Software Visualization Tools for Component Reuse

Craig Anslow Stuart Marshall James Noble Robert Biddle[1]

School of Mathematics, Statistics and Computer Science,
Victoria University of Wellington, New Zealand
[1]Human Oriented Technology Laboratory,
Carleton University, Canada
Email: {craig, stuart}@mcs.vuw.ac.nz

Abstract

This paper describes our experiences with our software visualization tools for web-based visualization of remotely executing object-oriented software. The motivation of this work is to allow developers to browse web-based software repositories to explore existing code components and frameworks by creating visual documentation. Components are test driven to capture their static and run-time information in program traces and are then transformed into useful visualizations. Visualizations can help developers understand what a component does, how it works, and whether or not it can be reused in a new program.

Keywords: software visualization, component reuse, web-based code repositories.

1 Introduction

To reuse a software component developers need to understand what a component does, how it works, and how it can be reused. However, this is difficult in practice. Helping developers understand components by creating visualizations means that they will potentially be able to reuse a component in a new program.

We have created a series of software visualization tools within our Visualization Architecture for REuse (VARE) to solve this problem. VARE is used for test driving reusable components to create meaningful visualizations so that developers can understand how components work.

To visualize a design or a software component, certain information has to be selected. Extracting the correct information and gathering it in program traces is a difficult procedure. There are many factors which can affect this procedure, such as the language a component is written in, or the design complexity.

One method for deriving this information is to test drive a component. Test driving is a method for examining the execution of a component and can be done in various ways such as using debuggers or modified execution environments. Test driving generates static and run-time information about a component such as class descriptions and the methods that have been invoked on objects. We have created two tools for examining C++ and Java programs.

Program traces are expensive to generate because they are extremely large and take a long time to create. We have created two XML based program trace languages for describing object-oriented programs. Our program traces are stored in an XML database and can be queried and then transformed into Scalable Vector Graphics (SVG) visualizations.

The paper is organised as follows. In section 2, we describe our motivation for visualizing reusable components. Section 3 describes the experiences with our software visualization tools. Section 4 addresses related work.

2 Motivation

The main reasons for wanting to reuse components are to save on time, effort, and costs in both development and maintenance of quality software. This will mean the developer will not have to implement a new solution to an old problem. Instead they can recycle existing components to solve their problem. Research into component reuse has been happening for a long time [11] and includes many areas of focus; several overviews are available [4, 9, 14].

There are many ways component reuse can be applied. For example, copying and pasting code into a new program, inheritance of classes, instantiation of common methods within programs, using a framework, and using an application programming interface. When reusing a component it may need to be modified or extended in some way so that it will meet the requirements of the new program. The assumption is that even modifying or extending a component will result in the reduction of time, cost and effort compared with designing the component from scratch.

A key benefit from reusing components is that when modifications, bug fixes or updates occur, the developer can save time by incorporating them into their program. Problems then don't have to be solved for every instance. This can happen on a global scale and examples include online updates of both proprietary and open source software.

We are interested in understanding reusable components so that developers can reuse them in their new programs. Currently several techniques exist to help understand how software works and these include documentation, experimenting, and visualizations. Documentation is sometimes provided with software either in online or in written form, but is often difficult to use, read and understand. Experimenting with reusable components means that developers will gain practical experience and learn how components work. Visualizing a component's static or run-time information can show developers how a component has been designed, and how it works when executed.

We are interested in visualizing reusable components for the purposes of understanding and we separate software visualizations into two categories:

- Static visualizations: can be created from investigating the source or binary files, which can contain class descriptions along with their methods and variables, inheritance hierarchies between classes, and dependency hierarchies amongst classes.
- Run-time visualizations: can be composed by examining or spying on programs during execution and gathering events in a program trace. The types of information that can be gathered include object creation and deletion, method calls and returns, field accesses and modifications, exceptions, and multi-threading issues.

When visualizing reusable components we have focused on three different types of information. These include understanding what a component does, how a component works, and how a component can be reused. For what a component does, it is important to look at the external side-effects and the results that occur as a consequence of interacting with a component's public interface. For how a component works, it is important to look at the internals of a component. This is because it may open up opportunities for modifying the component's behaviour to what is required by replacing sub-components, extending components or overloading methods. For how a component can be reused or modified, it is important to look at how it has previously been used.

