DO DEVELOPERS RESPOND TO CODE STABILITY WARNINGS?

by

Sylvie L. Foss

B.SEng., The University of Victoria, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2015

© Sylvie L. Foss, 2015
Abstract

Ideally, developers would always release code without bugs. Given the impossibility of achieving this ideal, there has been growing interest in ways to alert a developer earlier in the development process to code that may be more bug prone. A recent study found Google developers were unsure of how to act on file-level bug prediction information provided during code reviews as developers were confused about how files were flagged and how potential problems indicated by flagged files could be addressed. We hypothesize that developers may find simpler information provided earlier than code reviews easier to act upon. We introduce a plugin we built called ChangeMarkup that indicates code age and commit sizes via per-line markers in the editor. To understand if this approach has value, we performed a field study with five industry participants working on JavaScript code; our rationale was that warnings might be of more use to developers working in a dynamic language. We found that participants were interested in whether code is recent but do not care precisely how recent and that participants are generally unwilling to change their work habits in response to code stability warnings, regardless of the indicated risk level of performing an edit. Reasons for this reluctance were limited choice as to which edits must be performed and how, a reliance on resilient company release procedures such as dark launching, and confidence in their own work. Based on participant feedback, we propose future adaptations of ChangeMarkup such as an adaptive plugin that anticipates developer activities and presents information accordingly, and a version further simplified to mark only the most recently committed code.
Preface

This thesis and the research material presented therein, including the ChangeMarkup plugin, its evaluation, and field study results analysis, are the original and independent work of the author, performed under supervision of Dr. Gail Murphy. The field study of the ChangeMarkup plugin was conducted by the author in adherence to ethical practices as approved by the administrator of the UBC Research Ethics Board, with certificate number H14-01797.
# Table of Contents

Abstract................................................................................................................. ii  
Preface................................................................................................................... iii  
Table of Contents ................................................................................................. iv  
List of Tables ........................................................................................................ v  
List of Figures ......................................................................................................... vi  
Acknowledgements ................................................................................................ vii  
Dedication ............................................................................................................... viii  
Chapter 1: Introduction ....................................................................................... 1  
Chapter 2: Related Work .................................................................................... 4  
Chapter 3: Characteristics of JavaScript Program Evolution ............................... 7  
  3.1  Evolution........................................................................................................ 7  
  3.2  Fix-on-fix Validation .................................................................................... 14  
  3.3  Threats to Validity ....................................................................................... 19  
Chapter 4: The ChangeMarkup Plugin ................................................................. 20  
  4.1  Implementation ............................................................................................ 23  
Chapter 5: Evaluation ........................................................................................... 25  
  5.1  Methodology ............................................................................................... 25  
  5.2  Participants ................................................................................................. 28  
  5.3  Results ........................................................................................................ 30  
  5.4  Analysis of Logs and Developer Communication Diaries .......................... 35  
  5.5  Threats to Validity ..................................................................................... 39  
Chapter 6: Discussion .......................................................................................... 42  
  6.1  Future Changes and Uses of ChangeMarkup ............................................ 42  
Chapter 7: Conclusion ......................................................................................... 44  
Bibliography ........................................................................................................... 45
List of Tables

Table 3.1 Current characteristics of the four open source projects we studied ................................ 9
Table 4.1 Change size warning levels ............................................................................................. 21
Table 5.1 Questions used to guide initial interviews and distributed for questionnaires .......... 26
Table 5.2 Questions used to guide final interviews .......................................................................... 27
Table 5.3 User study participants .................................................................................................. 29
Table 5.4 Average time passed between P4 events ........................................................................ 39
List of Figures

Figure 2.1 ChangeMarkup running in IntelliJ, with purple age markers and orange change size marker ................................................................. 5
Figure 3.1 Daily number of files touched for a commit, with largest value marked ............ 10
Figure 3.2 Daily number of files touched for a commit, with largest value marked ............ 11
Figure 3.3 Daily number of files touched for a commit, truncated for visibility of smaller commits ........................................................................................................ 12
Figure 3.4 Daily number of files touched for a commit, truncated for visibility of smaller commits ........................................................................................................ 13
Figure 3.5 Example project history showing fix chain detection for file fileH.js in blue ........ 15
Figure 3.6 Industry partner JavaScript files with at least one bug fixing commit ................. 17
Figure 3.7 Distribution of number of changes to industry partner files of fix chain length .... 17
Figure 3.8 Firebug JavaScript files with at least one bug fixing commit .............................. 18
Figure 3.9 Distribution of number of changes to Firebug files of fix chain length .............. 19
Figure 4.1 Commit size data for industry partner ................................................................ 22
Figure 5.1 Tooltip views and communications of three participants (P1, P4, and P6) ........ 37
Acknowledgements

I would like to express my deepest gratitude to my supervisor Gail Murphy for her excellent advice, insight, and encouragement over the course of my studies. I would also like to thank Ivan Beschastnikh for reviewing my work and providing helpful comments.

I am grateful for the support I have received from my inspiring colleagues, and for the useful suggestions I received from Marc Palyart while I developed my plugin.

Special thanks to my study participants for volunteering their time and providing me with thoughtful and honest responses. This research would not have been possible without them.

