A Structured Demonstration of Program Comprehension Tools

Susan Elliott Sim
Dept. of Computer Science
University of Toronto
10 Kings College Rd, Toronto
ON, Canada M5S 3G4
+1 (416) 978 4158
simsuz@cs.utoronto.ca

Margaret-Anne D. Storey
Dept. of Computer Science
University of Victoria
PO Box 3055 STN CSC
Victoria, BC Canada V8W 3P6
+1 (250) 721 8796
mstorey@uvic.ca

Abstract

This paper describes a structured tool demonstration, a hybrid evaluation technique that combines elements from experiments, case studies, and technology demonstrations. Developers of program understanding tools were invited to bring their tools to a common location to participate in a scenario with a common subject system. Working simultaneously, the tool teams were given reverse engineering tasks and maintenance tasks to complete on an unfamiliar subject system. Observers were assigned to each team to find out how useful the observed program comprehension tool would be in an industrial setting. The demonstration was followed by a workshop panel where the development teams and the observers presented their results and findings from this experience.

Keywords

Empirical study, program comprehension, tool evaluation.

1. Introduction

During the past decade, many tools have been developed both in industry and research to support reverse engineering and program understanding. There is no doubt that better tools would have a huge impact economically, as the pressure to rapidly evolve and develop software systems increases. Unfortunately, few tools have achieved widespread acceptance in industry. One way to remedy this slow technology transfer is to undertake tool evaluations. These evaluations are done not only to assess the applicability of tools, but also to help improve them and to identify further requirements.

Unfortunately, the evaluations in the literature tend to be *ad hoc* at best.[15] Software tools are rarely evaluated in a formal way by users, and when they are evaluated, it is for a short time by people who do not have training or experience with the tool.[11, 13, 21, 22] Too often potential users base their opinions of the tool on superficial factors such as appearance, ease of learning, and number of features, rather than factors that are more important in the long run such as ease of use, flexibility, and scalability. Evaluations based on case studies, such as applying a particular tool to a subject system are informative but the results are difficult to generalize.[9, 12, 14]

Although program comprehension tools share the common goal of simplifying the task of understanding large bodies of source code, these tools differ at many levels: from their appearance to technical details to their philosophical approach. These differences and their relative strengths and weaknesses do not become apparent until the tools are seen side-by-side. Opportunities to see different tools perform the same tasks are highly illuminating. Some authors have compared tools independently or with colleagues.[8, 10, 17, 24] Chikofsky organized a Reverse Engineering Demonstration Project where researchers were invited to use their tools to analyze the WELTAB III Election System.[4]

With this in mind, the authors of this paper designed a structured tool demonstration where tool builders were invited to demonstrate their tools in a live setting. The idea was for the tool developers to apply their own tool to a common software system. Software tools that provide visualization and exploration facilities for program understanding were selected to participate in the demonstration. Working simultaneously, the tool teams were given reverse engineering tasks and maintenance tasks to complete on the subject system. Industrial observers were assigned to each team to learn how to use the program comprehension tool. They were asked to assess if the tool would be useful for their own development team in industry.

The research contributions of this work are threefold. First, they establish a benchmark that can be used to evaluate reverse engineering tools. Tool developers who use the *xfig 3.2.1* structured demonstration can compare their results with those from previous participants.

Tool	Description	Languages	Operating Systems
Lemma, IBM RTP	• displays software structure and code statements are various levels of abstraction • source code searching, navigation, code viewing, calling diagrams and control flow graphs	C/C++, Java, Fortran, Cobol, PL/I, Pascal, Rexx	Windows NT Linux OS/2
PBS, U. of Waterloo	• tool set for extracting, analyzing and visualizing software architecture	C, C++, PL/IX	Solaris Linux Windows
Rigi, U. of Victoria	• graph visualization and exploration tool, with scripting and some metrics facilities	C, C++, Cobol, PL/IX	UNIX Linux Windows
TkSee, U. of Ottawa	• source code searching tool with a GUI for very large software systems • history and task management capabilities	C, Pascal, Assembler	Linux
UNIX Tools (Red Hack)	• vi/emacs • compiler, debugger, profiler • grep	C, Fortran, others	UNIX
Visual Age C++, IBM	• a repository-based IDE with incremental compiler • includes editor, compiler, search capabilities, class browser	C++	Windows NT AIX

Table 1: Summary of Tool Characteristics

Second, they present a technique for combining usability testing with benchmarking to provide further evidence on the applicability of tools. Third, the materials developed for a structured demonstration encapsulate the knowledge necessary to perform an empirical tool evaluation. Consequently, it will be easier for someone with little knowledge of experimental design to conduct a reasonable study.

