Integrating Information Sources for Visualizing Java Programs

Jeff Michaud Margaret-Anne Storey Hausi Müller

Dept. of Computer Science
University of Victoria, BC Canada
email:{jmichaud,mstorey,hausi}@csr.uvic.ca

Abstract

This paper describes the integration of information sources to support the exploration of source code and documentation of Java programs. There are many public domain tools that are available for extracting information and documentation from Java programs. We describe how data integration and presentation integration were used to enable the visualization of this information within a software exploration environment.

1 Introduction

Software visualization is considered by many researchers to be a useful and powerful tool for helping programmers understand large and complex programs. Consequently, there are many visualization tools that have been developed for exploring software code and documentation .

Clearly the effectiveness of a particular visualization relies heavily on the pertinence and accuracy of the information being visualized. A visualization's usefulness depends on 1) the relevance and accuracy of the information being displayed, 2) the coherence of the representation used to display the information, and 3) the methods provided to the user for navigating and exploring the presented information. This paper focuses on how information that is needed during software maintenance can be integrated and subsequently browsed in a software exploration environment.

Information sources for software visualization can be roughly categorized into four general categories:

- Source code artifacts and relationships
- Architectural abstractions and relationships
- Documentation (for example, history information and design decisions)
- Metrics and other analysis results

All of these information sources are required at some point during software maintenance. When trying to understand a complex fragment of source code, the ability to cross reference information is required. For instance, a programmer trying to understand a class in Java may like to have instant access to colour-coded source code that has control flow and data flow dependencies available as hypertext links. In addition, the programmer may wish to rapidly access any available Javadocs, documentation or diagrams describing the role of the class in the system's architecture. In addition, there may be a need to view metrics that describe the size or complexity of the class under examination.

Ideally a maintainer would like to be able to access these pieces of information without having to run distinct tools for each of these items separately. If separate tools are used, integrating the collected information and saving it for future use is cumbersome and is often therefore not attempted. Moreover, missing information may lead to mistakes during future maintenance.

For Java, there are many public domain tools available, such as parsers, source code browsers, analysis tools and documentation generators. However, there are few environments that seamlessly integrate or even allow the seamless integration of these different tools to be used during software maintenance. This paper describes how some public domain tools have been integrated in a software exploration environment using data and presentation integration methods [1]. By reusing existing tools in this fashion, we avoid reinventing a wheel, and instead demonstrate how a powerful machine can be built from several wheels.

The rest of this paper is organized as follows. Section 2 briefly describes the SHriMP software exploration environment. Section 3 describes how information from public domain tools can be integrated within multiple cross-referenced views in SHriMP. Section 4 presents a scenario of how the resulting visualizations can be used during the exploration of a

Java program. Section 5 describes future work and suggests other information sources that could be additionally integrated into a visualization environment. Section 6 concludes the paper.

2 SHriMP Views

The SHriMP visualization technique was originally designed to enhance how programmers understand programs [2,3]. SHriMP presents a nested graph view of a software architecture. Program source code and documentation are presented by embedding marked up text fragments within the nodes of the nested graph. Finer connections among these fragments are represented by a network that is navigated using a hypertext link-following metaphor. SHriMP combines this hypertext metaphor with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user.

SHriMP employs a fully zoomable interface for exploring software. This interface supports three zooming approaches: *geometric*, *semantic* and *fisheye* zooming [4]. 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 causes other information to be elided. Fisheye zooming allows the user to zoom on a particular piece of the software, while preserving contextual information. Information that is of interest appears larger than other information which is reduced in size accordingly.

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 exists. Other possible views may include annotation information, code editors or other graphical displays. 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 source code view, SHriMP will zoom to the appropriate node and display the source code within that node

SHriMP is language independent and can be used for browsing any information space. In this paper we describe how SHriMP is applied to visualizing and exploring Java programs. Previously [5] we used a Java parser that is not available to the general public as well as some other *ad hoc* tools for obtaining program artifacts, relationships, HTML'ized source code, architectural information and documentation to display in SHriMP. In the next section we describe how a redesign of SHriMP enabled us to make use of various public domain tools for obtaining information for software visualization.

