On Designing an Experiment to Evaluate a Reverse Engineering Tool

M.-A.D. Storey[†‡] K. Wong[†] P. Fong[‡] D. Hooper[‡] K. Hopkins[‡] H.A. Müller[†]

[‡]School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

[†]Department of Computer Science
University of Victoria
Victoria, BC, Canada

Abstract

The Rigi reverse engineering system is designed to analyze and summarize the structure of large software systems. Two contrasting approaches are available for visualizing software structures in the Rigi graph editor. The first approach displays the structures through multiple, individual windows. The second approach, **S***imple* **H***ierarchical* **M***ulti-***P***erspective (SHriMP) views, employs fisheye views of nested graphs. This paper describes the design of an experiment to evaluate these alternative user interfaces. Various results from a preliminary pilot study to test the experiment design are reported.*

1 Introduction

Numerous reverse-engineering tools have been developed to assist in software maintenance by providing methods to uncover the original (or existing) design of software systems. The usability of these tools is critical to their effectiveness. This paper evaluates a particular reverse engineering tool called Rigi.

The Rigi system is suitable for extracting, analyzing, and documenting the structure of large software systems [1, 2]. The reverse engineering process involves parsing a subject software system, resulting in a graph where nodes represent system artifacts such as functions and datatypes, and arcs represent dependencies among the artifacts. A hierarchy is then imposed on the flat graph by building subsystem abstractions. Software maintainers can subsequently browse and annotate these software hierarchies to aid in program comprehension.

Currently, there are two alternative approaches available in Rigi for browsing subsystem hierarchies [3]. The first (original) approach displays a hierarchy using multiple, overlapping windows, where each window displays a portion of the subsystem hierarchy. A second (newer) approach, **S**imple **H**ierarchical **M**ulti-**P**erspective (SHriMP) views, employs a nested graph formalism to display a subsystem hierarchy in a single window [4]. A zoom algorithm, based on a fisheye-lens metaphor, automatically enlarges and shrinks portions of the graph to ease browsing and navigation in the hierarchy.

The SHriMP approach was developed in response to several deficiencies identified with the multiple window approach. For larger systems, the hierarchy may be very deep and many windows may need to be opened. Positioning and resizing these windows to keep pertinent information visible can be tedious. Since the relationships between windows are typically implicit, it is easy to lose context and become disoriented while navigating larger systems.

The SHriMP interface is implemented in the Tcl/Tk [5] language and is currently a library that has been integrated into the Rigi system. Although Tcl/Tk is a powerful tool for rapid prototyping, one of its shortcomings is that the graphics are very slow and not suitable for interactively browsing large software graphs in Rigi. The designers of the Rigi system intend to tightly couple this interface with the Rigi tool for improved performance. Before undertaking this task, it is wise to evaluate this interface and compare it to the existing Multiple Window interface in Rigi, to ascertain the value and focus of a reimplementation.

This paper describes the design of an experiment to evaluate these two approaches. The experiment design has been refined through its application in a pilot study. Preliminary results from the pilot study are reported.

The two interfaces are compared to each other and also to Unix command-line tools (`vi` and `grep`). Rigi can be used both for creating and browsing software hierarchies. The experiment presented in this paper only addresses the browsing capabilities of Rigi. However, observations were also made by the Rigi experts as they prepared software hierarchies for use in the pilot study.

Before undertaking the pilot study, we expected that Rigi would show the most significant advantage in tasks requiring the user to explore dependency relationships between the functions and data types in the program. We expected that the SHriMP interface would provide a significant speed and ease-of-use advantage over the standard Rigi interface when task completion requires the exploration of heavily nested dependency graphs. In addition, it was expected that the SHriMP interface would alleviate the *lost in space* syndrome experienced by users as they navigate deep hierarchies.

Section 2 describes the two available user interfaces for navigating software structures in Rigi. Section 3 outlines the experiment design and specifics of the pilot study. Section 4 presents the preliminary results of the pilot study. Section 5 interprets the pilot study results, suggests refinements which should be made to the experiment design, and provides recommendations for changes to improve the usability of the Rigi tool. Section 6 is the conclusion.

2 The Rigi system

Rigi is a system for extracting, analyzing, visualizing and documenting the structure of evolving software systems. Software structures are manipulated and explored using a graph editor. The following two subsections describe two alternative approaches for exploring software hierarchies in Rigi.

2.1 Multiple window approach

In the original Rigi approach, a subsystem containment hierarchy is presented using individual, overlapping windows that each display a specific portion of the hierarchy. For example, the user can open windows to display a particular level in the hierarchy, a specific neighborhood around a software artifact, a projection or flattening of the hierarchy, or the overall tree-like structure of the entire hierarchy,

Figure 1 shows the multiple window approach in Rigi for presenting the structure of a small sample program. The program root node, entitled src, is displayed in Fig. 1(a). A user displays the next layer in the hierarchy by double clicking on the src node, see Fig. 1(b). This layer consists of the main function and two subsystems, List and Element. Arcs in this window are called *composite* arcs and represent one or more lower level dependencies in the graph.