The demonstration was held as part of a workshop at CASCON99, an annual Canadian software technology conference that brings together researchers, industry and government.[1] The goal of the demonstration was not to find a *winner*, but to help researchers in this field learn which aspects of the studied tools would be useful for particular tasks. Given the differences in how the tools operate, comparing tools along a single dimension would have been difficult, if not impossible.

The remainder of the paper is organized as follows: Section 2 outlines our objectives and reasons for organizing this demonstration. Section 3 describes the participating tools, the format of the demonstration, the industrial observers assigned to the teams, and the assigned reverse engineering and maintenance tasks. Section 4 describes the results of the assigned tasks. Section 5 reviews some of the observations made by the industrial observers and the workshop chairs (the authors of this paper). Section 6 discusses the outcomes of the workshop and outlines future work.

2. Objectives

The overall idea behind this workshop was to provide a common playing field for tool builders to demonstrate their tools by having experienced users apply them in a live setting to an example software system. We wanted to capture the entire experience, i.e. observe each team receiving the subject system's source code and documentation right through to when the team used their tool to complete the assigned tasks.

By demonstrating the tools in a structured fashion, we could observe expert programmers and expert users using the tools. A drawback with other user studies is that it can be difficult to find expert users and it is impractical to expect users to spend a lot of time learning a tool for the sake of participating in a study. Furthermore, by assigning realistic tasks on an actual software system, this helped us consider ease of use, flexibility and capability rather than focusing solely on usability. It also allowed us to consider tool usefulness from the particular task perspectives of program comprehension and software maintenance.

For the tool developer participants, we expected that this demonstration would provide them with insights into their own tools, as well as enable firsthand viewing of other approaches being developed to provide support for the same tasks. We strove to give the tools an opportunity to excel within the structure of the demonstration, by assigning a variety of tasks. The subject system assigned was a novel experience for the development teams.

Tool	Team	Observer
Lemma, IBM RTP	Robert Mays—senior software developer	Jeremy Broughton—IBM, DB2 Development Environment and Build Support
PBS, U of Waterloo	John Tran—UofW graduate student Thomas Parry—UofW graduate student Eric Lee—UofW graduate student	Ryan Chase—IBM, DB2 UDB Administration Tools
Rigi, U. of Victoria	Johannes Martin—UofV graduate student Bruce Winter—UofV graduate student Kenny Work—U of Alberta faculty	Not available due to illness
TkSee, U. of Ottawa	Tim Lethbridge- UofO faculty Paul Holden—UofO undergraduate Sonia Vohra—UofO undergraduate	Jeff Michaud—UofV graduate student with previous industry experience
UNIX Tools	Piotr Kaminski—UofV graduate student Arthur Tateishi—Shelty Systems, consultant Andrew Walenstein—SFU graduate student	Not applicable
Visual Age C++, IBM	David McKnight—software developers Cindy Nie—software developers Jeff Turnham—software developers	Not applicable

Table 2: Characteristics of Participants

The demonstration was public; conference participants were invited to observe how the tools were being deployed for the assigned tasks. Teams were requested to have at least one representative available to explain their tools and methodology.

3. The Structured Demonstration

This section of the paper describes the participating tools, the format of the demonstration, the observers assigned to the teams, and the assigned tasks.

3.1 Participating tools

Five software development teams were invited to participate in the demonstration:

- Lemma, IBM RTP [23]
- PBS, University of Waterloo [3, 16]
- Rigi, University of Victoria [5, 16]
- TkSee, University of Ottawa [19]
- Visual Age C++, IBM Toronto Lab [6]

A sixth team of software developers (Red Hack) used a set of UNIX tools to solve the same set of tasks. A short description of the tools and other relevant characteristics can be found in Table 1. The team members are listed in Table 2. With the exception of TkSee, all the teams consisted of individuals who had experience with software development in C and their respective tools. TkSee had one team member who was not an experienced tool user.

3.2 Format

The workshop consisted of two phases. In the first phase the tool development teams demonstrated their tools in a live setting by applying their tools to a subject software system. We tried to find a subject system that was written in a programming language that was common to all the teams on an operating system that everyone could use. We selected the open source *xfig* drawing package, which runs on a variety of UNIX operating systems and is written in ANSI C consisting of approx 50,000 LOC.[7]

We recommended that teams consist of three members and collaborate using only one computer. However, the Lemma team had only one individual and the Rigi team made use of additional computers to fix a bug in their parsing tool. The teams were given the source code and handbooks shortly after 9am and were asked to complete their work by 5pm.

