Deep Intellisense: A Tool for Rehydrating Evaporated Information

Reid Holmes[†]
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
rtholmes@cpsc.ucalgary.ca

Andrew Begel
Microsoft Research
One Microsoft Way
Redmond, WA, USA
andrew.begel@microsoft.com

ABSTRACT

Software engineers working in large teams on large, long-lived code-bases have trouble understanding why the source code looks the way does. Often, they answer their questions by looking at past revisions of the source code, bug reports, code checkins, mailing list messages, and other documentation. This process of inquiry can be quite inefficient, especially when the answers they seek are located in isolated repositories accessed by multiple independent investigation tools. Prior mining approaches have focused on linking various data repositories together; in this paper we investigate techniques for displaying information extracted from the repositories in a way that helps developers to build a cohesive mental model of the rationale behind the code. After interviewing several developers and testers about how they investigate source code, we created a Visual Studio plugin called *Deep Intellisense* that summarizes and displays historical information about source code. We designed Deep Intellisense to address many of the hurdles engineers face with their current techniques, and help them spend less time gathering information and more time getting their work done.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: [User-centered design]; D.2.6 [**Programming Environments**]: [Integrated environments]

Keywords

Code Investigation, Mining Repositories

General Terms

Human Factors

1. INTRODUCTION

Studies of software developers' information needs [3, 8] have shown that their most common question is "why?" Why is this piece of code behaving this way? Why was it written this way? The answers can be found in source code, development specs, bug reports, checkin messages and email archives. When developers search through these repositories, they must find the snippets of information they are hunting for and then correlate them by inferring relationships between them. For example, a checkin message might mention a bug that is resolved, or a development spec might mention method names found in source code. Often, these links are implicit — references to code are embedded in the text of bug reproduction steps, in emails on a mailing list, or in source code owned by the developer who made the most recent checkin.

Usually, correlating this information is difficult. Documentation-based sources (e.g. specifications) are not well-trusted [9]; historical information (e.g. checkin messages) contains scant details; and narrative information (e.g. bug reports and emails) is inadequately tagged with metadata linking it to the source code in question. The overhead in manually correlating information across the repositories often causes developers to abandon their own investigation and ask their colleagues for help [1–5, 7, 8]. One could say that the information has *evaporated*. In response to these developer problems, developers can use software repository data mining tools to help infer links between related development artifacts [6, 10, 11]. We use one of these (Bridge [11]) in our work to provide the data for the focus of this paper, a code investigation user interface.

To determine how to effectively display code rationale information, we undertook a user-centered investigation and interviewed industrial developers and testers to gain insight into how they perform source code investigation tasks. What data sources do they use? What facts are useful to their investigation? How do they locate information? During this interview process we talked to 3 developers and 2 testers with varying levels of industrial experience. Based on what we learned, we designed a new tool called *Deep Intellisense* that automatically presents information about related development artifacts for selected source code entities in a way that promotes a cohesive understanding of the events that caused the code to reach its current state. We validated several low-fidelity mockups of Deep Intellisense with developers and testers to gain concrete guidance on what information they wanted to see, how it should be displayed, and how they wanted to interact with it.

2. INTERVIEWS WITH ENGINEERS

Developers and testers use tools to explore source code repositories and bug databases, but often face the problem that the answers they seek are too difficult to find or construct from available materials. We interviewed engineers at Microsoft to understand how information evaporation affects them today.

[†]This research was conducted while the first author was an intern at Microsoft Research.

MSR'08, May 10-11, 2008, Leipzig, Germany.

Role	Daily Tasks
Hotfix tester	Is given a bug and a diff by a developer, as such he knows exactly what was changed and who to talk to.
Performance tester	Investigates a lot of code to figure out where to put performance tracing markers.
Developer lead	Spends 70% of his time looking at code written by other developers. Lots of SCM command-line work.
New Developer	50% of time is reading code. Lots of bug database searching, works hard to find the diffs associated with a bug.
Feature Developer	Mostly new team working on an older product. Specifics of changes are not interesting, but knowing there has been a change is.

Table 1: Roles of developers and testers we interviewed.

2.1 Methodology

We interviewed three developers and two testers who worked on Microsoft Windows, Office and Silverlight in semi-structured 60 minute interviews to elicit from them how they perform daily code investigation tasks (Section 2.2). We first asked them about their typical daily work tasks that involved programming, debugging, or testing. We then asked them to describe and demonstrate the specific tools and techniques that they used to perform these tasks. Table 1 describes the roles and daily tasks of the engineers we interviewed. In the second half of the interview, we showed them a series of low-fidelity paper mockups we had created for Deep Intellisense (Section 2.4). The two testers were given Mockups #1 and #2 (shown in Figure 1). Based on their feedback and what we learned about their work practices, we created a tester-oriented Mockup #3 (Figure 1) and showed it to all of the developers. A final mockup (#4) was created from the feedback we learned at the first developer interview, and was shown to the last two developers. (This mockup closely resembles the final Deep Intellisense interface, so we omit it from the paper for lack of space.) We interviewed each engineer only once. While the subjects commented on all aspects of each mockup, pointing out their likes and dislikes, the developers preferred the later mockups that had incorporated feedback from our earlier interviews. These mockups seemed to embody a blend of ideas that they found appealing, making them more excited to discuss its details.