3 Integrating Information Sources

Many software visualization tools tend to focus on a very specific collection of information views to enhance the understanding of a software system. However, few tools address all of the information categories mentioned in Section 1 (i.e. source code artifacts and relationships, architectural abstractions, documentation and history information, metrics and analysis information).

More recently, there is a trend towards building extensible and customizable tools that promote the integration of additional views (e.g., PBS [5], Rigi [6], Shimba [7], Dali [8] and Bauhaus Rigi [9]). With the need for extensibility in mind, SHriMP has recently been redesigned and reimplemented using Java Beans [10]. A primary goal of its new component based architecture was to allow tool interoperability via data integration, control integration and presentation integration.

The next three subsections describe the different tools that we use to collect architectural information, HTML'ized source code, and Javadocs. The final subsection describes how these information sources are cross-referenced and subsequently displayed within SHriMP.

3.1 Extracting architectural information from Java programs

The first tool that is needed is one that can extract architectural information from the Java source code. The public domain tool, JavaRE [11], analyzes Java programs and outputs architectural information in the XMI format (cf. Fig. 1) [12]. XMI is the XML Metamodel Interchange format that represents a combined effort from the W3C and the OMG. XMI is built upon UML, MOF, and XML. It is standard industry practice to represent an object-oriented system in UML. What XMI provides is a standard method to serialize this information in a file in order to exchange the model information between tools. The XMI file thus contains expressed in XML all the UML architecture model elements such as packages, classes, methods, attributes, and some important relationships that exist between these model elements such as inheritance and associations. Rational Rose [13] and other modeling tools are now moving to support XMI as an exchange format.

Since the SHriMP tool does not import XMI as a format, we wrote an extractor (XMI2RSF) that extracts the relevant information from the XMI file and expresses it in RSF (Rigi Standard Format) [14]. RSF is structured as a flat text file and is an attributed nodes and arcs format. As the XMI is collected, a complete in-memory model of the architecture is created before writing it out as RSF (cf. Fig. 2). To do this, two steps are performed with each model element. First, the model element is parsed and then stored in an object designed to hold the type of model element in question. Second, the object is then inserted into the in-memory model such that it is pointed to by its owner. For instance, a method is owned by its class, a class by its package, etc.

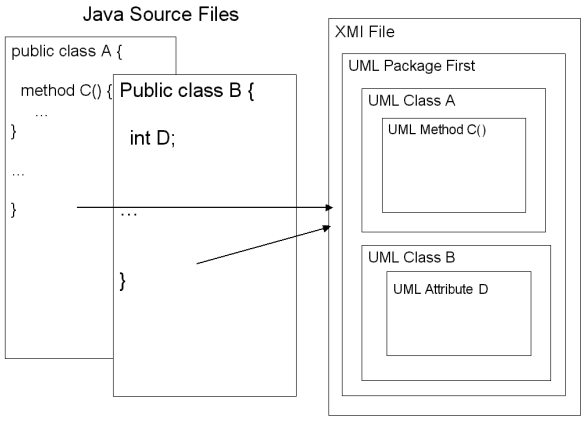

Figure 1: Step 1, the Java source code is parsed in order to extract UML class diagram architecture elements.

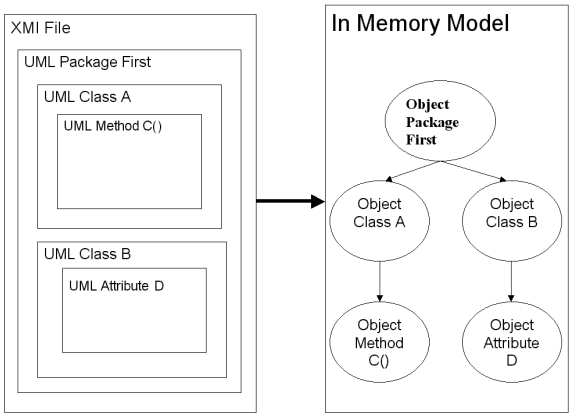

Figure 2: Step 2, The XMI is parsed and stored in an in-memory model that reflects its UML structure.

Once the entire model is in memory, the model is ready to be output in RSF format. Each class has a `toRSF()` method that serializes the model element into RSF. These `toRSF()` methods provide us with the opportunity to tie data provided by the other tools into the RSF. In Section 3.4 we will refer to these `toRSF()` methods again when we discuss how information gathered from different sources is cross-referenced.