The six teams were presented with the following scenario:

> *xfig* is a drawing application that runs on a variety of UNIX platforms. The current version is 3.2.1 and consists of about 50 000 lines of ANSI C. The old *xfig* team and manager quit the *xfig* project to join a start-up company.
>
> You have been assigned, along with some of your colleagues, to rescue the future development of the *xfig* application. You are placed under a new manager, a recent MBA graduate, who is impressed that you are going to use some fancy tools to get the new team up to speed.

> The first thing the new manager would like you to do is to use your tool(s) to create some documentation that would summarize the main structures and architecture of the *xfig* application. The manager would also like you to explore how you would go about implementing some of the changes that were identified in the inherited "TODO" list.

The scenario also contained a set of assigned tasks. These tasks were described in a handbook that also reviewed procedures to be followed throughout the day (see Section 3.4).

3.3 Observers

Impartial observers (all of whom had experience as software developers) were assigned to the teams to observe how each of the tools were used to solve the tasks. We did not recruit an observer for the UNIX Tools team as these tools are already widely used in industry. Unfortunately, two of the recruited observers were unable to attend due to work issues or illness. We were able to assign a graduate student to one team but we were not able to find an observer for the Rigi team. The observers and their backgrounds are listed in Table 2.

The observers were to act as "apprentices" with the goal of trying to develop a mastery of the tools over the course of the day. This experience allowed them to determine how a particular tool set could be used in their work as software developers. We asked the observers to take notes so that they could report on their experiences during the workshop panel. We briefed each of the observers before the demonstration and gave them an observer's handbook to guide them in their task[1]. During this phase workshop attendees were also invited to drop by and observe the tool developers as they progressed.

3.4 The assigned tasks

We used two principles in selecting the tasks for the structured demonstration. We wanted the tasks to be representative of those a software developer would face in his or her daily work. These were presented as problems, not as prescriptions for how the tools ought to be used. For example, most managers are more likely to ask people to repair a defect or add a feature, rather than perform data flow analysis or slicing. We also wanted the tasks to provide opportunities for the researchers to demonstrate the strengths of their tools. Consequently, we included tasks that required the teams to look at the subject system in different ways.

We assigned two reverse engineering tasks and three maintenance tasks. The teams were required to complete all the reverse engineering tasks and at least one of the maintenance tasks. Each task had a deliverable that the teams were required to hand in. The following task descriptions are taken from the handbook given to the developer teams.

3.4.1 Reverse engineering Tasks

Q1.1 Documentation

Provide a textual and/or graphical summary of how the *xfig* source code is organized. This documentation should provide the manager with an overview of the system, and may include a call graph, subsystem decomposition, description of the main data and file structures or any other appropriate information. Use whatever format you think is appropriate, such as text files, HTML, Word documents, graphics, etc.

Q1.2 Evaluate the structure of the application.

Your manager would like you to form an opinion on the structure of the *xfig* program. In particular, you should answer the following questions:

- Was it well-designed initially?
- Do you think the original design is still intact?
- How difficult will it be to maintain and modify?
- Are there some modules that are unnecessarily complex?

Are there any GOTO's? If so, how many, what changes would need to be made to remove them?

3.4.2 Maintenance Tasks

These tasks were extracted from the *xfig*'s TODO file. The teams were instructed to outline changes required to complete the task, but they were not asked to change the code.

Q2.1 Modify the existing command panel.

The buttons in the command panel (i.e. the tool bar) at the top of the window are somewhat unconventional. For example, the tool bar should be more consistent with those in other graphical user interfaces. The headings "File", "Edit", and "View" should be left justified and the "Help" menu item should be right justified. Also, the buttons in the command panel should be re-arranged as follows:

File	**Edit**	**View**	**Help**
New Load/Merge Save Save As Export Print Exit	Undo Paste Find Replace Spell Check	Landscape Portrait Redraw	Xfig HTML Reference Xfig tutorial in pdf Xfig man pages in pdf About Xfig

[1] A copy of the handbooks and other materials used in the demonstration are available at the workshop web site. [2]

Q2.2 Add a new method for specifying arcs.
Currently, arcs are created by specifying three points (you may want to run the program to try this out), which are then used to create a spline curve. Add a feature that allows a user to draw an arc by clicking on the centre of a circle and then selecting two points on the circumference, i.e. by specifying a radius and angle. Explain the approach you would take to implement this new feature.