2.2 Tools and Techniques

Our developers and testers employed a variety of tools and techniques to conduct code investigations. They had several tools in common: everyone used the Visual Studio IDE, a bug tracking system, an SCM system, and the Outlook email client. Several of our interviewees used a bug tracker on-screen "widget" that enumerated their open and assigned issues without having to query the bug tracker, and showed them a detailed view when an issue was selected. In concert with Visual Studio, a few developers used an alternative IDE which they used for searching through source code. Developers would regularly switch back and forth between Visual Studio and this alternative depending on their task. All of the interviewees usually accessed their SCM systems via the command line. They would issue SCM commands in series until they found a check-in they were looking for; they would then launch a graphical diff viewer. Sometimes, they would try to search for a change in the bug tracking system using the time, comments and author of the checkin as search keywords. Other times, this work flow was reversed — the developer or tester would start with a bug and work their way to the code change associated with a fixed bug.

Overall, we noticed that our interviewees frequently switched between different IDEs and tools to get their jobs done — none used any single tool for their entire job. The more experienced developers and testers always knew which tool held the information they sought, but would still have to switch between them to build a cohesive mental model.

2.3 Information Needs

During our interviews, several common information needs emerged between the developers and testers:

- *"What happened most recently?"* Both developers and testers wanted to consider the information available to them in reverse-chronological order. They felt that more recent artifacts were more likely to be relevant to their task.
- *"Who should I contact about this?"* After looking through bug and change history, if the developer decided to follow up with someone from an external team, it was important to identify the correct contact person. Who they talked to depended on the type of information they wanted — for example, it could be the person who closed the most recent bug, or created the original spec for the code.
- *"How can I filter the information available to me?"* Both developers and testers have an overabundance of accessible data. They wanted to initially filter to locate information that was relevant to the code they were investigating, and then filter further to find information that was relevant to the specific issue they were considering. Providing a flexible way to access the information they required was very important.

Two kinds of information were of specific interest to both groups:

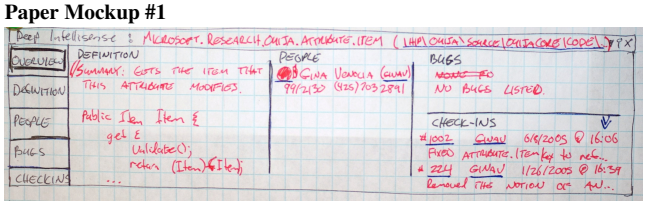
- *Inferred links.* Bugs, checkins, emails, and documents each satisfied specific information needs for our participants; however, they rarely provided all of the information they sought. The inferred relationships between these elements were crucial for gaining the understanding they required to satisfy their investigation.
- *Scoped information.* Developers and testers infrequently looked for information at the file-level; they were more interested in locating artifacts related to specific source code locations (methods, fields, etc.).

Two kinds of information were not interesting to either group:

- *Burstiness.* There was little interest in the density of changes. For example, the fact that there might have been four check-ins in two days, and then nothing until two months later was not interesting to any of the five developers or testers.
- *Relationship to product schedule.* While the strict ordering of events (this check-in before that one, or this bug opened before that check-in was made) was interesting, how these events related to the product schedule (*e.g.*, was this check-in before milestone three?) was not interesting.

We also found that developers and testers had specific needs.
Developers. Developers frequently investigate large amounts source code they did not write themselves. They often try to infer how a bug relates to a checkin (or vice versa). By relating a bug to a checkin, the developer can see both what was changed (from the diff in the checkin) and learn why it was changed (by looking at the rationale in the associated bug). This inferred link helps the developer better understand the *how* and *why* the code changed.
Testers. Testers work on tasks for specific developers. They are often assigned a bug and given a diff of the fixed source code by a specific developer. The developer is the explicit point of contact

Paper Mockup #1

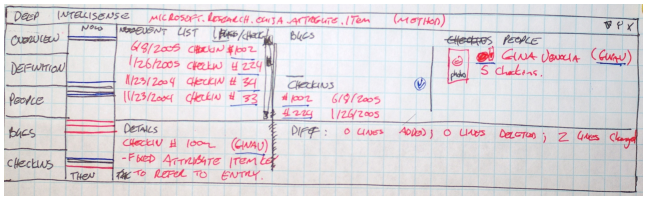


Details: Bugs and checkins are kept in separate panels in this mock-up, as this maps to how developers access these artifacts with their current tools. Only the two most recent bugs or check-ins are listed with one-click access to the full list. The people pane is populated by enumerating each person involved with the artifacts in the index.