3 Software Visualization Tools

We have created an architecture, VARE [8], for web-based visualization of remotely executing object-oriented software. VARE is based on the Program Mapping Visualization (PMV) conceptual model for describing program visualization systems [18, 16]. The design of VARE supports multiple programming languages and provides user control for the different parts in the visualization process.

VARE is a client-server architecture, see figure 1. The server contains repositories and processes. On the client side, the user manages the activities associated with creating and viewing a visualization. The component repository interface lets the user select a component from the repository to create a component set. Once this is created, the user can select an engine type from the engine repository to control the test driving of these components. Test driving is defined as "specifying a sequence of method invocation and field access/modifications and then executing the sequence on a component" [5]. The engine component is synonymous to the program component in the PMV model.

The engine generates a program trace/test drive trace as output, which is stored in the test drive report repository. A program trace contains all the information required to describe a program execution such as the order of object creation, method invocations, field accesses and field modifications. A program

trace is then used as input to a transformer, which is synonymous to the mapping component from the PMV model. The transformer repository interface lets the user select the transformer to use and the program trace to use with it. The transformer then transforms the program trace into an appropriate visualization.

Finally the finished visualization is stored in the visualization repository. The visualizations contain information such as a description of the components they are associated with, who created them, and notes that help the understanding of the visualization. The visualization interface lets a user choose a particular visualization and control its presentation. The following sections describe our experiences with our VARE tools.

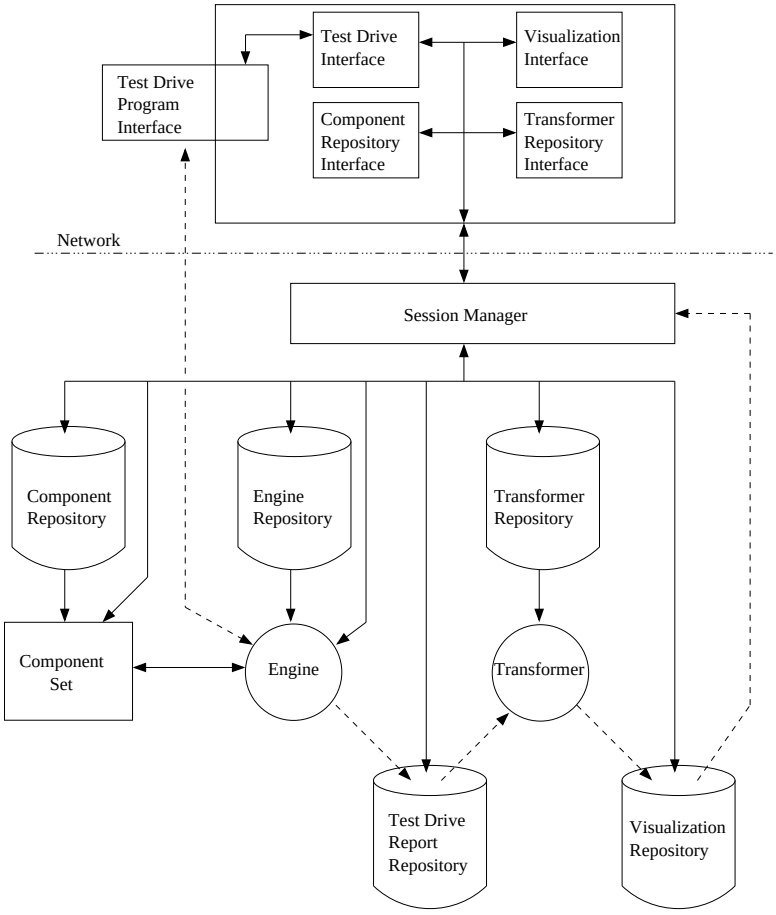

Figure 1: The VARE architecture is based on a client/server model, with the server being split into repositories and processes. Dashed lines represent test drive or visualization input/output, while solid lines represent control, queries or responses [8].

3.1 Test Driving

Abstraction Tool (AT) [10, 8] is an implementation of an engine from VARE, see figure 2. AT is a prototype utility that has been developed to extract information from applications and present the information using the Process Abstraction Language (PAL), so that visualizations tools can visually display the information to a developer. AT test drives programs written in C++, using the GNU Debugger (GDB). It is written in the Python scripting language. The main tasks of AT are to drive GDB, and to output XML based on what was seen during execution. AT also uses SOAP for remote method invocation, to allow AT to be controlled by another application.