3.2 Obtaining HTML'ized Java code

The Javasrc [15] tool is used to provide an enhanced set of HTML pages that allows browsing of the source code as if it were a website. All references in the code to classes, methods, and attributes are linked back to their definitions providing a quick and easy method for traversing the source code. As an added feature, Javasrc creates an organized and complete listing of all references for each class, method, and attribute. While browsing the source code, clicking on the source code definition of a class, method or attribute will bring the user to a special reference page containing a list of references to the selected item which can be used for numerous tasks such as impact analysis and determining how coupled a system is.

Java file and directory naming conventions are followed in the generation of all files and directories by the tool. This means that directory names reflect the package organization and file names match one to one with the class names (with a '.html' extension). Reference pages are also named after the classes but end with '_ref.html'. Every line of code within each of the enhanced HTML pages is accessible through an HTML anchor labeled with that line number (for example, UmlItem.html#25 would bring the user to line 25 of the UmlItem class). Similarly, methods have HTML anchors that are conveniently named after the method itself. The use of these naming conventions turns out to be useful when combining the data from each of these tools (as explained in Section 3.4)

3.3 Generating Javadocs

The Javadoc [16] tool works in a similar fashion to the Javasrc HTML generating tool just described. Javadocs can be generated for the entire program and can be a very useful means of exploring a software system. This tool generates several Javadoc files that provide information on several different levels of granularity including the project level, the package level, and the class level. API information, user documentation, and class structure are each detailed in the Javadocs in an easy to use structure. Javadoc takes the form of a series of HTML pages with file and directory naming conventions that once again follow the Java naming standards.

3.4 Integrating Information Sources

This subsection describes how data, control and presentation integration techniques are used to improve tool interoperability with SHriMP.

3.4.1 Data Integration

According to Wasserman [1], data integration involves the sharing of data among tools and the managing of relationships among data objects produced by different tools. Data integration can be achieved using file exchange or through interprocess communication. Another common approach is to rely on a shared repository that can be accessed by multiple tools.

In our case, data integration is achieved using small custom built programs and scripts to gather and cross-reference information from multiple sources. Figure 3 illustrates the different steps required. First the source code architecture is captured using the Java to XMI tool (JavaRE). Next the enhanced HTML version of the source code and the Javadoc are generated using Javasrc and Javadoc. Finally, the XMI2RSF tool is used to extract the architecture elements from the XMI while generating the necessary additional RSF node attribute information that cross-references the architectural elements with the enhanced HTML source code and the Javadocs.

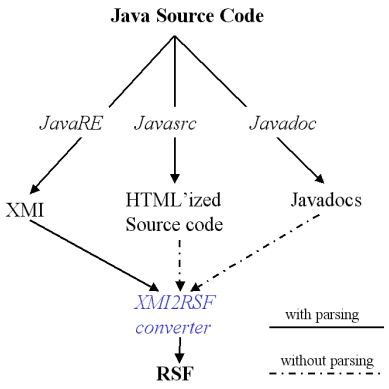

Figure 3: Three public domain tools (JavaRE, Javasrc, and Javadoc) in conjunction with one tool written by us are used to produce the needed RSF for SHriMP. Only the XMI is parsed in the XMI2RSF tool while the HTML'ized source code and Javadoc are incorporated without parsing.

As mentioned in Section 3.1, the XMI is parsed into an in-memory model and is ready to be output as RSF. Each of the objects has a `toRSF()` method that is called to serialize the model element contained within it. These `toRSF()` methods supply access points for inserting additional code to create the RSF that will link to the additional information generated by the other two tools. This additional code will make use of the package, class, method, and attribute names available from the in-memory model to compose links to the files (and parts of files) provided by the Javasrc and Javadoc tools. Since both of these tools consistently follow a set of conventions for naming their files and directories, writing the code to compose these links is simply achieved (cf. Fig. 4).

An important point here is that no parsing of the data provided by the Javasrc and Javadoc tools is required in order to effectively associate their data into the architecture.

