A Comparative Study of Three Program Exploration Tools

Brian de Alwis, Gail C. Murphy
Dept. of Computer Science
University of British Columbia
{bsd,murphy}@cs.ubc.ca

Martin P. Robillard
Dept. of Computer Science
McGill University
martin@cs.mcgill.ca

Abstract

Programmers need tools to help explore large software systems when performing software evolution tasks. A variety of tools have been created to improve the effectiveness of such exploration. The usefulness of these tools has been argued largely on the basis of case studies, small narrowly-focussed experiments, or non-human-based experiments. In this paper, we report on a more rigorously controlled study of three specialized software exploration tools in which professional programmers used the tools to plan complex change tasks to a medium-sized code base. We found that the tools had little apparent effect; the effects observed instead appear to be dominated by individual styles and strategies of the programmers and characteristics of the tasks. In addition to presenting the results of the study, this paper introduces the use of two experimental evaluation aids: the NASA *Task Load Index* (TLX) *for assessing task difficulty and distance profiles for assessing the degree to which programmers remain on-track.*

1. Introduction

Programmers spend a considerable amount of their programming efforts exploring source code [7, 10]. During such explorations, programmers sometimes miss seemingly obvious information due to difficulties in obtaining, recognizing, or synthesizing relevant information from the program text [17]. A number of exploration tools have been proposed to help overcome such problems [e.g., 13, 18]. The effectiveness of these tools has been argued largely on the basis of case studies, narrowly-focussed experiments, or non-human-based experiments (Section 2).

As part of our work on a new exploration tool, we undertook a controlled experiment to compare the effectiveness of three software exploration tools (Section 3) based on the following three questions:

- Do exploration tools alleviate the demands made upon a programmer's *mental resources* during a task?
- Do exploration tools help a programmer focus on the program elements relevant to the task and avoid irrelevant elements? (*exploration behaviour*)
- Do exploration tools help a programmer garner a better understanding of how the task should be carried out on the code? (*correctness*)

The experiment used the Eclipse integrated development environment (IDE) as a baseline and three specialized exploration tools: JQuery [5], Ferret, and Suade [12]. Each of these tools uses and presents structural information in a different way and represents a different point in the design space for a structurally-based program exploration tool. To help ensure a realistic setting for our experiment, we recruited professional programmers and had the programmers use the tools to plan complex changes to an open-source codebase.

We found that any observable effects appear to be dominated by individual styles and strategies of programmers and characteristics of the tasks. We found no evidence that the use of the exploration tools lead to better quality solutions (Section 4). We also found that what were seemingly similar tasks to us were not perceived as such by the participants. This paper makes four contributions.

1. We introduce the use of NASA Task Load Index (TLX) [2] to gauge how the participants view task difficulty.
2. We introduce the use of distance profiles to evaluate the degree to which a programmer remains on-task during their program exploration.
3. We present evidence that any effects of the exploration tools are likely dominated by the strategies employed by a programmer and by the task.
4. We show that the use of a small set of metrics to argue the effectiveness of exploration tools may not reflect the impact of the tools.

2. Related Work

Most tools for helping software programmers navigate and explore source code have been evaluated using qualitative case study methods [e.g., 13]. These studies provide rich detail about particular situations but are often limited in the range of situations and users considered. In performing our experiment, we were attempting to more precisely compare the effects of different tools.

There are only a few reports describing experiments conducted about source code exploration tools. Storey et al. [19] and Sim and Storey [16] describe comparative experiments where programmers attempted the same set of tasks using different exploration tools. Their results are largely qualitative. They found that some tools were better for different *types* of tasks, whether reverse-engineering tasks or program-modification tasks. Our experiment focuses specifically on program modification tasks.

We found five reports of hypothesis-testing experiments to assess the effectiveness of software exploration tool(s) [1, 3, 8, 9, 20]. All five of these experiments used students as their subjects. To assess the tools, all of these experiments used the correctness of answers to questions about a code base. Three of these experiments also measured time-to-completion [1, 3, 20]; the other experiments imposed completion times, which varied from 50 minutes to 5 days. Three of these experiments attempted to ascertain whether the subjects found their respective tools useful [8, 9, 20], but each used a different method. Most of these experiments used a number of elementary information-finding tasks [3, 8, 9, 20], such as "list procedures of functions called by *X*." Meyers and Reiss [9] asked one question to describe the changes necessary to implement new functionality. Binkley [1] asked participants to identify the source of an error in a program, in addition to a series of information recall questions. In contrast, our experiment attempted to determine if using specialized rather than general tools caused professional programmers to perform more effectively on real tasks.

Rajlich and Cowan [11] propose some starting points for standardizing comparative experiments for software tools, suggesting the recording of three measures: (i) accuracy of the answers the programmers give to questions about the program, (ii) the response time for accurate answers, and (iii) the response time for inaccurate answers. These measures do not capture any detail of the programmer's behaviour, nor the support afforded to the programmer. In contrast, we use measures in Section 3.2 intended to analyze and describe the effect of tools on the experience and behaviour of programmers.

3. Experiment

Our experiment sought to assess the effectiveness of three tools introduced to ease the exploration of source code.

