Supporting Software History Exploration

Alexander W. J. Bradley Gail C. Murphy
Department of Computer Science
University of British Columbia
{awjb, murphy}@cs.ubc.ca

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 paper, we consider two user interfaces for helping a developer 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 paper, 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 lab experiment, we found that our model can help predict which interface is helpful for a particular kind of historical question. 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 either directly from or mined from software repositories.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Integrated environments; K.6.3 [**Software Management**]: Software maintenance

General Terms

Design, Human Factors

Keywords

Software repositories, integrated development environment

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 [17]. In another study, LaToza and colleagues confirmed that developers faced challenges in trying to understand the rationale behind code [21]. 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 [19].

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 [5], and graphical interfaces to it such as Mozilla Bonsai [22] and Eclipse's "show annotations" feature [9], display the last user who changed each line of code in a source file. Tools such as Hipikat [7] and Deep Intellisense [13], which is built on Bridge [24], 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 paper, we investigate how these different presentation styles affect how developers explore historical information to answer questions related to why code was implemented as it was. We introduce a model of how different kinds of software repository information, such as bugs and code check-ins, are connected and we use this model to predict questions for which one interface style has a significant advantage over the other interface style (Section 3). For example, the model can help predict which interface style 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 compared to answering the question of which bugs have affected a given method.

To investigate the usefulness of the model and to understand more about how users explore software history information, we conducted a laboratory study in which eleven participants used one of two prototypes to answer a set of historical questions (Section 4). The first prototype was a replica of the Deep Intellisense user interface style (Section 2.1). The second prototype 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

MSR'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA

code was changed, who last changed it and why it was changed (Section 2.2). The results of our study confirmed the predictions of our model and identified deficiencies in each of the 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 what we have learned about user interfaces conforming to this model to help reason about appropriate interface designs for tools that expose software history information.

This paper makes three contributions:

- it introduces a new user interface, which we call 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 historical information.

2. TWO TOOLS FOR SOFTWARE HISTORY EXPLORATION

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 [13]. 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 retrieves metadata from a Bugzilla bug 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 [13]. As the original plugin is not available for use, we created a replica of its user interface for 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 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 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 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 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 [24] 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.

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 (i.e., in the background "underneath" the editor text.) Clicking the heading a second time reduces the transparency of the column (i.e., raises it to the foreground "above" the editor text.) Clicking the heading a third time hides the column again. Figure 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 revision that affected the line (if there is a related bug for the revision, it is displayed instead of the revision.) The background of the "When?" column is coloured to indicate the age of the code (darker entries are older). In the "Who?" column, each developer is assigned a colour and these colours are used as the background colours for the column 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 in 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. 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.

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

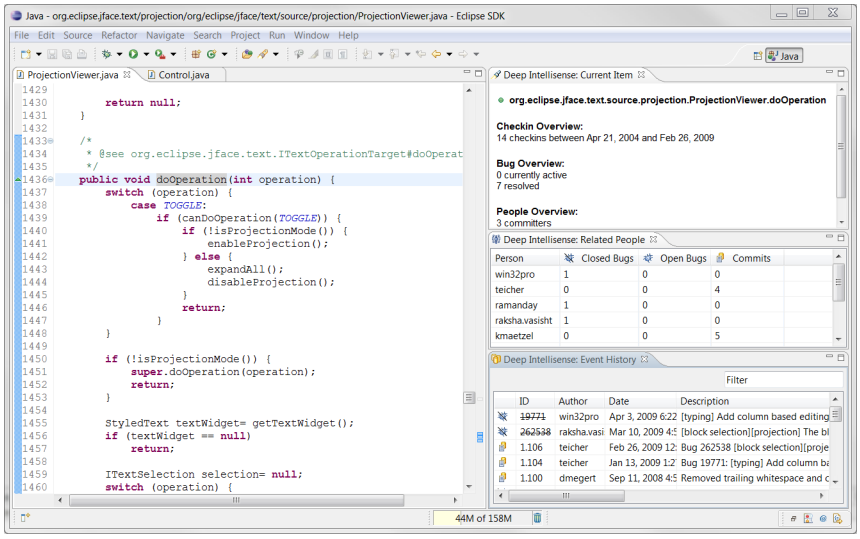

Figure 1: Deep Intellisense replica. Selecting a code element updates the three Deep Intellisense views (right) with information about related people and events.