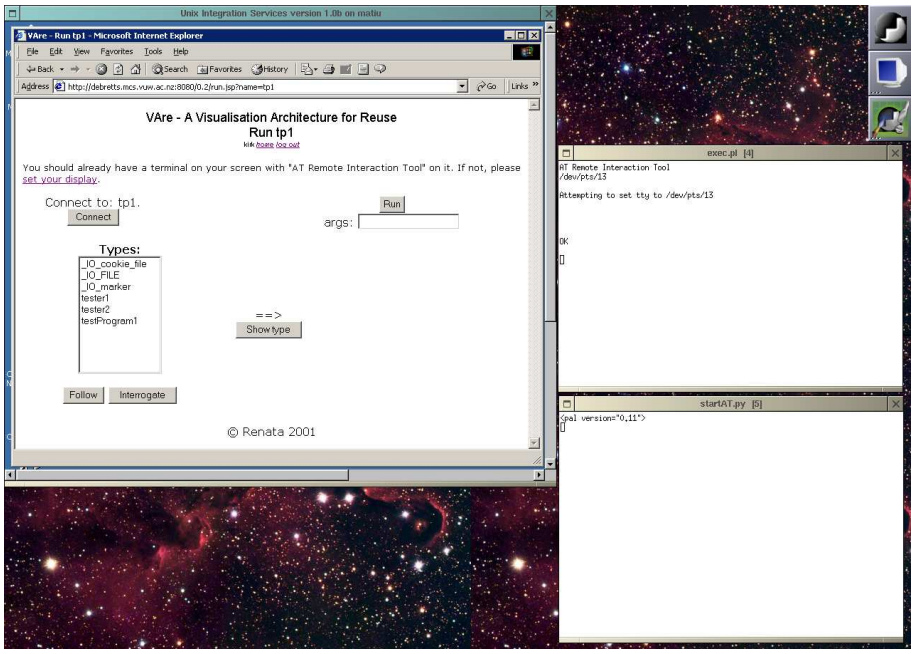

Figure 2: Abstraction Tool (AT) [8].

Spider [6] is a proof-of-concept prototype for exploring and documenting reusable components in a web environment, see figure 3. Spider documents components with Reusable Component Descriptions (RCD) and test drive traces with eXtensible Trace Executions (XTE) by interpreting information stored in a component, detecting events in the run-time environment, and interrogating the runtime environment's state. Spider uses existing Java libraries to achieve this functionality: notably the Java Reflection and Java Debugger Interface (JDI) technologies to extract information.

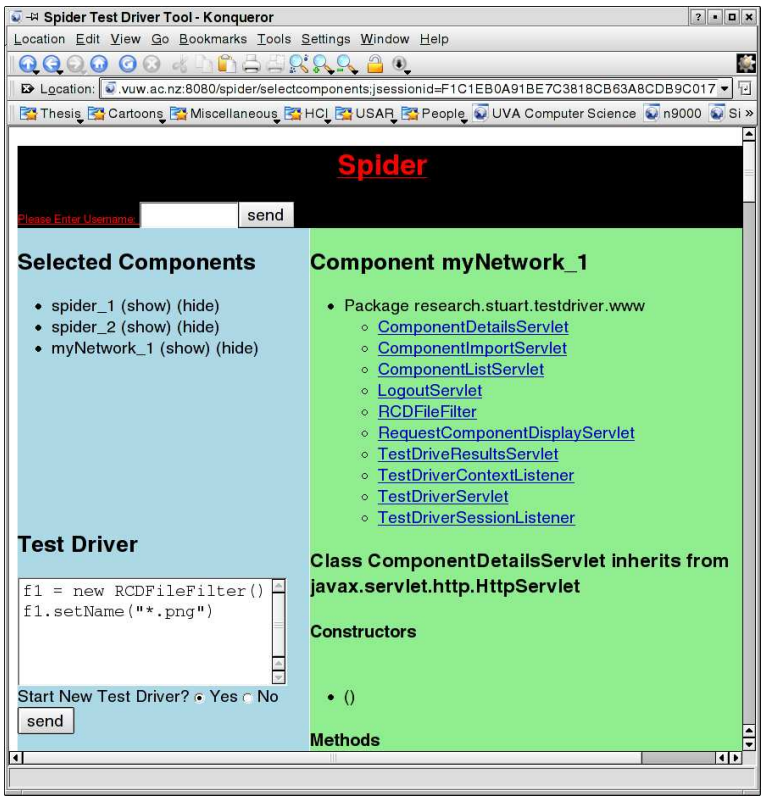

Figure 3: Spider [6].