The List subsystem has been opened in Fig. 1(c). Nodes in this window are leaf nodes and directly correspond to functions or datatypes in the software. Arcs in this window represent either call or data dependencies. Figure 1(d) shows an overview of the software hierarchy and provides context for the other windows. Arcs in the overview window are called *level* arcs as they represent the parent-child relationships in the hierarchy. Finally, Fig. 1(e) shows a *projection* from the src node. This operation has the effect of flattening the hierarchy and displays all of the lower level dependencies and artifacts in a single window.

2.2 SHriMP views

The SHriMP visualization technique offers an alternative approach for navigating and manipulating subsystem hierarchies in Rigi. In this approach, nested graphs represent the structure and organization of the software. The nesting feature of nodes communicates the hierarchical structure of the software (e.g. subsystem or class hierarchies). A fisheye-view visualization technique is used to enlarge nodes of current interest while concurrently shrinking the remainder of the graph. Fisheye views, an approach proposed by Furnas in 1986 [6], provides context and detail in one view. This display method is based on the fisheye-lens metaphor where objects in the center of the view are magnified and objects further from the center are reduced in size.

The same program is again used to demonstrate how this interface may be used for visualizing software. A user travels through the hierarchy by opening nodes. Nodes and arcs representing the next layer of the hierarchy are displayed inside the open node, as opposed to being displayed in a separate window. In Fig. 2(a) the src node is displayed as a large box. When this node is opened, its children are displayed inside the node as shown in Fig. 2(b). In Fig. 2(c) List's children are displayed inside the List node when it is opened. The Element node has been opened in Fig. 2(d). This view shows the same information as the overview window from the Multiple Window approach. The containment feature of the nested nodes depicts the parent-child relationships among nodes in the software hierarchy.

Composite arcs may be opened in the SHriMP views to show the lower-level dependencies that the arcs represent. A user opens a composite arc by double-clicking on it to display the lower-level arcs. In Fig. 2(e) composite arcs between the main function and the List and the Element subsystems have been opened. In this view, all of the lower level dependencies and artifacts are visible.

The next section in this paper describes the design of an experiment to evaluate these two interfaces in Rigi.

3 Experimental methods

This section describes the design of an experiment to evaluate the usability of three user interfaces:

Command-Line: online source code and documentation, with `vi` and `grep` Unix command-line tools;

Multi-Win: multiple window approach in Rigi;

SHriMP: SHriMP views approach in Rigi.

Each interface is tested by asking the users to complete a series of typical software maintenance tasks under controlled