I would also like to thank NSERC for providing funding for this research.
To my parents, for their encouragement, love, and support
Chapter 1: Introduction

The longer it takes to discover a bug, typically the greater the cost of fixing it [1, 2]. Ideally, if a developer on a project could be made aware of bugs as code is being written, he or she could resolve the bugs before code goes through extensive quality procedures or is released. Predicting where bugs might exist and focusing attention on those places early has been the focus of much recent research [4, 5, 8, 9, 11, 16, 17]. Although bug prediction algorithms, such as Rahman’s [8, 9, 11, 17], have been shown to have reasonable accuracy in retrospective studies, it has been difficult to have developers act on bug prediction information. A study of bug prediction in use at Google showed that the provision of comments about predicted bugs in code reviews did not change developer behaviour, largely because the information was not actionable and because many bug prediction warnings that were similar were attached to a variety of files, including auto-generated files [11].

In this thesis, we investigate whether more direct metrics about code age and change size, which we refer to as code stability, presented within a developer’s editor might be more amenable to developer attention and might be more actionable. Our investigation uses ChangeMarkup, a plugin we developed for the IntelliJ IDEA\(^1\) Integrated Development Environment (IDE) and PhpStorm\(^2\) IDE that presents code stability warnings as per-line colored markers in editor being used by a developer. ChangeMarkup provides two types of code stability markers: one to represent code age and the other change size. Code age is the number of commits that have passed since the given code was last modified. Change size is the number of added and modified lines that are currently uncommitted. We picked these two metrics as prior research indicates that the larger and more recent a change, the less stable it is [4, 5, 10, 13-16] and that more unstable code leads to more bugs, as can be evidenced by fix-on-fixes, where changes to resolve one bug lead to subsequent bugs [16].

---

\(^1\) [http://www.jetbrains.com/idea](http://www.jetbrains.com/idea), verified 2015/04/07
\(^2\) [http://www.jetbrains.com/phpstorm](http://www.jetbrains.com/phpstorm), verified 2015/04/10
We evaluated ChangeMarkup through a field study focused on two research questions:

**RQ1**
Do developers who are shown information that could potentially help avoid the introduction of bugs behave differently than without that information?

**RQ2**
How do developers notice and interact with new editor markers about potential bugs?

Five industrial developers used ChangeMarkup as part of their regular work for two weeks. We gathered data from several sources during this study: an initial interview, periodic questionnaires, a final interview, plugin logs and developer diaries. Analysis of this data resulted in three observations that pertain to RQ1 and one that pertains to RQ2.

In terms of developer behaviour (RQ1), we found:

Observation 1: Only binary indicators of age are important to developers

Observation 2: A number of code characteristics lead to developer precaution but change size warnings do not

Observation 3: Developers rarely discuss code outside of code reviews

In terms of presenting code stability information in the editor (RQ2), we found:

Observation 4: Developers are drawn to unfamiliar visual features, but may overlook drilldown features (i.e., detailed tooltip descriptions of the stability warnings)

These results suggest that code stability information might be better placed in other tools used during software development, such as code review tools, and that adaptive UI features that only show the information when it is pertinent to a developer may lead to more consideration of the information. These results can help inform toolsmiths of future similar tools.

This thesis makes three contributions:

- it shows that fix-on-fixes, a phenomena suggesting that unstable code leads to more bugs, occur at a nontrivial frequency in two JavaScript systems (21% in an industrial project,
and 52% in an open source project), confirming the nontrivial amount reported on the much older 5ESS system [16],

- it introduces ChangeMarkup, a plugin that presents code age and change size information to developers in the editor they normally use, and

- it describes how industrial developers perceive code stability information in terms of code age and change size based on use in their daily work.

The remainder of this thesis is organized as follows. Section II outlines prior research related to bug prediction, code stability and software visualization. In Section III, we validate that the fix-on-fix phenomenon occurs in newer code bases written in a dynamic language. Section IV describes the ChangeMarkup plugin and its implementation. Section V details our evaluation of ChangeMarkup. Section VI is the discussion. We conclude the thesis in Section VII.
Chapter 2: Related Work

A number of techniques have been proposed in recent years to predict where bugs may occur in code and to warn developers of those locations. Kim et al. developed and evaluated two algorithm variations, BugCache and FixCache, that predict the occurrence of software faults by caching the locations connected to known faults at the time they are fixed [9]. Their work is based on the idea that faults occur in temporal bursts. FixCache predicted faults with 73-95% accuracy at file level granularity. FixCache was further analyzed by Rahman et al. with the different goal of determining its usefulness in code inspections [17]. They discovered that though FixCache presented inspectors with larger files, these files had high bug density. They also developed a naïve prediction model that simply ranks files based on how many closed bugs they have, and gathers enough top files to total 20% of the entire project. This simple algorithm was found to be nearly as successful as FixCache when used for code inspections. Kim et al. also developed a successful file level bug prediction algorithm based on machine learning classification that was on average 78% successful at predicting buggy changes [8].