3.2 Program Trace Languages

We have created a set of requirements for a program trace language [7] and implemented two different XML based program trace languages. They each have features for representing both static and run-time information.

The Process Abstraction Language (PAL) [10, 8] defines an XML specification for object models designed to help visualization tools get the information they need to generate useful visualizations. PAL describes object-oriented programs and has been used for representing information that can be test driven from C++ and Java programs. It has elements for describing classes, superclasses, methods, and fields. PAL can also describe the run-time behaviour of programs, including objects, run-time representations of classes, method calls with their arguments and return values, and different threads of control.

Experiments with AT and Spider identified weaknesses in PAL. PAL combines both static and dynamic information, and stores the entire information needed to describe a specific test drive. This leads to large amounts of redundancy in situations where we were storing multiple test drives. The static

information is identical when we test drive the same component multiple times. The component may also depend on other common components that are shared by other components likely to be test driven. One example of this in the Java language is the use of the Java libraries packaged in the standard development kit. There are similar examples in all widely-used programming languages.

To remove the redundant information in the test drives, we decided it was necessary to store static information relevant to a single component separately from the dynamic information relevant to a single test drive. The result was Reusable Component Descriptions (RCD), for static information, and eXtensible Trace Executions (XTE), for dynamic information [6].

There are a number of benefits to this approach. Firstly, the files are smaller, which is beneficial both in storage and in transportation costs. Secondly, there was no easy means of identifying which version of a component was contained within a PAL file, however in RCD we have separate files for each component. Thirdly, there was no easy means of identifying which types comprised a component, and which types were used by the component.

3.3 Repositories

XML Data Storage Environment (XDSE)[1] is used for storing and retrieving PAL, RCD and XTE program traces, see figure 4. XDSE is an implementation of the test drive report and visualization repositories from VARE. The main feature of XDSE is to store PAL, RCD and XTE program traces and then query them for useful information to generate a visualization. XDSE is implemented with an Ipedo native XML database, SOAP, Apache Tomcat, and XQuery.

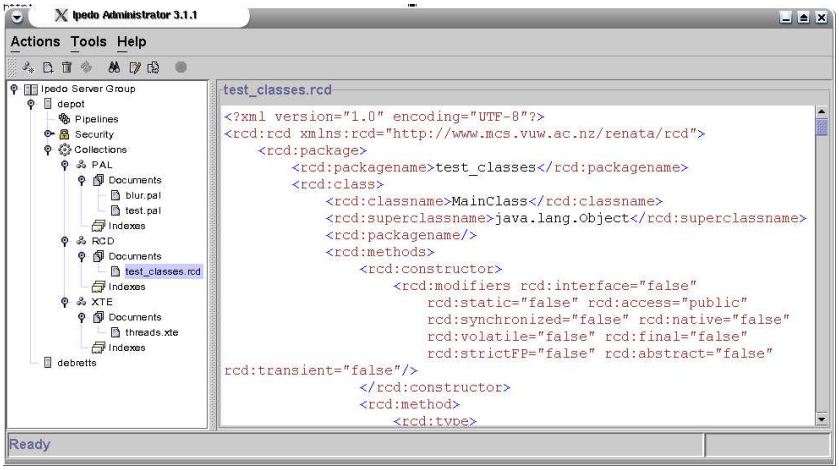

Figure 4: XML Data Storage Environment (XDSE) [1].

3.4 Visualizations

Blur [3] is an implementation of a transformer from VARE. Blur takes a PAL program trace and transforms it into a Scalable Vector Graphics (SVG) [20] visualization for viewing over the web. Blur is implemented as a Java Servlet running a version of Apache Tomcat.

Figure 5 shows a SVG UML interactive class diagram from a PAL program trace generated by Blur. When the mouse covers a piece of code in the right hand side frame, the left hand side highlights the appropriate class or method in the UML class diagram. This is a helpful tool for developers, because it shows where the code is located in a file, and how it is associated with other classes in a program.

