, such as *Notepad*, *WinWord* and *Emacs*, have some kind of support for finding arbitrary strings and regular expressions. Like *grep*, these tools rarely take into consideration semantic information. Furthermore, a user often requires repeated operations to locate a specific variable or function definition. However, their user interfaces

are simpler and easier.

**File finding tools:** The *find* tool [16] is one of the standard tools shipped with UNIX. This tool allows a user to search recursively through a list of directories for files that match an acceptable logical expression. In addition, it permits a user to specify which attributes can be associated with the found files. Despite its lack of a graphical interface, the *find* tool is much more configurable and powerful than the *File Finder* tool on the Windows operating system. The *find* tool is used a lot by UNIX programmers.

**Searching facilities of program understanding tools:** Some program understanding tools such as the *Searchable Bookshelf* [3, 6] and *TkSee* [7, 17] are capable of searching for arbitrary strings, regular expressions, and software artifacts with semantic meanings. According to Lethbridge, a code exploration tool should meet two functional requirements: (1) a user should be able to search for software artifacts by arbitrary strings or by regular expressions, and (2) the tool should be able to display some relevant attributes of the retrieved artifacts [7]. The tool should also meet a set of non-functional requirements such as dealing with millions of lines of code. Other relevant requirements are described in more detail in [7].

#### 4.1.2 The Searching Tool in SHriMP

The searching tool in SHriMP supports three searching strategies: *General Search*, *Artifact Search* and *Relation Search*. The General Search strategy provides a searching facility for arbitrary strings and regular expressions. The second strategy, Artifact Search, allows a user to search for a particular kind of software artifact. The third strategy, Relation Search, makes it possible to find relationships among software artifacts. All these searching strategies support incremental search over the current search results. That is, a user can refine search conditions and then search through the current search results again.

The general search is similar to the searching facilities of a text editor. A user can enter a string or a regular expression as the current search pattern and indicate the current search category (for example, search by node names, search in the source code or in the documentation). Since SHriMP uses a nested hierarchy to organize information, each node is treated as a collection of information, and has its own identification, name and pieces of source code and documentation.

Regardless of the specified search category, all the search results are organized into a selectable list according to node names (see Fig. 4). For a selected result, a user can press the “**Browse**” button to magnify the corresponding node in the primary view. Thus, the user can examine that node in more detail. This capability lets a user perform arbitrary navigation easily by choosing a search result.



Figure 4: General search. Search for nodes whose names match the regular expression of *Err\**.

The artifact search enables a user to find a variety of software artifacts according to certain requirements. SHriMP uses an in-memory data model to organize a variety of software artifacts extracted using certain parsers. For example, a C program parser from Rigi [13] can extract different types of software artifacts and their relationships. Software artifacts in C programs can be classified into categories such as *file*, *unknown*, *globalvar*, *collapse*, *proc* and *usertype*. Their relationships can be abstracted as *infile*, *access*, *call*, *composite*, etc. The artifact search allows a user to search for software artifacts using arbitrary strings or regular expressions, but the type information has to be provided. The search results are displayed in a table with four columns (*Artifact Name*, *Source File*, *Line Num* and *URL File*). Each column refers to a software artifact attribute (see Fig. 5).

The first two strategies are useful for finding software artifacts and string symbols. However, they are limited at revealing relationships among software artifacts. To solve this problem, SHriMP provides a relation search strategy (see Fig. 6). Using this strategy, a user only needs to choose a relation type and then enter search patterns for the source and destination nodes of the relationships of interest. Each relationship found contains its type

information, the name of its source node and the name of its destination node. The result table uses five columns (*Relation Type*, *Src Artifact*, *Src File*, *Dest Artifact* and *Dest File*) to describe basic relationship attributes.

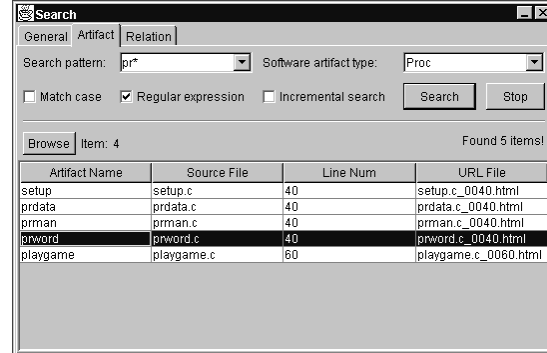


Figure 5: Artifact search. Search for *Proc* software artifacts whose names match *pr\**.