Q2.3 Bug fix: Loading library objects.
Loading objects from a library causes the program to crash. This error occurs when the user attempts to load a library object using the bookshelf icon on the left-hand side of the screen. When you click on this icon, a dialog box opens that allows you to select a Library and an object to load. This sequence of steps will result in a "Segmentation Fault" error.

In addition, we asked the teams to consider the following questions which they would need to address in their presentations:

- How long did it take you to read the source code into your tool?
- What difficulties did you encounter with your tool? Did it crash? Any other surprises?
- How long did you spend on the required tasks?
- What kind of documentation did you create?
- Which maintenance tasks did you do?
- How long did each of them take?

The next section in this paper provides highlights of the results provided by the teams in the deliverables.

4. Results

For each of the assigned tasks the teams had to hand in a deliverable that included a description of their solutions. This section describes the documentation and answers to the tasks that the teams provided. A table in the Appendix provides a more detailed summary. The complete results as submitted by the teams, as well as the source code for *xfig*, can be found at the website for the workshop. [2]

The Visual Age team was unable to complete the tasks because *xfig* is written in ANSI C and their IDE works with only ANSI C++. During the structured demonstration, they and the organizers learned that ANSI C++ is not a superset of ANSI C. This was unfortunate because this hampered their ability to participate. However, during the second phase of the workshop, the VisualAge team demonstrated how they would have solved the tasks. This presentation is also available at the workshop website.

4.1 Reverse engineering Tasks

Q1.1 Documentation
In general, the teams produced rather terse documentation. Red Hack and TkSee provided 3 paragraphs. Rigi provided one diagram. The PBS team provided about 4 pages, mostly consisting of diagrams. The Lemma team provided 8 pages of documentation, which included call graphs and code excerpts. The Red Hack team explained the brevity of their documentation for the task by arguing that since they didn't need it to complete the maintenance tasks then future maintainers would probably not require it either.

Q1.2 Evaluate the structure of the application.
The teams had varying differences of opinion on the architecture of the system, the quality of the code, and even the number of GOTO's in the program. All the teams used file name prefixes as the basis for clustering the files into five subsystems, corresponding to the letters, d, e, f, u, and w. However, they had different interpretations of what the prefixes meant. Lemma produced a second clustering based on the functional units in the user interface. Although the Red Hack team did not explicitly specify this clustering, they did criticize it in their design assessment. The PBS group pointed out that the subsystems formed using file prefixes contained more function calls and variable references to files outside the subsystem than to files within it.

In terms of quality, PBS said the subsystems exhibited low cohesion and high coupling, while Lemma said they exhibited low coupling and high cohesion. The PBS team thought that the original design had eroded since its inception, but the Rigi team thought that the design had improved over subsequent releases. Rigi also noted that some modules were unnecessarily complex. All the teams had complaints about the code, such as the lack of comments, function pointer usage, cloning, and duplicated names, but they did not find the code difficult to modify.

Rigi, Lemma, and Red Hack found 5 GOTO's, PBS found 4 and TkSee found 3. Three of the teams gave suggestions for how to remove all of them. The Red Hack team recommended removing only one, and the Rigi team suggested leaving them in the code.

4.2 Maintenance Tasks

The solutions given to the maintenance tasks were fairly consistent across the groups. While the groups were required to do at least one task, most did all or almost all of the maintenance tasks.

Q2.1 Modify the existing command panel.
The groups gave the same basic answer for this task: change an array containing function pointers in w_cmdpanel.c. Although they were asked to simply list the files or functions that were involved in the change, the teams provided answers with varying levels of detail and thoroughness. Some listed only the file names, while others explained in detail how to make the change.

Q2.2 Add a new method for specifying arcs.
There were two approaches to solving this task. The first approach involved modifying the mode panel and adding code in some new files. The second approach involved modifying existing functions to implement the new behaviour. Red Hack, PBS, and Lemma used the first approach, while the other teams used the second approach.