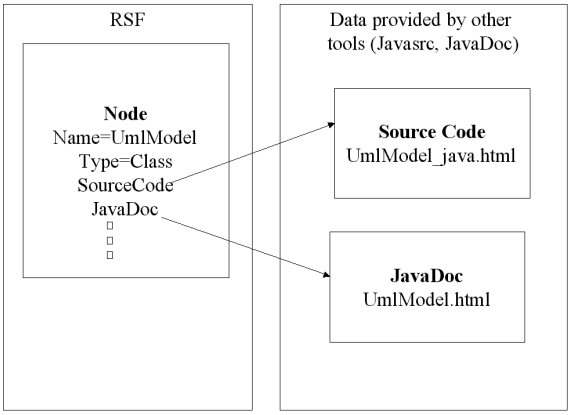

Figure 4: Step 3, while the RSF is generated for each model element, additional RSF is generated to link to the information provided by the other tools without having to parse the HTML'ized source code and Javadoc information.

In order to make it simple to repeat the data integration process, a script was created that simply calls all of the needed programs (all of which are in Java) in the correct order with the needed parameters. The script is setup to be easy to change in order to be applied to parse different Java programs. A mechanism for calling this script and inputting the required parameters will be integrated within the SHriMP user interface.

3.4.2 Presentation Integration

Presentation integration implies a mechanism to ensure consistency at the user interface level [1]. For example, we integrate Swing widgets [17] that can be easily tailored to have a common look-and-feel.

In addition to this we suggest that presentation integration should also reflect ease of navigation between and coherence of multiple views of information sources. A logical, organized structure is required so that all the information gathered can be appropriately displayed and easily manipulated when necessary. One choice for this organizing structure is to use a software architecture view. So far we have been using the implementation architecture that

documents the organization of the source code. For Java, the implementation architecture [9] consists of a hierarchy of packages that contain classes, interfaces or other packages. Classes may contain methods and attributes[1].

The hierarchical structure of the architecture is represented using a nested graph with the *parent-child* relationship showing subsystem containment (cf. Fig. 5). Nodes in the graph represent packages, classes, methods, attributes and so on. However, other relationships, such as inheritance, could alternatively be used for the parent-child relationships. The choice of parent-child relationship is fully configurable by the end-user at run-time. For example, parent nodes could represent superclasses, with their embedded children nodes representing subclasses.

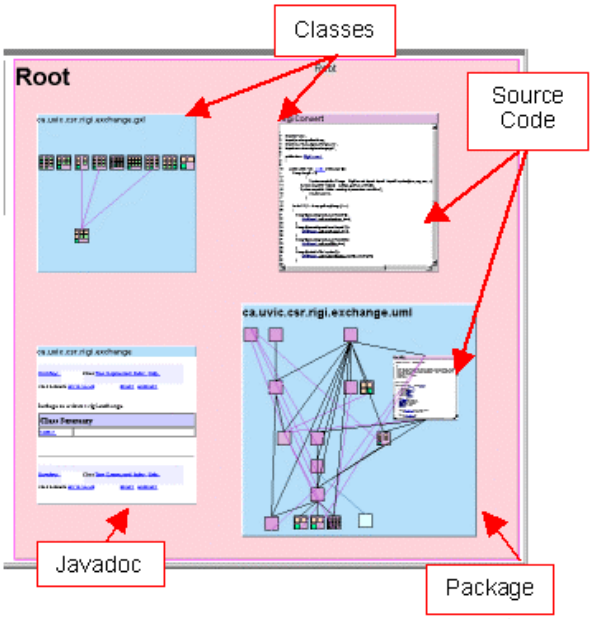

Figure 5: A SHriMP view of a Java program. Three of the displayed nodes (top left, bottom two) show packages in the program. The top left and bottom right nodes are opened to show the classes and interfaces in these packages. The bottom left node shows the Javadoc for that package. The top right node shows the source code for that class.

Additional relationships are visualized using arcs layered over the nested graph. In Fig. 5, coloured arcs represent relationships such as *extends* (i.e. an inheritance relationship), *implements* (when a class implements an interface) and *hastype* (when a class uses an object of a particular type).

Collected information is displayed at the appropriate architectural level. The nodes in the graph are used as containers for different views. For example, a package node can contain a graphical view of its children (classes and interfaces) or it may contain a view showing its Javadoc. A class node may contain a graphical view of its children (attributes and methods), its Javadoc, or its HTML'ized source code. Moreover, other views that are implemented using Swing [17] can be displayed within the nodes. Choosing which view to display is dependent on the stake holder's goal and reason for viewing the visualization.