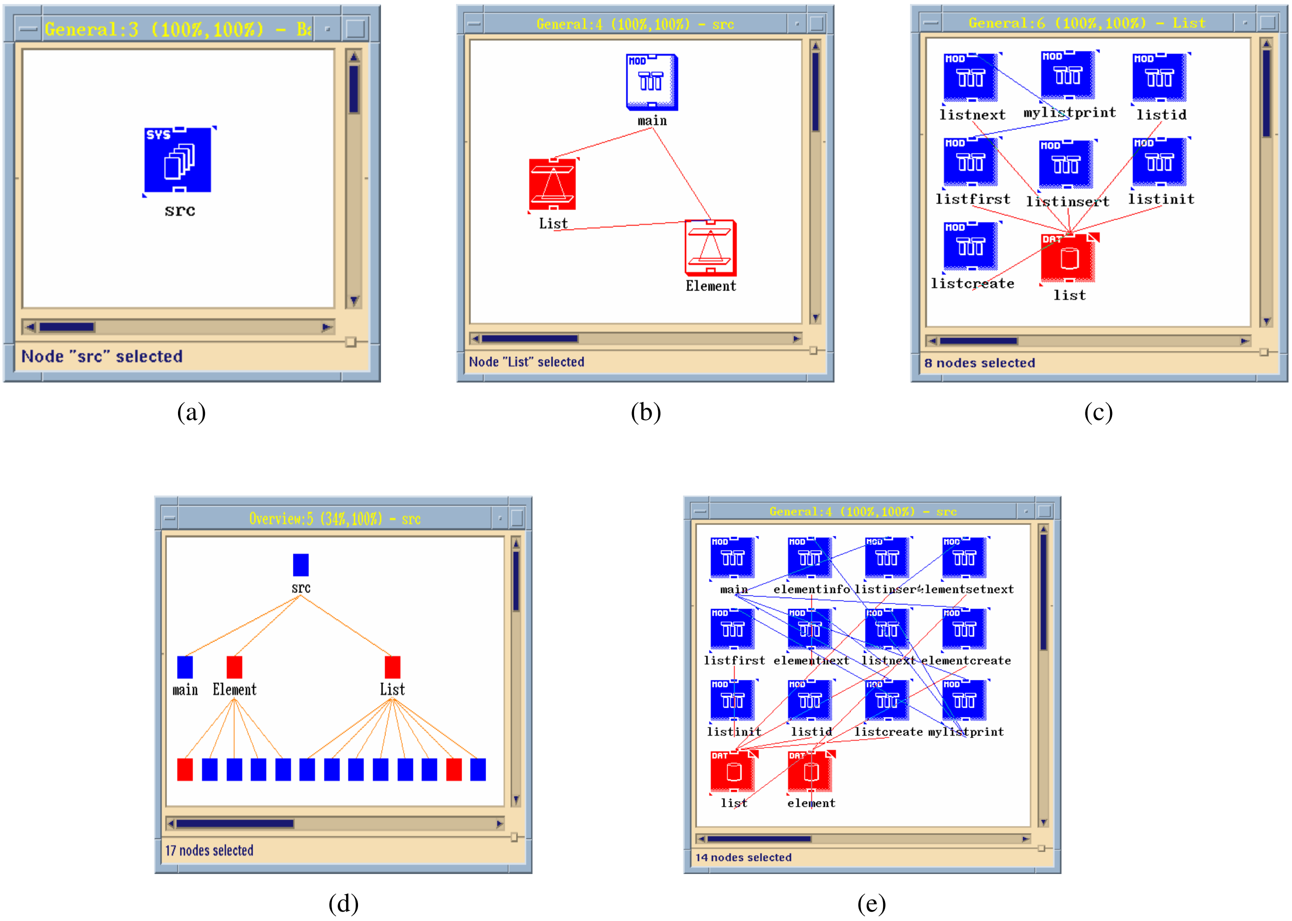

Figure 1: (a) This window contains the root node of the program, entitled src. (b) This window contains the children of src: main, List and Element. (c) This window appears when a user opens the List node. (d) This window is an *overview* window and provides context for the other windows. (e) A *projection* from the src node is performed to show lower level dependencies between the subsystems.

and supervised conditions. After finishing the tasks, the users are asked to complete a prepared questionnaire. Finally, informal interviews are conducted to stimulate the users into revealing relevant thoughts not expressed while answering the questionnaire.

A small pilot study was conducted at the University of Victoria and Simon Fraser University according to the experiment design. Parameters of this study to test the design are mentioned in the relevant following subsections.

3.1 Hypothesis

Null hypothesis: *Command-Line, Multi-Win, and SHriMP are (pairwise) equally effective under the same conditions.*

3.2 Experimental variables

The independent variables in the experiment are:

- the user interface,
- complexity of the test program,
- complexity of software maintenance task, and
- level of user expertise.

The following dependent variables are assumed to be influenced:

- correctness of tasks,
- time taken to complete tasks,
- subjective user satisfaction, confidence, and productivity.