Q2.3 Bug fix: Loading library objects.
This task was not completed by all of the teams and there was more variability in the solutions given. TkSee listed the files to change, but did not explain how or why to change these files. Lemma used static analysis to find a number of possible causes. Red Hack found a couple of ways to stop the program crashes, but they were unsatisfied with those solutions because they could not understand why those changes worked. They reverse engineered *xfig 3.2.2* to find the official solution and the root cause of the problem. Subsequently, they repaired the defect by setting a variable to 17 instead of 55.

5. Observations

This section of the paper details some of the observations made by the observers and by the workshop chairs during the structured demonstration. First, some general observations are offered, followed by some comments about each of the specific tools.

As is often the case with demonstrations, some things did not go according to plan. The day started late due to missing observers and minor troubles with library compatibility within the operating system. The teams, for the most part, completed the tasks within the allotted time (9am to 5pm), but some teams took longer than this to finish writing up their results. As organizers, we had our share of glitches. We had observers who arrived late or not at all and Visual Age did not have an operating system installed on the computer assigned to them.

The biggest difficulty for some teams was parsing the source code (a requirement for all tools except the UNIX tools). Although Lemma only spent 20 minutes parsing and loading the subject system, a bug in their tool slowed their progress initially. The others had to spend several hours modifying their parsers or customizing scripts to load the software.

The observers had many comments about their respective tools, which they presented during the workshop. While these comments were generally positive, there were some criticisms as well. The observer for the PBS team commented that the tool was useful for learning about the general architecture of the subject system as they were able to create diagrams that fit well with his mental image of the system. However, he found that the tool was not useful for the maintenance tasks. For these, the PBS team used basic UNIX tools, such as *vim* and *grep*. The observer concluded that although PBS does have some strengths, it is not a tool that could be easily integrated into his daily (maintenance) work. These observations are also borne out by the team's results. The software landscape diagrams allowed them to ask questions about the clustering that the other teams did not. On the other hand, they had to use alternative tools to complete the maintenance tasks.

The Rigi team, unfortunately, did not have an industrial observer, but we noted that they too had to use other tools to complete the maintenance tasks. Like PBS, they spent a long time parsing the subject system in order to display a visualization of *xfig*. But once they loaded the system, they had diagrams that could easily be used as documentation.

The TkSee observer expressed some frustration with difficulties parsing the system, but once the code was loaded into TkSee, he was very impressed with the tool. In particular, he liked the advanced searching functionality, search history and the to-do list management feature. However, he noted the lack of high-level visualization capabilities. The TkSee team attempted all the maintenance tasks. The observer felt this tool would be useful for his daily work but he expressed some doubts as to whether it would scale as it seemed a little slow at times.

The Lemma observer was very impressed by the comprehensiveness of the searching options in Lemma. While Lemma did provide some diagrams of call graphs and control flow, he was disappointed by the lack of high-level visualization. The Lemma team completed all the maintenance tasks. The Lemma observer wanted to participate in the structured demonstration so that he could determine whether his development team should adopt this tool. He reported that he would be recommending acceptance.

Using the basic UNIX tools, the Red Hack team was able to very quickly complete the assigned maintenance tasks. Although it was not required, they modified the source code and compiled a new executable. However, they produced very little documentation describing the system.

6. Discussion

In this section we present our own inferences based on the results and observations of the structured demonstration. These points are general in nature spanning several tools and the evaluation experience as a whole. This structured demonstration provides lessons for tool designers, potential tool users, and researchers who plan to design similar tool evaluations.

6.1 Lessons for Tool Designers

Everyone used grep, either at the command-line or built into their tool, which has interesting implications for us as researchers. As the Red Hack team noted in their presentation, UNIX tools already provide a great deal of support to programmers for a variety of programming languages in the form of editors, compilers, debuggers, profilers, and cross referencers. There are two issues here. One, these widely-available, popular tools represent a minimum standard which we must improve on to convince software developers and maintainers to use new tools. Two, industry is already able to accomplish a great deal using the tools they already have. Companies regularly release new programs that consist of hundreds of thousands of lines of source code with sophisticated functionality and we should not underestimate what they can tell us about designing successful software tools.

In the structured demonstration, the tools fell into three categories: visualization, advanced search, and code creation. Although this categorization is based on features and functionality, tools from a given category produced similar results. PBS and Rigi were designed for creating graph-based visual representations of software systems based on file clustering. Both teams focused on the same tasks and used diagrams in their documentation of the subject system. TkSee and Lemma had advanced features for searching and tracing through the source code. They both had grep-like functionality included in their tools. The observers for both teams were impressed with the functionality and were willing to use them in their daily work, but were concerned about the lack of high-level views. Finally, Visual Age and UNIX tools are development environments, intended to be used in the creation of new code. The assigned tasks were selected to provide each tool an opportunity to display its key features. We had expected the visualization tools to do better on the reverse engineering tasks and the search tools to do better on the maintenance tasks. These expectations were confirmed, both by the performance of the tools and by observer comments.