Navigation between views is facilitated by the data integration of the information sources. The links from one node to another are captured in the RSF node attribute information gathered when the XMI information is annotated by the Javasrc and Javadoc output. Section 4 will further clarify how navigation across views is achieved in SHriMP.

3.4.3 Control Integration

Control integration implies the ability for one tool to control another tool, either by directly activating functionality or by event notification [1].

The Java Bean design allows for easy integration of additional views within the SHriMP nodes. The views described so far are either non-editable views or embedded views that are part of the SHriMP tool itself. However, since a node can contain any Swing widget, it is possible to embed and subsequently control other tools within the nodes. Using this mechanism, we could potentially embed debuggers, metric analysis tools, version control systems, clustering tools and editors among others.

So far we have not had much experience embedding editable views within SHriMP. However, we have demonstrated how SHriMP supports control integration by embedding editable views from a knowledge management tool called Protégé [18,19] within SHriMP. Although this integration is not in the domain of software visualization, it has demonstrated to us that our java bean design is an effective control integration mechanism as we were able to integrate SHriMP and Protégé in just a few days.

4 A Scenario

To illustrate our approach, we provide a description of a working example of using the SHriMP tool. A small Java program (about 30 classes split amongst 3 packages) was parsed using the process summarized in Fig. 3. The program being visualized in this scenario is actually our own XMI to RSF tool that is written in Java. The broad goal of this scenario is to demonstrate how SHriMP facilitates an easier and more convenient

[1] Inner (nested) classes are currently not supported as the JavaRE tool does not identify them. We intend to design and implement this functionality if it is not provided by the next version of JavaRE.

environment for exploring software.

First the source code architecture of the XMI2RSF tool was captured using the JavaRE tool. Next the enhanced HTML version of the source code and the Javadoc were generated. Finally, the XMI2RSF tool was used to extract the architecture elements from the XMI and to generate the necessary additional RSF to cross-reference the enhanced HTML source code and the Javadoc. The script to run these different steps for this example ran in less than 30 seconds on a Pentium III class machine running Windows NT.

After the script is finished, the RSF file can be opened in SHriMP, and the exploration of the system can begin. The first view that a user sees is one similar to Fig. 6 except that the system package nodes have been filtered (however, one could imagine other scenarios where keeping them around could be useful).

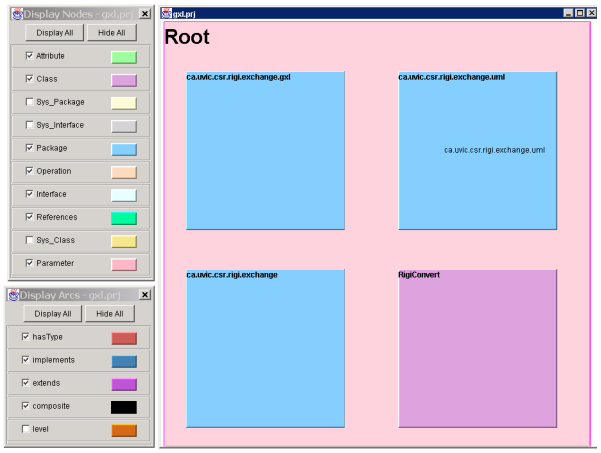

Figure 6: SHriMP view displaying the three packages and one class (which contains the main method) in the XMI2RSF tool. System nodes (packages, classes and interfaces such as those belonging to java.io or java.lang) have been filtered to simplify the view.

A few simple steps lead us to the view in Fig. 7. First, the user zooms in on the uml package node (top right node in Fig. 6). This package details the class structure of the in-memory model used to hold the source code architecture (modeled after UML). Next, some filtering is performed on the arcs to only retain those arcs related to inheritance (extends and implements). Next a Sugiyama layout is issued which arranges the nodes in a hierarchical form and attempts to minimize arc crossings. Lastly, the fisheye zoom is applied to the UmlItem node which increases its size while preserving the general layout to maintain context for the user.

The UmlItem node represents a class within the UML package and is the focus of the rest of this scenario. In the snapshot shown in Fig. 7, the user is shown an abstract graphical representation of the class. This graphical view allows the user to quickly estimate the number of methods and member variables (attributes) in the `UmlItem` class.