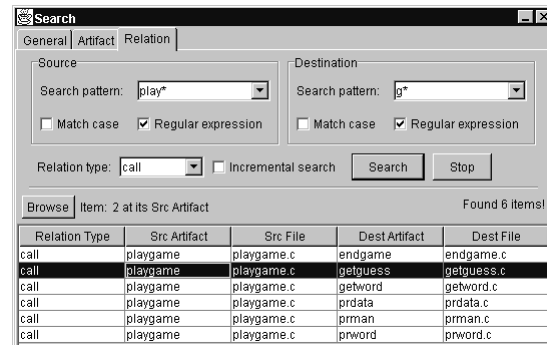


Figure 6: Relation search. Search for relationships that must be of type *call*. Each found relationship must have its source node match *play\** and have its destination node match *g\**.

## 4.2 Tracing Relations

When comprehending programs, a programmer often needs to search for various relationships, such as data access and method calls. The relation tracer in SHriMP is based on the relation search strategy, and it provides graphical representations to facilitate a user’s exploration of relationships.

### 4.2.1 Relation Tree

Views displayed in the relation tracer tool use horizontal tree layouts. Like the primary view in SHriMP, a relation tree deals with hierarchical

software structures, but it does not use a nesting presentation style. Nodes are simply linked by arcs to form a flat tree. When constructing a relation tree, a user can click on a node of interest to show its children or collapse its subtree. The user can also select a tree node and then zoom into the primary view to examine its detailed information. Figure 7 shows a screen snapshot of the relation tracer displaying a relation tree.

The default nesting relationship of a primary view is subsystem containment, which is applied statically through the entire hierarchy. However, the relation tracer can be used to explore other relationships. For example, in Figure 7, a user starts with the node of *main.c* that is linked to the other five nodes via the relationship of *infile*. If the user wants to explore the call hierarchy rooted at the node of *main*, (s)he can choose the tracing forward direction and the *call* relationship and then click the *main* node to find all the related function calls.

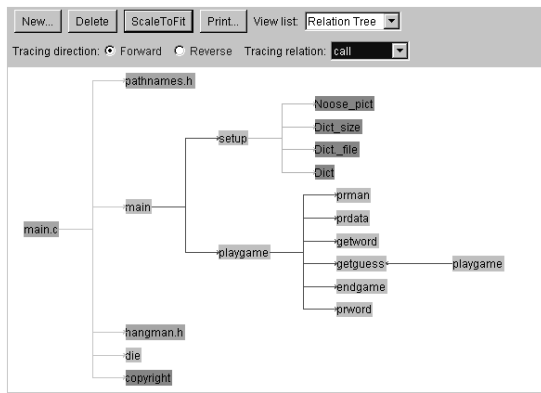


Figure 7: The relation tracer in SHriMP. A user is trying to find a complete *call* and *data access* hierarchy rooted at the root node of *main.c*.

The tracing direction can be specified as *forward* or *reverse*. If the forward direction is chosen, the underlying relation search strategy will find the destination nodes that have the specified relationship with the selected source node. If the reverse direction is used, the search strategy will find the source nodes that relate to the selected destination node via the specified relationship. The relation tracer allows a user to choose a tracing relation and a tracing direction. It is much more flexible and useful for exploring program dependencies, such as a call hierarchy, than the

primary view in SHriMP.

Because a relation tree is able to reveal program dependencies among various software artifacts, a programmer may use it to perform tasks relevant to program slicing. According to Weiser, program slicing is a method for decomposing programs by analyzing their data flow and control flow [18]. A program slice is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior [18]. For example, a call hierarchy is a syntactic slice that recursively finds the methods called by or calling a given method. The syntactic criteria for creating a call hierarchy consists of a root method and a call relationship.

#### 4.2.2 Managing Views

During program comprehension, a programmer may repeatedly reference preexisting knowledge or mental models. The relation tracer is capable of maintaining multiple relation trees. This makes it easier for a user to cross-reference views of previously constructed mental models and to do further exploration. For example, when understanding programs bottom-up, a programmer may continuously form a series of mental models of control flow and data flow [19].

Using the relation tracer, a user can create a new relation tree view that will be automatically added to and maintained by a view list. At any time there is only one view in the foreground of the display, but the user is allowed to choose any view from the list and switch it to the foreground. Views can be deleted from the list. This view management technique is useful for reducing the cognitive overhead a user may encounter in a multi-view system where the user has to find a specific view from many overlapping windows (see Section 2). Figure 8 illustrates how to choose a specific view from the relation tracer.

