Supporting Software History Exploration

by

Alexander Wilfred John Bradley

B.Sc. (Hons.), University of British Columbia, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

April 2011

Abstract

Software developers often confront questions such as "Why was the code implemented this way"? To answer such questions, developers make use of information in a software system's bug and source repositories. In this thesis, we consider two user interfaces for helping a developer to explore information from such repositories. One user interface, from Holmes and Begel's Deep Intellisense tool, exposes historical information across several integrated views, favouring exploration from a single code element to all of that element's historical information. The second user interface, in a tool called Rationalizer that we introduce in this thesis, integrates historical information into the source code editor, favouring exploration from a particular code line to its immediate history. We introduce a model to express how software repository information is connected and use this model to compare the two interfaces. Through a laboratory study, we found that our model can help to predict which interface is helpful for two particular kinds of historical questions. We also found deficiencies in the interfaces that hindered users in the exploration of historical information. These results can help inform tool developers who are presenting historical information from software repositories, whether that information is retrieved directly from the repository or derived through software history mining.

Preface

An earlier version of the work in this thesis was accepted for publication as a research paper:

> [3] A. W. J. Bradley and G. C. Murphy. Supporting software history exploration. In *MSR 2011: Proceedings of the 8th Working Conference on Mining Software Repositories.* ACM, May 2011.

The abovementioned paper was written in collaboration with the supervisor of this thesis, Dr. Gail Murphy. Under Dr. Murphy's guidance, the author performed all of the research reported in this thesis. Dr. Murphy contributed to the writing of the paper, and small amounts of material originally written by Dr. Murphy are incorporated into Chapters 1, 5, 6 and 7. The writing in this thesis is primarily the work of its author.

Human subjects research conducted for this thesis was approved by the UBC Behavioural Research Ethics Board (certificate number H10-02033).

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACM Association for Computing Machinery

CE code element (method, field, or class)

CVS Concurrent Versions System, a revision control system (see http://savannah.nongnu.org/projects/cvs)

ER Entity-relationship diagram (a data modelling technique initially introduced by Chen [7])

FA filter application

GEF Graphical Editing Framework (libraries for building rich graphical editors in the Eclipse IDE; see http://www.eclipse.org/gef/)

GOMS Goals, Operators, Methods, and Selection rules (a framework for usability analysis introduced by Card et al. [4])

IDE integrated development environment

ONV opening(s) of new view(s)

RGB Red–Green–Blue colour model

RHI reference(s) to historical information

SCE selection(s) of code element(s)

VS visual scan through highlighted code in editor, possibly involving scrolling

Acknowledgments

First of all, I thank my supervisor, Gail Murphy, for steering me through this thesis project. Gail let me explore interesting side-paths, but always made sure that I stayed on track. I'm grateful for our many discussions of this work, which frequently helped me to sharpen my thinking about the subject matter and taught me a lot about software engineering research. Thanks also go to my second reader, Joanna McGrenere, who returned feedback on my thesis draft quickly and provided useful comments.

I thank Thomas Fritz for providing comments on an early version of the paper that became this thesis. I am also thankful to Andrew Begel, Arie van Deursen, and attendees of the June 2010 Vancouver Eclipse Demo Camp for their feedback on early versions of the tools I implemented for this thesis project, and to all of our study participants for their time and feedback. Further thanks go to the anonymous MSR 2011 reviewers, whose comments helped to improve this work. This research was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

I am grateful for the support and companionship of many lab-mates past and present—Thomas, Emerson, Sarah, Nick, Roberto, Robin, Sam, Brett, James, Nima, Peng, Immad, Antonio, Ryan, Seonah, Peter, Apple, Rick, and Lloyd. Many of them have provided useful comments and suggestions that improved this work, and all of them have helped to make the Software Practices Lab a friendly and intellectually stimulating environment for research. I am also grateful to Steve from Imager Lab, who helped me to get access to the user study rooms when they were heavily booked, and was also a great TA back when I was an undergraduate trying my first HCI course.

I first conceived the idea that eventually became this thesis project in Tamara Munzner's Information Visualization course. I am grateful to Tamara for a stimulating course that introduced me to many interesting and useful visualization techniques and principles.

A special word of acknowledgment should go to my undergraduate thesis supervisor and former advisor, Kris De Volder. His encouragement and guidance helped to set me on the path to graduate school, and his support during my first year was much appreciated.

Thanks are also due to all the hard-working support staff in the CS Department who keep our research running smoothly. In particular, I would like to thank group assistants Hermie Lam and Holly Kwan and graduate program administrator Joyce Poon, who have always been ready to help with administrative details large and small, and ICICS technician Gable Yeung, who swiftly processed my many booking requests for user study rooms.

I am thankful for the support of my friends Alicja, Ciarán Llachlan, Daniel, and Sandra, who helped me make it through the more difficult parts of the last three years. I am also thankful for the support and understanding of my colleagues from my high school IT work: Frances, Bill, Bruno, and Lloyd.

Finally, I am profoundly grateful for the unfailing love and support of my family: Grandma Parkes, Grandma Bradley, Uncle Bill, Uncle Ian and Aunt Chris, cousins Alison, Bill and Cassie, and—most of all—my mother and father.

Chapter 1

Introduction

Developers working on large software development teams often confront the question, "Why was the code implemented this way?" In a study by Ko and colleagues, this question was found to be amongst the most time-consuming and difficult to satisfy of a number of information needs faced by developers [20]. In another study, LaToza and colleagues confirmed that developers faced challenges in trying to understand the rationale behind code [23]. A third study conducted by LaToza and Myers further confirmed these findings, reporting that questions about code history—when, how, by whom and why was code changed or inserted—were some of the most frequent questions developers needed to answer [21].

Various existing tools have attempted to assist developers in exploring a software system's history to answer such questions. Tools such as CVS' "annotate" (or "blame") feature [6], and graphical interfaces to it such as Mozilla Bonsai [25] and Eclipse's "show annotations" feature [11], display the last user who changed each line of code in a source file. Tools such as Hipikat [8] and Deep Intellisense [15], which is built on Bridge [27], provide artifacts (e.g., bugs, e-mails, or documents) that appear relevant to a piece of code.

These existing tools use one of two basic styles to support a user in exploring the historical information needed to answer the questions that frequently confront them. One approach is to integrate the historical information into or near the editor, as is the case with Eclipse's "show annotations" feature. This approach tends to favour displaying a breadth of historical information for large regions of the code

at once. The other approach is to use a set of integrated views that provide related historical information for a single element out of the context of the editor, as is the case with Deep Intellisense. This approach tends to favour displaying a depth of historical information about a single part of the code.

In this thesis, we investigate how these different presentation styles affect how developers explore historical information to answer questions related to why code was implemented a certain way. When exploring software history to answer a question of interest, a developer must typically traverse linkages between various pieces of information, such as code lines, code elements, revisions, and bugs. We created a model of common types of history information and the linkages between them, shown as an entity-relationship (ER) diagram in Figure 1.1. We have chosen bugs as the principal form of related historical information in this model; a more detailed model might include other kinds of related information such as e-mail, webpages, documents, or Mylyn task contexts [18]. In Chapter 3, we identify basic user interface operations by which the user can navigate between the entities in this model in each interface style and, by identifying the simplest sequences of operations necessary to answer certain types of questions, predict that one interface style sometimes has a significant advantage over the other. For example, the model can help to predict that one interface style is likely to be easier to use for finding out which bugs have affected a given method throughout its history (see Hypothesis 3.1), while the other is likely to be easier to use for answering a question about which sections of a source file were affected by a given bug fix (see Hypothesis 3.2).

To investigate the usefulness of the model and to obtain user feedback on the design of the tools, we conducted a laboratory study in which thirteen participants used first one of the tools, then the other to answer software history exploration questions. The first tool (described further in Section 2.1) was a replica of the Deep Intellisense user interface style. The second (presented in Section 2.2) is a new interface we created, called Rationalizer, that integrates historical information directly into the background of the source code editor using semi-transparent columns to directly answer questions about when code was changed, who last changed it and why it was changed. The results of our study (reported in Chapter 4) confirmed the predictions of our model and identified deficiencies in each of the

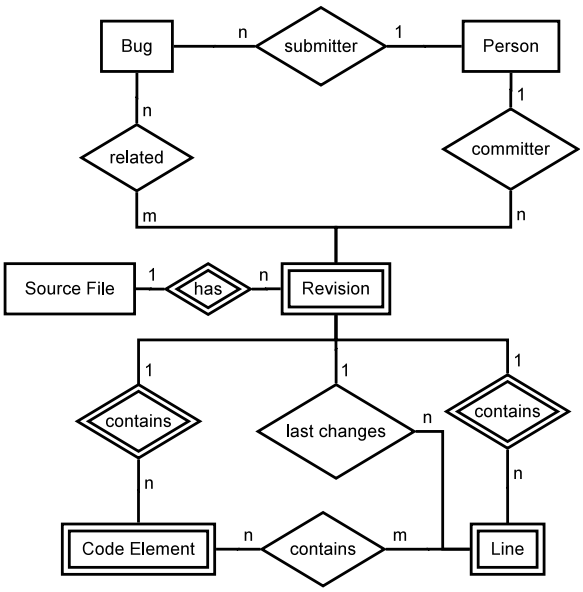


Figure 1.1: ER diagram of software history artifacts

tools—for instance, many users found scrolling through the large amounts of information provided by Rationalizer confusing and error-prone and requested better filtering, and several users found little use for Deep Intellisense's summary views. Tool developers can use the model we have introduced and our findings about user interfaces conforming to this model as a guide in reasoning about appropriate interface designs for tools that expose software history information.

This thesis makes three contributions:

- it introduces a new user interface for software history exploration, namely Rationalizer, that integrates more historical information into the editor than previous designs,
- it introduces a model of software repository information and shows how that model can be used to reason about and predict the performance of a user interface that supports exploration of software history information, and
- it provides a comparison of two different user interface styles for exploring software history information.

The remainder of this thesis is structured as follows. Chapter 2 describes the user interfaces of the Deep Intellisense and Rationalizer tools in detail. Chapter 3 presents our model of software history information and tool user interface operations and uses it to analyze the difficulty of answering different types of questions using the tools; we identify two situations (Hypotheses 3.1 and 3.2) where we predict a particular question will be significantly easier to answer with one tool than the other. Chapter 4 describes the study we performed to validate Hypotheses 3.1 and 3.2 and to solicit user feedback on the design of the tools. Chapter 5 discusses our results and suggests ways they can be used to improve the design of history exploration user interfaces; the chapter also recommends ways to improve our model and study design. Chapter 6 places our contributions in the context of related work, and Chapter 7 concludes the thesis.

Chapter 2

Two Tools for Exploring Software History

To support our investigations, we built two prototype tools that embody the two different user interface styles for exploring software history. The first prototype is a replica of Deep Intellisense [15]. The second prototype is a new interface we designed to expand the historical information presented in a code editor; we call this user interface approach and the prototype, Rationalizer.

Each of these prototypes accesses the same historical information. Each retrieves revision history information for the source code of interest from a CVS repository. Each scans check-in notes to detect possible bug IDs using simple regular expressions. Each uses bug IDs as search keys to retrieve bug metadata from a Bugzilla repository using an Eclipse Mylyn connector.[1] Each prototype caches revision and bug data to improve performance. Each tool could be generalized to gather historical information from other kinds of source and bug repositories.

We describe the user interfaces of each prototype in turn.

2.1 Deep Intellisense

Deep Intellisense was created as a plugin for the Visual Studio IDE [15]. As the original plugin is not available for use, we created a replica of its user interface for

[1] http://wiki.eclipse.org/Mylyn_Bugzilla_Connector, verified March 7, 2011

our research. This replica is implemented as a plugin for the Eclipse IDE, providing access to historical information about Java code. Like the original plugin, our replica provides three views that update whenever the user selects a code element (defined as a method, field, or class declaration) in an editor (Figure 2.1):

- The *current item* view, which gives an overview of the number of check-ins related to the element, the number of bugs related to the element, and the number of people responsible for those check-ins and bugs. For example, in Figure 2.1, the doOperation method of the ProjectionViewer class has been selected in the editor and the current item view (top right) displays a summary of its history.

- The *related people* view, which provides a list of the people (represented by usernames) relevant to the element, as well as the number of related open and closed bugs each person has submitted and the number of related check-ins they have committed. For example, in Figure 2.1, the related people view (middle right) shows the people who have changed the doOperation method or filed bugs affecting it. (Unlike the original, we did not provide further information such as job titles or email addresses for the people in this view.)

- The *event history* view, which provides an interleaved list of events (i.e., revisions and bugs) relevant to the element. The list is initially sorted chronologically, but other sort criteria can be chosen. A text search can be used to filter the items displayed. Double-clicking on a revision opens an Eclipse comparison viewer showing the differences between the selected revision and the revision preceding it; double-clicking on a bug opens its full Bugzilla report in a web browser. For example, in Figure 2.1, the event history view (bottom right) shows the revisions and bugs affecting the doOperation method.

Unlike the original plugin, our replica is not backed by the Bridge database [27] and does not provide other kinds of historical information such as e-mails, web pages or documents. This support was not needed for the investigations we conducted. It also omits the "thumbs up"/"thumbs down" buttons provided by the original for rating the usefulness of person and event results.

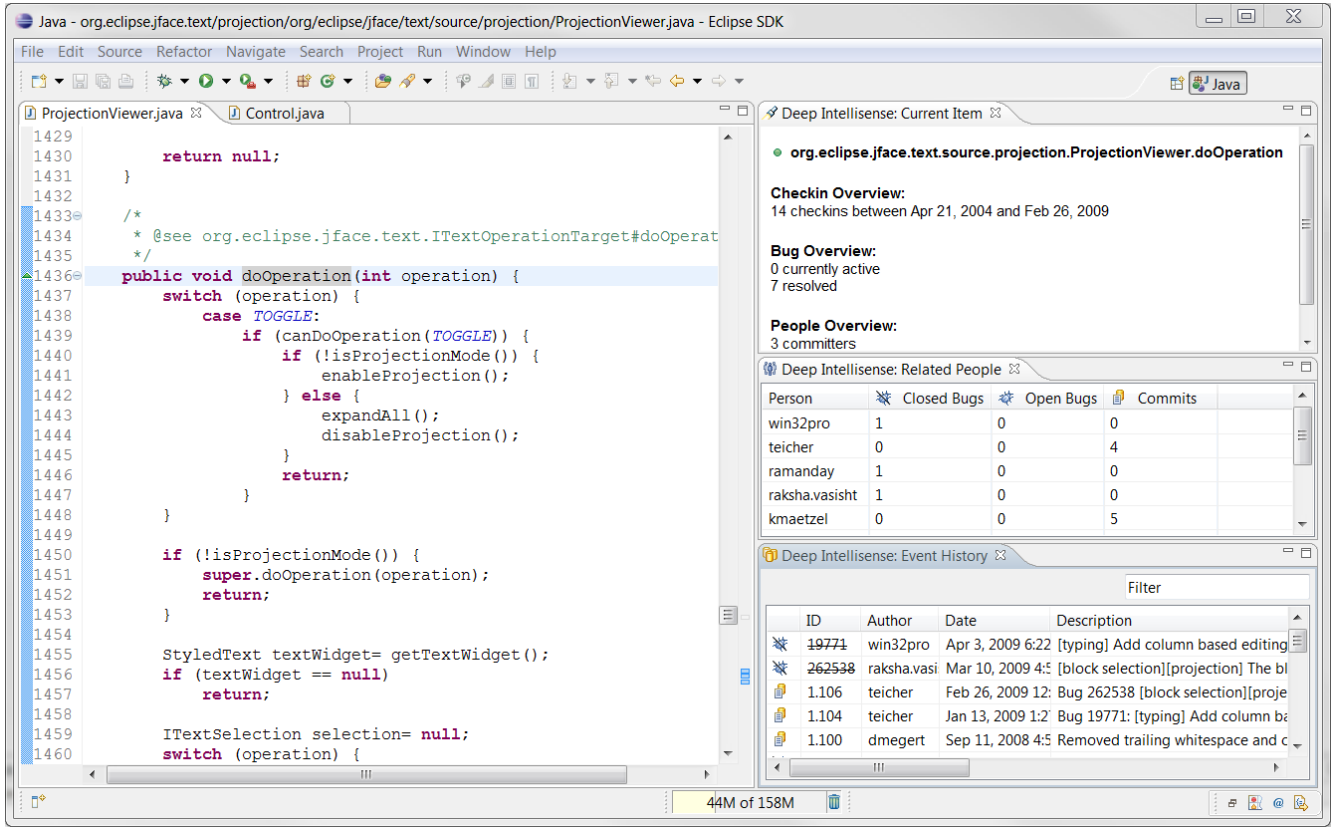


Figure 2.1: Deep Intellisense replica. Selecting a code element updates the three Deep Intellisense views (right) with information about related people and events.