Pros: "I like the concept of a people pane." "The abbreviated bits of information given for a checkin (change list #, person, time, and short description) seem right."

Cons: About splitting up bugs / check-ins — "Why can I only see 2 bugs or check-ins? I don't want to click to see more, they should just be there." "The additional information on the people pane is not the information I need."

Paper Mockup #2

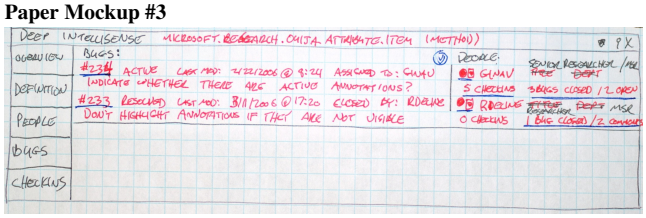


Details: This mock-up explores the concept of 'burstiness' and introduces the event list. The vertical bar provides a visualization of the time the code element has existed; the horizontal lines represent events (bugs opening and closing, checkins, etc.). Clicking on a line in the visualization would scroll the event list to reveal additional details.

Pros: "The event list looks great!" "Integration of the different kinds of events into one list works well." "The badge information in the people pane (aka, why am I listed here) is useful."

Cons: "The temporal visualization is not interesting." "The details view takes up too much space."

Paper Mockup #3



Details: This mockup resulted from our consultation with the testers. It tried to provide more information about the bugs they were looking at. If they wanted more detail, they could click on an item to populate the lower pane with a detailed view.

Pros: "I like having the position and dept. info on the people pane."

Cons: "I don't like the concept of the details pane; just take me to [our bug database], that's what I'm used to."

Figure 1: The three low-fidelity paper mockups of the Deep Intellisense user interface.

for any questions. At the same time, testers are still interested in seeing if the bug they are testing, or the associated code that fixes it, is related to any other bugs and checkins.

2.4 User Interface Mockup Evaluation

After interviewing each engineer about their work practice, we showed them a series of paper mockups of Deep Intellisense. The intent of this exercise was to focus our development effort on providing information that is relevant to developers and testers, in a format that is usable to them as they undertake their daily activities. The feedback we received on our mockups gave us insight into how the developers and testers would like to see code investigation information displayed to them. For our tool to be successful for our users, two characteristics of the user interface were going to be very important:

- *Flexibility of interaction.* The developers and testers knew what they were looking for. Their intuition drove them towards particular dates, artifacts, and keywords. Providing the ability to quickly filter the huge volume of information available to them down to those elements that they thought were most relevant should be a primary feature.

- *Interleaved event history.* Developers search for bugs, checkins, source code, and other artifacts using different tools. This makes it difficult to temporally compare each result set — the tools force them to think of each artifact set separately; however, this is not how they think of these artifacts. Our prototype user interface interleaves the artifacts by date to make as cohesive a presentation as possible.

3. DEEP INTELLISENSE

Deep Intellisense is a set of three views integrated into the Visual Studio IDE. It uses an implicit query interaction model, meaning that it automatically updates its view based on the source code element that is under the developer's cursor. Information is given for the most specific element under the cursor; for example, if the developer has clicked on a method call the view updates with information that is relevant only to the method being called. This specificity is significant as many tools operate at the file-level rather than the specific program element level; this enables developers to significantly focus their investigation using Deep Intellisense compared to file-level tools.

An overview screen shot of Deep Intellisense is given in Figure 2. The three views were chosen to provide all of the salient information about a source code element without requiring the engineer to query, type, or click anything. The views can be arranged in any manner the developer likes; this screen shot demonstrates the default horizontal orientation.

3.1 Views

Current item view. This view gives the developer a quick at-a-glance overview of the structural element currently under their cursor. The developer can see the fully-qualified name of the current item as well as an overview of the artifacts relevant to this item. The dates show when the item was first created as well as when it was last edited.