### 4.3 Summary

As a subsidiary view provider, the searching tool provides a very useful perspective, from which a programmer can examine source code and documentation. Unlike the Searchable Bookshelf, this searching tool does not use the complex GCL (Generalized Concordance List) [6, 20] as its query language. It uses a simple graphical interface that is similar to a text editor's searching facility. In comparison with TkSee, this tool is connected to

the primary view in SHriMP, which provides intuitive clues to assist a user in exploring source code. Therefore, a user is allowed to examine both the software architectural views and source code simultaneously. Another subsidiary view provider is the relation tracer, which can be used to trace various relationships and display them in the format of relation trees. It is especially useful for constructing call hierarchies.

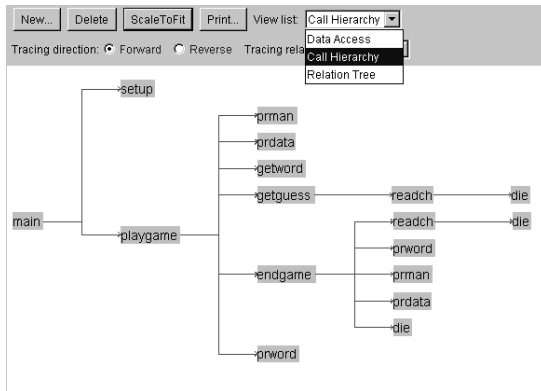


Figure 8: Choosing a call hierarchy view from the view list.

## 5 Conclusions and Ongoing Research

This paper describes the latest Java prototype of SHriMP. During the development of this prototype, we took into consideration the knowledge from previous prototypes and empirical evaluations. Three perspective views have been integrated in SHriMP: a primary view, a searching tool and a relation tracer. Together they provide a more complete multi-perspective software visualization environment, in which subsidiary views connect to the primary view via a zoom interface.

We are currently in the process of applying SHriMP to visualize the SHriMP software itself. We plan to use SHriMP during its own development as a type of introspective case study. We are using a Java parser (courtesy of Tarja Systa of Tampere University of Technology, Finland) to parse all the SHriMP relevant classes and interfaces and to extract artifacts (e.g., packages, classes, methods and fields) and their dependencies (e.g., method calls and class inheritances). The gathered data is first stored in plain text RSF files [13] and

then read into SHriMP for further introspective exploration. Figure 3 shows an architectural view of the **gui** package of SHriMP. We intend to use SHriMP to document and capture the evolutionary design process of the SHriMP tool. In addition, further user studies to evaluate the latest prototype are being planned.

Many reverse engineering and reengineering tools are in development. Closer collaborations between research groups will lead to better tools in shorter periods of time. This may be realized in several ways. For example, we intend to implement 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 plan to support the GXL exchange format [27], which is currently being adopted by several groups to enable the inter-operability of tools.

The Java Bean technology [28] in particular highlights an effective component-based approach for designing SHriMP. Every perspective view in SHriMP can be developed as a Java Bean. Thus, a variety of Java components can be composed into customized applications by end users. Moreover, this approach allows other researchers to integrate single components from SHriMP within their own environments.

For example, we are now working jointly with a research group with the Department of Medicine at Stanford University to integrate SHriMP into Protégé-2000. Protégé-2000 is a "meta-tool" for building domain-specific knowledge acquisition systems that application experts can use to enter and browse the content of electronic knowledge bases [29]. Part of SHriMP has been successfully added to Protégé-2000 as a plug-in component for visualizing various domain ontologies and content knowledge. Exploring collaborations with other tool designers is one of our primary research objectives.

## Acknowledgements

We wish to thank the other programmers of the Java prototype of SHriMP: Jamie Pettit, Anton An, Casey Best and Polly Allen. We would also like to thank Ben Bederson for his advice on how to integrate Jazz [30] within the SHriMP System. Jazz provides some of the animation zooming features in SHriMP.

This research was supported in part by CSER (Canadian Consortium for Software Engineering Research), IBM Canada, NSERC (National Sciences and Engineering Research Council of Canada), ASI (British Columbia Advanced Systems Institute) and the University of Victoria.

## About the Authors

Jingwei Wu got his M.Sc. in computer science at the University of Victoria in July 2000. He is now studying at the University of Waterloo towards a Ph.D. degree in computer science. His research interests include software engineering, program comprehension, design patterns and component technology.

Dr. Margaret-Anne (Peggy) Storey is an assistant professor in the Department of Computer Science at the University of Victoria. She is a fellow of the British Columbia Advanced Systems Institute (ASI) and is one of the investigators for CSER (Center for Software Engineering Research) developing and evaluating software migration technology. Her research interests include experimental software engineering, program understanding, human-computer interaction, information visualization and graph drawing.