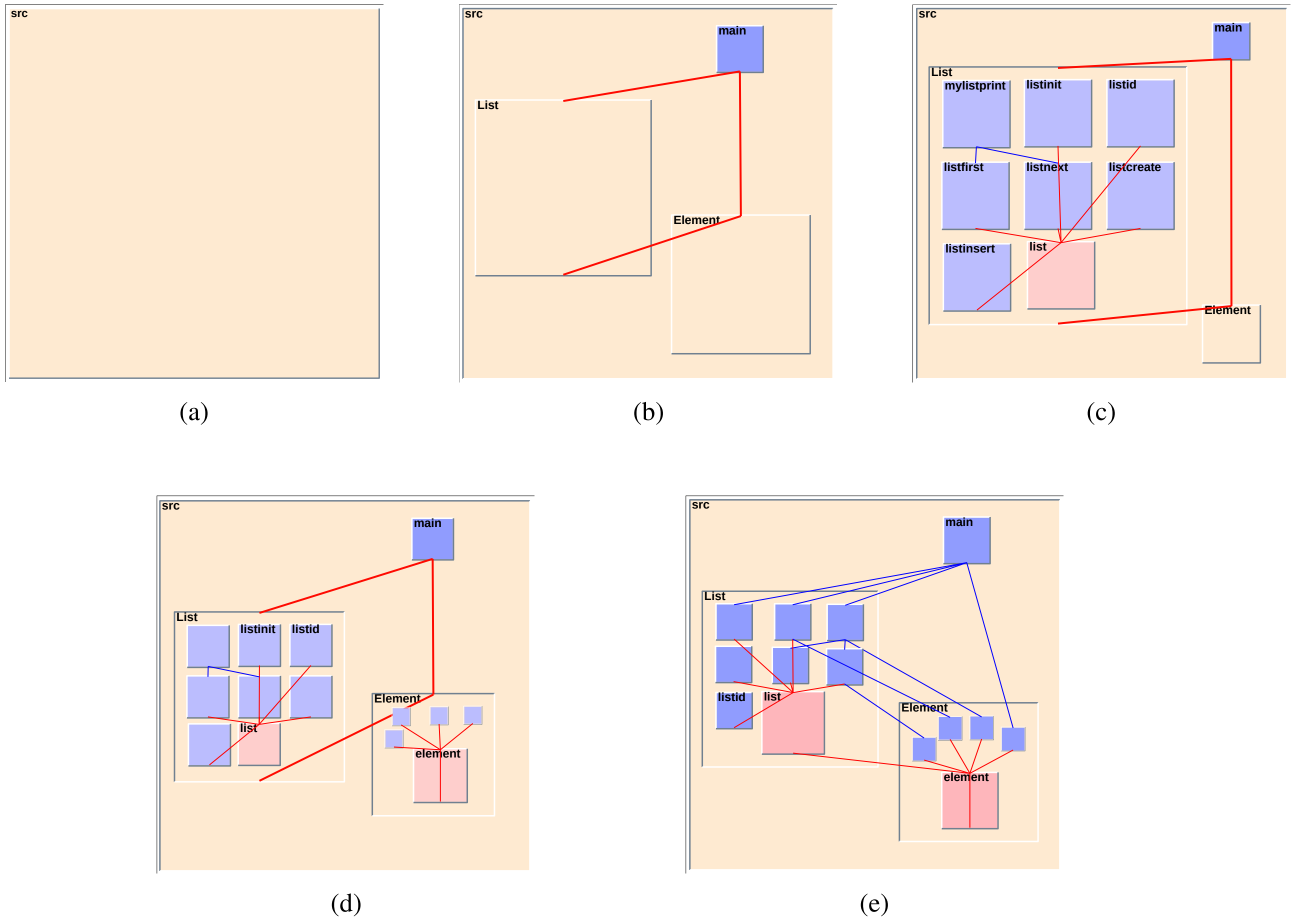

Figure 2: (a) This figure shows the root node of the program, entitled src. (b) This figure shows *src*'s children: main, List and Element, displayed inside src. (c) This figure shows how List's children nodes are displayed inside List when it is opened. (d) The Element node has also been opened to display its children showing an overview of the entire system. (e) Composite arcs are opened to display lower level dependencies.

3.2.1 User interfaces

To effectively increase the number of users in the pilot study, each user was assigned tasks using each of the three interfaces. This had the added advantage that the users can also compare the usability of the three interfaces. For each user, the Command-Line interface was tested first, followed by Multi-Win, with SHriMP last. Although some bias is introduced because of this fixed order, it is unavoidable unless the group of users is large enough to allow randomizing the order of the interfaces.

3.2.2 Test programs

If a single program is used throughout the experiment, then knowledge gained by a user from examining the program using one interface could be exploited while using a subsequent interface. To prevent this, a different program is needed for each interface tested by a user. Since each user tests three interfaces, three different programs are required. Some bias is introduced since the programs are necessarily different. To offset this bias, the assignment of a program to a user interface is randomized uniformly over all users in the experiment.

Because of this randomization, the three programs need not be of similar size or complexity. By selecting programs of varying size, it is possible to examine the effect of program size on the use of each interface.

In the pilot study, we used three programs that were similar in complexity but differed in size.

The programs were implementations of games written in the C language:

Fish: approx. 300 lines, one source file;

Hangman: approx. 300 lines, 12 source files;

Monopoly: approx. 1700 lines, 18 source files.

These lines of code counts do not include comments.

3.2.3 Tasks

A common series of tasks is assigned to each user. Ideally, complex software maintenance tasks involving several steps could be prepared. Due to time constraints, a trade-off between task complexity and task completion time is necessary. Instead of asking users to perform particular tasks (such as fixing a software bug), we chose to have them perform small tasks that are commonly done by software maintainers to attain larger goals of fixing errors or adding new features.

In the pilot study, there were two categories of tasks: *abstract* and *concrete*. Abstract tasks are high-level program understanding tasks and involve gaining an understanding of the overall structure or design of the program. Concrete tasks are low-level program understanding tasks and may involve understanding only small portions of the test program. Answers to the concrete tasks should be unambiguous.

Reasonable time limits on the individual tasks should be imposed to ensure that all tasks are at least attempted. In the pilot study, users were given 20 minutes to complete all eight tasks, where each task had a set time limit. If a user could not finish a task by the allotted time, we would remind the user to leave it and move on to the next task.