3.1. Exploration Tools

We wanted to compare features of three exploration tools: Ferret, our own unpublished tool; JQuery [5]; and a pre-release of Suade [12]. Each of these tools runs within the Eclipse environment.[1] We chose these tools as they represent a range of different approaches for obtaining and presenting static information extracted from program text, and all have a goal of improving program exploration.

JQuery is a query-based source code browser [5] that shows query results in context. A JQuery user can issue a variety of pre-written queries, a superset of the standard queries from Eclipse. Results of a query are displayed as a tree with expandable and collapsible nodes. Individual elements in the tree provide the context for further queries, whose results are shown in-place. This hierarchical display provides an explicit map of the navigation paths as the tree expansions capture the history of a programmer's exploration.

Ferret provides a structured display of information about the *local context* of a program element of interest, based on the assumption that understanding a program element also requires understanding how it relates to the rest of the system. The local context is structured as answers to a set of *conceptual queries* about that element, which include and build-on the standard queries supported by Eclipse. An example of a conceptual query is "what interfaces define (or specify) this method?" The conceptual queries are computed automatically as program elements are selected in the IDE, and the answers are displayed in a separate Eclipse view, categorized by whether they involve *declarations*, are about *inter-class* or *intra-class* relations, or are *hierarchical* in nature.

Suade is an implementation of the topology-analysis algorithm described by Robillard [12]. When requested by the user, Suade recommends additional program elements for investigation using an estimate of their structural relatedness to a previously-identified set of program elements. The recommended program elements are displayed in a separate Eclipse view, ranked according to their estimated relatedness. Elements deemed interesting by the programmer can be added to the input set and used for requesting additional suggestions.

3.2. Hypotheses and Measures

We transformed our study questions (Section 1) into the following three hypotheses.

Hypothesis 1: Programmers using an exploration tool should report a lower mental workload to complete a

[1] Versions: Eclipse R3.1.2 (www.eclipse.org), Ferret 0.2.14.2 (www.cs.ubc.ca/~bsd/research/ferret/), JQuery 3.1.5 (jquery.cs.ubc.ca), and Suade pre-release 0.1.2 (www.cs.mcgill.ca/~swevo/suade/); verified 2007/03/30

task as compared to Eclipse alone.

We expected that an exploration tool should alleviate the demands made upon a programmer's mental resources while undertaking a task. It may be that an exploration tool has little effect on the output produced by the programmer, but improves the programmer's ability to produce the output. We assess this support using the TLX [2], a well-validated and easy-to-administer questionnaire for assessing the *subjective mental workload* experienced by a subject performing a task. The TLX combines a subject's report of six different factors (the mental, physical, and temporal demands; the effort required; the subject's view of his performance; and the frustration experienced) to provide a subjective assessment of workload. To compare scores between subjects, the scores are normalized by taking the difference from a baseline task performed without the treatment of interest.

Hypothesis 2: Programmers using an exploration tool should not need to explore as much code as compared to Eclipse alone.

We expected that an exploration tool should help the programmers focus on the program elements relevant to the task and avoid examining irrelevant elements. We assess this hypothesis in two ways. We first compare the numbers of unique program elements viewed by the programmer in each of the tasks (*Viewed*). Recognizing that some program elements are more relevant to the task solution than others, we also examine the programmer's navigation patterns, comparing the distances strayed from the solution elements during the exploration.

Hypothesis 3: Programmers using an exploration tool should identify more of the salient elements for the task as compared to Eclipse alone.

We expected that programmers using an exploration tool should garner a better understanding of how the task should be carried out on the code through the improved support for exploring relations in the source code. We assess this hypothesis by comparing the quality of their solutions, determined by the percentage of program elements correctly reported from model solutions (*Quality*).

3.3. Design

Our within-subjects experiment compared a programmer's code exploration behaviour when using one of the exploration tools with the same programmer's behaviour when using the standard exploration facilities in the Eclipse IDE for Java development. A within-subjects design satisfied the requirements of the TLX to have a baseline for comparison, and allowed compensating for strong individual differences seen in programmers [e.g., 14, 15].

Each subject was asked to investigate and document a solution for two change tasks. The order of the tasks was

Table 1: Allocation of programmers per treatment group.

	AS–SR		SR–AS	
Tool Combination	T_1	T_2	T_1	T_2
Eclipse / JQuery	G_1	G_1	G_2	G_2
Eclipse / Ferret	G_3	G_3	G_4	G_4
Eclipse / Suade	G_5	G_5	G_6	G_6

randomized between subjects. Each subject used the normal Eclipse facilities for the first task, which served as a baseline for comparison as required for the TLX. For the second task, each subject was randomly assigned to use one of the three exploration tools. The two tasks (AS and SR) are described in Section 3.6. By stressing the documentation to guide another programmer in making the necessary changes, we aimed to emphasize the searching to understand the behaviour of the application.