While code creation tools are fundamental components of programming environments, this is not yet the case for visualization and advanced searching tools. Visualization and searching tools complement each other; the shortcomings of the visualization tools are matched by the strengths of the search tools, and vice versa.[18] Furthermore, these tools represent different approaches to dealing with large software systems. One approach is to make the code more manageable by supporting searching. The other approach is to abstract away details to make the system more manageable. Within research, it is important to explore different approaches to solving a difficult problem. Moreover, it is worthwhile to test a particular approach with multiple tools. During the panel discussion both the TkSee and Lemma teams identified elements of the other tools that they could use.

As a discipline matures there comes a point when the proliferation of tools is no longer productive. At this point it becomes more important to synthesize the lessons learned from separate explorations. Tool interoperability can be achieved either through a standard interchange format or APIs (application program interfaces) that allow programs to call each other directly. Such mechanisms would allow, for example, an advanced searching tool to leverage the capabilities of a visualization tool. Parsing was another problem that was common across the tools. A standard interchange format or API would allow tool designers to use a best of breed approach in selecting a parser rather than building a parser from scratch.

6.2 Lessons for Tool Users

When selecting a tool for program comprehension it is important to know what the tool will be used for. If the tool is to be used as part of a reverse engineering effort, where large-scale understanding is required, a visualization tool that provides architectural diagrams may be more appropriate. If the tool is to be used in day-to-day software maintenance with extensive effort focused on specific areas, then a searching tool may be more suitable. However, there are many subtle differences to be considered. For example, although TkSee and Lemma are both searching tools, TkSee has features to support exploration of unfamiliar code, whereas Lemma supports control flow diagramming features.

Software developers have a sophisticated set of skills that have been acquired over many years. These skills include knowledge of the problem domain, expertise in programming and experience within a working environment. It is not surprising that this background will influence their acceptance of a new tool. A new tool has a much better chance of being adopted permanently if it works with and complements existing tools. For example, a UNIX programmer who has been working with command-line tools for many years is likely to be biased against an integrated development environment with a feature-rich graphical user interface. However, matching interface styles by itself may not be sufficient. For example, the Red Hack team consisted of three people who had UNIX experience; one of them used vi and the other two used emacs, but preferred working with different highlighting modes.

There is a cost to installing and learning a new tool. Consequently, a task needs to be sufficiently large, difficult, or long-lived that the user can amortize the time investment and realize the benefits. There are other costs that are not immediately obvious. Learning to use the tool involves more than just learning the interface, the user also needs to understand the fundamental concepts underlying the tool. For example, PBS and Rigi can be used to depict any type of graph with attributes. The designers of these tools use them to construct particular views of a software system but the reverse engineering processes they follow are not necessarily described in the documentation. Another cost often not considered is that the tool may need to be tailored to work with the local environment and subject system. The modifications may involve changing the parser, writing scripts to automate tasks, or writing utilities to add information to a repository.

6.3 Lessons for Organizers of Evaluations

The structured demonstration provided a public opportunity for researchers and developers to demonstrate their tools on a common subject system using prescribed tasks. We developed the structured tool demonstration to overcome some of the flaws in other tool evaluation methods such as case studies, technology demonstrations, and experiments. It allowed us to see expert users and expert programmers using the tools on a medium-sized software system on realistic tasks.

The final design that we used in the tool demonstration was quite complex. It had many elements: required tasks, optional tasks, deliverables, observers, presentations and a panel. There are some things we could have done differently. Pilot testing is a very important stage in any experimental design, and we unfortunately overlooked it. A pilot test would have indicated that we should have included another task that was more difficult to complete in order to have a set of questions that was maximally discriminating. More inter-tool observations, that is, observations that compared the tools, in addition to having assigned observers to individual teams would have been helpful. We did some time-stamped observations, which were invaluable, but they were a last-minute idea.

There was a general tendency to create very terse documentation, both for task 1.1 and for the exercise as a whole. Most groups handed in a total of 3 pages. Lemma was a notable outlier, producing 24 pages, including diagrams and code excerpts. It is unclear why this occurred. There may have been a general reluctance to write documentation, or the participants may have felt constrained by the time limits or were unaccustomed to the artificial nature of the deliverables. In hindsight, we probably should have given more explicit instructions for documentation or asked for more detailed deliverables.