To determine whether these algorithms help developers, Lewis and colleagues performed an industry study to determine whether developers at Google came up with similar predictions of the bug proneness of files as did three algorithms: the naïve algorithm developed by Rahman et al. and two variations of FixCache [11]. Based on developer feedback following this process, they then implemented a modified version of the naïve algorithm and included it in the interface of Mondrian, the peer code review tool used at Google. The researchers had developers use this system for three months to determine if their behavior changed as a result, but it mostly did not. A main user complaint about this system was that the flags given to bug prone files did not come with any suggested action items to resolve the flag; the nature of the algorithm means it is not possible to do so immediately, but rather the overall technical debt of the file must be reduced. Lewis and colleagues surmised that the predominately negative feedback was “not … a failure of developer-focused bug prediction as a whole, but largely … a failure of [the algorithm].”
Our intent in creating ChangeMarkup was to understand if more straightforward information about the code on which a developer is working might be more likely to impact developer behavior. Our use of change size and code age as basic stability metrics within ChangeMarkup is supported by a substantial amount of available research into code stability. Such research has determined that newer, larger changes are more defect prone than smaller changes or code that has not been changed in a long time. A key implication that Eick et al. noted from evaluation of one of their bug prediction models was that the larger and more recent a change, the more likely it is to contain faults [4]. Graves et al. found that old code is stable, either having been tested and deemed stable or having had ample time to be fixed [5]. Mens and Demeyer consider the other direction, finding that newer code is more defect prone [13]. They state that if code has recently been under frequent modification, it is likely to continue being edited and thus will have more opportunities to become defective. Mockus and Weiss [14] and Leszak et al. [10] found that more, larger changes correlate with more defects. Eick et al. [4], Purushothaman and Perry [16], and Graves et al. [5] found that change size is a significant predictor of faults. Eick et al. found this to be a much better predictor than code complexity [4].

Based on these findings, in our plugin ChangeMarkup, we present code age and change size to the developer using editor markers with varying saliency depending on the stability implications of those metrics; code that is newer or part of a large change is shown as more salient for a stronger warning of potential instability. Figure 2.1 shows ChangeMarkup markers in the editor gutter, and related tooltips.

Figure 2.1 ChangeMarkup running in IntelliJ, with purple age markers and orange change size marker
There are three existing plugins that have influenced the design of ChangeMarkup. Code Orb visualizes code volatility warnings from data such as code churn. These warnings are presented via a pie chart occupying a dedicated lower panel of the IDE, color coded by warning level, and color corresponding per-line gutter markers [12]. The goal of Code Orb was to warn developers of bug prone areas of code. Cook et al. present a code age editor based on Caise, their framework for concurrent development, which highlights components based on age [3]. CoderChrome is an Eclipse plugin that showcases many possibilities for color-coded visualization within the IDE, including colored markers in the editor gutter [7]. Code Orb and Caise were both intended to inform developers, but their respective authors do not report on user studies of these tools. CoderChrome received positive feedback from a small user group including student colleagues from the authors’ university, but further study would be necessary to determine whether there truly is potential for acceptance of developer centric, IDE integrated color coding. We aim to help fill this gap by evaluating ChangeMarkup in use by practicing developers.

Design of ChangeMarkup was also guided by the findings of Sensalire and Ogao, who presented industry software developers with three code visualization tools and elicited feedback. Their key findings were: developers dislike switching contexts, so IDE integration is preferred; unresponsive tools will be considered not useful and abandoned; there must be minimal user effort required in order to generate the visualizations; data must be displayed in a simple, clear manner; and users must be able to manipulate any visualization as needed. This developer desire for simplicity supports the color based interfaces of Code Orb and CoderChrome, and so was another factor in our decision to represent ChangeMarkup data in the form of plain colored shapes. The hover or click functionality of ChangeMarkup markers to produce further details is also in line with the need for ease of manipulation.
Chapter 3: Characteristics of JavaScript Program Evolution

Our argument for displaying code stability in terms of change size and code age is based in research conducted on older, stable languages, such as C. The use of newer languages, like Javascript, that are more dynamic, may result in applications with different evolution characteristics for which different code stability metrics may apply. We focused on providing tool support for JavaScript because many of the analysis tools that help developers to produce stable static code are unavailable to the dynamic code developer, and Javascript is one of the most popular dynamic languages today. To investigate whether the research conducted about applications built in more stable languages carries over to newer applications written in JavaScript, we conducted two investigations. First, we analyzed the evolution of four open source systems, comparing the characteristics of two written predominantly in more traditional languages and two in JavaScript. Second, we compared fix-on-fixes, referring to the number of times a bug fix leads to another bug, in our industrial partner’s JavaScript application with rates reported in the literature for more traditional languages.

3.1 Evolution

To determine if there are similar evolution patterns between applications written in traditional languages and JavaScript, we selected four open source systems to study. To ensure we were studying evolution of realistic systems with enough data to minimize effects of unique developer styles, we selected large scale projects with at least two years of history and more than 25 developers. For accessibility of rich and relatively uniform data, we limited our choices to open source projects maintained under Git version control. Table 3.1 summarizes current characteristics of the four open source projects. We obtained the primary language percentages, commits, and numbers of developers from GitHub\(^3\), and the project sizes from Black Duck Open HUB\(^4\). Etherpad Lite is an example of a dynamic application, written in JavaScript, that has now

\(^3\) [https://github.com/](https://github.com/), verified 2015/04/07

\(^4\) [https://www.openhub.net/](https://www.openhub.net/), verified 2015/04/07
been evolving for 4 years and currently has 135 developers. Firebug is another predominately JavaScript project that has was started over 7 years ago and now has 31 active developers. The GIMP graphics editor is an example of a traditional application written largely in C that has been evolving for 18 years, currently with 174 current developers. Hibernate ORM is the core functionality of Hibernate, comprised almost entirely of Java code, and has been developed for nearly 8 years. It now has 146 developers. These project statistics are current but the range of project history we studied varies for each project. Table 3.1 lists the project history ranges over which we performed our analyses.