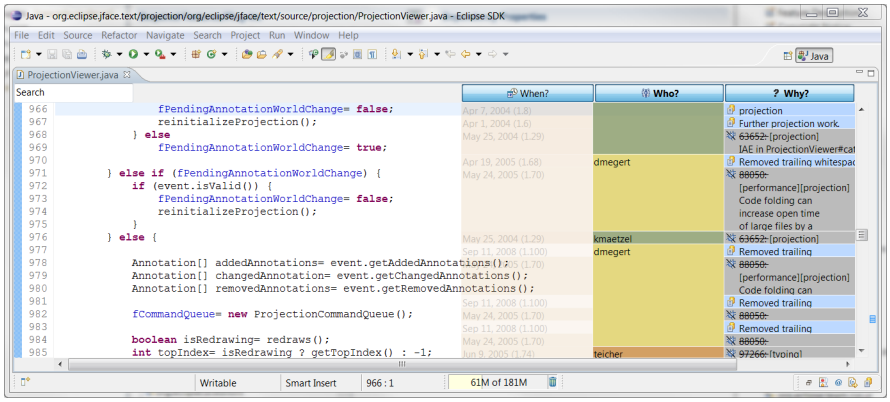

Figure 2: Rationalizer user interface. Columns can be activated by the user with low or high transparency. A text search (top left) allows filtering of information displayed in the columns.

3. MODELING HISTORY EXPLORATION

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. In Figure 3, we provide a model of this information in the form of an entity-relationship (ER) diagram [6]; this model captures how information is conceptually related, not how it is stored in any particular tool. 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-mails, webpages, documents, or Mylyn task contexts [16].

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

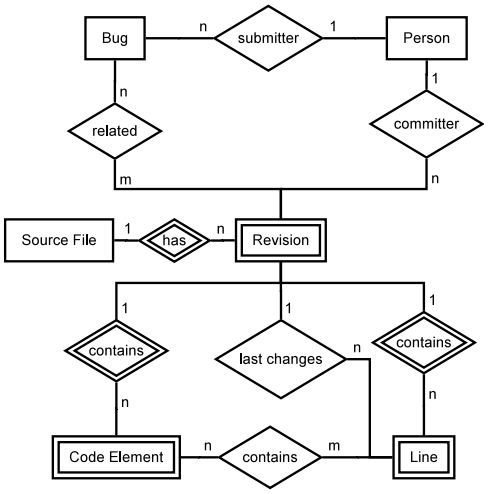

Figure 3: ER model of software history artifacts.

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 this model and these basic operations 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 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 Section 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 [8] 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 [19] 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 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, $O(E)$ 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.

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 specific design characteristics of the tools: 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 Study Design

Our study used a within-subjects design; participants were asked to answer four questions about software history with each tool. Two "question sets" (designated A and B) of four questions each were prepared for this purpose. These question sets may be found in the appendix of the first author's master's thesis [2]. Participants were randomly assigned to one of four groups, spanning the four possible orderings of tools and question sets.

Before working with a tool, participants were given a paper tutorial 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 allowed to refer back to the tutorial while working on a task. After each task, 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".

After a participant had finished both question sets, 15 minutes were allocated for a follow-up discussion, in which the interviewer (the first author of this paper) asked about participant preferences between the tools and the kinds of software history information desired by the participants.

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.) Specific questions are shown in Table 2. For each question, we recorded the total time taken, and the experimenter noted the participant's major actions, with particular attention to the operations described in Section 3. As a backup to the experimenter notes, the prototypes were instrumented to log view opens, filter applications, and code element selections. Participant answers to the questions were later scored for correctness. We expected that for the questions in Table 2, users might produce less correct answers and report lower levels of satisfaction with the tool that we predicted would be at a disadvantage.