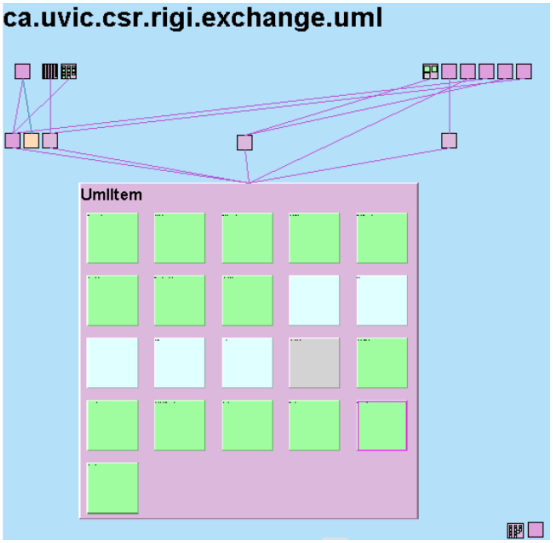

Figure 7: SHriMP view displaying the uml package focusing on the parent class UmlItem. Displayed within UmlItem are its children nodes (methods and attributes).

With one mouse press the user can change the contents of the UmlItem node to display the enhanced HTML source code (cf. Fig. 8). Now the user can navigate the system using hyperlinks. When the user clicks on a link, SHriMP animates and navigates from the source node to the destination node and will display the HTML code in the destination node if it was not already visible.

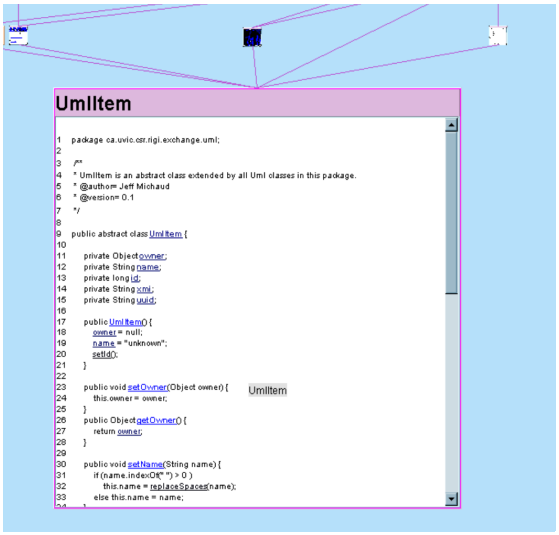

Figure 8: With one click the user has instant access to the enhanced HTML source code for the UmlItem class.

As mentioned in Section 3.2, clicking on the definition of a class, method or attribute will bring the

user to a page that lists all references to the class, method or attribute. If the user selects the `setOwner(Object owner)` shown in Fig. 8, SHriMP will animate the view and display the reference page for the UmlItem class appropriately scrolled to the references for the `setOwner` method (cf. Fig. 9).

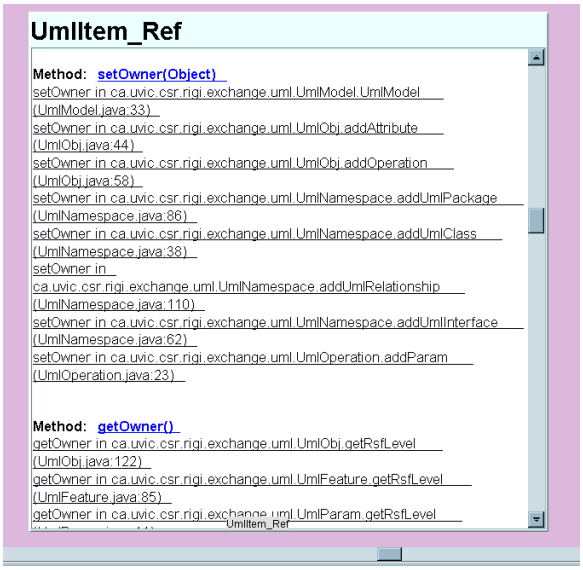

Figure 9: The reference page contains a complete list of references for each method and attribute in the UmlItem class.