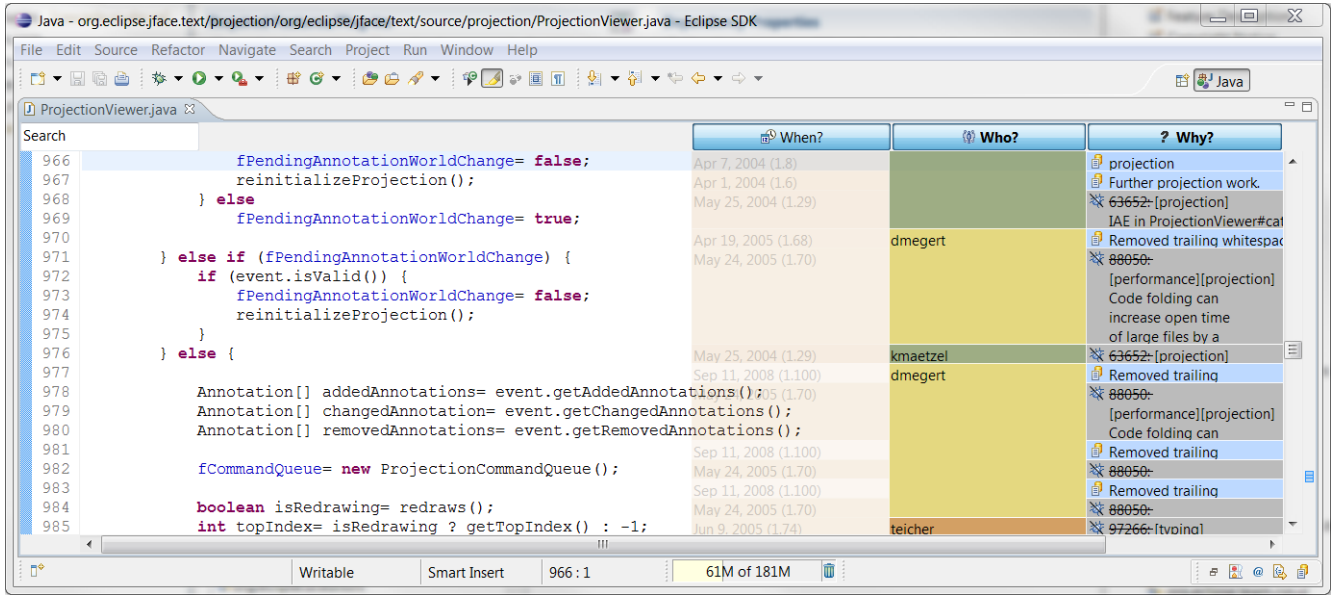


Figure 2.2: Rationalizer user interface. Columns can be activated by the user with low or high transparency. A text search (top left) allows filtering of information displayed in the columns.

2.2 Rationalizer

The Rationalizer interface we designed integrates historical information into the background of a source code editor through three semi-transparent columns entitled "When?", "Who?", and "Why?". Clicking a column heading reveals the column at a high level of transparency[2] (i.e., in the background "underneath" the editor text.) Clicking the heading a second time reduces the transparency of the column[3] (i.e., raises it to the foreground "above" the editor text.) Clicking the heading a third time hides the column again. Figure 2.2 provides an example of the two different transparency levels: the "When?" column (left) is at high transparency, while the "Who?" and "Why?" columns (middle and right) are at low transparency.

For each line of code, the "When?" column gives the date on which the line was last modified; the "Who?" column gives the username of the developer who made the modification; and the "Why?" column provides the check-in note of the last source code revision that affected the line (if there is a related bug for the revision, it is displayed instead of the revision.) Within a column, if the same information applies to multiple consecutive lines, a single block is rendered spanning all of those lines; for example, in Figure 2.2, user dmegert was the last person to modify lines 970–975, so a single block is rendered in the "Who?" column for those lines. The background of the "When?" column is coloured to indicate the age of the code[4] (darker entries are older). In the "Who?" column, each developer is assigned a colour[5] and these colours are used as the background colours for the column

[2] Alpha blend ($\alpha \approx 0.16$) with the editor contents as background.

[3] Alpha blend ($\alpha \approx 0.78$) with the editor contents as background.

[4] A revision with UNIX timestamp t is assigned an alpha blend of background colour $(199, 134, 57)$ and foreground colour $(241, 225, 206)$ with mixing factor

$$\alpha = \begin{cases} (t - t_o)/(t_n - t_o), & t_n \neq t_o; \\ 1, & t_n = t_o, \end{cases}$$

where t_o and t_n are the UNIX timestamps of the oldest and newest revisions, respectively. Colours are given as RGB tuples. This colour palette is based on one found in the source code of the Eclipse user interface for CVS "blame" (class org.eclipse.jface.internal.text.revisions.RevisionPainter); however, Eclipse uses a different colour mixing algorithm and assigns darker colours to *newer* revisions.

[5] Colours are assigned from a 28-colour palette taken from the source code of the Eclipse user interface for CVS "blame" (class org.eclipse.team.internal.ccvs.ui.operations.CommitterColors). This source code is distributed under the Eclipse Public License [10], which permits the creation of deriva-

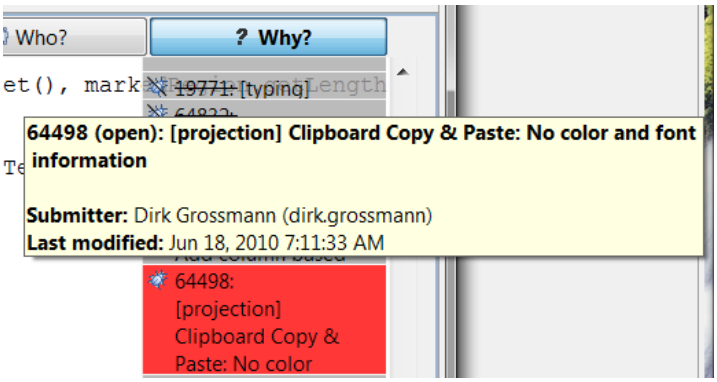


Figure 2.3: Hovering over a bug in Rationalizer shows the title, submitter, and last modification date for the bug.

entries. In the "Why?" column, revision entries are coloured light blue, closed bugs are coloured grey and open bugs are coloured red. A text search is available at the upper left-hand corner of the editor to filter the items displayed in the columns.

Hovering over column items provides more detailed information if available. For instance, hovering over a bug in the "Why?" column shows the title, submitter and last modification date for the bug (Figure 2.3). Some column items also have hyperlinks, which can be activated by Ctrl-clicking on the items. Ctrl-clicking a bug item opens its full Bugzilla report in a web browser. Ctrl-clicking on a revision opens an Eclipse comparison viewer showing the differences between the selected revision and the revision preceding it. The comparison viewer thus opened also includes the three Rationalizer columns, allowing the user to explore further into the past.

2.3 Contrasting the Tools

We will note three major areas in which the user interfaces of the tools just presented are different from each other. First, Rationalizer presents software history information line-by-line, while in Deep Intellisense, history exploration starts from code elements and cannot be activated directly from individual lines. Second, Rationalizer integrates history information directly into the editor, while Deep Intel-

tive works.

lisense displays it in views separate from the editor. Third, Rationalizer facilitates exploring software history information in a breadth-first manner (allowing the user to access recent information on every line at once) while Deep Intellisense is oriented towards depth-first exploration (allowing the user to see deep into the history of code elements, but one element at a time). The last of these differences may have a serious impact on user performance for certain types of history exploration tasks, as will be shown in more detail through the model we introduce in Chapter 3 and our user evaluation of the tools in Chapter 4. Our evaluation will also report user feedback on the other contrasting properties described above.

Chapter 3

Modeling Software History Exploration

User interfaces of tools to help explore software history differ in the operations that they provide to view entities and follow relationships between entities. We have identified basic operations in each prototype by which these linkages are traversed. For example, in Deep Intellisense, the operation of *selecting a code element* (SCE) navigates from the element to a list of related items (revision or bug references) from its history. In both prototypes, the operation of *opening a new view* (ONV) is necessary to navigate from a revision reference to a display of the changes it introduced, or to navigate from a bug reference to the full bug report. In Rationalizer, the user must *visually scan* (VS) through some number of highlighted lines of interest to find the information of interest associated with them. Finally, in both prototypes, a *filter application* (FA)—i.e., use of the text search—can limit the historical information displayed to show only items of interest.

3.1 Predictions

With the model in Figure 1.1 and the basic operations above established, we can predict the simplest strategies a user can employ to perform traversals across the various linkages in the model. We will analyze three principal cases (Sections 3.1.1–3.1.3): navigating from code to its *immediate* history (i.e., the last revision or

bug that affected it); navigating from code to its *deep* history (i.e., all revisions and bugs that traceably affected its development); and navigating from historical items (persons or bugs) back to their effects on the current state of the code. For each of these cases, we consider starting the navigation both from code lines (as facilitated by Rationalizer) and from code elements (as facilitated by Deep Intellisense).

In our analysis, we distinguish between *references* to historical information (RHI)—e.g., blocks in Rationalizer columns or table rows in the Deep Intellisense "Event History" view—and the full descriptions of the historical information—e.g., full Bugzilla bug reports or comparisons showing the changes introduced by a revision. For completeness, the procedure for navigating from RHI to full descriptions is analyzed in Section 3.1.4.

Our predictions, explained in detail below, are summarized in Table 3.1. We identify two areas in which our model predicts that one tool will have a significant advantage over the other (Sections 3.1.2 and 3.1.3); these two predictions are evaluated through a user study that is described in Chapter 4.

3.1.1 Finding Immediate History for Code

With Rationalizer, navigating from source lines or code elements to RHI for the bugs and revisions that *last* affected them is relatively straightforward, since these references are immediately available though Rationalizer's columns. Finding the last RHI for ℓ source lines takes an $O(\ell)$ visual scan of the column blocks next to the lines. Finding the last RHI for e code elements takes an $O(e)$ visual scan of the column blocks next to the elements (the efficiency of which can be increased by using code folding [9] to collapse the elements.)

With Deep Intellisense, navigating from code elements to the last RHI affecting them may be slightly more difficult, requiring the user to select each element in turn. Formally, finding RHI for e code elements requires $O(e)$ SCE. Finding the last RHI for specific lines is somewhat more tricky. If we consider ℓ lines which are contained in e code elements (and assume the elements have a maximum of r related revisions affecting them), the user would need to select each element in turn, then open a comparison view for every revision affecting each element to see if it affected the line. This amounts to $O(e)$ SCE $\times$ $O(r)$ ONV.

3.1.2 Finding Deep History for Code

LaToza and Myers [21] noted that developers sometimes wanted to know the entire history of a piece of code, not just its most recent change. In Rationalizer, navigating from a source line to the last r revisions that affected it requires Ctrl-clicking revision hyperlinks $O(r)$ times to step back through past revisions one at a time. Navigating from a code element to its last r revisions requires a similar strategy. In either case, $O(r)$ ONV are required.

In Deep Intellisense, navigating from a single code element to the last r revisions that affected it takes only one code element selection. Navigating from a source line to the last r revisions that affected it is more complicated, as the user would have to open a comparison view for each of the past revisions from the "Event History" view for the enclosing code element and check which revisions actually affected the line. Formally, $O(r)$ ONV would be required.

Based on this analysis, we make our first major prediction:

Hypothesis 3.1. *For code elements that have been affected by many past revisions, Deep Intellisense will have a significant advantage over Rationalizer for finding those elements' entire history.*

3.1.3 Finding Related Code for History Item (Person or Bug)

Sometimes a developer needs to know which code a co-worker has touched in an object-oriented class. One way to address that question is to find areas of source affected by a particular co-worker of interest. (More generally, one could look for areas of source affected by various persons or bugs of interest.)

In Rationalizer, the most efficient way to find areas of code last touched by a given user (or affected by a given bug) is to filter the "Who?" column by that user's name (or filter by a bug ID) and then perform a visual scan of the blocks that remain visible in the columns. Formally, if the user or bug affected ℓ_a lines (or code elements containing ℓ_a lines), 1 FA and $O(\ell_a)$ VS are required.

In Deep Intellisense, the most efficient way to find which code elements in a class were last touched by a given user or affected by a given bug would be to select every code element in the class in turn, and for each code element, apply a filter to the "Event History" view to check if the user or bug affected the element.

Formally, if there are E code elements in the class, $O(E)$ SCE and $O(E)$ FA would be required. Finding which specific lines were last touched by a given user or affected by a given bug would require more work, as it would require examining every revision since the user or bug first affected the class to find all the altered lines and check whether they had been overwritten by other revisions since they were committed. Formally, if the file has had R revisions since the user or bug first affected it, $O(R)$ ONV would be required.

Based on this analysis, we make our second major prediction:

Hypothesis 3.2. *Rationalizer has a significant advantage over Deep Intellisense for finding sections of code affected by persons or bugs if a large number of code elements or lines are affected and/or a large number of past revisions directly affect regions of the current source code.*

3.1.4 Following References

Once the user has found a RHI of interest, following that reference for more information is equally easy in Deep Intellisense and Rationalizer. In Rationalizer, references to historical items are realized as blocks in the "When?" or "Why?" columns; Ctrl-clicking on the block for a bug reference opens a browser with the full bug report, while Ctrl-clicking on the block for a revision reference opens a comparison window showing the changes introduced by that revision. In Deep Intellisense, references to past revisions or bugs are realized as table rows in the "Event History" view; double-clicking on a bug or revision row has the same effect as Ctrl-clicking a bug or revision block in Rationalizer. In either prototype, following a historical information reference takes 1 ONV.

3.2 Limitations of Analysis

Our analysis has a number of limitations. First, it models the ideal path to the right answer taken by a user who is expert at using each tool and takes no wrong steps. It does not take into account the difficulty of *finding* the correct operation to perform at any stage, or backtracking if an incorrect path is taken. Second, the respective difficulty of model operations has simply been estimated based on the elementary actions (mouse clicks/drags or key presses) of which the operations are

composed; no precise weights have been assigned to each action and the difficulty of individual operations has not been measured empirically.

Table 3.1: Different types of navigation tasks and their predicted complexity under Rationalizer and Deep Intellisense

History exploration task	Rationalizer	Deep Intellisense
Finding immediate history for code (cf. Section 3.1.1)		
ℓ source lines (contained in e CEs with r related revisions) → last related revisions and associated RHI	$O(\ell)$ **VS**	$O(e)$ SCE × $O(r)$ ONV†
e CEs → last related revisions and associated RHI	$O(e)$ **VS**	$O(e)$ SCE
Finding deep history for code (cf. Section 3.1.2)		
Source line → r last related revisions and associated RHI	$O(r)$ ONV	$O(r)$ ONV†
CE → r last related revisions and associated RHI	$O(r)$ ONV	**1 SCE**
Finding related code for history item (person or bug) (cf. Section 3.1.3)		
Reference to author or bug → all ℓ_a source lines in file affected by same reference (where file has had R revisions since first affected by given author or bug)	**1 FA + $O(\ell_a)$ VS**	$O(R)$ ONV
Reference to author, bug → all CEs (of E total) affected by same reference in file (where affected CEs contain ℓ_a source lines)	**1 FA + $O(\ell_a)$ VS**	$O(E)$ SCE + $O(E)$ FA
Following references (cf. Section 3.1.4)		
RHI → previous source revision	1 ONV	1 ONV
RHI → full bug report	1 ONV	1 ONV

Acronyms: CE: code element (method, field, or class); FA: filter application; ONV: opening(s) of new view(s); RHI: reference(s) to historical information; SCE: selection(s) of code element(s); VS: visual scan through highlighted code in editor, possibly involving scrolling.
Bold font is used to indicate situations where we predict that one tool will have a significant advantage over the other.
† It would be necessary to check every related revision for the enclosing CE(s) to see if it modified the line(s).