This experiment uses a two-way, repeated measures, factorial design. The *two-way* refers to the two independent variables (or between-subjects variables): *Tool*, the exploration tool used for the second task; and *Task Order*, the order in which AS and SR were tackled. A *factorial design* tests all combinations of the independent variables, and allows studying the effect of each factor as well as interactions between the factors. We thus have six treatment conditions. Our design has repeated measures as each subject was involved in two trials, referred to as the variable *Trial*. As we were comparing three exploration tools, with two possible orderings of the tasks, and two trials, this is a $3 \times 2 \times 2$ design. This design may be diagrammed as in Table 1, where T_i represents the trial and G_i represents the treatment conditions.

Both the choice of exploration tool and task order were randomized within blocks [6], where a block is a sequence of experiment runs exercising all combinations of variables. This randomization resulted in six treatment conditions; we chose to have three subjects in each treatment condition, resulting in 18 subjects.

3.4. Procedure

Each subject was asked to work on two change tasks to an open-source editor, jEdit 4.1-pre6.[2] This version of jEdit comprises approximately 65 KLOC[3] and 679 classes. The subjects had 40 minutes to perform each task. As the exploration tools use only static program information, the subjects were instructed to not use the debugger. The experiment was performed on a 2 GHz duo-core laptop equipped with 1 GB of RAM and a 1400×1050 screen.

[2]www.jedit.org, verified 2007/03/30.

[3]Thousands of lines of code, only counting non-blank lines of code.

Each subject was instructed to create a plan for performing the tasks, identifying the program elements that either need to be changed or that need to be understood. The plan was to be appropriate for a senior student to rapidly turn into working code. The plan was to be captured using a specialized Relevant Elements view (Section 3.7).

After completing the first task, each subject followed a 10 minute tutorial on the exploration tool to be used in the second task. Each tutorial demonstrated the particular tool's use on an unrelated codebase, JHotDraw 6.0b1.[4] The tutorials were assembled from introductory materials provided by the developers of each tool. To help a subject retain information from the tutorial, we had them use the assigned tool to answer a relatively simple question about JHotDraw; if unsuccessful, the experimenter provided the solution and demonstrated how the tool might be used to obtain the answer. The experimenter also answered any questions and provided tips on tool strategies.

Each subject was asked to use their assigned tool to its fullest extent possible. If the subject was not successful at finding the information, he was allowed to revert to the standard facilities in Eclipse. Because finding a starting point is difficult, we provided an initial *seed*, a single pertinent element to help start each task.

The experiment was piloted on three students. Minor changes were made in light of comments from the first two students, and the third student, who was the most experienced, was able to successfully complete the experiment. Each of these students felt the tasks were of equivalent difficulty. Complete details of the tasks, and the provided source code, are available online.[5]

3.5. Subjects

We recruited 18 professional programmers; each met our criterion of Eclipse proficiency having either used Eclipse for at least 6 months, or having successfully developed an Eclipse plug-in. These programmers came from six different companies or organizations, one of which involved programmers from two very different business divisions. All programmers were male, had a minimum of three years experience writing software in an industrial setting, and had worked on software projects with at least five other programmers. All but one had a minimum of two years experience programming with Java; one programmer had only one year of Java experience.

JEdit was unknown to all but two programmers, neither of whom had seen the code for at least three years. In our follow-up interviews, neither claimed any advantage from that previous exposure.

[4]www.jhotdraw.org, verified 2007/03/30.

[5]www.cs.ubc.ca/~bsd/research/icpc2007/

3.6. Tasks

The experiment involved identifying the changes required for two tasks. One of the tasks, AS, involved adding a capability to the autosaving functionality, and was chosen to be identical to that used in previous work [14]. The other task, SR, involved reloading settings when the settings files were edited within jEdit. We identified this task by examining the jEdit source code and identifying a small piece of functionality that was easily removed. The SR task seemed comparable to AS: both interacted with only small amounts of the overall system, and both used already-existing functionality and required little new code.

Model solutions to the tasks were created by identifying the methods, fields, and types in the existing source that were either necessary to understand or that required modification. The model solutions did not include new methods or new types that might have been necessary. Types were only included when the solution required them to be subclassed, or when their entire workings were required to be understood in detail. The model solution for the AS task was identified from the solution from Robillard et al.'s original study. As the SR task was created by removing existing code, we simply examined the code that was removed.

3.7. Data Collection

We had two primary sources for data collection. The Eclipse workspace was instrumented with the Mylar Monitor [10] to collect information on Eclipse and tool usage. Programmers were also asked to record their development plan using a custom tool called the Relevant Elements view.

The Mylar Monitor provides a record of the exploration and interaction undertaken by a subject during the experiment, called the *monitor history*. Events registered include the selections of user interface (UI) elements, switches between editors, and command executions. Additional instrumentation recorded the program source elements currently visible or made visible through scrolling.

The Relevant Elements view was built for this experiment for the programmers to capture and describe the program elements that are relevant to the task. This view supports one-way interaction only, like paper, meaning that it is easy to add elements to the view, but cannot be used to re-open those elements, so as to prevent its being used as an exploration replacement by the programmers.

The solutions created by the programmers required some clean-up before being processed. Although programmers were encouraged to reference all elements necessary for understanding or implementing the change, most did not. We processed the programmer solutions to add elements that were referred to in their comments. For example, one programmer instructed in his solution:

> This is done by iterating over JEdit.getBuffers() and calling buffer.getAutoSaveFile().delete().