To compare the evolution of these systems, we computed a number of metrics. Figure 3.1 and Figure 3.2 show the daily commit size of two traditional and two dynamic projects respectively, where commit size is the total number of files touched for a commit. A file can be counted more than once per day if it was part of multiple commits on a given day. Figure 3.3 and Figure 3.4 show the same respective data, with the upper bound reduced so that smaller commit sizes can be more easily viewed. Values extending beyond the upper bound are simply truncated in these figures.

Our comparison of these metrics shows some general trends in traditional versus dynamic applications. Figure 3.1 and Figure 3.2 show that the daily number of committed files is generally much larger for traditional applications, with their record largest commits being an order of magnitude larger than those of dynamic applications. The truncated representations of this data, Figure 3.3 and Figure 3.4, also show this discrepancy in commit size; the truncation bounds required for visibility of small commits varies depending on whether the project is dynamic or traditional. This difference in bounds reflects the significantly larger size we observed in commits of traditional applications.

We also observed that in dynamic applications, most defects occur in the presentation layer as opposed to traditional applications, in which most defects occur in the logic layer [7]. We looked for trends in release frequency but did not notice any patterns specific to either dynamic or traditional applications.
<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>Type</th>
<th>Primary language</th>
<th>History studied</th>
<th>Commits</th>
<th>Size (LOC)</th>
<th>Developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etherpad</td>
<td>Collaborative document editor</td>
<td>Dynamic</td>
<td>92% JavaScript</td>
<td>Mar. 26, 2011 to Sep. 21, 2013</td>
<td>4,247</td>
<td>44.7K</td>
<td>135</td>
</tr>
<tr>
<td>Lite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firebug</td>
<td>Web browser development extension</td>
<td>Dynamic</td>
<td>84% JavaScript</td>
<td>Aug. 15, 2007 to Sep. 15, 2013</td>
<td>12,139</td>
<td>486K</td>
<td>31</td>
</tr>
<tr>
<td>GIMP</td>
<td>Graphics editor</td>
<td>Traditional</td>
<td>97.5% C</td>
<td>Jan. 1, 1997 to Sep. 22, 2013</td>
<td>35,071</td>
<td>730K</td>
<td>174</td>
</tr>
<tr>
<td>Hibernate ORM</td>
<td>Object-relational manager, component of Hibernate</td>
<td>Traditional</td>
<td>99.5% Java</td>
<td>Jun. 29, 2007 to Sep. 22, 2013</td>
<td>5,804</td>
<td>Unknown; Hibernate is 1.13M</td>
<td>146</td>
</tr>
</tbody>
</table>

Table 3.1 Current characteristics of the four open source projects we studied
Figure 3.1 Daily number of files touched for a commit, with largest value marked
Figure 3.2 Daily number of files touched for a commit, with largest value marked
Figure 3.3 Daily number of files touched for a commit, truncated for visibility of smaller commits
Figure 3.4 Daily number of files touched for a commit, truncated for visibility of smaller commits
3.2 Fix-on-fix Validation

Our approach builds on earlier research that suggests recently changed code of larger size is more prone to future bugs. In particular, Purushothaman and Perry’s study of fifteen years of history of Lucent Technologies’ 5ESS telephone switch showed that 40% of changes for a bug fix introduced errors themselves [16]. Before building a plugin that assumes code stability is important in a dynamic language environment, we wanted to understand if the fix-on-fix characteristics of the open source Firebug and the industrial, dynamic language codebase of our partner are similar to the fix-on-fix characteristics of the Purushothaman and Perry’s earlier work.

A fix-on-fix is a commit performed to resolve bugs introduced by a previous commit. Since fix-on-fixes cannot be detected with certainty, we used a particular sequence of commit types to be our indicator of a possible fix-on-fix. The commit messages of our industry partner consistently designate a commit as being part of a fix or an enhancement and include the ID of the addressed issue. We used this data when examining the commit history of each file, searching for fix chains: at least two consecutive commits, of the given file, for fixes in which the issue number varies. For example, a fix chain that is two commits long suggests that the first commit contained at least one bug, meaning one fix-on-fix. Likewise, a chain that is eight commits long suggests seven buggy commits and seven corresponding fixes. We omit committed enhancements from our fix-on-fix detection as these typically are not intended to address bugs. We looked for a varying issue ID in the commit message because one or more consecutive commits for the same issue number might mean the developer was performing a single fix incrementally, with commits serving as personal checkpoints. Figure 3.5 shows examples of fix chain detection for a file within the project commit history.
Figure 3.5 Example project history showing fix chain detection for file fileH.js in blue
We analyzed the repository data of our industry partner’s main project, filtered to consider JavaScript files only, in search of fix chains. The available data spanned exactly two years of project history from 27 January 2012, the date of their transition to the JIRA issue tracker which told us whether each commit was for a fix or an enhancement, until 27 February 2014, the day we performed our analysis. Our industry partner consistently notes the addressed issue number in every commit message and so we looked up these numbers in JIRA to determine if each was for a fix or an enhancement. We also searched over six years of Firebug history, spanning from September 8, 2007 until February 23, 2014, for fix chains. To differentiate between fixes and enhancements for Firebug, we parsed the commit messages in search of the case insensitive term “issue”, present in some but not all commit messages, which is followed by the number of the issue addressed by the commit. We then looked up these numbers against the online Firebug issue tracker\(^5\) to determine whether each referred to a fix or an enhancement.