Chapter 4

Evaluation

We conducted a laboratory study to investigate the two major predictions of our model (Hypotheses 3.1 and 3.2). We also wanted to elicit feedback on the other contrasting design characteristics of the tools mentioned in Section 2.3: Do developers prefer software history information to be tightly integrated into the source code text editor (as in Rationalizer) or presented in views that are visually separate from the text editor but linked to it (as in Deep Intellisense)? Do developers prefer software history information to be associated with individual lines of code (as in Rationalizer) or with higher-level code elements, such as methods and classes (as in Deep Intellisense)?

4.1 Methodology

4.1.1 Question Sets

We prepared two question sets (designated A and B), each of which contained four questions about software history. These questions can be found in Appendix A.4.3. The question sets were based on code drawn from the open source Eclipse Graphical Editing Framework (GEF) codebase[1] and bugs drawn from the associated Eclipse Bugzilla database[2]. This codebase was chosen because it was compatible with our tool and we believed it would be relatively easy to understand, even if the

[1] http://www.eclipse.org/gef/, verified March 25, 2011
[2] http://bugs.eclipse.org/bugs/, verified March 25, 2011

participant did not have an Eclipse development background. To test the predictions of our model, we ensured that each question set contained at least one "deep history" question (which Hypothesis 3.1 predicts to be harder in Rationalizer) and at least one "history to related source" question (which Hypothesis 3.2 predicts to be harder in Deep Intellisense.) These question sets were not isomorphic; this issue is discussed further in Section 5.3.

4.1.2 Design

Our study used a within-subjects design; participants were asked to complete one question set with each tool. Participants were randomly assigned to one of four groups, spanning the four possible orderings of tools and question sets.

4.1.3 Procedure

The experimenter began each session by welcoming the participant and reading the initial instructions found in Appendix A.5.1. Before working with a tool, participants were given a paper tutorial (cf. Appendices A.4.1 and A.4.2) describing the features of the tool they were about to use. Participants then worked on the question set they were assigned for that tool; they were allowed a maximum of seven minutes per question. Participants were asked to think aloud while working on the questions, and were allowed to refer back to the tutorial while working on a question. After each question, participants were asked to rate the tools on a 1–5 scale, where 1 meant "not helpful at all" and 5 meant "did everything I wanted" (cf. Appendix A.4.4).

After participants had finished both question sets, 15 minutes were allocated for a follow-up discussion, in which the experimenter asked about participant preferences and perceptions concerning the tools and the kinds of software history information desired by the participants. The specific questions used to guide the follow-up discussion can be found in Appendix A.5.4.

4.1.4 Quantitative and Qualitative Measures

We predicted the simplest possible strategy for answering each question; our predicted strategies are given in Appendix A.5.2. Table 4.1 summarizes these strate-

Figure 4.1: Photograph of experiment setup. Participants used the computer on the left; the experimenter used the computer on the right.

gies by showing the predicted minimum operation counts for each question; visual scans are not included in the predictions, since they cannot be measured accurately in our experimental setup. We recorded the time taken for each question, and the experimenter noted the participant's major actions, with particular attention to the operations described in Chapter 3. Furthermore, the prototypes were instrumented to log view opens, filter applications, and code element selections. Participant answers to the questions were later scored for correctness; the scoring guidelines can be found in Appendix A.5.3. We expected that for the hypothesis-testing questions highlighted in Table 4.1, users might take more time, produce less correct answers and report lower levels of satisfaction with the tool that we predicted would be at a disadvantage.

The follow-up interview (cf. Appendix A.5.4) contained questions designed to assess participant preferences and perceptions concerning specific design characteristics of the tools and to gather examples of software history questions from

participants' real-world experience. Questions 1 and 2 asked participants for their preferences as to whether code history information should be presented integrated into the editor or in separate views and whether code history searches should be line-based or element-based. Question 3 aimed to determine which tool users would prefer in four specific situations. Questions 4 and 7 aimed to elicit user feedback on strengths and weaknesses of the tools, and question 6 asked whether users had a preference for one tool over the other. Finally, question 5 was designed to elicit examples of real questions participants encountered in their development work.

4.1.5 Participants

We recruited 13 participants from the membership of the Vancouver Java User Group and from the UBC undergraduate and graduate student populations, focusing on students studying Computer Science or Electrical and Computer Engineering. Participants were required to have experience programming in Java using Eclipse (e.g., at least one university course project, or professional experience), and to have experience using version control and issue tracking systems for software development. Participants were compensated with a $20 gift card for a 90-minute session.

Before scheduling a session, participants were asked to fill out an initial online questionnaire (Appendix A.3) that requested demographic data and information about their background with IDEs, version control systems, and issue tracking systems. Of our participants, twelve were male and one was female; six were aged 19–29, three were aged 30–39 and four were aged 40–49. Four of the participants were undergraduate students, seven were graduate students, four were commercial software developers, three were open source developers, one was a software tester, one was a test and infrastructure manager, and one was a web developer. The "student" and "developer" categories were not mutually exclusive. Of the students, nine were in the Computer Science program, one was in the Information Science program, and one was unclassified. Participants' reported experience with version control and bug tracking systems is shown in Figure 4.2.

During the follow-up discussion, most participants stated that they frequently investigated historical information about code in their own work or had done so

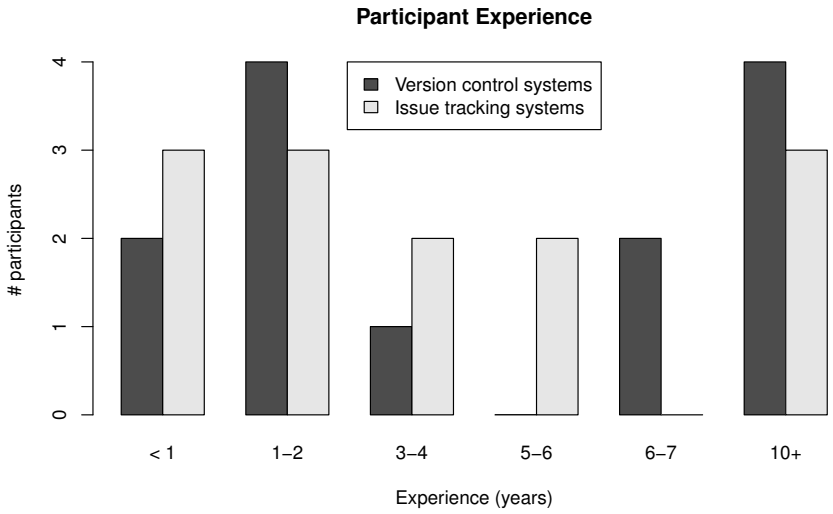


Figure 4.2: Participant experience with version control and issue tracking systems ($n = 13$)

in past work. They provided examples of history exploration questions from their own experience, including questions about the rationale and evolution of code, the activity of team members, finding code relevant to bugs, and finding other code changes related to a particular change. Appendix B.1 provides further details on participants' history exploration experience.

Participants were assigned a participant number between 4 and 16. Participant numbers 1–3 were used during pilot tests.

4.1.6 Apparatus

Figure 4.1 shows the experimental setup. The participant laptop was a ThinkPad T43 (Pentium M 1.86GHz processor, 1.5GB RAM) running Windows XP Professional SP3. On this laptop, the prototype plugins were run using Eclipse SDK 3.6.1 (build M20100909-0800), and Firefox 3.6.13 was used for accessing full Bugzilla reports. The Eclipse workspace contained projects org.eclipse.draw2d, org.eclipse.gef, org.eclipse.zest.core, and org.eclipse.zest.layouts, all checked out from the dev.eclipse.org CVS repository with a date tag of midnight on December

Table 4.1: Predicted numbers of FA and minimal numbers of SCE and ONV for each question. For hypothesis-testing questions, the row is highlighted with a grey background and the tool predicted to be easier is highlighted in **bold** font. Appendix A.5.2 provides full descriptions of the predicted strategies for each question and tool.

Question	Hypothesis	Deep Intellisense	Rationalizer
A1	3.1	**1 SCE, 1 ONV**	2 ONV
A2	3.2	31 SCE	**1 FA**
A3	3.1	**1 SCE, 1 FA, 1 ONV**	2 ONV
A4	–	1 SCE, 2 ONV	2 ONV
B1	3.1	**1 SCE, 1 ONV**	2 ONV
B2	–	1 SCE, 1 ONV	1 ONV
B3	3.2	18 SCE	**1 FA**
B4	–	1 SCE, 1 FA, 1 ONV	1 FA, 1 ONV

Acronyms: FA: filter application; ONV: opening(s) of new view(s); SCE: selection(s) of code element(s).
FA operations are always optional, since the user can still find the necessary data without them. Single initial SCE operations are not considered in comparisons since, in our experimental setup, they occur automatically when the Mylyn task context for a question is opened.

15, 2010. Eclipse Mylyn 3.4.2 was used to provide task contexts for each question on the participant worksheet. The participant laptop was kept closed and connected to a $24''$ LCD monitor (Dell model no. 2405FPW) using a screen resolution of 1900×1200. The participant laptop was also connected to a Logitech optical USB mouse with a scroll wheel (model no. M-BZ96C), a Microsoft Comfort Curve Keyboard 2000 v1.0, and a Logitech microphone. The experimenter used a laptop running a Python script to time the participants during the questions and play warning sounds near the end of the allocated time (a short sound when there was one minute remaining, and a longer sound when no time remained.)

4.2 Results

4.2.1 Model Predictions

Difficulty of Observed Strategies

To validate our model predictions, we reviewed the experiment notes and logs to see how each participant answered each question in the question sets. We used the notes to review the strategy a participant used by forming the sequence of basic operations (filter applications, view opens, and code element selections) performed, and used the logs to verify operation counts. We then compared the participant's sequence of operations to those in the strategy predicted in Appendix A.5.2 (summarized in terms of operation counts in Table 4.1) for that question and tool. Table 4.2 shows statistics for the observed operation counts, and Table 4.3 summarizes how well the observed user strategies conformed to our predictions, in the sense of being "as hard as predicted". Since our predictions are meant to provide lower bounds for the difficulty of answering the questions using the tools, we considered our predictions successful if the user (a) performed as many SCE and ONV operations as predicted; (b) performed a different sequence of operations that was of a similar level of difficulty; or (c) failed to produce a strategy sufficient to address the major demands of the question. Cases in category (c) are excluded from the operation count statistics in Table 4.2. We found that the strategies employed by participants usually fell into categories (a) or (c); exceptions are described below.

In the case of the "history to source" questions (A2 and B3), most Deep Intellisense users (5 of 7 participants for question A2 and 5 of 6 participants for question B3) employed a strategy in category (b): instead of selecting each method in turn, they used the "event history" view to identify the past revisions by anyssen (there were two for A2 and three for B3) and opened comparison views to examine the changes introduced by these revisions. We do not believe this strategy is easier than our predicted strategy as it requires examining many methods in multiple revisions and is more error-prone (some users identified methods that no longer existed as "modified" based on revisions earlier than the current one.)

Table 4.2: Summary of observed event counts. n_s denotes the number of participants who formed a strategy sufficient to address the major demands of the question. Cases where the minimum number of ONV or SCE was lower than predicted are highlighted in light grey.

(a) Deep Intellisense

Question	n_s	ONV			SCE			FA		
		Min.	Med.	Max.	Min.	Med.	Max.	Min.	Med.	Max.
A1	7	1	2	4	1	1	4	0	0	0
A2	6	0	3	6	1	6	34	0	0	2
A3	7	1	2	5	1	1	25	0	1	2
A4	7	3	5	7	1	1	2	0	0	4
B1	4	1	4	6	1	1	3	0	0	1
B2	6	0	2	4	1	4	7	0	0	0
B3	6	1	2	9	1	2	19	0	0	1
B4	6	2	2	3	1	1	1	0	0	2

(b) Rationalizer

Question	n_s	ONV			FA		
		Min.	Med.	Max.	Min.	Med.	Max.
A1	1	2	2	2	0	0	0
A2	6	0	0	0	0	1	1
A3	0						
A4	4	2	3	5	0	0	0
B1	6	1	4	7	0	0	3
B2	7	1	2	3	0	0	2
B3	7	0	0	3	0	1	2
B4	6	1	1	3	1	2	3

Table 4.3: Conformity of observed strategies to predicted difficulty level

Question	Proportion of users whose strategy was as hard as predicted	
	Deep Intellisense	**Rationalizer**
A1	7/7	6/6
A2	7/7*	6/6
A3	7/7	6/6
A4	7/7	6/6
B1	6/6	5/7*
B2	4/6*	7/7
B3	6/6*	7/7
B4	6/6	7/7

* Actual strategies differed from prediction; see discussion in Section 4.2.1.

There were four cases in which users found strategies that were easier than predicted. In the case of question B1, we had expected that Rationalizer users would have to go back two revisions from the current revision to find the change of interest, but two Rationalizer users used filter criteria that we had not anticipated, discovered a line outside the target method that had been affected by the correct bug, and were therefore able to reach the correct revision in one step. In the case of question B2, the Deep Intellisense event history showed only two revisions, 1.1 and 1.44, and one bug affecting the method. Since 1.1 was the initial revision of the file and 1.44 had the same description as the bug, users could guess that 1.44 was the correct answer without opening a new view to check that it actually made the change. Two users therefore achieved lower ONV counts than predicted.

Other Measures

We also validated the predictions of our model by comparing the user task completion times, satisfaction ratings, and answer correctness scores for Rationalizer and Deep Intellisense. For these comparisons, we used a rank-based non-parametric statistical analysis since the results were not always normally distributed. Table 4.4 reports our comparisons of the median task completion times, satisfaction ratings and correctness scores, using the Wilcoxon-Mann-Whitney test to assess signifi-

cance. Figure 4.3 provides boxplot summaries of completion times, satisfaction ratings and correctness scores.

For the "deep history" questions (A1, A3, and B1), median task completion time was lower with Deep Intellisense (significantly so for B1, not significantly so for A1 and A3); median user satisfaction was higher (significantly so for B1, not significantly so for A1 and A3); and median correctness was significantly higher for A1 and A3, but (non-significantly) lower for B1. For the "history to source" questions (A2 and B3), median task completion time was lower with Rationalizer (significantly so for A2, not significantly so for B3); median user satisfaction was higher (significantly so for A2, not significantly so for B3); and median correctness was also higher, but not significantly so for either question. In all cases where we obtained statistically significant differences in the median task completion times, user satisfaction ratings, or answer correctness scores for Rationalizer and Deep Intellisense, those results were in line with the predictions of Hypotheses 3.1 and 3.2 as outlined in Table 4.1.

4.2.2 Design Characteristics

In answer to our questions about preferences for specific design characteristics, participant opinion was almost evenly split on both characteristics. Five participants (P6, P7, P13, P14, P15) preferred the separate views provided by Deep Intellisense, as opposed to seven (P4, P8, P9, P10, P11, P12, P16) who preferred Rationalizer-style integration and one (P5) whose preference depended on the type of question. Six participants (P4, P6, P8, P11, P12, P16) preferred to use a line-by-line interface style as in Rationalizer[3], six (P7, P9, P10, P13, P14, P15) preferred to start their code search from code elements as in Deep Intellisense, and two (P5, P13) said their preference depended on the type of question (P13, as noted, leaned towards element-based exploration.)

[3] P16 reported trying to access code on a line-by-line basis with Rationalizer, but not succeeding due to lack of familiarity with the tool. However, this participant preferred line-by-line access to historical information in general.

Table 4.4: Comparisons of median task completion times, satisfaction ratings and correctness scores. U is the Wilcoxon-Mann-Whitney U-statistic. Statistically significant differences $(p < 0.05)$ are highlighted in light grey.

(a) Differences in median participant completion times. The lower median for each question is bolded.