3.2.4 User expertise

The level of user expertise and skill will affect an individual's performance. Also, user familiarity with the `vi` and `grep` tools gives an unfair advantage over the Rigi interfaces. However, we tried to offset this advantage by training the users on the Rigi interfaces and by having experts prepare software hierarchies of the test programs for each of the interfaces. In the pilot study, 12 users of similar skill level participated in the experiments. The users volunteered their time and were unpaid. These 12 users consisted of 10 graduate students and 2 senior undergraduate students from the University of Victoria and Simon Fraser University.

Domain knowledge can give a user a head start by providing useful preconceptions. This knowledge may contribute significantly to program understanding and must be considered. For the pilot study, the first task asks whether a user is familiar with the game implemented by the test program.

3.3 Experimental procedure

The experimental procedure for each user is outlined in Fig. 3. Experiments may be run in parallel but in separate rooms. In this case, it may be best to train multiple users at the same time. In the pilot study, each user experiment lasted between 1.5 and 2 hours.

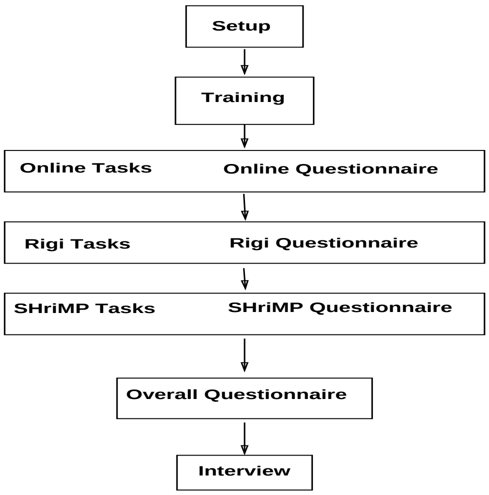

Figure 3: Phases of the experiment.

3.3.1 Setup

In any experiment, properly controlled conditions are needed to obtain results with reasonable confidence. The experimenter's handbook details what must be done during each phase of the experiment. The handbook specifies how to introduce the users to the experiment and provides instructions on setting up the workstation for each phase. These protocols ensure that the experiment proceeds smoothly and consistently, reducing the likelihood of mishaps that might affect user performance.

3.3.2 Training

For each user interface, a specific training module in the experimenter's handbook outlines the features to be used by the users, along with demonstrations of several example tasks.

In the pilot study, we emphasized that the interfaces were being tested, not the users. To reduce frustration due to time constraints, we also told them that we did not expect them to complete all the tasks, but that we were more interested in how they attempted to solve a task using a particular interface. This helped relax the users considerably, although it appeared that they did strive to complete the tasks correctly. The training time took between 30 and 40 minutes for each user. The user did not perform any practice tasks. We stressed that users did not have to remember how to access all of the features. They could ask for help during the experiment, but not ask for assistance in completing a task.

3.3.3 Tasks

The abstract tasks used in the pilot study were:

1. Show familiarity with the game.
2. Summarize what subsystem x does.
3. Describe the purpose of artifact x.
4. On a scale of 1-5, how well was the program designed?

The concrete tasks for the pilot study were:

5. Find all artifacts on which artifact x directly or indirectly depends.
6. Find all artifacts that directly or indirectly depend on artifact x.
7. Find an artifact that is not used.
8. Find an artifact that is heavily used.

3.3.4 Questionnaire

The questionnaire is designed to evaluate and compare the usability of the interfaces through user feedback. The design of the usability questionnaire is based on the IBM Post-Study System Usability Questionnaire (PSSUQ) [7]. The questionnaire is presented to a user after all tasks have been completed with a given user interface.

For the pilot study, we adapted the PSSUQ slightly to ask 20 questions in 5 categories:

overall: all 20 questions evaluate overall user satisfaction;

sysuse: 8 questions evaluate interface usefulness;

interqual: 3 questions evaluate interface quality;

organization: 4 questions evaluate helpfulness of module organizations in the interface;

confidence: 4 questions evaluate user confidence in the answers generated by the interface.

Questions in a category are subtle rewordings of each other to help stimulate responses. The ordering of all questions were randomized.

In addition, the following questions were asked in the pilot study after a user had completed testing all of the user interfaces.

1. Rank the three systems in order of their perceived effectiveness at helping to understand the software.
2. Hypothetically choose a system for a future software maintenance project.
3. Name the three most preferred features in the user interfaces tested.

3.3.5 Interview

An informal interview is held at the close of each experiment. The purpose here is to determine what difficulties the users encountered in using each interface and to extract more about their opinions of usability.