Figure 3.6 and Figure 3.8 show the fix chains determined from our partner’s data and from Firebug, respectively, and Figure 3.7 and Figure 3.9 show statistics on the number of times the files with those fix chain lengths were changed. The 355 detected fix-on-fixes of our partner account for approximately 21% of the total 1665 commits for a fix in the repository. For Firebug, we found that approximately 52% of Firebug fixes introduced at least one defect; we detected 1345 fix-on-fixes out of a total 2603 commits for a fix. These findings indicate that the fix-on-fix phenomenon observed by Purushothaman and Perry is still prevalent in modern systems. The 19% discrepancy between their 40% finding and the 21% we calculated from our partner could be attributed to differences between the JavaScript files we studied, the C and C++ of 5ESS, and the differing ages of the systems. The 52% for Firebug is relatively large, and this could be attributable to the project being open source. One possible hypothesis is that since Firebug is for Firefox, a major web browser, and free and open source, there could be a larger userbase to notice and report bugs than for the other two projects we studied.

\(^5\) Hosted at [http://code.google.com/p/firebug](http://code.google.com/p/firebug) at the time we conducted the study
Figure 3.6 Industry partner JavaScript files with at least one bug fixing commit

Figure 3.7 Distribution of number of changes to industry partner files of fix chain length
Figure 3.8 Firebug JavaScript files with at least one bug fixing commit
Figure 3.9 Distribution of number of changes to Firebug files of fix chain length

3.3 Threats to Validity

We observed that traditional projects have larger commits than dynamic projects. Though the discrepancies are significant, it is possible that they can be attributed to content that is not code, such as images or XML files, or licensing information and other comments within code files. We also observed that JavaScript code in dynamic projects seems to get left behind when other content is deleted, but again this may be attributable to deletion of content other than code.

We determined whether a commit was for a fix by parsing commit messages for issue numbers which we then cross referenced against the projects’ issue trackers. Our industry partner attempts to cite the relevant issue number consistently in every commit but such does not seem to be the case for Firebug; some Firebug commits do not cite any issue number. This could have affected our fix-on-fix study.
Chapter 4: The ChangeMarkup Plugin

To determine whether developers respond to code stability warnings, we developed the ChangeMarkup plugin to present developers with such warnings directly in their editor. We developed our plugin for two popular JetBrains IDEs, IntelliJ IDEA and PhpStorm, as these are the editors of choice at our industry partner where we performed our field study of ChangeMarkup. As Figure 2.1 shows, when in use, ChangeMarkup places colored markers in the left and right gutters. The markers in the left gutter represent code age and change size per line; the right gutter is an overview of the entire file, showing all markers in the file, scaled to fit the window. This overview is a feature of IntelliJ. Each marker can be clicked or hovered for an explanatory tooltip.

We define code age of a line as the number of project commits that have occurred since that line was most recently committed, including the most recent commit of that line. For example, a line that was part of the most recent project commit is considered one commit old. If it is not included in the next commit, it will become two commits old. We decided to mark code as old as ten commits as it becomes difficult to visually discern too many age levels; lines more than ten commits old are not marked partly for this reason but mainly because our intention is to draw the user’s attention to recent code only. To lower the learning curve associated with using ChangeMarkup and help developers to quickly and accurately associate marker meanings with colors, we used only one color per metric. Therefore to represent a range of ages, we use multiple shades of a single color in order of salience such that the most recent code is marked most saliently and the marker color for ten-commit-old code is only slightly discernable from the editor gutter.

We also show the change size of uncommitted code. This change size information is effectively the code age data of any uncommitted code on which a developer is working, augmented with additional details about the size of the entire change to which it currently contributes. For example, an uncommitted line of code that has been edited as part of a large change will be marked saliently to warn the developer that it contributes to a large, and therefore potentially
unstable, change. We focused on the size of the uncommitted change because developers would be unable to change the size of already committed code and a given line might be part of multiple commits.

We use three coarse granularity warning levels to show change size rather than precise numbers; for example, “low” rather than “3% likelihood of containing a defect”. The determination of the change size warning levels for ChangeMarkup was driven partly by the work of Purushothaman and Perry [16]. In their study of 5ESS, they charted the trends they saw in the number of lines modified or added versus the percentage of those changes that introduced one or more errors into the system. As a result, we chose to compute change size in terms of added and modified lines and based the bounds of our defect warning levels on their numbers. We could not use their exact numbers since, as outlined in Section 3.2, their system varies significantly from that of our industry partner. To ensure the bounds we chose would be relevant to the system we studied, we analyzed the commit sizes of our industry partner. Figure 4.1 shows that approximately 50% of changes are 15 lines or fewer and approximately 75% are 58 lines or fewer, while only 3% of commits are a single line. We used this information to implement our warning level ranges as listed in Table 4.1. We omit deletions from ChangeMarkup because Purushothaman and Perry’s analysis “did not produce any credible evidence that deletion of less than 10 lines of code resulted in errors,” [16] and also because they cannot be marked in the editor gutter in a style consistent with the rest of ChangeMarkup, since they are an absence of lines.