The question sets were based on code drawn from the open source Eclipse Graphical Editing Framework (GEF) codebase[2] and bugs drawn from the associated Eclipse Bugzilla database[3]. This codebase was chosen because it was compatible with our tool and we believed it would be relatively easy to understand, even if the participant did not have an Eclipse development background.

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

[3] http://bugs.eclipse.org/bugs/, verified March 25, 2011

Table 3: Participant experience with version control and issue tracking systems.

Experience type	Experience (years)					
	< 1	1–2	3–4	5–6	6–7	≥ 10
Version control	2	4	1		2	2
Bug tracking	3	3	2	2		1

4.2 Participants

We recruited 11 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.

Participants were asked to fill out an initial online questionnaire that requested demographic data and information about their background with IDEs, version control systems, and issue tracking systems. Of our participants, ten were male and one was female; six were aged 19–29, three were aged 30–39 and two were aged 40–49. Four were undergraduate students, six were graduate students, four were commercial software developers, three were open source developers, one was a software tester, and one was a web developer. The "student" and "developer" categories were not mutually exclusive. Participants' reported experience with version control and bug tracking systems is shown in Table 3.

4.3 Results

4.3.1 Model Predictions

To validate our model predictions, we reviewed the experiment notes and logs to see how each participant answered each question in Table 2. We used the notes to form the sequence of basic operations (filter applications, view opens, and code element selections) a participant performed, and used the logs to verify operation counts. We then compared the participant's sequence of operations to those in the strategy predicted by our model (Section 3) for that question and tool. In cases where the participant seemed to have difficulty even forming a strategy and never arrived at a correct answer, as was sometimes the case, we considered the question to be as hard as predicted. We found that the strategies employed by participants were, for all question-tool combinations except one, at least as hard as our model predicted. Table 4 provides a summary of how well user strategies conformed to our predictions for the five model validation tasks. In the table, the notation "4/6" means that 4 out of 6 participants used a strategy that was as hard as our predicted strategy.

The results diverged from our expectations in two particular cases. 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 the "history to source" questions (A2 and B3), our statement that all users' strategies were at least as hard as predicted requires some qualification. For these questions, most Deep Intel-

Table 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 $\times O(r)$ 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 and Hypothesis 3.1)		
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 and Hypothesis 3.2)		
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, 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 an 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).

Table 2: Questions predicted to be harder with a particular tool.

Task	Hypothesis	Question
A1	3.1	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? (3) Who committed the change, and what was the revision ID?
A2	3.2	Which methods of LayoutEditPolicy were last modified by anyssen?
A3	3.1	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).
B1	3.1	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.
B3	3.2	Which methods of ConstrainedLayoutEditPolicy were last modified by anyssen?

Table 4: Conformity of observed strategies to predicted difficulty level.

Task	Hypothesis	Hard as predicted?	
		Deep Intellisense	Rationalizer
A1	3.1	All	All
A2	3.2	All*	All
A3	3.1	All	All
B1	3.1	All	4/6
B3	3.2	All*	All

* See discussion in Section 4.3.1.

lisense users (4 of 6 participants for question A2 and 4 of 5 participants for question B3) employed a different strategy than predicted: 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.)

We also validated the predictions of our model by comparing the user 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. Tables 5 and 6 report our comparisons of the median satisfaction ratings and correctness scores, respectively, using the Wilcoxon-Mann-Whitney test to assess significance. In both tables, n is the number of participants using a given tool for a given question, U is the Wilcoxon-Mann-Whitney U-statistic; in cases where the medians differ, the higher median is bolded, and statistically significant differences ($p < 0.05$) are highlighted in light grey. Figures 4 and 5 provide boxplot summaries of satisfaction ratings and correctness scores, respectively.

For the "deep history" questions (A1, A3, and B1), median user satisfaction was equal or higher with Deep Intellisense (significantly higher for B1, equal for the other two tasks); median correctness was also equal or higher (significantly higher for A1 and A3, equal for B1). For the "history to source" tasks (A2 and B3), median user satisfaction was higher with Rationalizer (significantly so for A2, not significantly so for B3); median correctness was also higher, but not significantly so for either question. These results are consistent with our predictions in Table 2.