Question	Rationalizer		Deep Intellisense		U	p
	n	Median (s)	n	Median (s)		
A1	6	398.5	7	**258**	44.5	0.6
A2	6	**193**	7	393	69	0.002
A3	6	420	7	**410**	42.5	0.3
A4	6	**280.5**	7	331	57.5	0.2
B1	7	373	6	**233**	24.5	0.009
B2	7	261	6	**136**	28.5	0.06
B3	7	**366**	6	367.5	42	1
B4	7	351	6	**315**	39	0.7

(b) Differences in median participant satisfaction ratings. The higher median for each question is bolded.

Question	Rationalizer		Deep Intellisense		U	p
	n	Median	n	Median		
A1	6	3.5	7	**4**	54.5	0.5
A2	6	**5**	7	3	33	0.02
A3	6	3	7	**4**	53.5	0.6
A4	6	**4**	7	3	42	0.4
B1	7	3	6	**4**	60.5	0.004
B2	7	4	6	**5**	51.5	0.2
B3	7	**4**	6	3	39.5	0.8
B4	7	3	6	**4.5**	55.5	0.06

Continued on next page...

Table 4.4: Comparisons of median task completion times, satisfaction ratings and correctness scores (continued)

(c) Differences in median participant correctness scores. The higher median, if any, for each question is bolded.

Question	Rationalizer		Deep Intellisense		U	p
	n	Median	n	Median		
A1	6	0%	7	**100%**	69.5	0.002
A2	6	**100%**	7	40%	42.5	0.4
A3	6	0%	7	**83%**	70	0.0006
A4	6	83%	7	**100%**	50	0.9
B1	7	**100%**	6	83%	40	0.8
B2	7	100%	6	100%		
B3	7	**90%**	6	60%	37.5	0.5
B4	7	100%	6	100%		

4.2.3 Tool Preferences

In response to the question "How likely would you be to use each prototype for your own code?", five participants said they would be more likely to use Deep Intellisense (P4, P6, P10, P13, P15), five said they would be more likely to use Rationalizer (P5, P8, P11, P12, P16), and three (P7, P9, P14) indicated they would be likely to use both without expressing a preference either way. Appendix B.2 provides additional observations concerning participants' stated tool preferences in particular situations.

4.2.4 General Comments

We recorded general feedback from participants in a number of areas.

Participants expressed appreciation for various aspects of the tools. Ten liked having a Deep Intellisense-style list of all changes to a method (P4, P5, P6, P7, P8, P9, P10, P13, P14, P15), while eight liked Rationalizer's integration with the editor (P4, P5, P8, P9, P10, P11, P12, P16). Two participants (P14, P16) said Deep Intellisense had a "traditional" user interface that felt more familiar to them; one (P15) commented that Deep Intellisense was easier to use without prior experience, and that Rationalizer had a steeper learning curve.

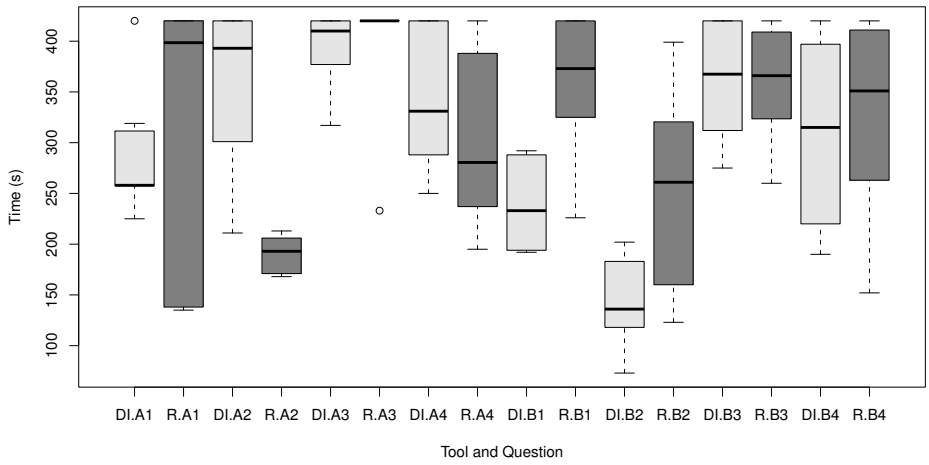
A few participants showed general enthusiasm about the tools. One said that Rationalizer had a more "intelligent" user interface than Deep Intellisense (P12). One participant had not been aware these kind of tools existed and found them exciting (P7); another volunteered that they seemed "a hundred times better" than the tools that participant currently used (P14).

Some participants wanted to extend or integrate the tools. Five expressed a desire to bring some features of Rationalizer into Deep Intellisense, or vice versa (P4, P5, P7, P8, P10). One participant wanted to adopt the tools and customize them for their own needs (P10).

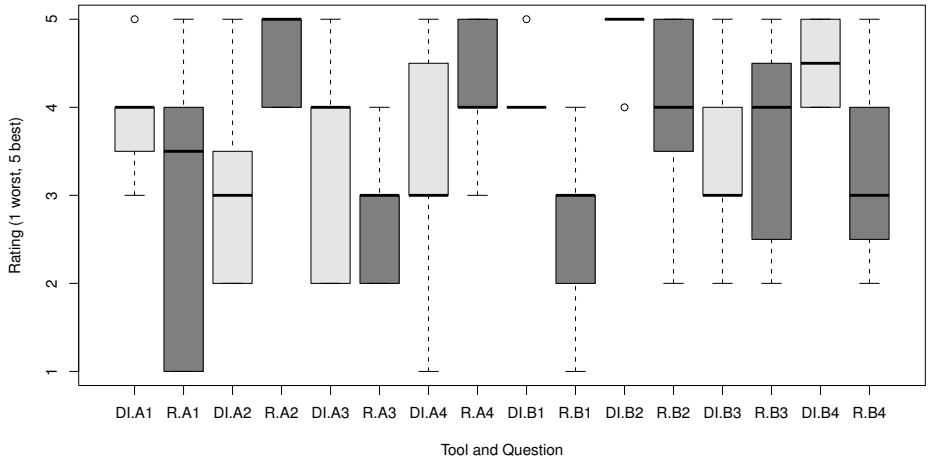
Participants identified various problems with the tools and made suggestions for improvement. Five wanted linkages between bugs and changesets to be more clearly indicated (P5, P7, P9, P11, P14). Seven participants expressed confusion with scrolling through large volumes of column information in Rationalizer (e.g., P7 remarked that they "could miss something") and/or desired more powerful filtering (P5, P7, P9, P10, P12, P14, P15). Similarly, one participant found Rationalizer's columns to be too "busy" and cluttered (P14); however, another thought the columns could fit *more* information (P11). Three participants stated that the user interface for navigating back to past history in Rationalizer was hard to use (P5, P10, P15). Five participants found Deep Intellisense's summary views ("current item" and "related people") to be unintuitive or not useful (P4, P5, P8, P12, P16). Finally, four participants expressed concerns about the interfaces intruding into their regular work, or the easiness of turning the interface on and off. Two found Deep Intellisense more intrusive (P4, P8), while two found Rationalizer more intrusive (P6, P14).

4.2.5 Summary

Participant performance on history exploration questions provided evidence for the validity of Hypothesis 3.1 (which predicted that "deep history" tasks would be easier with Deep Intellisense) and Hypothesis 3.2 (which predicted that "history to related source" tasks would be easier with Rationalizer.) With few exceptions, the strategies participants used to address the questions were at least as difficult as we predicted. We observed statistically significant median differences in one or



(a) Question completion times



(b) Participant satisfaction ratings

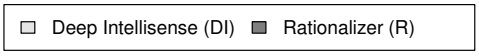
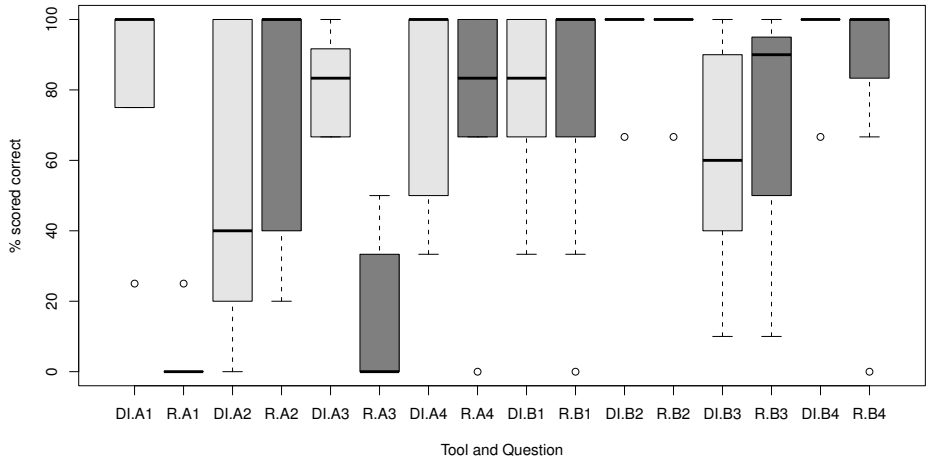


Figure 4.3: Boxplots of question completion times, satisfaction ratings and correctness scores. $n = 6$ for Rationalizer used in question set A and Deep Intellisense used in question set B; $n = 7$ otherwise.

Continued on next page...



(c) Participant answer correctness scores

Figure 4.3: Boxplots of question completion times, satisfaction ratings and correctness scores (continued)

more of our measures of interest (participant question completion times, satisfaction ratings and answer correctness scores) for four of the five questions that were designed to test our hypotheses.

Participant opinion on design characteristics (integration vs. separate views and line-based vs. element-based presentation) was almost evenly split in both cases. Participants also provided a variety of general comments about Deep Intellisense and Rationalizer, showing appreciation of various features (e.g., Deep Intellisense's event history or Rationalizer's editor integration), giving suggestions for extending or integrating the tools (e.g., merging the most-liked features of both tools), and identifying a number of usability problems (e.g., insufficiently clear indication of linkages between bugs and changesets, confusion due to the large amount of information in Rationalizer's columns, or finding Deep Intellisense's summary views to be unnecessary.)

4.3 Threats to Validity

A number of factors threaten the internal validity of our results. It is possible that the difficulty of questions may have been affected by factors other than the op-

erations predicted by our model, such as user disorientation upon encountering a confusing comparison view; this problem could be addressed in future work by creating larger and more rigorously tested question sets in an attempt to eliminate obstacles that do not correspond to model operations, and/or by extending the model to take new kinds of operations and obstacles into account. It is also possible that unconscious experimenter bias, prior knowledge of the existence of Deep Intellisense, or prior awareness of the experimenter's graduate work might have led some participants to perceive that Rationalizer was the experimenter's own work and consequently be biased to give positive feedback about it. The experimenter attempted to combat this risk by sticking to a pre-planned script (Appendix A.5.1) in introducing the study, by following a policy of commenting as little as possible while participants were answering the questions, and by declining to answer questions about the origin of the tools until after the study session was complete.

The generalizability of our results is threatened by two factors. First, our study involved only eight software history investigation questions related to a system unfamiliar to the participants. These questions may not be representative of participant behaviour on software history questions encountered on familiar systems in the course of real-world development work. However, we believe this risk is mitigated by the facts that participants generally showed good comprehension of what the questions were asking and several participants mentioned real-world experience with similar types of questions. Second, our 13 participants were self-selecting volunteers; it is possible that they may not be representative of the general developer population. As the majority of the participants had over a year's experience with bug and with source repositories, we believe the participants had sufficient background to represent non-novice developers.

Chapter 5

Discussion

The user study we conducted shows the variety of ways in which users tackle questions about the history of a software system and the differences that exist between the various strategies developers employ. In this chapter, we discuss how the results of the user study can be used to improve the design of software history exploration interfaces; argue that our model of history exploration could be improved by taking into account the potential for user mistakes and disorientation; and suggest some improvements to our study design for future researchers conducting similar studies.

5.1 Improving User Interfaces for History Exploration

Future tools could support multiple interface styles in each of the user interface dimensions we identified: they could provide breadth *and* depth on demand; allow line-based *and* element-based searches and result presentations; and make editor-integrated or separate-view interfaces *both* available depending on the user's wishes. New history exploration tools could merge the strengths of both of our tools. For instance, clicking on a revision in a Deep Intellisense-style events view could (as requested by some users) highlight the lines changed by that revision in the editor. Another possibility would be to make a Deep Intellisense-style events view available on demand from a Rationalizer-style breadth-first interface so that users could explore a line's history more deeply if desired. Future investigations may explore these and other ways in which the contrasting user interface

approaches we have studied through our two tools could be combined.

One lesson we learned from observation of user interactions with Rationalizer and from user comments in follow-up interviews is that good filtering is important to the success of a tool that, like Rationalizer, integrates into the editor and displays a large amount of information. Rationalizer's filtering was helpful but had an important shortcoming in practice: it was not always clear whether the filter had found any results, and users scrolling through long files sometimes missed the column blocks matching their filter and concluded that there were no matches for their filter. We recommend that Rationalizer-style interfaces should have filtering mechanisms that show the number of matches for a given filter and make it easy to jump from match to match.

5.2 Modeling User Mistakes

Our model of task difficulty, based as it is on operation counts, assumes that users know the right steps to take and that the difficulty of a task is proportional to the number of steps. However, in our observation of participant interaction with the tools, we found that long delays and failures to complete tasks usually resulted from taking wrong steps or having difficulty in finding the next correct step. Some of these episodes of user disorientation occurred because particular users were still learning how to use the tools and were not aware of some of their features (e.g., they did not realize it was possible to access past revisions through hyperlinks in Rationalizer); these arguably might occur less for users who were experts with the tools. However, some types of disorientation might affect even a user who was experienced with the tools. Users may decide to explore a revision that is not actually helpful for answering their question and expend time and effort discovering that it is not relevant. They may form mistaken hypotheses about code rationale based on bugs that seem relevant at first glance, then realize their mistake and backtrack. They may reach a confusing revision comparison view in which many changes have occurred and the change to a method of interest is difficult to find because the comparison view has not succeeded in aligning the old and new versions of the method for side-by-side comparison.

These observations suggest that an improved model of software history explo-

ration should help experimenters to identify potential sources of user mistakes and confusion. For instance, an experimenter estimating the difficulty of a task might consider how many paths are available at each step and estimate how many wrong paths the user might take before finding a correct one and how long it might take the user to recover from a wrong decision. This estimation might be based on some combination of empirical data and heuristics (e.g., conjecturing that longer wrong paths might require lengthier exploration).

5.3 Improvements to Study Design

While conducting the study, we became aware of a number of deficiencies in the study design. In this section, we describe those deficiencies and suggest ways of correcting them.

The initial guidance given to the users (through the tutorial and initial experimenter instruction script) could be improved in three respects:

- At least one participant did not find it clear from the tutorial that each information block in Rationalizer's columns was associated with the line(s) next to which it was displayed. One participant instead thought that Rationalizer's columns displayed deep historical information in the manner of Deep Intellisense and that clicking on different lines would update the columns. Future tutorials for Rationalizer-style tools should aim to avoid this misunderstanding through careful writing and extensive piloting. Experimenters might consider using video tutorials or asking users to perform training tasks to improve the users' understanding of the tools.

- The initial guidance provided no instructions on how to use Bugzilla[1]; particularly, it was not made clear whether users could perform bug searches directly in Bugzilla independently of the tools. Although users were required to have some experience with an issue-tracking system, they were not required to have any prior experience with Bugzilla. In the experiments, some users did in fact use Bugzilla for bug searches independent of the tools, while others did not. Also, some users made futile attempts to solve the task

[1] http://www.bugzilla.org/, verified March 30, 2011

problems using unexpected Bugzilla features such as dependency trees.

To mitigate the confounding effect of varying levels of user familiarity with Bugzilla, we recommend adding explicit directions to the initial guidance that make it clear that the participant can start searches from Bugzilla at any time and describe the features of Bugzilla that the participant is expected to use.

- The initial instructions did not specify whether users were allowed to use Eclipse's "Show Annotation" (cvs annotate) feature or history view. One participant was familiar with the "Show Annotation" feature and attempted to use it; the experimenter intervened to prevent this. We recommend using the initial instructions to direct participants not to use the "Show Annotations" feature and/or history view.

Furthermore, the initial questionnaire (Appendix A.3) did not ask users about their background with software history exploration tools such as cvs annotate. It is possible that prior user familiarity with such tools could affect user responses to our tools; for example, users familiar with CVS' history annotations might be better prepared to use Rationalizer than users who were not familiar with them. The questionnaire should be revised to ask potential participants whether they have used cvs annotate or other history exploration tools (e.g., Fisheye[2]). Data collected in an experiment using this revised questionnaire could be examined for any effect experience might have on performance with our tools.