From here, the user can view all references to the `setOwner(Object)`method described by the package, class, method and line number where the reference occurs. If the user selects any of the links in the reference list, SHriMP animates the view so that the user is brought to that instance where the `setOwner(Object)` method is called.

Alternatively, a user may not wish to be overwhelmed by the amount of detail shown in the source code directly and may instead wish to view the Javadoc. From either of the views shown in Figures 7 or 8, the user can switch the view so that the UmlItem node displays the Javadoc (cf. Fig. 10).

Through this brief tour of the XMI2RSF tool, we have shown that the user has complete freedom to view the software system as she/he wishes. Having instant access to various views (enhanced source code, Javadoc, nodes and arcs, and others) allows the user to explore the system without having to worry about switching applications for viewing the various representations available for the Java system.

Although we have not evaluated this paradigm for exploring software information, we have observed members of our team using SHriMP for exploring Java programs. Early input seems to indicate that it may be more useful for programmers browsing unfamiliar code. However, as other views (in particular editable views) are integrated within our environment, we expect its appeal will broaden.

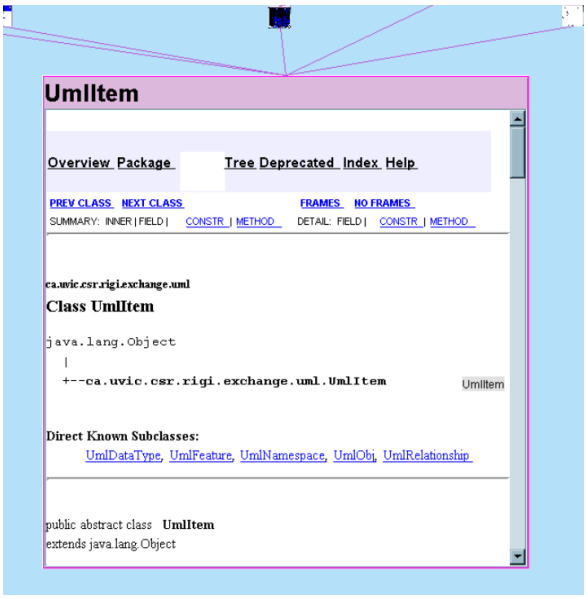

Figure 10: Viewing the Javadoc for the UmlItem provides access to inheritance information, the API, and the programmer's comments.

5 Future Work

The integration of information from the JavaRE, Javasrc, and Javadoc tools to provide cross-referenced information for visualizing Java programs has been very encouraging. Our next step is to see if additional functionality can be integrated with these existing views. In particular we are interested in exploring metric tools, abstract syntax tree parsers, source code repositories (to provide history information), source code editors, clustering tools, graph layout tools and other visualization approaches.

Determining which combinations of tools could be useful in an integrated environment will require extensive user studies. However, by making use of existing tools, we have removed some of the risk of having to implement components that may not be necessary or required by maintainers. User studies (similar to those described in [20]) are scheduled for Spring 2001 to start investigating these questions.

Recently, there has been a move in the reverse engineering and reengineering community towards a standard exchange format (GXL, [21]). GXL takes advantage of all the benefits of XML. As explained, SHriMP employs a format called RSF that is one of several flat text formats for encoding and exchanging

architecture information. One next step for our work is to implement a GXL data bean to allow our tool to interoperate with many more tools through data integration.

Both the Javasrc and JavaRE tools were created using a tool called ANTLR [22]. ANTLR is a tool that provides generic parsing abilities. Since it is a generic parsing tool, it is designed to handle all grammar parsing needs which include parsing Java and C++. The XMI format is a general purpose modeling format and can be used to model other object based languages such as C++ and Visual Basic. For these reasons, it may be possible to find tools to generate the XMI and HTML representations needed for creating a browsing environment for other languages.

6 Conclusions

By taking advantage of existing tools and the conventions that they use to store their data, a powerful environment for visualizing and exploring Java programs was created. The only code written was a program to serialize XMI into RSF and was written within 2 weeks by one developer (the first author of this paper). Software visualization tools depend on the data extracted from parsers and tools such as these. A parser that is simple to understand, simple to expand and simple to extend is as crucial as the software visualization tools. We hope that the description of our experiences using public domain tools as part of a research prototype will be of benefit to other researchers interested in pursuing a similar approach.