4.3.2 Design Characteristics

In answer to our questions about specific design characteristics, participant opinion was almost evenly split on both characteristics. Four participants (IDs 6, 7, 13, 14) preferred the separate views

Table 5: Differences in median participant satisfaction ratings.

Task	Rationalizer		Deep Intellisense		U	p
	n	Median	n	Median		
A1	5	4	6	4		
A2	5	**5**	6	2.5	24	0.02
A3	5	3	6	3		
B1	6	3	5	**4**	43	0.01
B3	6	**4**	5	3	27	0.6

Table 6: Differences in median participant correctness scores.

Task	Rationalizer		Deep Intellisense		U	p
	n	Median	n	Median		
A1	5	0%	6	**87.5%**	50.5	0.006
A2	5	**100%**	6	60%	30	0.3
A3	5	0%	6	**75%**	51	0.002
B1	6	100%	5	100%		
B3	6	**80%**	5	60%	29	0.9

provided by Deep Intellisense, as opposed to six (IDs 4, 8, 9, 10, 11, 12) who preferred Rationalizer-style integration and one (ID 5) whose preference depended on the type of question. Five participants (IDs 4, 6, 8, 11, 12) preferred the line-by-line interface style, five (IDs 7, 9, 10, 13, 14) preferred to start their code search from code elements, and two (IDs 5, 13) said their preference depended on the type of question (participant 13, as noted, leaned towards element-based exploration.)

4.3.3 Tool Preferences

In response to the query "How likely would you be to use each prototype for your own code?", four users said they would be more likely to use Deep Intellisense (IDs 4, 6, 10, 13), four said they would be more likely to use Rationalizer (IDs 5, 8, 11, 12), and three (IDs 7, 9, 14) indicated they would be likely to use both without expressing a preference either way.

4.3.4 General Comments

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

Participants expressed appreciation for various aspects of the tools. Nine liked having a Deep Intellisense-style list of all changes to a method (IDs 4, 5, 6, 7, 8, 9, 10, 13, 14), while seven liked Rationalizer's integration with the editor (IDs 4, 5, 8, 9, 10, 11, 12). One participant (ID 14) said Deep Intellisense had a "traditional" user interface that felt more familiar to them.

A few participants showed general enthusiasm about the tools. One said that Rationalizer had a more "intelligent" user interface than Deep Intellisense (ID 12). One participant had not been aware these kind of tools existed and found them exciting (ID 7); another volunteered that they seemed "a hundred times better" than the tools that participant currently used (ID 14).

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 (IDs 4, 5, 7, 8, 10). One participant wanted to adopt the tools and customize them for their own needs (ID 10).

Participants identified various problems with the tools and made suggestions for improvement. Five wanted linkages between bugs and changesets to be more clearly indicated (IDs 5, 7, 9, 11, 14). Six participants expressed confusion with scrolling through large volumes of column information in Rationalizer (e.g., participant 7 remarked that they "could miss something") and/or desired more powerful filtering (IDs 5, 7, 9, 10, 12, 14). Similarly, one participant found Rationalizer's columns to be too "busy" and cluttered (ID 14); however, another thought the columns could fit *more* information (ID 11). Two participants stated that the user interface for navigating back to past history in Rationalizer was hard to use (IDs 5, 10). Four participants found Deep Intellisense's summary views ("current item" and "related people") to be unintuitive or not useful (IDs 4, 5, 8, 12). Finally, four participants expressed concerns about the interfaces intruding into their regular work, or the easiness of turning the interface on and off. Two of these found Deep Intellisense more intrusive (IDs 4, 8), while the other two found Rationalizer more intrusive (IDs 6, 14).

4.4 Threats to Validity

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 in the wild. 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, we do not know how representative the 11 participants in our study are 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.