Finally, analysis of the study results could be given significantly greater power by devising versions of question sets A and B that are isomorphic—that is, making each question in question set A sufficiently "equivalent" to a question in question set B that direct comparisons could be drawn between the results from such pairs of questions. The precise standard to be used for "equivalence" of history exploration questions would require careful consideration. Ideal isomorphic question pairs would probably deal with classes of identical structure with identically structured revision and bug histories, where only the names and content of methods, the names of authors and the content of check-in notes and bugs differed. It might not

[2] http://www.atlassian.com/software/fisheye/, verified March 11, 2011

be feasible to find questions in real codebases that shared class and history structure to that degree; therefore, creating isomorphic question sets might require creating artificial revision histories loosely based on real code.

Chapter 6

Related Work

A number of recent studies have considered the information needs of software developers. Sillito and colleagues [26] focused on eliciting the questions programmers ask as they work on software evolution tasks. LaToza et al. [23] used surveys and interviews to investigate what problems developers typically encountered and found that many problems arose from expending effort trying to reconstruct an appropriate mental model for their code; 66% of developers in one survey found understanding the rationale for a piece of code to be a serious problem, and 51% said the same of understanding code history. Ko et al. [20] observed working developers to identify their information needs, and found that questions about why code was implemented a certain way were frequent but often difficult to answer, and that developers sometimes used revision histories and bug reports to attempt to answer such questions. LaToza and Myers [21] conducted a large survey of professional developers and identified 21 categories of questions the developers found hard to answer; their most frequently reported categories dealt with the intent and rationale behind code. In this thesis, we have investigated how different user interfaces may support a subset of these questions related to software history.

Previous work in software history mining has proposed models for the entities and relationships involved in software history. In describing their Hipikat tool, Čubranić and Murphy [8] provided an artifact linkages schema which is similar to our ER model (Figure 1.1), but incorporates other types of historical artifacts such as project documents and forum messages. Venolia [27] modeled a similarly di-

verse variety of software-related artifacts from multiple repositories as a directed multi-graph; her Bridge tool attempted to discover relationships between the artifacts using textual allusions. Both of these models differ in purpose from ours; they are designed as a basis for implementing software history exploration tools rather than for user interface analysis.

LaToza and Myers [22] argue that understanding the strategies that software developers choose and modeling how they decide between strategies is an important way to identify challenges that should be addressed by tools; they call for a theory of coding activity that could, among other things, help designers identify assumptions made by tools. Our modeling of software history navigation in terms of basic operations attempts to predict the simplest strategy possible under particular tool designs, although it does not attempt to predict how developers might choose between different possible strategies. Our operation model has some resemblance to the classic GOMS model introduced by Card et al. [4, 5]; however, it does not attempt to model operation sequences at the level of fine-grained detail found in classical GOMS analyses and does not attempt to predict precise task execution times. It shares a key limitation of GOMS, in that it postulates an expert user who knows the correct sequence of operations to perform for any task, and does not take into account learning, user errors, mental workload or fatigue.

As mentioned in the introduction, a number of common tools, such as CVS' "annotate" feature, provide developers with access to historical information in current IDEs. IBM Rational Team Concert [16] takes these tools a step further by augmenting the Eclipse "annotate" feature with references to work items. As these references are provided only at the margin of the editor in minimal space, they must be traversed one-by-one to extract their full information. The Rationalizer user interface introduced in this thesis is similar to the tools just described, but attempts to provide more information in an immediately visible manner to the user.

A number of research software visualization systems have assigned colours to code lines for various purposes. The early SeeSoft system created by Ball and Eick [2] shows large codebases in a condensed graphical representation; files are shown as tall, thin rectangles wherein each code line is represented by a thin coloured horizontal line. The colouring could represent code age or indicate where bug fixes had affected files. Subsequent systems such as Augur [13] and Tarantula [17] built

on this idea. Tarantula shows codebases coloured according to test suite coverage. Augur shows multiple code files coloured according to properties such as code age, and allows users to explore changes in the code over time using a slider. Voinea and colleagues [24, 28, 29, 30] have explored the domain of software evolution visualization extensively and have produced a variety of visualization techniques for exploring the history of source code files and entire codebases. All of these tools focus on providing large-scale overview visualizations of files, modules or projects. By contrast, the tools we have evaluated in this thesis focus on helping developers answer questions about particular lines and code elements in the context of a file editor.

Grammel et al. [14] argue that software engineering research tools are insufficiently customizable and inadequately grounded in the practices of real developers. They suggest a style of software analysis tool that works like a web mashup to integrate different sources of information. They recognize Deep Intellisense as a step towards leveraging multiple sources of information but note that its interface is not highly customizable. They cite Fritz and Murphy's "information fragments" tool [12] as an example of the more flexible kind of information composition they advocate. They believe the "conceptual workspace" of Ko et al. [19] also realizes their vision because of its support for "collecting task-related information fragments" so they can be seen "side-by-side". We believe the Rationalizer style of interface may provide a step towards aspects of their vision, as it could in principle be extended to support showing many kinds of information in resizable background columns side-by-side with code on demand.

Chapter 7

Conclusion and Future Work

Developers frequently ask questions about the rationale behind pieces of source code, and often find these questions difficult to answer. Exploring source code history is a common way of addressing such questions. Many approaches and tools have been developed to mine software history and present access to interesting historical information; however, little research has been devoted to designing user interfaces with which to present such information. In this thesis, we have compared two user interfaces for software history exploration: Deep Intellisense, which presents information in separate views and facilitates depth-first searches from code elements, and Rationalizer, which presents information integrated into the source code editor and facilitates a breadth-first display of historical information on a line-by-line basis. We have introduced a model of software history information and, through a laboratory user study, shown how it can be used to predict how the difficulty of performing different types of historical exploration tasks varies between different interface styles. Through our study, we have also obtained user feedback that can inform the design of future software history exploration tools; user requests include clear indications of linkages between history artifacts such as bugs and changesets, well-designed filtering in Rationalizer-style interfaces, and interfaces that merge the strengths of both of our tools. The results of our study can give tool developers insight into how their user interface design choices impact the ease and efficiency of history exploration tasks.

Our results leave a number of areas for future work. Further laboratory studies

could attempt to isolate the individual effects, if any, of different design choices in each of the areas we have considered in this thesis (namely, breadth vs. depth, line-based display vs. element-based searches, and editor integration vs. separate views) on user satisfaction and performance by comparing variants of our tools that differed in only one of those areas. Field studies could be conducted to test the usability of our tools in real development work. Future field studies could also help to determine the relative frequency in real-world development of the types of historical questions we identified and to gather examples of such questions that occur in actual practice; such data would permit more realistic tests of software history exploration tools and allow stronger conclusions to be drawn about which interface styles are better for such tools.

Bibliography

[1] ACM, Inc. ACM copyright policy, version 6, Jan. 2011. URL http://www.acm.org/publications/policies/copyright_policy. → p. iii

[2] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4): 33–43, 1996. ISSN 0018-9162. doi:10.1109/2.488299. → p. 40

[3] A. W. J. Bradley and G. C. Murphy. Supporting software history exploration. In *MSR 2011: Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, May 2011. To appear. → p. iii

[4] S. K. Card, T. P. Moran, and A. Newell. Computer text-editing: An information-processing analysis of a routine cognitive skill. *Cognitive Psychology*, 12(1):32–74, 1980. ISSN 0010-0285. doi:10.1016/0010-0285(80)90003-1. → pp. ix and 40

[5] S. K. Card, T. P. Moran, and A. Newell. The keystroke-level model for user performance time with interactive systems. *Commun. ACM*, 23:396–410, July 1980. ISSN 0001-0782. doi:10.1145/358886.358895. → p. 40

[6] P. Cederqvist et al. *Version Management with CVS (release 1.11.23)*, May 2008. URL http://ximbiot.com/cvs/manual/cvs-1.11.23/cvs_16.html#SEC126. Cf. appendix A.8. → p. 1

[7] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, March 1976. ISSN 0362-5915. doi:10.1145/320434.320440. → p. ix

[8] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. doi:10.1109/ICSE.2003.1201219. → pp. 1 and 39

[9] P. Deva. Folding in Eclipse text editors. Eclipse Corner article, Mar. 2005. URL http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html. → p. 13

[10] Eclipse Foundation. Eclipse Public License, version 1.0, 2004. URL http://www.eclipse.org/legal/epl-v10.html. → p. 9

[11] *Determining who last modified a line with the Annotate command (Eclipse documentation)*. Eclipse Foundation, 2010. URL http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-cvs-annotate.htm. → p. 1

[12] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi:10.1145/1806799.1806828. → p. 41

[13] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. doi:10.1109/ICSE.2004.1317461. → p. 40

[14] L. Grammel, C. Treude, and M.-A. Storey. Mashup environments in software engineering. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 24–25, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-975-6. doi:10.1145/1809198.1809207. → p. 41

[15] R. Holmes and A. Begel. Deep Intellisense: a tool for rehydrating evaporated information. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 23–26, Leipzig, Germany, 2008. ACM. ISBN 978-1-60558-024-1. doi:10.1145/1370750.1370755. → pp. ii, 1, and 5

[16] *Viewing annotations (IBM Rational Team Concert 2.0.0.2 online help)*. IBM Corporation, 2009. URL http://publib.boulder.ibm.com/infocenter/rtc/v2r0m0/topic/com.ibm.team.scm.doc/topics/t_annotate.html. → p. 40

[17] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference*

on Software Engineering, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi:10.1145/581339.581397. → p. 40

[18] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006. ACM. ISBN 1-59593-468-5. doi:10.1145/1181775.1181777. → p. 2

[19] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006. ISSN 0098-5589. doi:10.1109/TSE.2006.116. → p. 41

[20] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi:10.1109/ICSE.2007.45. → pp. 1 and 39

[21] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0547-1. doi:10.1145/1937117.1937125. → pp. 1, 14, and 39

[22] T. D. LaToza and B. A. Myers. On the importance of understanding the strategies that developers use. In *CHASE '10: Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, pages 72–75, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-966-4. doi:10.1145/1833310.1833322. → p. 40

[23] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi:10.1145/1134285.1134355. → pp. 1 and 39

[24] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: An interactive toolset for source code investigation. In *INFOVIS '05: Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 4, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9464-x. doi:10.1109/INFOVIS.2005.33. → p. 41

[25] Mozilla Developer Centre. Hacking with Bonsai, May 2009. URL https://developer.mozilla.org/en/Hacking_with_Bonsai. → p. 1

[26] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM. ISBN 1-59593-468-5. doi:10.1145/1181775.1181779. → p. 39

[27] G. Venolia. Textual allusions to artifacts in software-related repositories. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 151–154, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2. doi:10.1145/1137983.1138018. → pp. 1, 6, and 39

[28] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, June 2009. ISSN 1382-3256. doi:10.1007/s10664-008-9068-6. → p. 41

[29] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software Visualization*, pages 47–56, New York, NY, USA, 2005. ACM. ISBN 1-59593-073-6. doi:10.1145/1056018.1056025. → p. 41

[30] S. L. Voinea. *Software Evolution Visualization*. PhD thesis, Technische Universiteit Eindhoven, 2007. URL http://repository.tue.nl/629335. → p. 41

Appendix A

Study Materials

Contact information has been redacted from all of the following documents.

A.1 Recruitment Materials

A.1.1 Recruitment Email

```
Subject: Call for Study Participants - Answering Questions about
         Software History ($20 gift card)

We are conducting a study to evaluate different techniques for
answering questions about software history.

The study will take approximately 90 minutes of your time. You will be
compensated for your participation with a gift card valued at $20.

Who: To participate, you must...
  * be age 19 or older
  * have a strong command of spoken and written English
  * have normal colour vision
  * not have any physical or mental impairment that makes it difficult
    for you to use a computer without assistive technologies
  * have experience programming in Java
  * have experience using version control systems and issue/bug
    tracking systems for software development
```

Where: The interview can be conducted at UBC Point Grey (ICICS Building, 2366 Main Mall) or at your workplace.

Should you wish to participate, please visit this website to sign up and fill out an initial questionnaire:

REDACTED

We look forward to hearing from you.

[Version: October 12, 2010]

A.1.2 Recruitment Advertisement

The University of British Columbia
Department of Computer Science

201 - 2366 Main Mall
Vancouver, B.C.
V6T 1Z4

CALL FOR PARTICIPANTS

Answering Questions about Software History

Co-Investigator: Alex Bradley, M.Sc. Student ([redacted])
Principal Investigator: Dr. Gail Murphy ([redacted])

Earn a $20 gift card!

If you are **reasonably proficient in Java** and have **experience using version control and bug/issue tracking systems for software development**, we invite you to participate in a study about different techniques for answering questions about software history. The study session will take at most **90 minutes.**

Interested? Want to sign up? See
[redacted]

This study is part of Alex Bradley's graduate thesis research, in which he is supervised by Dr. Murphy.

Version: October 12, 2010

[redacted] Code History Study

A.1.3 Recruitment Website

Answering Questions about Software History

a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
201–2366 Main Mall
Vancouver, B.C.
V6T 1Z4

Answering Questions about Software History

We are seeking reasonably proficient Java developers who are familiar with version control and issue tracking systems for a study, and would like to ask for your participation. The overall purpose of the project is to improve the design of user interfaces for integrating information from source code history into software development environments. This study will compare the usability and effectiveness of two software tools with that purpose.

This study is being conducted by Alex Bradley and Gail Murphy and will result in a public document. This study is part of Alex Bradley's graduate thesis research, supervised by Dr. Murphy.

This study will consist of (1) an initial online questionnaire and (2) one interview session involving a tool-usage experiment with a follow-up discussion. We will train you on the use of two software tools for exploring source code history and then ask you to use each tool for a number of tasks. We will screen capture your usage of the tools, as well as audio record and take written notes of your comments. At the end, the interviewer will ask you questions about your experience with the tools. The interview session will take a maximum of 90 minutes.

All data will be anonymized before any publication or presentation. You are free to stop participating in the study at any time without penalty. If you have any questions or desire further information with respect to this study, you may contact Dr. Gail Murphy (email: [redacted] tel: [redacted]).

Participation criteria: To participate, you must...

- be age 19 or older
- have a strong command of spoken and written English
- have normal colour vision
- not have any physical or mental impairment that makes it difficult for you to use a computer without assistive

[redacted]

technologies
- have experience programming in Java using the Eclipse IDE (e.g., at least one university course project, or professional experience)
- have experience using version control systems (e.g., CVS or Subversion) and issue/bug tracking systems (e.g., Bugzilla) for software development

The study session can be conducted at **UBC Point Grey** (ICICS building) or at **your workplace** (if your workplace is located in the City of Vancouver.) **You will be compensated for your participation with a gift card valued at $20.**

If you wish to participate, please review the consent form carefully, then proceed to the initial questionnaire.

Version: October 13, 2010

[redacted]

A.2 Consent Form

The University of British Columbia
Department of Computer Science

201 - 2366 Main Mall
Vancouver, B.C.
V6T 1Z4

Consent Form

Answering Questions about Software History

Principal Investigator: Dr. Gail Murphy, Dept. of Computer Science [redacted]

Co-Investigator: Alex Bradley, M.Sc. Student, Dept. of Computer Science [redacted] This study is part of Alex Bradley's graduate thesis research, in which he is supervised by Dr. Murphy.

Purpose:
The overall purpose of the project is to improve the design of user interfaces for integrating information from source code history into software development environments. This study will compare the usability and effectiveness of two software tools with that purpose. This study is funded in part by NSERC.

Study Procedures:
This study will consist of (1) an initial online questionnaire and (2) one interview session involving a tool-usage experiment with a follow-up discussion. We will train you on the use of two software tools for exploring source code history and then ask you to use each tool for a number of tasks. We will screen capture your usage of the tools, as well as audio record and take written notes of your comments. At the end, the interviewer will ask you questions about your experience with the tools. The interview session will take a maximum of 90 minutes.

Known Risks:
There are no known risks from this study beyond those involved in normal workplace activities.