3.4 Recording observations

It is not possible to extract all the required results from task answers and questionnaires alone. To determine expected and unexpected difficulties, experimenters need to record observations of the users completing the task sets. For example, a user may correctly answer a task by using an unorthodox method or even by pure chance. The experimenter verifies assumptions about what the user is thinking by asking appropriate questions, taking care not to unduly interrupt. After the task set has been completed and while the user fills in the questionnaire, the experimenter also records a summary of how the user performed.

In the pilot study, we used several methods of recording observations:

Think aloud: The users were asked to verbalize their thoughts as they attempted a task. This allowed the experimenter to gain a better understanding of what each user was trying to accomplish.

Video taping: One or two video cameras recorded each of the experiments, where one camera captured actions on the computer screen and the other captured the user's facial expressions and verbal comments.

Experimenter comments: Most of the experiments had two experimenters present. One experimenter interacted with the user while the other served as a silent observer.

3.5 Analyzing the results

To maintain consistency while assessing the correctness of the tasks, experimenters make use of prepared answer keys. The assessment of answers to the abstract tasks are somewhat subjective.

In the pilot study, for the task results, we looked for non-normality of the samples, performed an ANOVA with the

Scheffé method, and computed two-sample t tests, where possible, to determine instances where the null hypothesis could be rejected.

4 Pilot study results

The purpose of the pilot study was to evaluate the experiment rather than the interfaces. Nevertheless some interesting results were observed that could serve as interesting hypotheses for the next experiment. This subsection describes the results from the pilot study.

4.1 Task results

The tasks were judged using a prepared answer key. Due to the small sample size, tasks 1 and 4 were not included in the analysis. (Task 1 determined the user's domain knowledge of the game and task 4 enquired about the user's mental model of the program.) The results of the other tasks appear in Table 1. There were some findings where the null hypothesis was rejected (one interface found less effective or worse than another). For concrete tasks on the large Monopoly program, Command-Line was worse than Multi-Win ($P = 0.01$) and Command-Line was worse than SHriMP ($P = 0.0005$). For concrete tasks on the very small Fish program, Command-Line was worse than SHriMP ($P = 0.05$) and Multi-Win was worse than SHriMP ($P = 0.005$), with Command-Line tending to be somewhat better than Multi-Win ($P = 0.1$).

4.2 Questionnaire results

Preliminary results seem to suggest that the users were more satisfied with SHriMP than Multi-Win, and more satisfied with Multi-Win than Command-Line. A different picture emerges, however, when the results are divided according to the three test programs (see Fig. 4). Looking at the "overall" questionnaire category, user satisfaction with SHriMP is lower than Multi-Win for the Monopoly test program. The same pattern holds for the other questionnaire categories.

When asked to hypothetically choose a user interface for their next software maintenance project, 8 users chose SHriMP, 3 chose Multi-Win, and only 1 user chose Command-Line.

4.3 Observations

This subsection describes observations made for each of the three interfaces. The quotes relating to each of the interfaces were made by users during the experiments.

4.3.1 Command-Line

> *"If I knew the structure of the program maybe I could guess what is called frequently."*

For the most part, the users were able to effectively utilize the `vi` and `grep` tools, due to previous programming experience with these tools. For those with extensive programming experience, their performance with this interface was quite successful.

Some of the tasks may have been unrealistic for the Command-Line tools and may have been biased towards the Multi-Win and SHriMP interfaces. For example, a task which asks to name all functions called directly or indirectly by another function is a much easier task for the Rigi tool. More experienced users often used heuristics, or "guesses" to try to answer these types of tasks. When a user had an understanding of how the games are played, they would use this knowledge to answer the question. Other users went about these tasks in an ad hoc manner, and quickly gave up. Only a few attempted to thoroughly and accurately complete the tasks.

4.3.2 Multi-Win

> *"It would be necessary to get more familiar with Rigi [Multi-Win] in order to properly judge it."*

In general, many of the users seemed quite pleased with the graphical representation of the software. However, some problems were often observed. Most of the users had difficulties understanding the purpose of the overview window. Arcs in this window show the parent-child relationships of subsystems, but these arcs were often confused with call or data dependency relationships that are shown in the general windows.

In addition, many users did not at first remember that a composite arc represents one or more lower-level arcs. Indeed, they had to be reminded that the projection feature in Multi-Win should be used to view the lower-level dependencies. Some had to be reminded of this more than once.

The training time for Multi-Win was too short. This was obvious since the users were initially unsure how to solve the first few tasks using Multi-Win. They did improve their performance during the experiment, but they still had to ask for help with the interface.

Also, users often opened windows that were already displayed. This increased the user's cognitive load as they scanned the windows trying to identify pertinent artifacts.