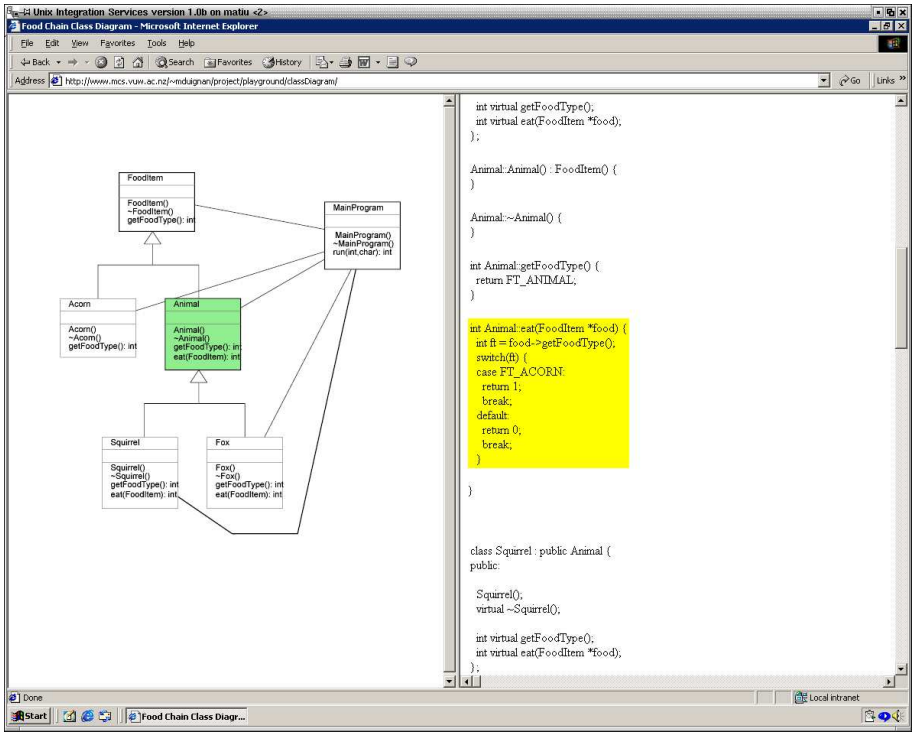

Figure 5: A SVG interactive class diagram generated by Blur [3].

Figure 6 shows a SVG sequence diagram generated by Blur from run-time information found in the same PAL program trace as that of figure 5. The sequence diagram is interactive and allows the user to navigate, zoom-in-out, and fold and unfold call sequences to better understand the diagram.

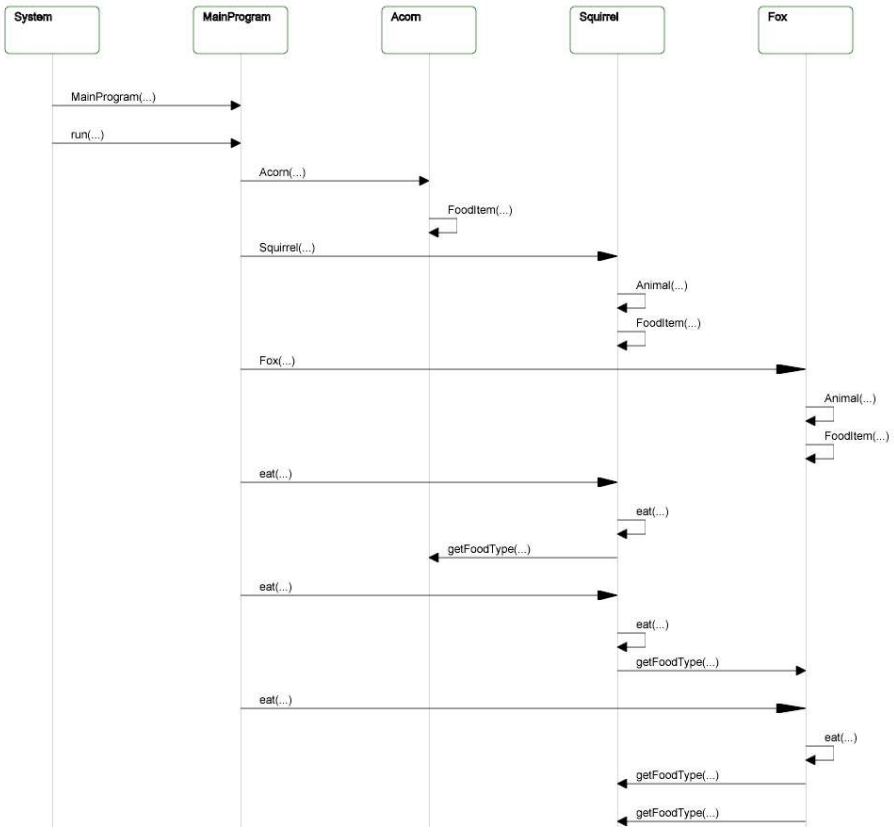

Figure 6: A SVG interactive sequence diagram generated by Blur [3].