From this description, we added jEdit.getBuffers() and Buffer.getAutosaveFile().

Not all situations were so clear cut. We were conservative, adding elements only when referred to in the solution. For example, in the following excerpt:

> (The setting can be retrieved from JEdit properties)

This programmer had referenced the fetching of properties elsewhere, and so we added jEdit.getProperty().

3.8. Data Analysis

We analyzed the data both visually and statistically to test our three hypotheses.

3.8.1. Visual Assessments

One form of visual analysis entailed plotting and comparing various measurements grouped by the different task orderings and tools.

A second form involved the analysis of plots, called *distance profiles*, of the programmers' exploration of the programs by comparing the *distance* they strayed from the solution elements during the exploration. To produce these plots for a particular task, we first transform the jEdit source code into a directed graph. We map types, methods, and field in the source to nodes in the graph. We add edges for the following relations between program elements: declares field, method, and inner type; extends and implements type; returns type; has argument of type; references field; calls method; creates object instance; throws exception; and catches exception. These edges correspond to relations that can be found in a single step from direct examination of the source code or by using a simple query. We then assign a distance to each node calculated as the node's minimum distance d (ignoring edge directionality) to any of the nodes identified in the model solution (Section 3.6) for that task. A node with $d = 0$ is a solution element. A node with $d = 1$ is a program element that either uses or is used by a solution element. Nodes with $d > 1$ have no direct tie to a solution element.

A plot for a particular programmer using a particular tool is then created by processing the interaction history collected by Mylar Monitor. We determine the distance for each new element that appeared on the programmer's screen and assigned the distance of the screen to be the minimum distance of any elements currently on-screen.

3.8.2. Statistical Assessments

Statistical analyses were used to assess support for our findings through use of ANOVAs. ANOVAs, or Analyses of Variance, are a set of statistical techniques to identify sources of variability between groups. A *one-way* ANOVA assumes a single independent variable; a *two-way* ANOVA assumes two independent variables. ANOVAs allow dealing not only with the individual effects of the independent variables, but also any interactions that may arise between them. The result of an ANOVA is a set of statistical assessments of the different individual effects and effect combinations. We chose a 10% significance level in assessing the statistical tests[6] as it is generally considered suitable for exploratory analysis.

We were unable to test the samples for normality; such tests generally require seven or more samples, and our design with 18 programmers resulted in only three samples per group. However ANOVA is generally robust to non-normality providing the variances in the groups are the same [4], which was confirmed.[7]

4. Results

We present the results of the experiment by examining each of the three hypotheses. We note that all programmers did use their assigned exploration tool throughout their task, although two of the Ferret users, one of the JQuery users, and three of the Suade users also reverted to the Eclipse search facilities on occasion.

4.1. Mental Support

Hypothesis 1: Programmers using an exploration tool should report a lower mental workload to complete a task as compared to Eclipse alone.

Figure 1 shows the differences in TLX scores reported by the programmers, grouped by the different exploration tools and the task orderings, as well as grouped by only the task orderings.[8] Each point corresponds to an individual programmer and corresponds to $t_2 - t_1$ where t_i is the TLX score from task i, and where the second task was undertaken with the exploration tool. A TLX difference greater than zero indicates that the programmer felt that the second task (always undertaken with the exploration tool) was harder than the first, and a difference less than zero that the second task was easier. The data supports the first hypothesis if the TLX TLX differences are generally less than zero, indicating that the tasks in which the exploration tool were used were easier.

From Figure 1, we see that the TLX differences lie both above and below zero regardless of the exploration tool

[6]All statistical tests performed using R 2.3.1 (www.r-project.org) on NetBSD 4.99.1 on a 2 GHz Intel Core Duo.

[7]Homogeneity of variances assessed using Bartlett's test.

[8]We present many of the results graphically using boxplots showing the 0, 25, 50, 75, and 100 percentiles; percentiles are particularly appropriate for summarizing data with skew. Results from individual programmers are represented by single points, which may be jittered to reveal overlaps.

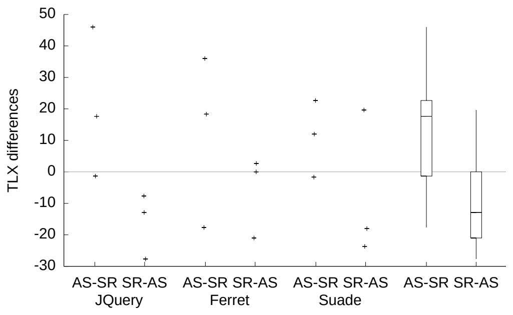

Figure 1: Differences in TLX scores, separated by tool and task order, and by task-order only.

used. Grouping the differences by task order, we see that the differences are reflected around zero, indicating that the AS and SR tasks were perceived as having different degrees of difficulty. A two-way ANOVA comparing the TLX differences by task ordering and exploration tool reports the only statistically significant difference being by task order ($F(1,12) = 7.06, p = 0.02$). From Figure 1, the AS task appears to impose less workload than the SR task, regardless of tool.

As a result of the overwhelming effect of the task on the result, the TLX difference cannot be attributed only to the tool, and we have no evidence to support our hypothesis. The exploration tools may actually alleviate subjective mental workload, but the effect associated with the tasks outweighs the effect associated with the tools.