<table>
<thead>
<tr>
<th>Change type</th>
<th>Number of lines changed for warning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>Addition</td>
<td>0-14</td>
</tr>
<tr>
<td>Modification</td>
<td>0-9</td>
</tr>
</tbody>
</table>

Table 4.1 Change size warning levels
Purushothaman and Perry found that the potential for a given change to introduce errors varies depending on whether it is a modification or an addition [16]. For this reason, we chose to maintain two separate calculations of change size. One is the total number of lines modified and the other the number of lines added. We present both to the user via a single range of marker colors as both are indications of stability based on change size and it is stability that we wish to present to developers, not the type of change. The latter would be a distraction if simultaneously presented but since change type helps to determine warning level associated with a line of code, this information is accessible via a simple hover or click interaction with a marker. We make this information available to the user in the interest of transparency of our metrics. The screenshot in Figure 2.1 shows examples of tooltips corresponding to these user actions.

Figure 4.1 Commit size data for industry partner
Markers in our plugin are ten shades of purple to represent the range in ages of the ten most recent commits and three shades of orange corresponding to the three levels of change size warning. We chose orange for change size markers as their purpose is to warn developers of potential for code instability; orange is the closest color to red and therefore most suitable to use for warnings without conflicting with the red of IntelliJ’s error indicators. We chose purple for age markers because it is dark enough to display in varying shades, not easily confused with orange even by colorblind viewers [6], and does not conflict with any IntelliJ markers used by our participants. We take advantage of varying saliency of the orange and purple marker colors to draw the user’s eye to stronger instability warnings. The most recently committed lines of code are marked by the most salient shade of purple, while older lines are increasingly pale, approaching the color of the editor gutter. This gives the effect of age markers fading away as the code ages. Similarly, the most saturated shade of orange marks those lines that make up the largest and therefore most unstable changes, while small changes are marked by less salient, more yellow markers.

4.1 Implementation

ChangeMarkup makes use of the IntelliJ OpenAPI⁶ and Git4Idea⁷ libraries and is currently configured to be used with the Git version control system. ChangeMarkup works for all project files regardless of language, but was tested primarily with JavaScript and Java files. ChangeMarkup maintains a log of interactions with the markers, capturing hovers and clicks and any corresponding details presented to the user.

The current implementation of ChangeMarkup, using the version control functions from the IntelliJ API, has two potential problems that can arise regarding commit date. The history of a merged file begins at the date and time of the merge as if the file was newly placed under version control at that time; any history prior to the merge is lost and all lines are recorded as new additions. Similarly, renamed files take on the date and time of the rename and appear to be newly added at that time. These factors affect the age markers of ChangeMarkup. For example, if

---

⁶ http://confluence.jetbrains.com/display/IDEADEV/PluginDevelopment, verified 2015/04/07
⁷ https://plugins.jetbrains.com/plugin/3033, verified 2015/04/07
a file that has been on a merge branch for more than ten commits gets merged, all its lines will now be marked as being one commit old. However, retaining accurate ages of each line in the event of merging or renaming a file might not be any more informative in practice since there are no available data regarding the semantics of branches, merging, or renaming. Treating a merged or renamed file as newly committed code may be the best option as in such scenarios the user is informed that some major event occurred involving that file.
Chapter 5: Evaluation

Our overall research goals are to determine the potential for developer adoption of any of the significant number of bug prediction algorithms developed and proven in academia so far, and how best to present code information to developers such that their interest is captured and retained. Our hope is that achievement of these two goals will reduce the number of bugs introduced into a system during implementation. To gauge industry adoption potential of code metrics and gain some insight about how to present them, and to build on the work of Lewis and colleagues by presenting our metrics to developers during code development rather than at the code review stage [11], we performed a small field study of ChangeMarkup in industry. Our study was predominately qualitative as tool adoption depends largely on conscious choices and sentiments of the user, data that is most easily captured via conversation.

For our field study, we initially recruited six developers at an industrial partner. The partner is a company of more than 700 employees, developing primarily in JavaScript. To recruit subjects, we had an employee at the company post a call for participants internally. The call requested participants who use IntelliJ to work on a predominately JavaScript project under Git version control. Six developers responded to this call and became our study participants. Our aim with this study was to gain knowledge about the following two research questions:

RQ1 Do developers who are shown information that could potentially help avoid the introduction of bugs behave differently than without that information?

RQ2 How do developers notice and interact with new editor markers about potential bugs?

5.1 Methodology

In the first part of the study, we interviewed each participant individually for thirty minutes using semi-structured interview format. Table 5.1 and Table 5.2 detail the questions used to guide the interview sessions. Within five weeks of these initial interviews, each participant had installed the ChangeMarkup plugin, agreeing to use the plugin for two weeks.
1. What is your process when editing unfamiliar code (e.g. a file that has been edited by somebody else since you last worked on it)? For example, you may just read and trace through it, review related revision logs, analyze it with a tool or plugin, ask people, a combination of these approaches, or others.