Remuneration:
You will not be compensated for participation in the initial online questionnaire. You will be compensated for your participation in the interview session with a gift card valued at $20. You may still obtain the gift card if you withdraw before completion of the interview.

The screen capture, the written notes and the audio record will not be stored in a form that contains any personally identifying information about you. Identifying information collected in the online questionnaire will be kept confidential. You will be identified by numbers or pseudonyms in any written reports or publications. All data (on paper and electronic) will otherwise be kept in locked rooms, locked filing cabinets, or access-protected by password.

Contact for information about the study:

If you have any questions or desire further information with respect to this study, you may contact Dr. Gail Murphy at [redacted].

Contact for concerns about the rights of research subjects:

If you have any concerns about your treatment or rights as a research subject, you may contact the Research Subject Information Line in the UBC Office of Research Services at [redacted].

Consent:

Your participation in this study is entirely voluntary and you may refuse to participate or withdraw from the study at any time.

Your expression of agreement to the terms of this consent form via the first question of the online questionnaire indicates that you consent to participate in the questionnaire. If you decide not to schedule an interview session or withdraw consent during the interview session, all data collected from you in the questionnaire will be discarded.

Your signature below during the interview session indicates that you have received a copy of this consent form for your own records. Your signature indicates that you consent to participate in this study. You do not waive any legal rights by signing this consent form.

_____

Name of participant (please print)

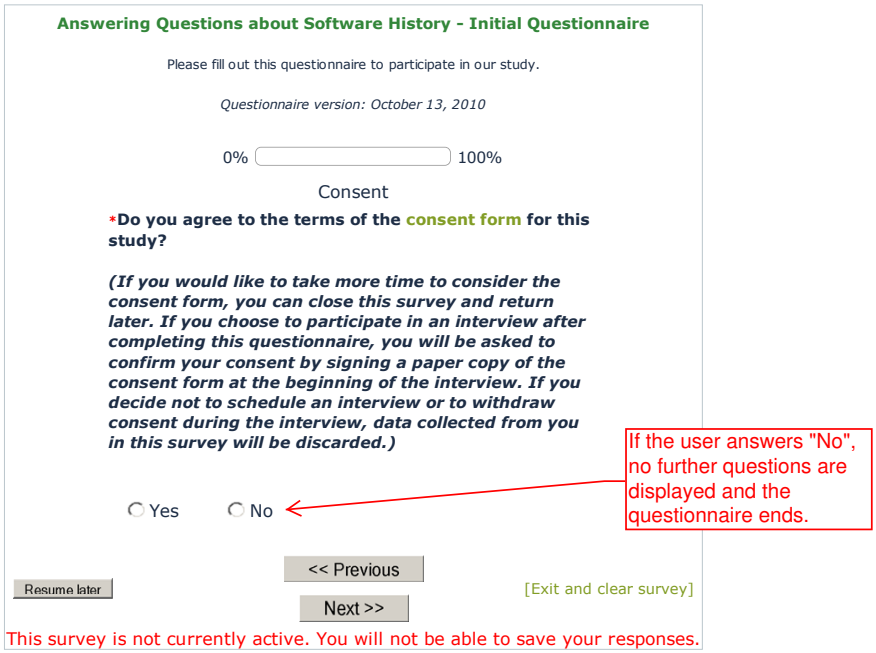
_____

Participant's Signature Date

_____

Investigator's Signature Date

A.3 Initial Online Questionnaire (Annotated)



Answering Questions about Software History - Initial Questionnaire

Please fill out this questionnaire to participate in our study.

Questionnaire version: October 13, 2010

0% 100%

Personal information

***What is your name?**

***What is your e-mail address?**

What is your gender?

○ Female ○ Male ◉ No answer

What is your age?

Choose one of the following answers

- ○ 19-29
- ○ 30-39
- ○ 40-49
- ○ 50-59
- ○ 60+
- ◉ No answer

Resume later << Previous Next >> [Exit and clear survey]

This survey is not currently active. You will not be able to save your responses.

Answering Questions about Software History - Initial Questionnaire

Please fill out this questionnaire to participate in our study.

Questionnaire version: October 13, 2010

0% 100%

Experience

***Which of the following roles currently apply to you?**

Check any that apply

- ☐ Undergraduate student
- ☐ Graduate student
- ☐ Software developer (commercial)
- ☐ Software developer (open source)
- ☐ Software tester
- ☐ Software development manager
- ☐ Other:

***What is your current program of study?**

Choose one of the following answers

- ○ Computer Science
- ○ Electrical and Computer Engineering
- ○ Master of Software Systems
- ○ Other:

This question is displayed only if the answer to the previous question is "Undergraduate student" or "Graduate student".

***How many years of experience do you have developing software using a version control system?**

Choose one of the following answers

- ○ No experience
- ○ Less than 1 year
- ○ 1-2 years
- ○ 3-4 years
- ○ 5-6 years
- ○ 6-7 years

- 8-9 years
- 10 years or more

*How many years of experience do you have developing software using an issue (bug) tracking system?

Choose one of the following answers

- No experience
- Less than 1 year
- 1-2 years
- 3-4 years
- 5-6 years
- 6-7 years
- 8-9 years
- 10 years or more

*What IDEs have you used for software development?

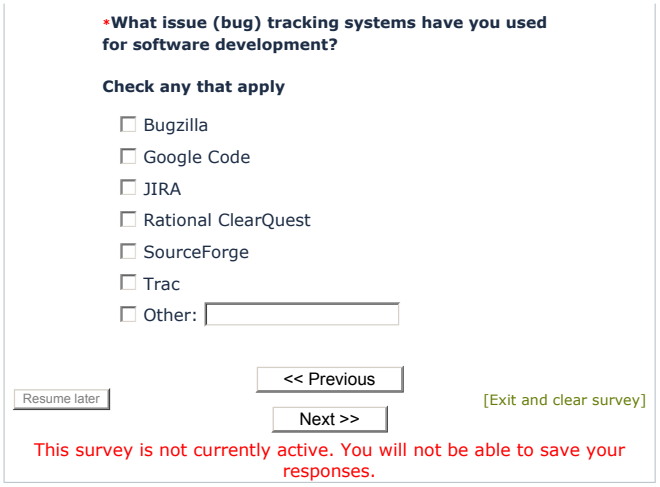
Check any that apply

- CodeGear
- Eclipse
- IntelliJ
- NetBeans
- Visual Studio
- Other:

*What version control systems have you used for software development?

Check any that apply

- CVS
- Git
- Mercurial
- Perforce
- Subversion
- Other:



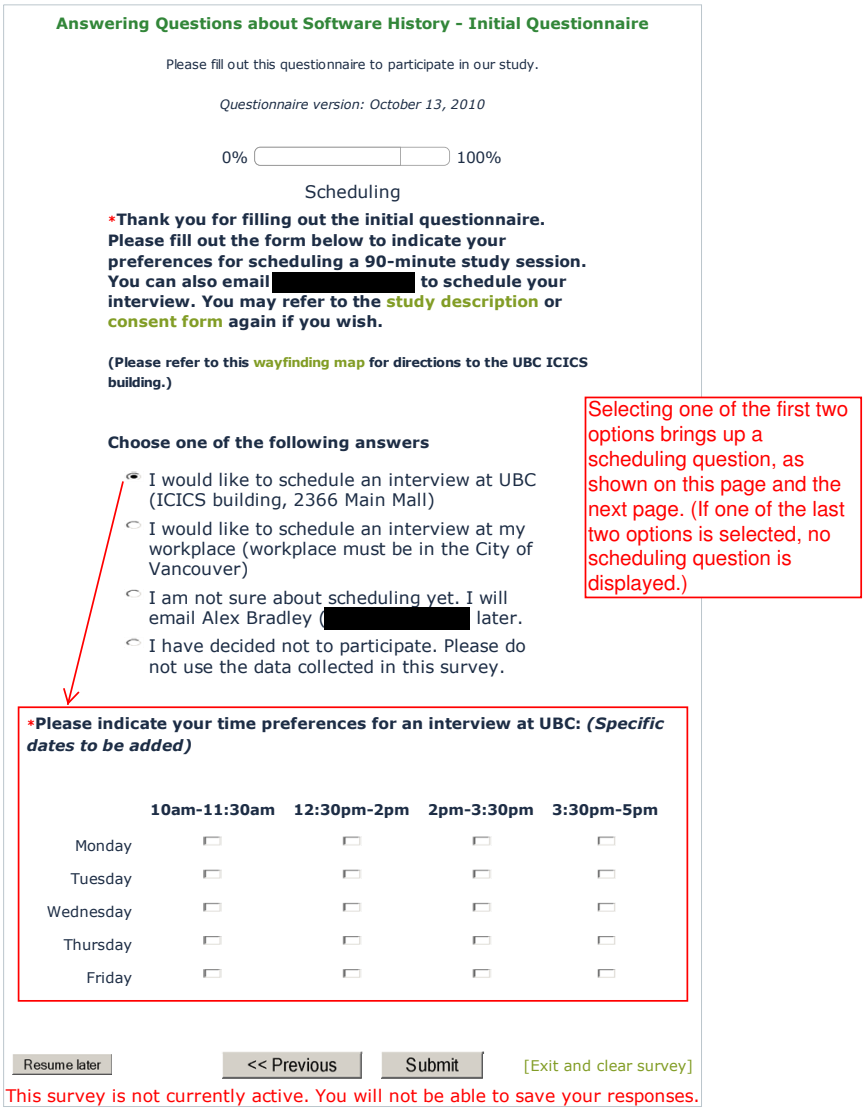
*What issue (bug) tracking systems have you used for software development?

Check any that apply

- [ ] Bugzilla
- [ ] Google Code
- [ ] JIRA
- [ ] Rational ClearQuest
- [ ] SourceForge
- [ ] Trac
- [ ] Other:

Resume later << Previous Next >> [Exit and clear survey]

This survey is not currently active. You will not be able to save your responses.



Answering Questions about Software History - Initial Questionnaire

Please fill out this questionnaire to participate in our study.

Questionnaire version: October 13, 2010

0% 100%

Scheduling

***Thank you for filling out the initial questionnaire. Please fill out the form below to indicate your preferences for scheduling a 90-minute study session. You can also email [redacted] to schedule your interview. You may refer to the study description or consent form again if you wish.**

(Please refer to this wayfinding map for directions to the UBC ICICS building.)

Choose one of the following answers

- (•) I would like to schedule an interview at UBC (ICICS building, 2366 Main Mall)
- ( ) I would like to schedule an interview at my workplace (workplace must be in the City of Vancouver)
- ( ) I am not sure about scheduling yet. I will email Alex Bradley ([redacted] later.
- ( ) I have decided not to participate. Please do not use the data collected in this survey.

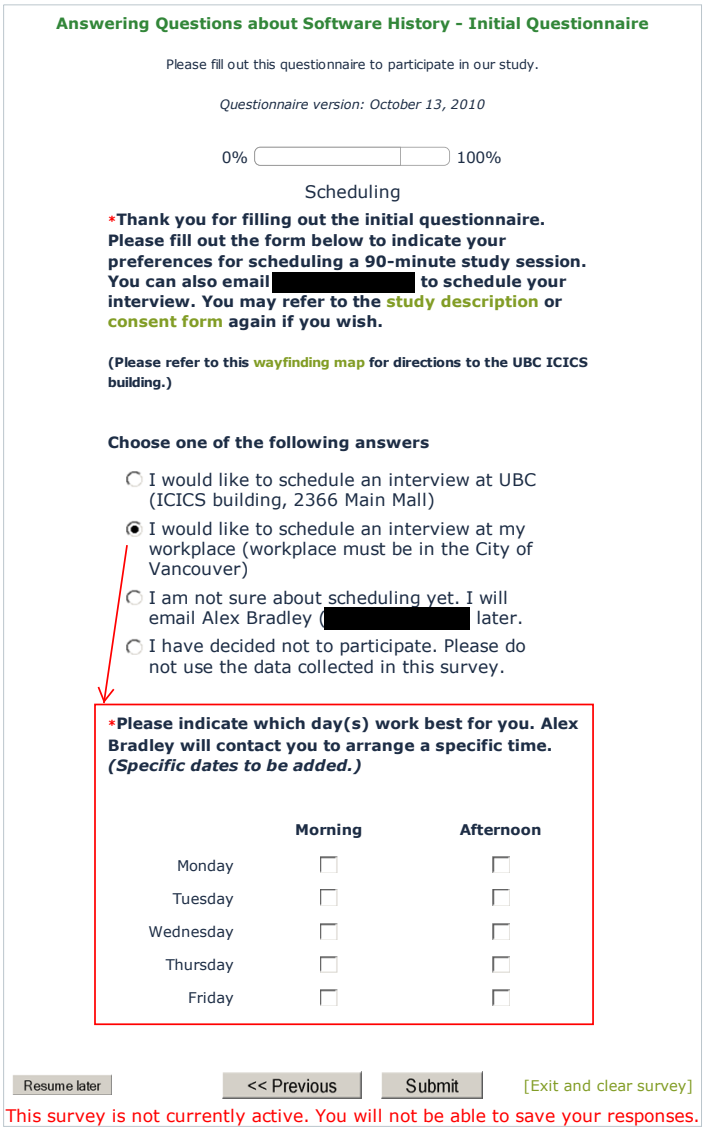
Selecting one of the first two options brings up a scheduling question, as shown on this page and the next page. (If one of the last two options is selected, no scheduling question is displayed.)

Please indicate your time preferences for an interview at UBC: *(Specific dates to be added)

	10am-11:30am	12:30pm-2pm	2pm-3:30pm	3:30pm-5pm
Monday	☐	☐	☐	☐
Tuesday	☐	☐	☐	☐
Wednesday	☐	☐	☐	☐
Thursday	☐	☐	☐	☐
Friday	☐	☐	☐	☐

Resume later << Previous Submit [Exit and clear survey]

This survey is not currently active. You will not be able to save your responses.



Answering Questions about Software History - Initial Questionnaire

Please fill out this questionnaire to participate in our study.

Questionnaire version: October 13, 2010

0% 100%

Scheduling

***Thank you for filling out the initial questionnaire. Please fill out the form below to indicate your preferences for scheduling a 90-minute study session. You can also email [redacted] to schedule your interview. You may refer to the study description or consent form again if you wish.**

(Please refer to this wayfinding map for directions to the UBC ICICS building.)

Choose one of the following answers

- ○ I would like to schedule an interview at UBC (ICICS building, 2366 Main Mall)
- ◉ I would like to schedule an interview at my workplace (workplace must be in the City of Vancouver)
- ○ I am not sure about scheduling yet. I will email Alex Bradley ([redacted] later.
- ○ I have decided not to participate. Please do not use the data collected in this survey.

Please indicate which day(s) work best for you. Alex Bradley will contact you to arrange a specific time. *(Specific dates to be added.)

	Morning	Afternoon
Monday	☐	☐
Tuesday	☐	☐
Wednesday	☐	☐
Thursday	☐	☐
Friday	☐	☐

Resume later << Previous Submit [Exit and clear survey]

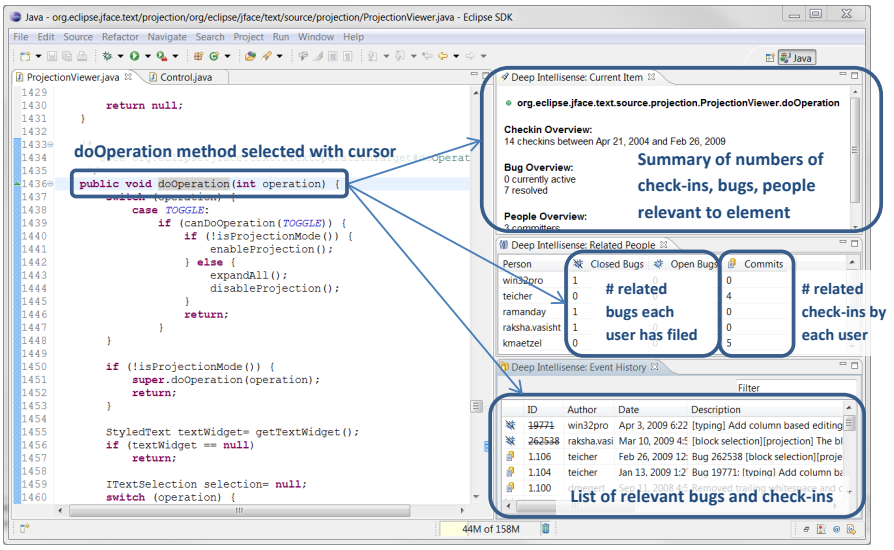
This survey is not currently active. You will not be able to save your responses.