In our follow-up interviews, we asked whether the programmers found the tools useful: four of six reported Ferret as useful; four of five reported JQuery as useful, and two of five reported Suade as useful. The two non-respondents for JQuery and Suade were ambivalent about the tool used. Two of the Suade dislikers attributed their dislike to difficulty in evaluating the reasons behind Suade's recommendations. Several programmers asked about the availability of the tools.

4.2. Context Explored

Hypothesis 2: Programmers using an exploration tool should not need to explore as much code as compared to Eclipse alone.
We expected that programmers using the tools should be more focused on the program elements relevant to the task.

4.2.1. Number of Unique Program Elements Viewed

Figure 2 compares the within-subject differences in number of unique *visible elements*, separated by task order and tool. Visible program elements are those whose source had been visible on-screen at some point during the programmer's investigation. This value serves as a ceiling on the number of program elements actually viewed and understood by the programmer. Each point corresponds to an individual programmer and corresponds to $v_2 - v_1$ where v_i is the number of program elements viewed in task i, and where the second task was undertaken with the tool. A number greater than zero means that the programmer viewed more items during the second task when the exploration tool was used. These numbers capture neither repeat visits to previously-viewed program elements, nor the amount of time spent examining the program elements.

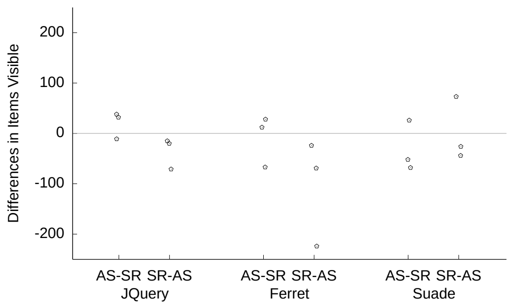

Figure 2: Within-subject differences in number of unique program elements visible to programmers, separated by task and tool (jittered).

From Figure 2, we see that most of the points lie below zero, indicating that programmers seemed to look at more code for their first task when Eclipse was used. A two-way repeated-measures ANOVA comparing the *Viewed* scores by task ordering and exploration tool, reports the only statistically significant effect being an effect by trial ($F(1,12) = 3.56, p = 0.08$). There is no statistical support for a difference between tools. This difference between trials can be interpreted in two ways: (i) the exploration tools had an equal and relatively uniform effect, or (ii) there were carry-over effects between trials.

4.2.2. Detailed Examination of Navigation Paths

We also undertook a more detailed qualitative analysis of the programmers' actual program exploration by comparing the distance they strayed from the solution elements (described in Section 3.8.1). For both AS or SR tasks, the maximum distance assigned to any program element was six steps, and all programmers remained within a distance of three steps (i.e., $d \leq 3$).

Figure 3 shows examples of distance profiles for several users from their two tasks. Space constraints prevent us from including all graphs. A distance profile indicates only that the code displayed by Eclipse had the minimum distance listed; the programmer may have spent some of that time looking at another artefact, such as the task descriptions. These graphs have three additional annotations, an '$\times$' marks where a solution element was first seen, a '$\circ$'

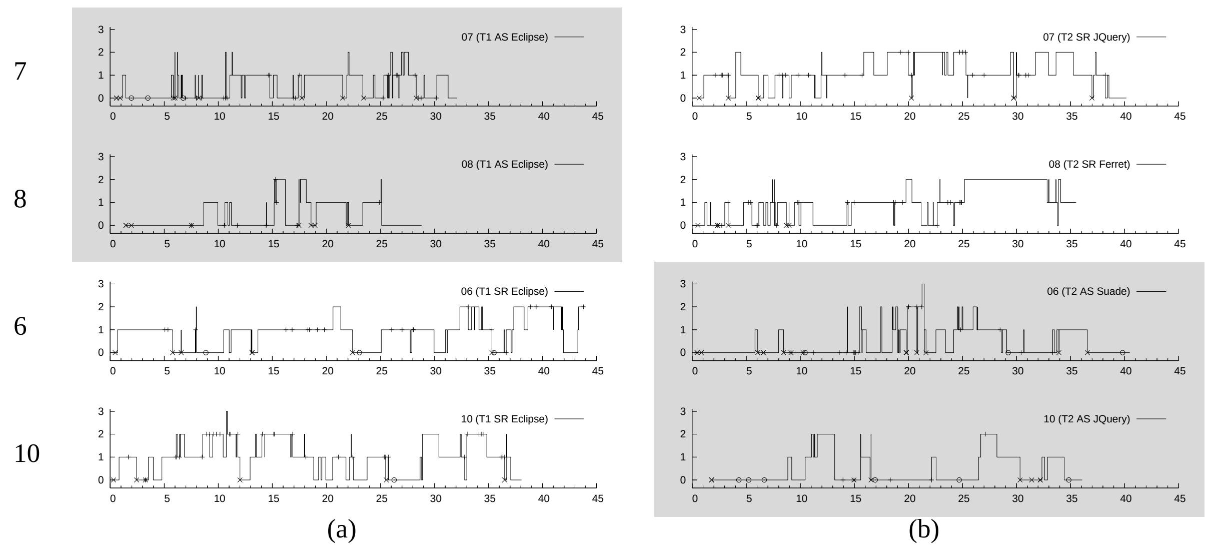

Figure 3: Distance profiles for users 6, 7, 8, and 10: column (a) is from the first task, always undertaken with Eclipse; (b) is from the second task, undertaken with one of the exploration tools. The greyed profiles (7(a), 8(a), 6(b), and 10(b)) are from the AS task; the white profiles (7(b), 8(b), 6(a), and 10(a)) are from the SR task.

where a solution element was first correctly identified, and a '+' when a different file was viewed.