People view. People are associated with events solely as attributes on events; e.g. a person sends an email, files a bug, or submits a checkin. Deep Intellisense considers people as first-class entities;

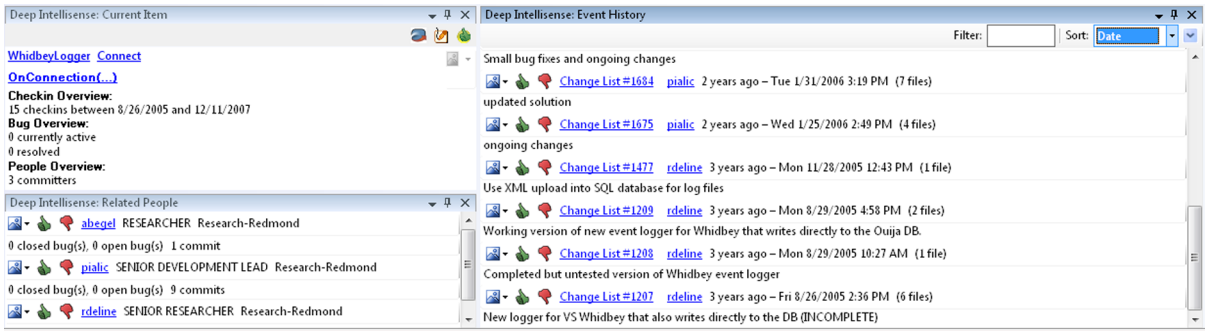


Figure 2: Deep Intellisense

the tool enumerates all of the people involved with the current item and adds them to a list. During our interviews we found that engineers were most interested in knowing the title and department of the people involved, along with their email alias and an explanation of their relationship to the current item. We include thumbs up and thumbs down buttons to filter out people (and their associated events) who are not relevant to a particular code investigation.

Event history view. This view is the heart of Deep Intellisense; it provides quick access to all of the relevant events (such as checkins, modifications to bugs, and emails) to the current item. The events are displayed in a simple, interleaved, initially chronologically-ordered list. The list can be sorted according to date, person, event kind, and if relevant, the number of files affected by the event. A text filter has also been included to allow the list to be interactively pared down. Clicking on any bugs, checkins, emails, web pages, or documents in this view opens up their respective native viewer to the correct artifact; this was specifically requested when we presented our mockups.

3.2 Mining backend

Deep Intellisense surfaces data found in isolated repositories utilized by software development teams. These data sources include source code, SCM repositories, bug report, feature request and work item databases, and emails sent to mailing lists. Each data source is mined for descriptive information, as well as links that can be made to other data items.

The Bridge [11] is the primary source of mined data for Deep Intellisense. The Bridge is a graph constructed by a series of data source crawlers, each of which is specialized for a particular kind of data. The SCM repository crawler indexes every change made to the source code. At Microsoft, bug reports, feature requests and work items are contained in a common database that we crawl in chronological order — all text found within is inserted into nodes in the Bridge graph. A series of regular expressions are run over the text in each node to look for plain-text allusions to other graph nodes. For example, the crawler would scan an email message that contains the text, "Last night I fixed bug 4567 by incrementing the counter at the end of the Account.Add method," and produce links between the graph node for bug 4567 and for any graph node labelled by the class Account and method Add.

We have built databases for seven months of Windows Vista development (4.7 million nodes, 9.6 million edges, 19% of which were derived from textual allusions), and for our own source code base (375,000 nodes, 830,000 edges, 53% of which were derived from textual allusions).

4. CONCLUSION

While developers have an abundance of historical artifacts available to them that are related to the source code they are investigating, finding the right ones and creating a cohesive view from them can be difficult using the disparate tools available to search and locate them. We created the Deep Intellisense tool to present information from various data repositories in a more effective way for developers. In developing Deep Intellisense we talked to several developers and testers, as well as referred to recent research that engaged a very diverse development community. We found that developers and testers employ a variety of techniques to determine how and why source code changes. We developed Deep Intellisense to automatically provide these developers and testers with context-sensitive code history within the Visual Studio IDE to conduct their code investigations. Deep Intellisense provides an interface that can surface links between normally disconnected artifacts such as bugs, emails, checkins, and specs. By rendering these historical artifacts in an integrated manner, Deep Intellisense aims to help developers better understand the history of the source code. We plan to evaluate our hypotheses about Deep Intellisense with a trial deployment to several industrial software development teams.

5. REFERENCES

[1] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *CACM*, 31(11):1268–1287, 1988.

[2] J. D. Herbsleb and R. E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of ICSE*, pages 85–95, 1999.

[3] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE*, pages 344–353, 2007.

[4] R. E. Kraut and L. A. Streeter. Coordination in software development. *CACM*, 38(3):69–81, 1995.

[5] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE*, pages 492–501, 2006.

[6] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM TOSEM*, 11(3):309–346, 2002.

[7] D. E. Perry, N. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.

[8] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of FSE*, pages 23–34, 2006.

[9] J. Singer. Practices of software maintenance. In *Proceedings of ICSM*, pages 139–145, 1998.

[10] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of ICSE*, pages 408–418, 2003.

[11] G. Venolia. Textual alusions to artifacts in software-related repositories. In *Proceedings of MSR*, pages 151–154, 2006.