4 Related Work

Two similar software visualization systems are BLOOM [15, 21] and Jinsight [12, 13, 17]. Other systems have also been documented [19, 2].

BLOOM is a system for doing software understanding through visualization. It provides facilities for static and dynamic data collection. It offers a wide range of data analysis. It includes a visual query language for specifying what information should be visualized. All these are used in conjunction with a back end that supports a variety of 2D and 3D visualization strategies.

Jinsight is a tool for visualizing and analysing the execution of Java programs. The aim of Jinsight is to help a user better understand, tune, and debug a program. It is useful for performance analysis, memory leak diagnosis, debugging, or any task in which a user needs to better understand what a Java program is really doing.

5 Conclusion

In the future, we plan to integrate the different parts of VARE to provide an overall system for an end user. We are currently exploring test driving of reusable components and program trace 3D visualization.

In this paper we have described our experiences with our software visualization tools. Visualizations are created by test driving components and extracting static and run-time information. Our visualizations help developers understand what a component does, how it works, and whether or not it can be reused in a new program.

References

[1] Craig Anslow, Stuart Marshall, Robert Biddle, Kirk Jackson, and James Noble. Xml database support for program trace visualisation. In *Proceedings of the Australian symposium on Information visualisation*. Australian Computer Society, Inc., 2004.

[2] Stephan Diehl. Revised lectures on software visualization, international seminar, 2002.

[3] Matthew Duignan, Robert Biddle, and Ewan Tempero. Evaluating scalable vector graphics for use in software visualisation. In *Proceedings of the Australian symposium on Information visualisation*, pages 127–136. Australian Computer Society, Inc., 2003.

[4] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[5] Stuart Marshall. Understanding code for reuse. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1999.

[6] Stuart Marshall, Robert Biddle, and James Noble. Using software visualisation to enhance online component markets. In *Proceedings of the Australian symposium on Information visualisation*. Australian Computer Society, Inc., 2004.

[7] Stuart Marshall, Kirk Jackson, Craig Anslow, and Robert Biddle. Aspects to visualising reusable components. In *Proceedings of the Australian symposium on Information visualisation*, pages 81–88. Australian Computer Society, Inc., 2003.

[8] Stuart Marshall, Kirk Jackson, Robert Biddle, Michael McGavin, Ewan Tempero, and Matthew Duignan. Visualising reusable software over the web. In *Proceedings of the Australian symposium on Information visualisation*, pages 103–111. Australian Computer Society, Inc., 2001.

[9] Carma McClure. *Software Reuse Techniques: Adding Reuse to the System Development Process*. Prenctice-Hall Inc., 1997.

[10] Mike McGavin. Extracting software reuse information for visualisation tools. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, October 2001.

[11] M. D. McIllroy. Mass produced software components. In P Naur and B Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, 1968.

[12] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Lecture Notes in Computer Science*, volume 821, pages 163–182, Bologna, Italy, July 1994. European Conference for Object Oriented Programming, Springer Verlag.

[13] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by analysis of running programs. In *Proceedings for Workshop on Software Visualization*, Toronto, Canada, May 2001. International Conference on Software Engineering.

[14] Jeffrey S. Poulin. *Measuring Software Reuse: principles, practices, and economic models*. Addison-Wesley Longman Inc., 1997.

[15] Steven P. Reiss. An overview of BLOOM. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5. ACM Press, 2001.

[16] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12), December 1993.

[17] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *Proceedings for TOOLS Europe 2001*, Zurich, Switzerland, March 2001. Technology of Object-Oriented Languages and Systems (TOOLS) Conference Series.

[18] John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[19] John T. Stasko, Marc H. Brown, and Blaine A. Price. *Software Visualization*. MIT Press, 1997.

[20] World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.1 specification, 2003. `http://www.w3.org/TR/SVG11`.

[21] Kang Zhang. *Software Visualization: From Theory to Practice*, chapter 11. Kluwer Academic Publishers, 2003.