In comparing the distance profiles, we see three slight trends, all involving *spikes*. Spikes are meaningful in that they represent occasions where the programmer has started to investigate beyond elements directly related to the solution (i.e., where $d > 1$). Of interest are the duration and frequency of the spikes: how long was it before the programmer realized that the line of inquiry was not actually related to the task?

The first trend we observe is a difference by task: generally the distance profiles from SR appear to be *spikier* than those from AS, with more frequent movements between the different distance levels. Such spikes may arise from examining an element and quickly realizing its irrelevance to the task. Although the SR task involved fewer actual solution elements, it did require examining a class hierarchy, which may account for some of the rapid descents. Further support for this observation is provided by comparing the average amounts of time spent per task in examining items with $d \leq 1$ versus $d > 1$. Using only the times from the first trial, which was always undertaken with Eclipse, the programmers spent more time exploring items with $d \leq 1$ for AS (92%, s.d. $\pm$9%) as compared to SR (73%, s.d. $\pm$12%).

The distance profiles also reflect individual differences. Even though the programmers generally started from the same locations (the *seeds*), the profiles immediately diverge as programmers chose different areas or items to explore. We also see wide variances in the amounts of time spent at the different distance levels d.

The final trend is that JQuery appeared to have a less-pronounced profiles: spikes appear to be less frequent, but longer-lasting. This may arise as the query tree structure promotes more breadth-first-search examination of query results; with the other tools, a new query generally causes the query context to be lost, perhaps promoting more depth-first-search examination.

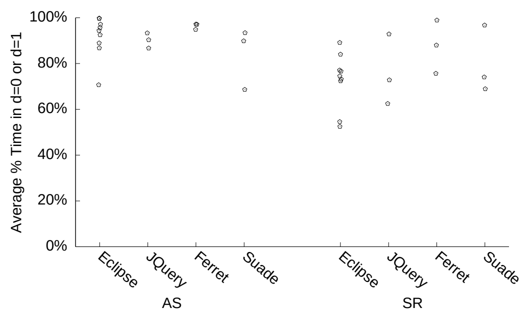

Figure 4: Comparisons of the percent of time spent looking at code at distance $d \leq 1$ for AS and SR.

Figure 4 compares the percentage of the time spent investigating program elements with $d \leq 1$. Programmers using Eclipse seem to spend most of their time viewing such elements for the AS task, but less so for SR. The tools may have helped somewhat in SR, given that the mean of time spent in $d \leq 1$ appears higher for the exploration tools for SR. This effect is slight at best, and a two-way ANOVA reports no statistical support for this speculation.

From examining the '$\times$' in the distance profiles, programmers came across new relevant elements the entire duration. We see no evidence of an early broad strategy, followed by a period of refinement.

Generally we did not see any consistent behaviour across the users. If there had been a strong tool effect than we would have expected to see more consistent patterns.

4.3. Solution Items Identified

Hypothesis 3: Programmers using an exploration tool should identify more of the salient elements for the task as compared to Eclipse alone.
We expected that programmers using the exploration tools should have garnered a better understanding of how the task should be carried out on the code. Their identified solution sets should be missing fewer of the key elements required for a correct solution.

Figure 5 compares the percentage of solution program elements available to the programmers, distinguishing between *selected*, *visible*, and *identified* program elements. *Selected elements* refers to the explicit selection of user interface elements corresponding to program elements: the programmers *asked* to view these elements. *Visible elements* refers to program elements whose definitions were either partially or wholly on-screen. Finally, *identified elements* refers to the program elements correctly identified by programmers as being required for the solution. The numbers shown are expressed as the percentage of the solution program elements identified, as normalized by the number of solution program elements for each task.

We report these three separate measures as no single measure captures the information required. The program elements selected value underestimates the program elements investigated as it does not account for program elements made subsequently visible through scrolling. The visible program elements value overestimates the program elements investigated as it assumes the programmer has examined all of the program elements available on-screen. In examining the program elements visible, we see that none of the programmers using Eclipse and only a very few of those programmers using the exploration tools saw all of the solution program elements required. None of these programmers correctly identified all solution program elements. We use the percentage of solution program elements correctly identified as the quality of the solution.