4.3.3 SHriMP

> *"When you gave the tutorial ... I thought that SHriMP would be the worst ... but it turned out that it was easier."*

Table 1: Task Results

User Interface	Test Program	Task Type	Mean	Std Dev	Variance
Command-Line	Fish	Abstract	0.72	0.36	0.13
		Concrete	0.75	0.38	0.14
	Hangman	Abstract	0.83	0.30	0.09
		Concrete	0.56	0.44	0.19
	Monopoly	Abstract	0.47	0.47	0.22
		Concrete	0.52	0.45	0.20
Multi-Win	Fish	Abstract	0.84	0.23	0.05
		Concrete	0.55	0.42	0.18
	Hangman	Abstract	0.65	0.43	0.18
		Concrete	0.68	0.47	0.22
	Monopoly	Abstract	0.60	0.42	0.18
		Concrete	1.00	0.00	0.00
SHriMP	Fish	Abstract	0.88	0.31	0.09
		Concrete	0.96	0.10	0.01
	Hangman	Abstract	0.88	0.23	0.05
		Concrete	0.79	0.40	0.16
	Monopoly	Abstract	0.75	0.35	0.13
		Concrete	0.95	0.15	0.02

The SHriMP interface appeared to be quite intuitive. The users liked being able to see all of the nodes in one window because they could better see how everything was connected. In particular, opening composite arcs seemed intuitive. However, we did observe some users would only open composite arcs connected to the immediate parent node when trying to view lower-level dependencies connected to a particular node. They would often overlook composite arcs which were connected to higher levels of subsystem abstractions.

Displaying everything in one window did lead to some complaints. Users had difficulties in determining the nodes that an arc connected. This happened especially when several composite arcs were opened to show many lower-level arcs. Most users dealt with this complexity by moving irrelevant nodes to one side to give a clearer view of the arcs of interest.

Tcl/Tk was useful for rapid prototyping of the SHriMP interface. However, the responsiveness of the resulting interface was poor for large graphs. Operations to move and scale nodes were particularly tedious. Many users quickly realized this and gave up trying to move or scale nodes in larger graphs.

5 Discussion

In this section, we discuss the results from the pilot study experiment. These include an interpretation of the tasks and questionnaires, suggested refinements to the experiment, and recommendations for changes to the Multi-Win and SHriMP interfaces.

5.1 Interpretation of results

From the task results (which measure the effectiveness of the systems), there was a slight tendency for Multi-Win to outperform Command-Line and for SHriMP to outperform Multi-Win. However, this may be due to the bias of fixing the order of the interfaces for each user. The users probably gained knowledge on how to tackle the tasks using the first two interfaces even though test programs differed.

Based on the concrete task results, the users seemed to use Command-Line more effectively than Multi-Win for smaller programs. This contrasts with the questionnaire results which suggest that the users preferred Multi-Win even for the smaller test programs. This confirms other experiments that compared graphical and textual representations of software. In those experiments, user performance did not improve with graphical representations, even though the users perceived them as more effective [8].

The questionnaires ranked the Multi-Win interface over the SHriMP interface for the larger Monopoly program. This

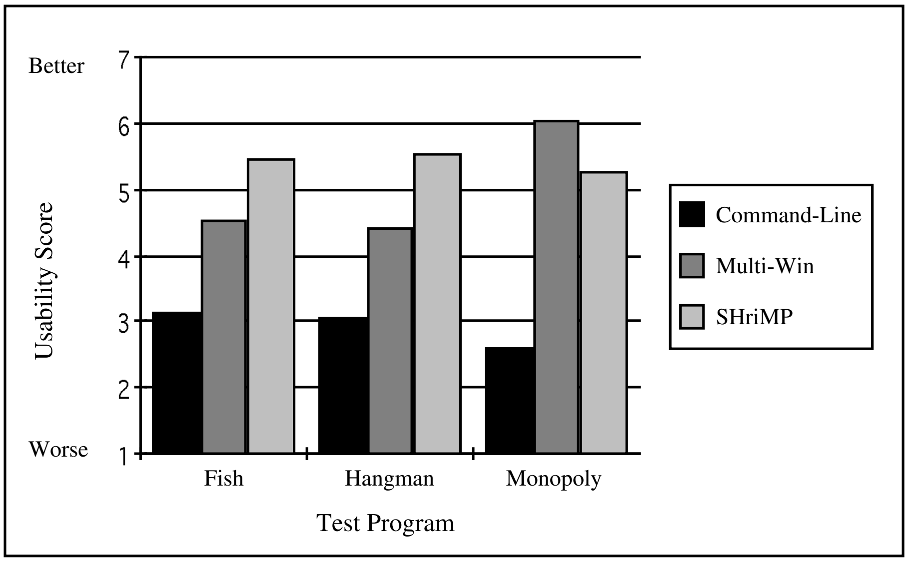

Figure 4: This chart shows the usability scores for the overall questionnaire category.

suggests that user satisfaction might be sensitive to the program size; users are less satisfied with SHriMP when they are dealing with a large program. Two plausible explanations are: (1) responsiveness of the SHriMP interface was slow; (2) too many arcs cluttered the SHriMP window.