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 components in their own tools. In addition, we can import other views and editors into SHriMP as shown in Fig. 5. Using Java beans has proved to be an effective facility to allow us to interoperate with other tools using data, control and presentation integration techniques. This ability to integrate gives us the capability to innovate and create new tools from existing tools [23].

As a concluding statement we would like to stress that although we have focused our discussion in this paper on *how* to integrate tools, the question of *which* tools we should be integrating remains unanswered.

Acknowledgements

This research was supported in part by CSER, NSERC, IBM, Stanford University, the University of Victoria, the University of Texas at Austin and the Space and Naval Warfare Systems Center San Diego. The content of the information does not necessarily reflect the position or the policy of any of the universities nor the US or Canadian government and no official endorsement should be inferred.

References

1. A. Wasserman. Tool Integration in Software Engineering Environments. In F. Long (ed.) *Software Engineering Environments*, *International Workshop on Environments Proceedings*, Lecture Notes in Computer Science, No. 467, pp. 137-149, Springer-Verlag, Sep. 1989.
2. M.-A. Storey, K. Wong, F. Fracchia and H. Müller. On Integrating Visualization Techniques for Effective Software Exploration. In *Proc. of the InfoVis'1997*, pages 38-45, Phoenix, USA, 1997.
3. M.-A.Storey and H.A. Müller. Manipulating and Documenting Software Structures using SHriMP Views. In Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95), Opio (Nice), France, pp. 275-284, October 1995.
4. J. Wu and M.-A. Storey. A Multi-Perspective Software Visualization Environment. In *Proc. of CASCON'2000*, November 2000.
5. P. Finnigan, et al. The Software Bookshelf. IBM Systems Journal, Vol. 36, No. 4, pp. 564-593, Nov. 1997.
6. H.A. Müller and K. Klashinski. Rigi-A System for Programming-in-the-large, IEEE International Conference on Software Engineering (ICSE), Raffles City, Singapore, pp. 80-86, April 1988.
7. T. Systa. Static and Dynamic Reverse Engineering Techniques for Java Software Systems, Ph.D. Thesis, Tampere University, Finland, May 2000.
8. R. Kazman and J. Carriere. Playing Detective: Reconstructing Software Architecture from Available Evidence. Journal of Automated Software Engineering, Vol. 6, No. 2, pp. 107-138, April 1999.
9. R. Koschke, Atomic Architectural Component Recovery for Program Understanding and Evolution, Ph.D. Thesis, Institute for Computer Science, University of Stuttgart, 2000.
10. Sun Microsystems. *JavaBeans API specification*, Version 1.01, http://java.sun.com/beans, 1997.
11. Andersson, Marcus. JavaRE - Java Roundtrip Engineering, http://javare.sourceforge.net/index.php
12. S. Brodsky and T. Grose. Mastering XMI; Java Programming with the XMI Toolkit, XML and UML, John Wiley, 2001.
13. T. Quatrani. Visual Modeling with Rational Rose

and UML, Addison-Wesley, 1998.

14. K. Wong. Rigi User's Manual. Department of Computer Science, University of Victoria, June 1998.
15. *Javasrc: An HTML Java Cross Reference Tool*, http://home.austin.rr.com/kjohnston/javasrc.htm
16. Sun Microsystems. Javadoc Tool Home Page, http://java.sun.com/j2se/javadoc/index.html
17. K. Walrath and M. Campione. The JFC Swing Tutorial: A Guide for Constructing GUIs, Addison-Wesley, 1999.
18. 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.
19. N. F. Noy, R. W. Fergerson, and M. A. Musen. The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility. International Conference on Knowledge Engineering and Knowledge Management (EKAW '2000), Juan-les-Pins, France, 2000.
20. 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.
21. Holt, A. Winter, A. Schürr, S. Sim. GXL: Towards a Standard Exchange Format in Proceedings of WCRE 2000 - 7th Working Conference on Reverse Engineering, November, 23 - 25, 2000, Brisbane, Queensland, Australia, 2000.
22. ANTLR Complete Language Solutions, http://www.antlr.org/.
23. J. Wu. Integrating Techniques for Software Visualization, M.Sc. Thesis, Department of Computer Science, University of Victoria, August 2000.