Figure 6 compares the within-subject differences in the solution quality, as assessed by the percentage of solution program elements correctly identified as compared to our ideal solutions, differentiating by task and tool. Each point corresponds to an individual programmer and corresponds to $q_2 - q_1$ where q_i is the percentage of solution program elements correctly identified from task i, and where the second task was undertaken with the exploration tool. A number greater than zero would imply that the programmer produces a higher quality solution during the second task when an exploration tool was used. The only visible effect seems to be by task ordering and there does not appear to be any

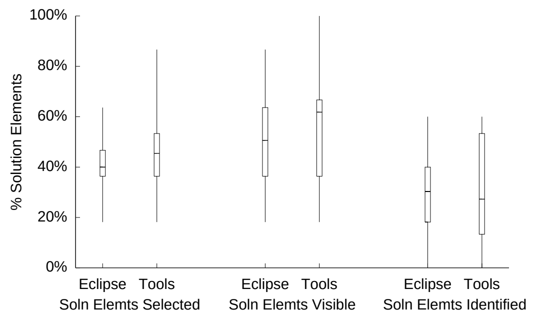

Figure 5: Between-subjects percentage of solution program elements respectively selected, visible, and identified by the programmers.

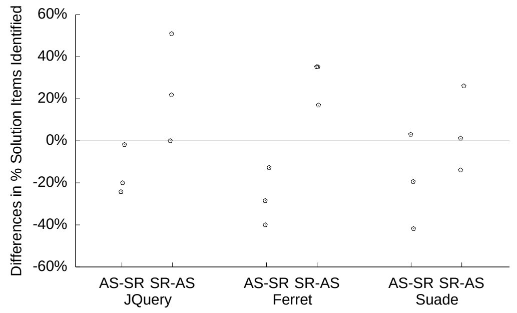

Figure 6: Within-subject differences in the percentage of solution elements correctly identified, separated by tool and task.

effect by exploration tool. A two-way repeated-measures ANOVA comparing the *Quality* scores by task ordering and exploration tool confirms this observation, showing a statistically significant interaction effect between the task ordering and the trial ($F(1,12) = 21.4, p = 0.0006$). This interaction effect confirms the findings of a task ordering effect as found in Section 4.1.

From our follow-up interviews, it seemed that many of the programmers appeared to have made more headway than would appear from their solutions. It is possible that they were willing to speculate in the interview, but would not put their speculations to 'paper', so to speak.

5. Discussion

5.1. The Absence of Practical Effects from Tools

We found no evidence of any practical effect from the exploration software exploration tools. There are several possible explanations.

There were effects from the tools but our measures were inaccurate or do not capture the phenomena of interest. Our analysis showed no effect from the tools used, despite positive comments made by the subjects about the tools tested and requests from those subjects for the tools. It may be that we require more sensitive measures than the simple, unweighted scores used, or require a sample with more than 18 programmers. Our qualitative analysis relied largely on the distance profiles that take an information-centric view of how the task was approached by programmers. As there may be other possible solutions to the tasks, the measures of distance from ideal solutions should be treated with some caution; however virtually all the solution elements would be required in some form by any solution. The deep cognitive nature of programming work may not be reflected in these behavioral measurements, both quantitative and qualitative, that we used.

Our experience does shed doubt on the usefulness of measures proposed and used in other experiments, such as the response times for answers [11]. We believe a more comprehensive set of measures needs to be developed than has been proposed to date. We suggest such a set include the use of the TLX. Although we were unable to establish the necessary baseline conditions to use the TLX to determine the perceived impact of the tools because of the unanticipated differences in perceived difficulty of the tasks, we were able to use it to gauge the programmers' perception of task difficulty.

Any effects from the exploration tools are being confounded by differences between the tasks, or confounded by carry-over (or learning) effects. The two tasks used in our experiment were sufficiently complex that no programmer successfully identified all the relevant elements, and their perceived difficulty confounded the results from TLX (Section 4.1). It is possible that the other two measures of effectiveness are also affected by task difficulty, thus confounding an effect by the tool. Selecting a different initial seed, a different task, a different domain, or even using a different coding style, might produce a very different result.

To assess the likelihood of learning effects, we computed the distance profile for one task using the model solution of the other task. We found no statistical significance in correlation between the percentage of time in $d = 0$ and the solution qualities.[9] This result suggests there was little carry-over effect between the tasks.

The exploration tools have no practical effect for these particular tasks. We see two possible explanations. First, it is possible that the tools are providing useful and relevant information to the programmers, but that the programmers are not taking in the information. Programmers may need further explanations for the information, or may need to discover the information on their own to make use of it.

Second, since none of the programmers correctly identified all relevant elements for either task, it may be that the static program information is insufficient to identify all of the relevant information for a program change task. It may be that Eclipse already does a good job, and that these new tools are simply repackaging readily-available information in new forms—finessing existing solutions, rather than providing new means to solve the problems.

[9]Pearson's $r = 0.36$ at the 10% level ($n = 18$).

5.2. Impacts on Validity

In addition to the possible construct validity threats mentioned above, there are several other factors that pose threats to our experiment.

The first factor arises from our choice to use real tasks as realistic tasks rarely have a single possible approach. This situation arose in our study where two of our subjects proposed very different solutions to the SR task as each supposed additional requirement beyond what was stated.

Another factor comes from the imposition of a time limit. We chose a 40 minute limit as it seemed long enough for the programmers to make at least some headway in understanding the task, and yet minimized total necessary time so as to avoid turning off potential subjects. Six of the 18 subjects finished their tasks before the time allotted.