2. When you are working on some code, do you talk to other developers?
   a. How do you decide who to talk to?
   b. Do you tend to talk to them before, during, or after completing your edit?

3. Do you ever take special care when editing certain code segments over others?
   a. If so, what prompts you to do this? Some examples are because you’re new to the component, it looks more complex, some tool or plugin has given you certain information, or other developers have told you it is complicated.

4. Are you using any plugins that help you understand the code’s history (e.g. blame) or what it does? Which ones?

**Table 5.1 Questions used to guide initial interviews and distributed for questionnaires**
1. When you were about to edit code, did you ever check first for colored markers?

2. About how often did you read the marker mouseover tooltips?

3. Did the presence of colored markers make you think twice before editing the related piece of code?
   a. Was it the orange (defect) markers, purple (code age) markers, or both?
   b. Was their appearance and saliency alone enough to make you stop and think?
      Or did you mouse over or click to get more details before worrying about it?

4. Did the markers or their tooltip details ever cause you to deviate from your original work plan (eg: instead of jumping in and editing, you reread the code one more time)?
   a. Was there any particular type of situation where this happened more often?
   b. Did it prompt you to talk to another developer? Is this the same person you would have talked to if you were not using this plugin?

5. Which of the plugin information did you find helpful? If none, were you curious enough to keep looking at it anyway?

6. Did you ever disable either the defect markers (orange) or the age markers (purple) via the gutter menu?

7. If you used other plugins to help you understand the code’s history or what it does, are you still using them? Which ones?
   a. Is the decision to continue or stop using them related to your experience with my plugin?

8. What features did you like or dislike? What would you change?

9. What features would you have liked to see?

10. Any other thoughts about this project?

| Table 5.2 Questions used to guide final interviews |

While we provided instructions for installation, we did not specifically teach participants how to use ChangeMarkup. We made this choice for two reasons. First, we were interested in observing their natural plugin usage behavior. Second, we wanted to be close to the real life scenario of not receiving an accompanying tutorial when installing a new plugin. We also did not formally
explain to participants the meaning of ChangeMarkup’s markers so as not to interfere with how they might choose to use the data.

As participants were using ChangeMarkup, the initial interview questions were distributed four times in the form of a questionnaire, approximately every two business days, such that the first questionnaire was distributed after this two day interval and the last at the end of the ChangeMarkup usage period. We used the same questionnaire to capture any changes in developer behavior as a result of using ChangeMarkup. We also asked participants to maintain a diary of their code related communications with other developers over the course of their two week usage of ChangeMarkup. These logged communications were to include, for each item, the reason for the conversation and their colleague’s position.

At the end of the study, we collected our plugin generated logs and performed individual thirty-minute final interviews in person. In these interviews, we asked participants how they used ChangeMarkup and also for general feedback, including any criticism of the plugin and suggestions for improvement.

5.2 Participants

All participants used IntelliJ platform IDEs, some using IntelliJ Ultimate Edition and others using PhpStorm. We allowed participants to use their preferred IDE color scheme, as ChangeMarkup was implemented with consideration given to both dark and light schemes. Table 5.3 summarizes our participant group. Academic software development experience is listed for only those participants with limited industry experience. We began our study with six participants but one participant (P5) withdrew from the study after the initial interview due to a heavy workload, leaving five participants to use ChangeMarkup and complete the remainder of the study. P2 used ChangeMarkup for only one day due to a busy schedule, but completed his final interview, answering all questions.
<table>
<thead>
<tr>
<th>ID</th>
<th>Employment duration</th>
<th>Company</th>
<th>Current position</th>
<th>Concurrent projects</th>
<th>JavaScript</th>
<th>Industry</th>
<th>Academic</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3 years</td>
<td>Senior software engineer</td>
<td>3</td>
<td>12 years</td>
<td>10 years</td>
<td>N/a</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>5 years</td>
<td>Engineering lead</td>
<td>1</td>
<td>9 years</td>
<td>9 years</td>
<td>N/a</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>3 months</td>
<td>Software engineer</td>
<td>1</td>
<td>More than 9 years</td>
<td>10 years</td>
<td>N/a</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>More than 1 month</td>
<td>Software engineer co-op</td>
<td>1</td>
<td>More than 1 month</td>
<td>N/a</td>
<td>2 years</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>6 months</td>
<td>Software engineer</td>
<td>1</td>
<td>6 years</td>
<td>6 years</td>
<td>N/a</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>1.5 months</td>
<td>Software developer intern</td>
<td>1</td>
<td>1.5 months</td>
<td>1 year</td>
<td>5 years</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3 User study participants
5.3 Results

We analyzed the interviews and questionnaires by iteratively reviewing the data, sorting portions of the data into emerging topic headings and grouping topics into larger headings as common themes emerged. The majority of the data are from the interviews as the questionnaires did not reveal any significant change in process over the course of the study, perhaps due to the short duration of the study. From this analysis, we noted four observations: the first three pertain to RQ1 and the last pertains to RQ2.

Observation 1: Age of code is important, but only as a binary indicator of recent versus not recent.