A.4 Study Materials Given to Participant

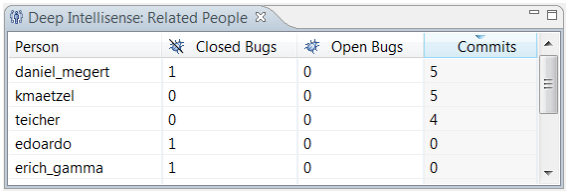
A.4.1 Deep Intellisense Tutorial

Tutorial for Deep Intellisense Tool

The Deep Intellisense tool displays information about the history of source code in views that are separate from the source code editor. When a code element (method or class) is selected in the editor, the Deep Intellisense views update with historical information about that element. Deep Intellisense provides three views: *Current Item*, which displays a summary of the current element's history (number of check-ins, committers and related bugs); *Related People*, which lists the people who have committed changes or filed bugs affecting the element; and *Event History*, which displays a list of past events (commits or bug updates) related to the element. Here is an annotated screenshot of all three views displaying historical information for the `doOperation` method:



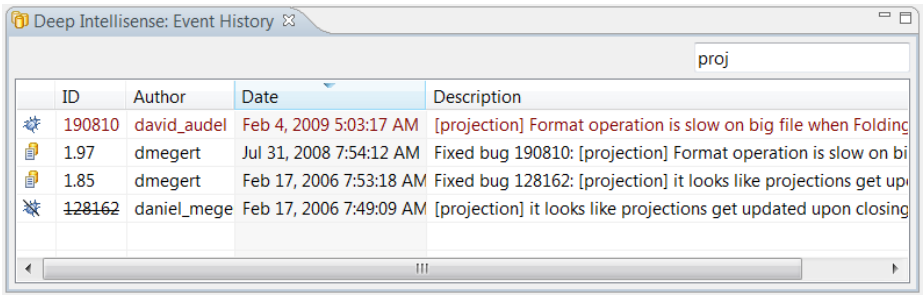
The *Related People* and *Event History* tables can be sorted on any column by clicking the column heading. For example, in the following screenshot, the *Related People* table has been sorted in descending order of number of commits:



Deep Intellisense: Related People

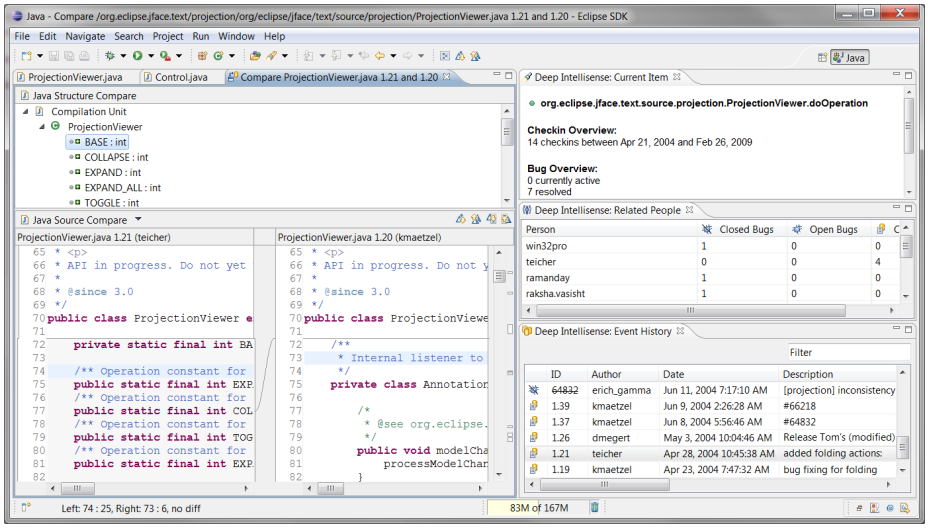
Person	Closed Bugs	Open Bugs	Commits
daniel_megert	1	0	5
kmaetzel	0	0	5
teicher	0	0	4
edoardo	1	0	0
erich_gamma	1	0	0

The *Event History* table can be filtered by entering filter text in the "Filter" box in the upper right-hand corner of the view. In the screenshot below, the table has been filtered to show only events matching the text "proj":



The above screenshot also demonstrates all the different kinds of events that can be shown. Check-ins are denoted by the icon. Open bugs are denoted by the icon and red text, while closed bugs are denoted by the icon and a crossed-out bug number.

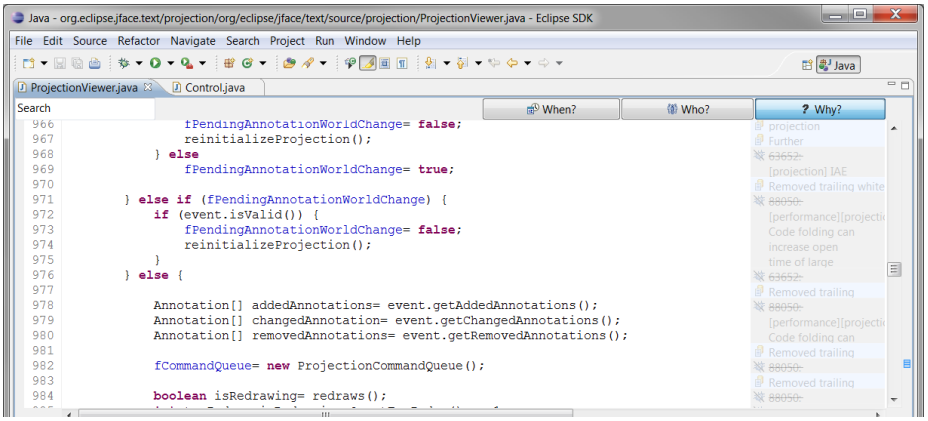
Double-clicking on the row for a bug event will open a web browser displaying the full Bugzilla entry for the bug. Double-clicking on the row for a check-in will open a comparison view showing the differences between that checked-in revision and the revision before it. For example, in the screenshot below, double-clicking on the check-in row for revision 1.21 has opened a comparison between revisions 1.21 and 1.20:



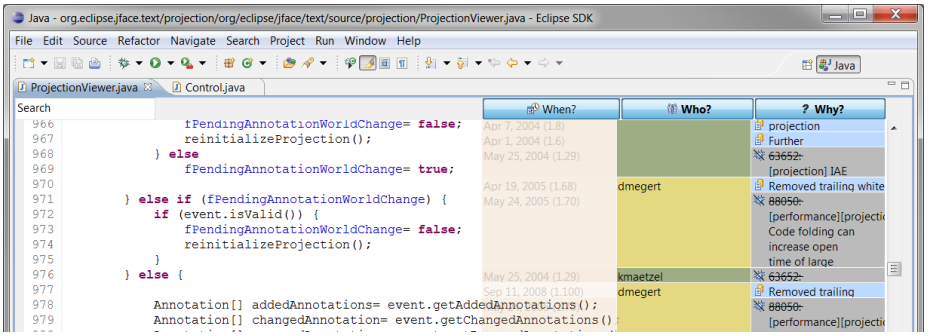
A.4.2 Rationalizer Tutorial

Tutorial for Rationalizer Tool

The Rationalizer tool displays information about the history of source code line-by-line in semi-transparent columns behind the text in a source code editor. The columns provide information that aims to answer three questions about each code line: (1) *When* was it last modified? (2) *Who* last modified it? (3) *Why* was the change made? You can enable a column by clicking on its heading. In the screenshot below, the "Why?" column has been enabled, showing information from bugs and check-in notes related to the last time each line was changed.



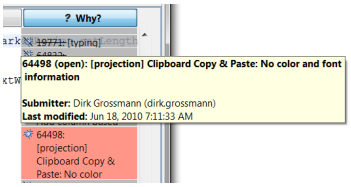
Clicking the column heading a second time brings the column into the foreground. Clicking the column heading a third time hides the column again. In the screenshot below, the "Who?" and "Why?" columns have been enabled and brought into the foreground, while the "When?" column has been enabled but remains in the background.



Rationale Information in the "Why?" Column

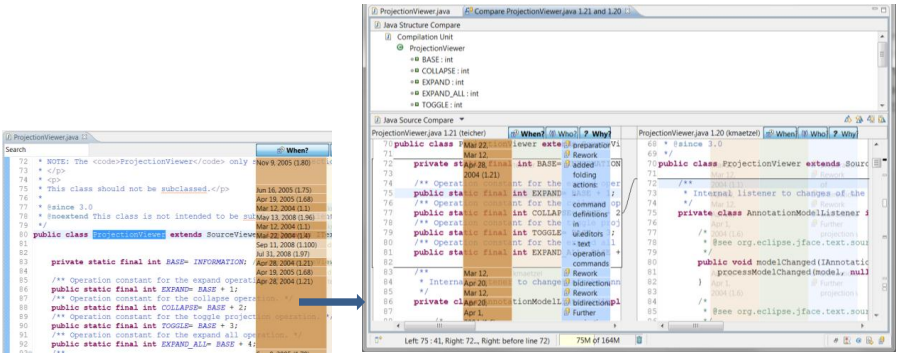
The "Why?" column currently displays rationale information drawn from two sources: version control check-in notes and associated bugs. Information drawn from check-ins has a icon and blue background colour. Information from closed bugs has a icon and grey background, while information from open bugs has a icon and red background. Examples can be seen in the screenshot below.

Hovering over column items provides more detailed information if available. For example, in the screenshot at right, hovering over the item for bug 64498 provides more information about the bug.



Hyperlinks from Column Items

Some column items have hyperlinks, which can be accessed by clicking on the item while holding `Ctrl`. For instance, `Ctrl`-clicking a bug entry in the "Why?" column will open a web browser displaying the full entry for the bug. `Ctrl`-clicking an entry in the "When?" column or a check-in note in the "Why?" column will open a comparison view showing the differences between the revision that last modified the line and the revision before it. In the screenshot below, clicking on the item for revision 1.21 in the "When?" column has opened a comparison between revisions 1.21 and 1.20:



Note that the comparison viewers also include "When?", "Who?" and "Why?" columns which function in the same way as in the regular editor. The same kinds of hyperlinks are available, allowing further navigation into the past.

Filtering

The information displayed in the columns can be filtered using the search text box in the upper left-hand corner of the editor. In the screenshot below, the search text box has been used to search for lines with historical information that mentions "folding".

A.4.3 Question Set Worksheets

Participant # _____ Tool: ❑ DI ❑ Rationalizer

QUESTION SET A

Question 1. Find how FlowLayoutEditPolicy.isHorizontal obtained layout container in the past

Consider the assignment "figure = getLayoutContainer()" in the isHorizontal method of FlowLayoutEditPolicy (line 171). Was a different method call used to obtain the layout container in the past? If so, please specify

(1) What was the old method call? _____

(2) Which bug, if any, led to the change? (State the bug number:) _____

(3) Who committed the change, and what was the revision ID? _____

Question 2. Which methods of LayoutEditPolicy were last modified by anyssen? *(Circle all that apply.)*

activate()
createChildEditPolicy(EditPart)
createListener()
createSizeOnDropFeedback(CreateRequest)
deactivate()
decorateChild(EditPart)
decorateChildren()
eraseLayoutTargetFeedback(Request)
eraseSizeOnDropFeedback(Request)
eraseTargetFeedback(Request)
getAddCommand(Request)
getCloneCommand(ChangeBoundsRequest)
getCommand(Request)
getCreateCommand(CreateRequest)
getCreationFeedbackOffset(CreateRequest)
getDeleteDependantCommand(Request)
getLayoutContainer()
getMoveChildrenCommand(Request)
getOrphanChildrenCommand(Request)
getSizeOnDropFeedback(CreateRequest)
getSizeOnDropFeedback()
getTargetEditPart(Request)
setListener(EditPartListener)
showLayoutTargetFeedback(Request)
showSizeOnDropFeedback(CreateRequest)
showTargetFeedback(Request)
undecorateChild(EditPart)
undecorateChildren()
getLayoutOrigin()
translateFromAbsoluteToLayoutRelative(Translatable)
translateFromLayoutRelativeToAbsolute(Translatable)

Question 3. What sections of FlyoutPaletteComposite were changed by the fix for bug 71525?

(1) What method(s) were changed? _____

(2) What revision made the fix? _____

(3) Briefly describe the nature of the change(s).

_____

_____

Question 4. Why are longs used in Geometry.segmentContainsPoint?

In Geometry.segmentContainsPoint, why are longs used for the square distance calculation?

(1) Identify the revision where they were first introduced. _____

(2) Identify (give bug ID: _____) and describe the bug that prompted it.

_____

Participant # _____ Tool: ❑ DI ❑ Rationalizer

QUESTION SET B

Question 1. Find last bug associated with null check in PaletteViewer.setActiveTool

What bug led to the introduction of the "editpart != null" check in the setActiveTool method of PaletteViewer? Please specify

(1) the bug number: _____

(2) the revision in which the bug was fixed: _____

(3) who committed the revision: _____

Question 2. Find why "pending layout request" check was added to Graph.applyLayout

Why was the hasPendingLayoutRequest check added to the applyLayout method of Graph? Please state

(1) which revision added the check: _____

(2) who committed the revision: _____

(3) the number of the bug that led to the change: _____

Question 3. Which methods of ConstrainedLayoutEditPolicy were last modified by anyssen?
(Circle all that apply.)

createAddCommand(ChangeBoundsRequest, EditPart, Object)
createAddCommand(EditPart, Object)
createChangeConstraintCommand(ChangeBoundsRequest, EditPart, Object)
createChangeConstraintCommand(EditPart, Object)
createChildEditPolicy(EditPart)
getAddCommand(Request)
getAlignChildrenCommand(AlignmentRequest)
getCommand(Request)
getConstraintFor(ChangeBoundsRequest, GraphicalEditPart)
getConstraintFor(Request, GraphicalEditPart, Rectangle)
getConstraintFor(Point)
getConstraintFor(Rectangle)
getConstraintFor(CreateRequest)
getConstraintForClone(GraphicalEditPart, ChangeBoundsRequest)
translateToModelConstraint(Object)
getResizeChildrenCommand(ChangeBoundsRequest)
getChangeConstraintCommand(ChangeBoundsRequest)
getMoveChildrenCommand(Request)

Question 4. What sections of ScaledGraphics were changed by the fix for bug 126747?

(1) What method(s) were changed? _____

(2) What revision made the fix? _____

(3) Briefly describe the nature of the change(s).

_____

_____

A.4.4 Per-Question Satisfaction Worksheets

Participant # _____ Tool: ❑ DI ❑ Rationalizer

QUESTION SET A

Question 1. Find how FlowLayoutEditPolicy.isHorizontal obtained layout container in the past

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Question 2. Which methods of LayoutEditPolicy were last modified by anyssen?

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Question 3. What sections of FlyoutPaletteComposite were changed by bug 71525?

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Question 4. Why are longs used in Geometry.segmentContainsPoint?

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Participant # _____ Tool: ❑ DI ❑ Rationalizer

QUESTION SET B

Question 1. Find last bug associated with null check in PaletteViewer.setActiveTool

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Question 2. Find why "pending layout request" check was added to Graph.applyLayout

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Question 3. Which methods of ConstrainedLayoutEditPolicy were last modified by anyssen?

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

Question 4. What sections of ScaledGraphics were changed by the fix for bug 126747?

How helpful did you find the tool while answering this question?