5.2 Refinements

In conducting the pilot study, several minor difficulties and a few major problems with our initial experiment design were uncovered.

We performed a *dry run* of the experiment using an experienced Rigi user. This early test identified major problems which were remedied for the pilot study. Admittedly, we did not have the foresight to develop an experimenter's handbook. The necessity of such a document was realized immediately upon running this test. We also realized that the original prescribed tasks were not simple enough to be completed in the time allotted. Some tasks were removed. The final task set used in the pilot study was described in Sec. 3.3.3.

To support a useful statistical analysis, more users, more tasks, task timings, and tighter controls over the running of the experiment are needed.

A concern with the current experiment design is that users can learn from performing tasks with preceding interfaces, influencing their performance with subsequent interfaces. Given enough users, future experiments must either randomize the order of the user interfaces or normally distribute the users into three groups where each group tests only one interface.

A longer experiment time would help since the training phase was too short for users to learn how to use all three interfaces effectively. Practice tasks should be a part of the user training.

All users had difficulty overcoming idiosyncrasies in the Multi-Win and SHriMP interfaces, due to the prototypical nature of both interfaces. These problems are discussed in the next subsection.

5.3 Recommendations

Based on observations and user comments, several improvements to the Multi-Win and SHriMP interfaces are recommended.

In Multi-Win, users often forgot (or never discovered) the context of individual windows. They often opened several windows of the same view, failing to recognize that these views were already available. Some way of emphasizing the relationship of the open windows to the corresponding composite nodes is needed.

There was also confusion between the interpretation of the general windows and the hierarchy overview. Some users misinterpreted the parent-child relationships in the overview as call or data dependencies. The appearance of the overview window should differ from the general windows. This might be achieved by simply having different background colors for the different window types.

The single most important problem with SHriMP views was the slow response of the interface. Since SHriMP views are based on direct manipulation, users expecting immediacy were disturbed by the slow response. This must be addressed in a future reimplementation of the SHriMP interface in Rigi.

Another problem with SHriMP was that it is possible to become intimidated by the large number of arcs revealed by opening several composite arcs. Methods to make it easier to identify arcs of interest and filter uninteresting arcs are required.

For the experiments, four Rigi experts created software hierarchies for each of the three programs. One set of hierarchies was then selected to be used in the pilot study. For the smaller programs, it took around 30 minutes to create a software hierarchy, and around 45 minutes for the Monopoly software hierarchy. These experts made use of both interfaces, but were particularly satisfied with the ability to see multiple levels of abstraction concurrently in the SHriMP views. The SHriMP interface was deemed more desirable for the *drag and drop* paradigm of adding nodes to subsystem abstractions.

In general, both the Multi-Win and SHriMP interfaces have advantages and disadvantages. Future versions of Rigi should include the ability to seamlessly switch between the two interfaces when reverse engineering a software system.

6 Conclusions

This paper describes the design of an experiment for evaluating two contrasting interfaces in a reverse engineering tool. The experiment design has been refined through its application in a pilot study held at the University of Victoria and Simon Fraser University, using 12 users. This experiment will be implemented with a larger number of users at the University of Victoria and Simon Fraser University in Spring 1997. The user group for this larger experiment will include professionals from industry. In the meantime, smaller experiments will be performed to test individual components of the reimplementation of the SHriMP interface. In the future, we would also like to perform experiments using larger software examples and to evaluate not only how software engineers browse software hierarchies, but also how they make use of these tools for creating software hierarchies when documenting or reverse engineering a software system. We look forward to analyzing the results from these future experiments.[1]

[1] For more information, please email: mstorey@csr.uvic.ca.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the University of Victoria, and Simon Fraser University. The authors thank Jim McDaniel and the anonymous reviewers for their helpful comments.

References

[1] S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4), December 1994.

[2] K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.

[3] M.-A.D. Storey and H.A. Müller. Manipulating and documenting software structures using shrimp views. *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95), Opio (Nice), France*, October 16-20, 1995.

[4] M.-A.D. Storey and H.A. Müller. Graph layout adjustment strategies. In *Proceedings of Graph Drawing 1995,* (Passau, Germany, September 20 - 22, 1995), pages 487–499. Springer Verlag, 1995. Lecture Notes in Computer Science.

[5] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[6] G.W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86,* (Boston, MA), pages 16–23, April, 1986.

[7] James R. Lewis. IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instruction for Use. *International Journal of Human-Computer Interaction*, 7(1):57–78, 1995.

[8] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.