5. DISCUSSION

Historical information about a software system is accessible through a structured, but non-trivial, web of information from multiple, inter-connected repositories, as shown in Figure 3. Through the study we conducted, we showed that the difficulty of performing certain kinds of software history investigation tasks with a tool depends partly upon the number of user interface operations required to answer the question in that tool. These results raise the question of what kind of user interface is best for supporting software history investigations. The answer rests in which kinds of tasks developers frequently perform. Gaining this frequency information requires more in-depth studies of developers at work.

Even before general interfaces are designed for supporting software history investigations, the results of our study provide useful input to researchers creating mining and other tools that display software history information as part of their results. Researchers can use the model we have introduced to help gauge which information is most likely to be needed by users of their tool and thus which style of interface is most likely to support the users best.

Future investigations may also consider how the contrasting user interface approaches we have studied through our two tools (breadth vs. depth, editor integration vs. separate views, line-based vs. code element-based searches) could be combined in different ways. 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.

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

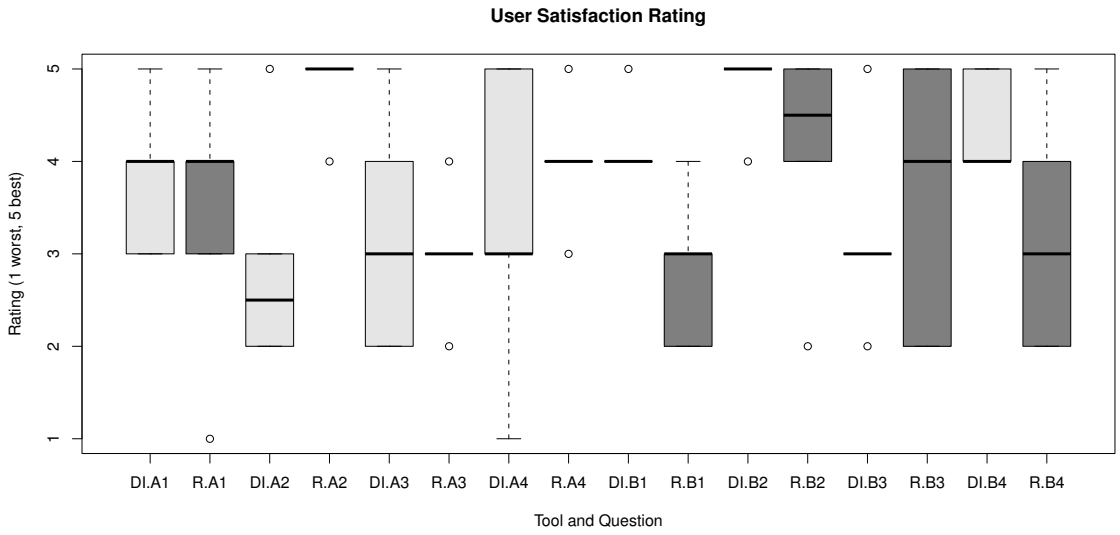

Figure 4: User satisfaction ratings.

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; for instance, they might form mistaken hypotheses about code rationale based on bugs that seem relevant at first glance, then realize their mistake and backtrack. These observations suggest that an improved model of software history exploration should attempt to take into account potential sources of user mistakes and confusion. For example, it 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.

6. RELATED WORK

A number of recent studies have considered the information needs of software developers. Sillito and colleagues [23] focused on eliciting the questions programmers ask as they work on software evolution tasks. LaToza et al. [21] found that many developer problems arose from expending effort trying to reconstruct an appropriate mental model for their code. Ko et al. [17] 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 [19] 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 paper, 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. Hipikat [7] provided an artifact linkages schema which is similar to our ER model (Figure 3), but incorporates other types of historical artifacts such as project documents and forum messages. Bridge [24] modeled software-related artifacts from multiple repositories as a directed multi-graph, and attempted to discover relationships between artifacts using textual allusions.