Another possibility is that the groups may have found the maintenance tasks to be more appealing. After providing a 3-paragraph description of the subject system, the Red Hack team wrote “We headed straight for the interesting tasks.” Despite the fact that the teams were required to do only one of the maintenance tasks, they opted to do all or almost all of them. In general, the teams began with the maintenance tasks and left the documentation tasks till the end. One possible explanation for this is that there is a general tendency by programmers to avoid writing documentation. Moreover, by performing the maintenance tasks first, this allowed them to glean information about the system that they later used to complete the reverse engineering tasks. This approach is consistent with Singer and Lethbridge’s model of just-in-time program comprehension. [19, 20]

During the panel portion of the workshop, a participant recommended that a single day wasn’t really enough time for the additional capabilities of the research tools to prove themselves and to justify the initial costs of loading the subject system. We would agree with this point and say that the design of the workshop is not perfect.

Despite some criticisms and imperfections, there were a lot of successes. One benefit that was not anticipated was the community building that occurred over the three days. We deliberately did not give any instructions on collaboration, neither condoning nor forbidding it. At first, the teams were quite competitive, but when they realized that they were having similar problems they began to work together. However, this sharing did not extend to comparing results as is evident in section 4 and in the Appendix. During the workshop presentations there was a great deal of laughter as the participants looked back at their struggles. The structured demonstration allowed them to see the flaws in each other’s tools fostering a feeling of familiarity that paper presentations and normal technology demonstrations normally do not.

Based on the success of this event we are planning another structured demonstration. This one will focus on parsing tools, because parsing was so problematic for many of the teams in this demonstration. We feel that such a demonstration would have benefits not only for the direct participants, but also to the wider community.

Acknowledgements

We thank the tool development teams and observers for participating in the structured demonstration. We also thank the CASCON organizers from IBM, in particular Homy Dayani-Fard, for their efforts to accommodate our many requests. This work is being supported by IBM Canada Ltd., sponsored by CSER, and funded by NSERC. Our thanks also to Mechthild Maczewski, Bruce Phillips and the anonymous reviewers for their comments.

References

[1] "CASCON Home page" <Available at http://www.cas.ibm.com/cascon>.

[2] "A Collective Demonstration of Program Comprehension Tools." Available at <http://www.csr.uvic.ca/cascon99>.

[3] "The PBS Home Page." <Available at http://www.turing.toronto.edu/pbs>.

[4] "Reverse Engineering Demonstration Project Home Page". <Available at http://pathbridge.net/reproject/cfp2.htm>.

[5] "Rigi Group Home Page." Available at <http://www.rigi.csc.uvic.ca>.

[6] "Visual Age C++ Home Page." Available at <http://www.software.ibm.com/ad/visualage_c++/>.

[7] "*Xfig* Home page." Available at <http://www.xfig.org>.

[8] M. N. Armstrong and C. Trudeau, "Evaluating Architectural Extractors," presented at Working Conference on Reverse Engineering, Honolulu, HI, 1998.

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Partial Redesign of Java Software Systems Based on Clone Analysis," presented at Sixth Working Conference on Reverse Engineering, Atlanta, GA, 1999.

[10] B. Bellay and H. Gall, "A Comparison of Four Reverse Engineering Tools," presented at 4th Working Conferences on Reverse Engineering (WCRE '97), Amsterdam, The Netherlands, 1997.

[11] R. W. Bowdidge and W. G. Griswold, "How Software Tools Organize Programmer Behavior During the Task of Data Encapsulation," *Empirical Software Engineering*, vol. 2, pp. 221-267, 1997.

[12] I. T. Bowman, R. C. Holt, and N. V. Brewster., "Linux as a Case Study: Its Extracted Software Architecture," presented at International Conference on Software Engineering, Los Angeles, CA, 1999.

[13] K. Brade, M. Guzdial, M. Steckel, and E. Soloway, "Whorf: A Visualization Tool for Software Maintenance," presented at 1992 IEEE Workshop on Visual Languages, Seattle, WA, 1992.

[14] T. Bruckhaus, N. H. Madhavji, I. Janssen, and J. Henshaw, "The Impact of Tools on Software Productivity," *IEEE Software*, pp. 29-38, 1996.