## References

1. B. Lientz and E. Swanson. *Software Maintenance Management: A Study of Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, MA USA, 1980.
2. H. Müller and K. Klashinsky. Rigi – A System for Programming-in-the-large. In *Proc. of the 10th International Conference on Software Engineering*, pages 80-86, Raffles City, Singapore, 1988.
3. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley and K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4): 564-593, 1997.
4. SNIFF+. *User's Guide and Reference (V3.2)*, TakeFive Software, <http://www.takefive.com>, November 1999.
5. M.-A. Storey, K. Wong, F. Fracchia and H. Müller. On Integrating Visualization Techniques for Effective Software Exploration. In *Proceedings of the 1997 IEEE Symposium on Information Visualization*, pages 38-45, Phoenix, Arizona USA, 1997.
6. S. Sim, C. Clarke, R. Holt and A. Cox. Browsing and Searching Software Architectures. In *Proc. of International Conference on Software Maintenance (ICSM'99)*, Oxford, England, September 1999.
7. T. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. *Computer Science Technical Report TR-97-07*, School of Information Technology and Engineering (SITE), University of Ottawa, Canada, 1997.
8. H. Müller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley and K. Wong. Reverse Engineering: A Roadmap, *The Future of Software Engineering* (ISBN: 1-58113-253-0), ACM Press, 2000.
9. M.-A. Storey, H. Müller and K. Wong. Manipulating and Documenting Software Structures. In P. Eades and K. Zhang, editors, *Software Visualization*, pages 244-263. World Scientific Publishing Co., 1996.
10. M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proc. of the 3rd Working Conference on Reverse Engineering*, pages 31-40, Monterey, CA USA, November 1996.
11. M.-A. Storey, K. Wong and H. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proc. of the 4th Working Conference on Reverse Engineering*, pages 12-21, Amsterdam, Holland, October 1997.
12. M.-A. Storey. *A Cognitive Framework for Designing and Evaluating Software Exploration Tools*. Doctoral thesis, School of Computing Science, Simon Fraser University, Canada, 1998.
13. K. Wong. *Rigi User's Manual*. University of Victoria, 1996.
14. M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of ACM*, 38(6): 33-44, June 1995.
15. M.-A. Storey and H.A. Müller. Graph Layout Adjustment Strategies. In *Proceedings of Graph Drawing 1995*, pages 325-337, Passau, Germany, September 1995.
16. A. Robbins. *Unix in a Nutshell: A Desktop Quick reference for System V Release 4 and*

- Solaris 7*, 3rd Ed. O'Reilly & Associates Inc., 1999.
17. TkSee. *Introduction to TkSee 2.0*, University of Ottawa, <http://www.site.uottawa.ca/~tcl/kbre/>, May 1996.
  18. M. Weiser. Program Slicing. *IEEE Transaction on Software Engineering*, 10(4), pages 352-357, July 1984.
  19. N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19: 295-341, 1987.
  20. C. Clarke, A. Cox and S. Sim. Searching Program Source Code with a Structured Text Retrieval System. In *Proc. of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 307-308, Berkeley, CA USA, August 1999.
  21. M. Sarkar and M.H. Brown. Graphical Fisheye Views of Graphs. In *Proc. of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'92)*, pages 83-91, New York, NY USA, May 1992.
  22. R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 19: 543-554, 1987.
  23. P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149-160, May 1984.
  24. K. Sugiyama and K. Misue. Visualization of Structural Information: Automatic Drawing of Compound Digraphs. *IEEE Transactions on System, Man and Cybernetics*, 21(4): 876-892, 1991.
  25. J. Wu. *Integrating Visualization Techniques to Support Program Comprehension*. Master's thesis, Department of Computer Science, University of Victoria, Canada, 2000.
  26. A. Von Mayrhauser and A. M. Vans. Program Comprehension during Software Maintenance and Evolution. *IEEE Computer*, pages 44-55, August 1995.
  27. R. C. Holt, A. Winter and A. Schürr. GXL: Towards a Standard Exchange Format, *Fachberichte Informatik 1-2000*, Universität Koblenz-Landau, Institut für Informatik, Koblenz, Mai 2000.
  28. Sun Microsystems. *JavaBeans API specification*, Version 1.01, <http://java.sun.com/beans>, 1997.
  29. 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). *SMI Report No. 1999-0801*, Department of Medicine, Stanford University, USA, 1999.
  30. B. Bederson and B. McAlister. Jaz: An Extensible 2D+Zooming Graphics Toolkit in Java. *HCI Technical Report No. 99-07*, Department of Computer Science, University of Maryland, USA, May 1999.