LaToza and Myers [20] 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. [3, 4]; 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 [14] 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 paper 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 [1] showed large codebases in a

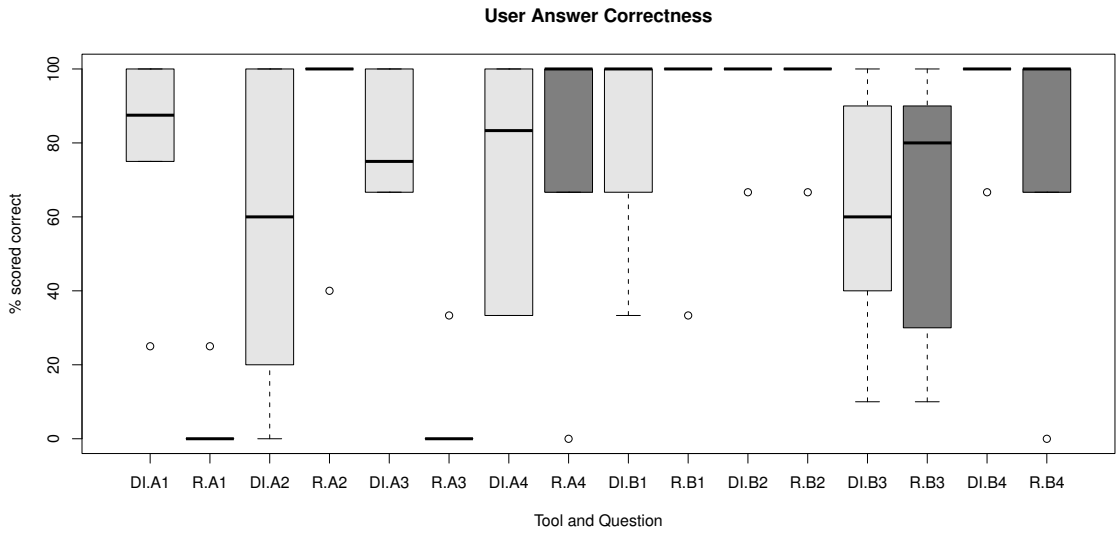

Figure 5: User answer correctness scores.

condensed graphical representation; files were shown as tall, thin rectangles wherein each code line was 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 [11] and Tarantula [15] built on this idea. Tarantula showed codebases coloured according to test suite coverage. Augur showed multiple code files coloured according to properties such as code age, and allowed users to explore changes in the code over time using a slider. Voinea and colleagues [25, 26] explored the domain of software evolution visualization extensively and 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 paper focus on helping developers answer questions about particular lines and code elements in the context of a file editor.

Grammel et al. [12] 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 [10] as an example of the more flexible kind of information composition they advocate. They believe the "conceptual workspace" of Ko et al. [18] 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.

7. SUMMARY

Developers frequently ask questions that require software history information to answer. 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 finding the best style of user interface with which to present such information. In this paper, we have introduced a model of software history information and shown how it can be used to reason about the difficulty of performing historical investigations with different styles of user interfaces. We have shown that this model can be used to predict how the difficulty of performing different types of historical exploration tasks varies between different interface styles. Further investigation will be necessary to determine the relative frequency in real-world development of the types of historical questions we identified. The results of our study can give tool developers insight into how their user interface design choices impact the efficiency of history exploration tasks.

8. ACKNOWLEDGMENTS

We thank Thomas Fritz for his comments on an earlier version of this paper. Our thanks also go to Andrew Begel, Arie van Deursen, and attendees of the June 2010 Vancouver Eclipse Demo Camp for their feedback on early versions of our tools, and to all of our study participants for their time and feedback. We thank the anonymous reviewers for their helpful comments. This work was partially supported by NSERC and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

9. REFERENCES

[1] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[2] A. W. J. Bradley. Supporting software history exploration. Master's thesis, University of British Columbia, expected April 2011.

[3] 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.

[4] 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.

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

[6] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, March 1976.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Mozilla Developer Centre. Hacking with Bonsai, May 2009. https://developer.mozilla.org/en/Hacking_with_Bonsai.

[23] 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.

[24] 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.

[25] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Softw. Engg.*, 14(3):316–340, June 2009.

[26] 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.