[15] B. Curtis, "By the Way, Did Anyone Study Any Real Programmers?," presented at First Workshop on Empirical Studies of Programmers, Washington, D.C., 1986.

[16] P. J. Finnigan, R. C. Holt, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong, "The software bookshelf," *IBM Systems Journal*, vol. 36, pp. 564-593, 1997.

[17] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An Empirical Study of Static Call Graph Extractors," *ACM Transactions on Software Engineering and Methodology*, vol. 7, pp. 158-191, 1998.

[18] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. M. Cox, "Browsing and Searching Software Architectures," presented at International Conference on Software Maintenance, Oxford, England, 1999.

[19] J. Singer, T. Lethbridge, and N. Vinson, "An Examination of Software Engineering Work Practices," presented at CASCON '97, Toronto, Canada, 1997.

[20] J. Singer and T. C. Lethbridge, "Just-In-Time Comprehension vs. the Full-Coverage Strategy," presented at Workshop on Empirical Studies of Software Maintenance, Bethesda, MD, 1998.

[21] M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Muller, "On Designing an Experiment to Evaluate a Reverse Engineering Tool," presented at Working Conference on Reverse Engineering, Monterey, CA, 1996.

[22] M.-A. Storey, K. Wong, and H. A. Muller, "How do Program Understanding Tools Affect How Programmers Understand Programs?," presented at WCRE '97, Amsterdam, Holland, 1997.

[23] A. von Mayrhauser and S. Lang, "On the Role of Static Analysis during Software Maintenance," presented at International Conference on Program Comprehension, Pittsburgh, PA, 1999.

[24] N. Wilde, S. W. Dietrich, and F. W. Calliss, "Designing Knowledge-Based Tools for Program Comprehension: A Comparison of EDATS and IMCA," University of Florida, Technical Report SERC-TR-79-F, December 1995.

Appendix: Summary of Results

	PBS	**Rigi**	**Lemma**	**TkSee**	**Red Hack**
Total Deliverables	approx. 6 pages, incl. 6 diagrams	3 pages, incl. 1 diagram	24 pages	3 pages	3 pages
Q1.1 Documentation	approx. 4 pages including 5 diagrams	1 diagram	8 pages, including call graphs and code excerpts	3 text paragraphs	3 text paragraphs
Clustering?	-file name prefix: draw, edit, file, GUI, util	-file names and containment -clusters not specified	1. Functional: cmd_panel, mode_panel, ind_panel 2. Components: draw, edit, file, update, window	-file name prefix: drawing, events, file, utilities, window	-file name prefix: drawing, edit, file, user interface, windows
Q1.2 Design assessment	-low cohesion and high coupling -a single menu item distributed over several files and subsystems	-understandable overall, but interdependent	-high cohesion and low coupling -use of global vars not excessive	-code uncommented, lots of external variables, lots of function pointers	-duplicated function names, cloned code, poor naming conventions, poor modularization, global state vars
Well designed initially?	-believe it was good	-can't tell	-can't tell -need to see previous versions	-well designed and divided according to clustering scheme	-can't tell
Initial design still intact?	yes, but eroding	-original design complex but later changes improved organization	-can't tell	yes	yes
Difficult to maintain and modify?	-not bad now, but will worsen over time	no	-reasonably easy	-a little difficult	no
Some modules unnecessarily complex?	no	yes (did not name)	no	no	no
GOTOs—how many and removal	4 2 in main.c 2 in f_wrgif.c -suggested using flags to remove	5 -do not degrade quality, so don't remove	5 2 in main.c 2 in f_wrgif.c 1 in f_readgif.c -suggestions plus time estimates	3 2 in f_wrgif.c 1 in f_readgif.c -a suggestion for each	5 -remove 1 in main.c, leave the rest
Q2.1 Change command panel	-modify array in w_cmdpanel.c -plus other files	-modify array in w_cmdpanel.c -added 3 structs	-modify array in w_cmdpanel.c -verify calling context	-modify array in w_cmdpanel.c	-modify array in w_cmdpanel.c -plus other files
Q2.2 New method for specifying arcs	-change w_modepanel.c and add new files with functionality according to naming convention	-changed existing arc specification method	-change w_modepanel.c and add new files with functionality according to naming convention	-changed existing arc specification method	-change w_modepanel.c and add new files with functionality according to naming convention
Q2.3 Bug fix	-did not complete task—could not crash program	-did not complete task	-identified possible causes	-gave long list of files to change but no description	-magic constant not set correctly