Not helpful at all 1 2 3 4 5 Did everything I wanted

A.5 Study Materials for Experimenter Use

A.5.1 Study Procedure Guidelines

Participant # _____

Answering Questions about Software History—Plan for User Study
Alex Bradley

Materials:

- **Hardware**
  - ❑ study laptop for participant (ThinkPad T43, Pentium M 1.86GHz, 1.5GB RAM, WinXP Pro SP3, screen resolution 1900x1200)
  - ❑ experimenter laptop (ThinkPad X60, Intel Core 2 Duo T7200 2GHz, 2GB RAM, Ubuntu 10.04)
  - ❑ power adapters for both laptops
  - ❑ USB keyboard and mouse
  - ❑ 24" LCD monitor
  - ❑ power bar
  - ❑ microphone (Logitech, model unknown)
  - ❑ pens
  - ❑ clipboard/writing surface
- **Printed materials**
  - ❑ Consent form (2 per session: 1 to be signed, 1 extra copy for user on request)
  - ❑ This checklist/procedure/initial instructions sheet (1 per session)
  - ❑ Rationalizer tutorial (version: Dec 21) (1 for all sessions)
  - ❑ DI tutorial (version: Oct 12) (1 for all sessions)
  - ❑ Question set worksheet (2 per session: 1 for participant, 1 for experimenter notes)
  - ❑ Per-question followup satisfaction rating worksheet (1 per session, for participant)
  - ❑ Follow-up interview question sheet (1 per session, for experimenter)
  - ❑ Gift certificate valued at $20 (1 per session)
  - ❑ Receipt for gift certificate (1 per session)
  - ❑ Tracking sheet for gift certificates (1 for all sessions)
  - ❑ *Optional:* Question set answer key (for experimenter, **not to be shown to participant**)

Procedure:

1. Reset Mylyn tasks and make sure version data cache is warm.
2. Participant assigned a user number.
3. Participant assigned to one of four question set/tool orderings:
   1. Rationalizer for question set A then DI for question set B
   2. Rationalizer for question set B then DI for question set A
   3. DI for question set A then Rationalizer for question set B
   4. DI for question set B then Rationalizer for question set A
4. Participant signs consent form. (2 minutes)
5. Experimenter gives initial instructions (overleaf) to participant (1 minute)
6. Participant does two tool sessions as follows: (70-74 minutes allocated)
   1. Read tutorial for tool (5-7 minutes)
   2. *(Videorecorded)* Perform 4 tasks with tool (7.5 minutes maximum each, so 30 minutes maximum in total)
      For each task:
      1. Experimenter activates Mylyn task context on study machine and starts timer on experimenter machine

Participant # _____

   2. User performs task
   3. Experimenter stops timer on experimenter machine and deactivates Mylyn task
   4. User gives answer to follow-up satisfaction question.
7. *(Audio recorded)* Followup interview with participant (Remainder of time, 13 minutes worst case)
8. Participant receives gift certificate and signs receipt. (1 minute)
9. Get log data from Mylyn, Rationalizer/DI log, experimenter laptop timer

---

Answering Questions about Software History—Initial Directions to Study Participant

Alex Bradley

- You will be trying out two tools for answering questions about the history of source code.
- You will read a tutorial for each tool then use that tool to answer a set of questions; this should take about 35 minutes for each tool.
- Each question will focus on a particular source code file from an Eclipse graphics framework. You don't need to have any prior knowledge of the framework.
- The Mylyn plugin will be used to bring up the files for each question. You don't need to know how to use Mylyn; I will do this for you.
- There are 4 questions in each set and you will have 7 minutes per question. When there is 1 minute remaining, you will hear a warning sound (*play warning sound*). When time is up, you will hear this sound (*play time-up sound*).
- Please write your answers to the questions on the question sheet provided.
- You may refer back to the tutorial at any time while working on the questions.
- After each question, you will be asked to use this sheet (*show rating sheet*) to rate how helpful you found the tool while working on the question.
- I cannot provide any information to you beyond what is written in the tasks and tutorials; if you find something unclear, please say so, but I cannot provide help.
- Finally, when you are working on the questions, please try to “think aloud”: that is, tell me what you're thinking as you work.

A.5.2 Predicted Strategies for Each Question and Tool

A1. *Deep Intellisense:* Select the isHorizontal method, identify revision 1.15 as the most plausible candidate based on its description, and open a comparison view (1.15–1.14) to confirm that it actually made the change. The bug number (191829) can be found in the revision description.

Rationalizer: Revision 1.17 was the last revision to affect the method, so open a comparison view for it (1.17–1.16) from the "When?" column. Scroll through the comparison view to find the getLayoutContainer() call in revision 1.16. The "When?" and "Why?" columns will indicate that revision 1.15 and bug 191829 last affected the line. Opening a comparison view (1.15–1.14) will confirm that revision 1.15 changed the way the layout container was obtained.

A2. *Deep Intellisense:* Select each of the 31 methods in the class and for each one, check whether anyssen was the last person to commit a change affecting it.

Rationalizer: Apply the filter anyssen and scroll through the file while looking at the "Who?" column to see which methods anyssen last affected.

A3. *Deep Intellisense:* Select the class FlyoutPaletteComposite. Apply the filter 71525. There will be one result: revision 1.31, which is described as providing fixes for bugs 71525 and 67907. Opening a comparison view (1.31–1.30) will show the methods affected by the fix.

Rationalizer: Filtering for 71525 will give an empty result, since none of the latest changes to the file are related to that bug. The user will need to jump back one or more revisions to find lines affected by the bug. The easiest way is probably to notice that there is a comment beginning "Fix for Bug# 71525" near the beginning of the file, use the "When?" column for that comment to open a comparison view (1.56–1.55), scroll up to the same comment in revision 1.55, and use the "When?" column there to get back to revision 1.31.

A4. *Deep Intellisense:* Select the segmentContainsPoint method. Four revisions

have affected the method. Start with the most recent by opening the comparison view for 1.9; it did not introduce any changes. Next, try 1.8.2.1; it did introduce the use of longs. The description of 1.8.2.1 is sufficient to provide a cursory rationale description; opening the full description for bug 313731 (which is the most recent bug in the event history, and is mentioned in the description of 1.8.2.1) will provide a fuller rationale explanation.

Rationalizer: The "Why?" column immediately shows that bug 313731 last affected the lines where longs are used. Use the "When?" column to open a comparison view for 1.9–1.8.2.1, and from there, use the "When?" column to open a comparison view for 1.8.2.1–1.8. The user may wish to open bug 313731 to obtain a fuller rationale explanation.

B1. *Deep Intellisense:* Select the setActiveTool method. Two bugs have affected the method; 270028 seems more likely to be relevant since it mentions a null pointer exception (NPE). Revision 1.44 can be identified as the fix for this bug since its check-in note has the bug number and description. Opening a comparison view (1.44–1.43) confirms that this revision added the null check.

Rationalizer: Use the "When?" column to open a comparison view for the last revision (1.45) to affect the line with the null check. The "Why?" column in revision 1.44 will show bug 270028. Opening the comparison view 1.44–1.43 will show that revision 1.44 made the change.

B2. *Deep Intellisense:* Select the applyLayout method. Only one bug (267558) affects the method, and there is one revision (1.44) that shares its description. Opening a comparison view (1.44–1.43) confirms that that revision made the change.

Rationalizer: Revision 1.44 and bug 267558 are immediately visible in the "When?" and "Why?" columns. Opening a comparison view (1.44–1.43) confirms that revision 1.44 made the change.

B3. *Deep Intellisense:* Select each of the 18 methods in the class and for each one, check whether anyssen was the last person to commit a change affecting it.

Rationalizer: Apply the filter anyssen and scroll through the file while looking at the "Who?" column to see which methods anyssen last affected.

B4. *Deep Intellisense:* Select class ScaledGraphics. Apply the filter 126747. There will be one result: revision 1.42, which mentions the bug in its description. Opening a comparison view (1.42–1.41) will show the changes made to the drawTextLayout method.

Rationalizer: Apply the filter 126747. Scroll through the class while looking at the "Why?" column to find areas affected by the bug; drawTextLayout is the only method affected. Use the "When?" column to open a comparison view (1.42–1.41) and find what changes were made.

A.5.3 Marking Guidelines for Question Set Worksheets

TASK SET A **ANSWER KEY**

Task 1. Find how FlowLayoutEditPolicy.isHorizontal obtained layout container in the past [4 marks]

Consider the assignment "figure = getLayoutContainer()" in the isHorizontal method of FlowLayoutEditPolicy (line 171). Was a different method call used to obtain the layout container in the past? If so, please specify

(1) What was the old method call? **(GraphicalEditPanel)get…()).getContentPane() [1 mark]**
(2) Which bug, if any, led to the change? **191829 [1 mark]**
(3) Who committed the change, and what was the revision ID? **Alexander Nyssen / 1.15 [1 mark each]**

Task 2. Which methods of LayoutEditPolicy were last modified by anyssen? (Circle all that apply.)

[Determine # right (/5) and # wrong]

activate()
createChildEditPolicy(EditPart)
createListener()
createSizeOnDropFeedback(CreateRequest)
deactivate()
decorateChild(EditPart)
decorateChildren()
eraseLayoutTargetFeedback(Request)
eraseSizeOnDropFeedback(Request)
eraseTargetFeedback(Request)
getAddCommand(Request)
getCloneCommand(ChangeBoundsRequest)
getCommand(Request) 1.27
getCreateCommand(CreateRequest)
getCreationFeedbackOffset(CreateRequest)
getDeleteDependantCommand(Request)
getLayoutContainer()
getMoveChildrenCommand(Request)
getOrphanChildrenCommand(Request)
getSizeOnDropFeedback(CreateRequest)
getSizeOnDropFeedback()
getTargetEditPart(Request)
setListener(EditPartListener)
showLayoutTargetFeedback(Request)
showSizeOnDropFeedback(CreateRequest)
showTargetFeedback(Request) 1.26
undecorateChild(EditPart)
undecorateChildren()
getLayoutOrigin() 1.26 (same for next 2)
translateFromAbsoluteToLayoutRelative(Translatable)
translateFromLayoutRelativeToAbsolute(Translatable)

Task 3. What sections of FlyoutPaletteComposite were changed by the fix for bug 71525? [3 marks]

(1) What method(s) were changed? **Sash.layout, ButtonCanvas.init, PaletteComposite.updateState [Determine # right (/3) and # wrong; disregard mouse*() calls. Give 1 mark toward total if any methods are right and 1 or fewer are wrong]**
(2) What revision made the fix? **1.31 [1 mark]**
(3) Briefly describe the nature of the change(s). **Adding a null check to the Sash's layout method; new flag to control transfer of focus "from the button in the vertical sash title to the button in the horizontal paletteComposite title" [1 mark if substantially right, 0.5 if just mentions right keywords]**

(N.B. This is a tricky question. There is some judgment involved in disentangling the fixes for Bugs 71525 and 67907 (66589), which are both fixed by 1.31. SashDragManager.mouse*() are for 66589.)

Task 4. Why are longs used in Geometry.segmentContainsPoint? [3 marks]

In Geometry.segmentContainsPoint, why are longs used for the square distance calculation?
(1) Identify the revision where they were first introduced. **1.8.2.1 [1 mark]**
(2) Identify and describe the bug that prompted it. **313731 [1 mark] - Integer overflow which led to incorrect results from method [1 mark if substantially right, 0.5 if just mentions right keywords]**

TASK SET B **ANSWER KEY**

Task 1. Find last bug associated with null check in PaletteViewer.setActiveTool [3 marks]

What bug led to the introduction of the "editpart != null" check in the setActiveTool method of PaletteViewer?

Please specify

(1) the bug number; **270028 [1 mark]**
(2) the revision in which the bug was fixed; **1.44 [1 mark]**
(3) who committed the revision. **ahunter [1 mark]**

Task 2. Find why "pending layout request" check was added to Graph.applyLayout [3 marks]

Why was the hasPendingLayoutRequest check added to the applyLayout method of Graph? Please state

(1) which revision added the check; **1.44 [1 mark]**
(2) who committed the revision; **Ian Bull [1 mark]**
(3) the number of the bug that led to the change. **267558 [1 mark]**

Task 3. Which methods of ConstrainedLayoutEditPolicy were last modified by anyssen? (Circle all that apply.) [*Italicized answers = only comments changed, not method body*]

[Determine # right (non-comment changes, /10) and # wrong. Ignore *comment-only* changes.]

createAddCommand(ChangeBoundsRequest, EditPart, Object) 1.28, *1.29*
createAddCommand(EditPart, Object) 1.29
createChangeConstraintCommand(ChangeBoundsRequest, EditPart, Object)
createChangeConstraintCommand(EditPart, Object) 1.29
createChildEditPolicy(EditPart) 1.30
getAddCommand(Request) 1.28
getAlignChildrenCommand(AlignmentRequest) 1.28
getCommand(Request)
getConstraintFor(ChangeBoundsRequest, GraphicalEditPart) 1.28, 1.29
getConstraintFor(Request, GraphicalEditPart, Rectangle) 1.28
getConstraintFor(Point) 1.28
getConstraintFor(Rectangle)
getConstraintFor(CreateRequest) 1.28, 1.29
getConstraintForClone(GraphicalEditPart, ChangeBoundsRequest) 1.28, *1.29*
*translateToModelConstraint(Object) 1.28 (trivial /**)*
getResizeChildrenCommand(ChangeBoundsRequest) 1.28
getChangeConstraintCommand(ChangeBoundsRequest) 1.28, *1.29*
getMoveChildrenCommand(Request)

Task 4. What sections of ScaledGraphics were changed by the fix for bug 126747? [3 marks]

(1) What method(s) were changed? **drawTextLayout [1 mark]**
(2) What revision made the fix? **1.42 [1 mark]**
(3) Briefly describe the nature of the change(s). **Added a null check for variable scaled and try/finally around method call [1 mark if substantially right, 0.5 if just mentions right keywords]**

A.5.4 Follow-Up Interview Questions

1. Did you prefer having code history information integrated into the editor (as in Rationalizer) or displayed in separate views (as in Deep Intellisense)?

2. Did you find it more useful to have access to code history on a line-by-line basis or to start your code history search from code elements such as methods?

3. Which tool do you think works better for...

   (a) determining the history of a particular code element (method/class)?

   (b) determining the history of code line-by-line?

   (c) exploring code to find sections that have been affected by bugs?

   (d) filtering the code history information displayed?

4. During the tasks, did you have questions about the history of the code that these tools did not help you to answer? Are there any features you would like to add to either tool?

5. Do you frequently (once/month or more) investigate historical information about code in your work?

   (a) *If answer is yes:* [Without divulging any confidential information,] could you describe situations in your own work that required investigating code history?

6. How likely would you be to use each prototype for your own code?

7. What did you like and dislike about each tool? General comments?

Appendix B

Additional Observations from Follow-Up Interviews

B.1 History Questions from Participants' Experience

In response to follow-up question 5, six participants (P8, P9, P12, P14, P15, P16) said they frequently investigated historical information about code in their own work. Four (P4, P5, P10, P11) said they had done so frequently in a past job or programming project and one (P7) said that they did so only for "larger" projects. One participant (P13) answered this question in the negative.

Participants identified a variety of history exploration questions that arose in their own work. Three participants (P5, P8, P12) mentioned needing to reconstruct the rationale behind their own old code; P5 and P8 also mentioned finding the rationale behind code in general. P4 and P6 needed to trace the evolution of methods or pieces of code. P5 and P14 needed to identify other things that changed at the same time as a piece of code; P9 needed to be able to determine if a bug fix that worked for one piece of code could be used for another, similar piece of code. P4 and P14 needed to identify the change that caused a newly appeared bug; P9 needed to find why functionality stopped or started working. P6 needed to find team members that worked on a section of code; P7 needed to see team members relevant to a bug; and P15 needed to see what changes particular team members had made. Finally, P9 was interested in having an "overall picture" of what changes had happened to the

codebase.

B.2 Tool Preferences in Specific Situations

Two participants (P8, P12) found Rationalizer better for determining the history of code elements, ten (P4, P6, P7, P9, P10, P11, P13, P14, P15, P16) found Deep Intellisense better, and one (P5) said the answer depended on the situation. Eleven participants (P4, P5, P6, P7, P8, P11, P12, P13, P14, P15, P16) found Rationalizer better for determining the history of code line-by-line, while two (P9, P10) found Deep Intellisense better for that type of question. Seven participants (P4, P5, P8, P10, P11, P12, P13) found Rationalizer better for finding code sections affected by bugs, while six (P6, P7, P9, P14, P15, P16) found Deep Intellisense better. One participant (P4) found Rationalizer better for filtering the code information displayed, ten (P6, P8, P9, P10, P11, P12, P13, P14, P15, P16) found Deep Intellisense better, and two (P5, P7) did not have a preference either way.