With the exception of disallowing the use of the debugger, we did not control what tools the programmers could use. The dynamic information from debugging could have been significant and helped diagnose some of the trickier interactions.

Finally, we also noticed one exploration strategy that was used almost exclusively by programmers from a single organization. These programmers used Eclipse's Call Hierarchy view which provides a means of drilling down through method calls and whose results are shown in-place in a tree. This view possibly confers similar benefits to JQuery. Several of these programmers commented that they had been shown this view by another programmer in their organization. This kind of *cultural* communication of helpful strategies may pose additional threats to validity.

5.3. Implications for Future Evaluations

It is possible that the programmers were hampered by unfamiliar UIs. This might be remedied by additional training or possibly by having more adaptive tools, such that the tool will suggest itself for use. Longitudinal evaluation approaches might be more appropriate in such circumstances, providing opportunities for subjects to learn and integrate the tool into their strategies.

Previous studies report finding effects for particular exploration tools when using elementary search tasks (Section 2), whereas our study failed to find any noticeable effects for complex tasks of a longer duration. This suggests that the effects from improvements to elementary search operations are dominated by the cognitive requirements of programming tasks.

Might we have had different results with a different set of tasks? It is possible that the tools studied were unsuitable for the particular tasks used. However, it is not clear why this might be: jEdit is implemented using plain Java, and neither task involved complicated architectures or other paradigms. Future work should address understanding more precisely the characteristics of the tasks we used that made programmers perceive them differently.

6. Summary

We undertook a study to compare the effect of three software exploration tools on programmer efficiency. We had 18 industrial programmers plan two complex change tasks, their first task using Eclipse and their second task using one of the exploration tools. We found that the tools had little apparent effect. Rather, we found that the behaviour of programmers seemed impacted by the tasks chosen, and we found evidence of effects from individual styles and strategies. Although this may seem obvious to those conducting comparative studies of tools, it has not been clearly reported previously. Finally, we discussed experimental issues and lessons we learned from this experiment.

Acknowledgments. We thank our subjects in the study, Peggy Storey for comments on an earlier version of this paper, Frédéric Weigand-Warr for help with Suade, and David Vickers for very helpful conversations. This work was funded by NSERC and IBM.

References

[1] D. Binkley. An empirical study of the effect of semantic differences on program comprehension. In *Proc. Int. Worksh. on Prog. Compr. (IWPC)*, pages 97–106, 2002.

[2] S. G. Hart and L. E. Staveland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Human Mental Workload*, volume 52 of *Advances in Psychology*, pages 139–183. North-Holland, 1988.

[3] T. D. Hendrix, J. H. Cross II, and S. Maghsoodloo. The effectiveness of control structure diagrams in code comprehension activities. *IEEE Trans. Softw. Eng.*, 28(5): 463–477, 2002.

[4] D. C. Howell. *Statistical Methods for Psychology*. Duxbury, Belmont, USA, 4 edition, 1997.

[5] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. Conf. Aspect-Oriented Softw. Dev. (AOSD)*, pages 178–187, 2003.

[6] A. E. Kazdin. *Research Design in Clinical Psychology*. Allyn and Bacon, 3 edition, 1998.

[7] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proc. Int. Conf. Softw. Eng. (ICSE)*, pages 126–135, 2005.

[8] J. Koeskinen. Experimental evaluation of hypertext access structures. *J. Softw. Maint. & Evol.: Research and Practice*, 14(2):83–103, 2002.

[9] S. Meyers and S. P. Reiss. An empirical study of multiple-view software development. In *Proc. ACM SIGSOFT Symp. on Softw. Dev. Envir. (SDE 5)*, pages 47–57, 1992.

[10] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23:76–83, 2006.

[11] V. Rajlich and G. S. Cowan. Towards standard for experiments in program comprehension. In *Proc. Int. Worksh. on Prog. Compr. (IWPC)*, pages 160–161, 1997.

[12] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proc. Joint Europ. Softw. Eng. Conf. and Int. Symp. Foundations of Softw. Eng. (ESEC/FSE)*, pages 11–20, 2005.

[13] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proc. Int. Conf. Softw. Eng. (ICSE)*, pages 406–416, 2002.

[14] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.

[15] M. B. Rosson. Human factors in programming and software development. *ACM Comput. Surv.*, 28(1):193–195, 1996.

[16] S. E. Sim and M.-A. D. Storey. A structured demonstration of program comprehension tools. In *Proc. Working Conf. Rev. Eng. (WCRE)*, pages 183–193, 2000.

[17] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.

[18] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. Müller. On integrating visualization techniques for effective software exploration. In *IEEE Symp. on Inf. Vis. (INFOVIZ)*, pages 38–45, Phoeniz, Arizona, 1997.

[19] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Programming*, 36(2–3): 183–207, 2000.

[20] S. Sulaiman, N. B. Idris, S. Sahibuddin, and S. Sulaiman. Re-documenting, visualizing and understanding software system using DocLike Viewer. In *Proc. Asia-Pacific Softw. Eng. Conf. (APSEC)*, pages 153–163, 2003.