The majority of participants (P2, P3, P4, and P6) said that they used ChangeMarkup’s code age information to see which code was recently modified. Three of these participants (P3, P4, and P6) particularly liked being able to see the most recent commit marked in the editor. One participant in this group (P4) checked for markers before performing edits, mostly to see if someone else had recently edited the code. This participant also stated that seeing the most recent commit helps him to stay focused on the task at hand and helps him track his personal progress. Another of these participants (P3) said that if he saw an age marker, he knew somebody had worked on that code before and therefore he should be extra careful. Only one of the participants (P2) indicated an interest in specific code age as this participant found age information helpful when dealing with unfamiliar code.

Despite the fact that most participants wanted to be aware of recent code, and some wanted to know which code is part of the most recent commit, participants did not otherwise care about the precise age of recent code. Two participants (P4 and P6) did not realize during the study that multiple ages of code were represented. In response to learning about the different code ages during his final interview, one of these participants (P6) said, “It gives me no benefit to know how old [a code segment] is, especially if it’s mine. I would not notice the difference because I’m looking at other stuff, but now that you mention it I think yeah there were other colors, but I didn’t really think about it.” Two other participants (P1 and P3) said they did know that a range of code ages was shown and that they were not interested in this information.
Participants also implicitly expressed their lack of interest in specific code ages during final interviews by speaking in terms of code being old versus new. They did not mention particular ages other than to give examples, as in this comment by P3: “I guess [exact code age is] sort of useful but I’m not sure if [it is] in my case. I can see the potential but I don’t think in my case I care that much if it’s four commits old or [just] recent.”

**Observation 2:** A number of code characteristics lead to participant precaution but change size warnings do not.

Questions in the initial interview asked what code characteristics cause developers to take special care when attempting to understand or edit code. Though all participants indicated that they do take special care editing certain code segments over others, only half (P2, P3, and P4) reported that perceived code instability is a driver of this extra care. Both P2 and P4 said they take greater caution when editing code that is brittle. P2 described the situation in more detail as some code segments being older and more fragile, perhaps poorly written, or poorly tested and with inadequate test coverage. P2 feels that code written not more than one month ago, if well tested, does not require special care to edit. P3 commented in his initial interview that he is aware that recent changes have increased potential to contain bugs, and that since he prefers to work on stable code, he appreciates being able to see recent changes. Since P2 and P3 talked about code age in relation to stability, we expected their behavior to be affected by ChangeMarkup’s age markers. P3 did state in his final interview that he liked the plugin’s markers in general and was prompted by them to talk to other developers, but he did not mention stability or specify the age markers in this comment.

We expected participants to respond to our change size indicators with extra coding care because we present this metric in the form of warnings that the affected code is more likely to contain defects. However, these indicators had little to no affect on participants’ coding behavior. Two participants (P1 and P6) said they never explicitly checked for any of ChangeMarkup’s markers. Two other participants (P2 and P4) said they ignored the implications of our change size warnings because they had strong habits and as such their workflows are unlikely to change regardless of any code comprehension data presented to them. For one participant (P4), ignoring the markers was also due to a lack of confidence in the metric. He realized the change size
markers are generated solely based on the number of lines of uncommitted code and felt this is an inaccurate measure of risk for two reasons: comments are inherently free from defects, and he is confident in his own ability to produce low risk code. P1 cited other reasons for ignoring change size markers: a given coding task must be completed regardless of risk, and a developer often does not get to decide which code or file needs to be edited for a particular fix or implementation. This was said during our final interview with P1, while in her first questionnaire she also commented on risk, saying “we are dark-launching features to minimize the risk.” Also based on risk, P6 felt that warnings in general are unnecessary, stating, “If you change something and break something you’re going to see it on your dev [sic], so it doesn’t really matter. There’s no real repercussion because even if you break the whole thing, it’s only on your dev machine.” He spoke further on this point, adding that the warnings might be interesting if there were no such safety net and instead code went directly to a customer, but that integration testing and other measures render warnings negligible. He also stated other reasons for ignoring change size markers: he did not fully understand their meaning, and they vanish on commit. For both age and change size markers, he simply said that he did not make mental connections between the marker meanings and the task at hand.

Observation 3: Participants rarely discuss code outside reviews.

We anticipated that the appearance of a stability warning in the editor by the ChangeMarkup plugin might prompt a developer to ask a colleague about marked code. Only one participant (P3) reported that ChangeMarkup directly caused him to discuss code with his colleagues, stating that age markers made previously touched code more apparent, and at times, prompted him to discuss that code with another developer working on the same code. Such discussions would occur before P3 began his own work on that code. P3’s is the only case clearly indicative of ChangeMarkup affecting developer communication.

Most developers indicated a reluctance to ask colleagues questions about code unless absolutely necessary. The more familiar the developers were with the code (P1 and P5), the more reluctant the developers were to discuss code. P1 said she asks questions only “if [she] really [doesn’t] get what’s going on or if [she’s] in a hurry,” or if working on an “emergency” such as a showstopper bug requiring an immediate fix. She said she otherwise rarely asks questions as she has been at
the company long enough to be familiar with the codebase. P5 sees communication about code as a bothersome interruption to colleagues’ activities. He therefore avoids it where possible, and said, “interrupting has a big cost.” He said when faced with unfamiliar code, he does not talk to others unless it would be much quicker than studying the code independently, and then only if he knows the other developers are working on the component that contains the unfamiliar segment. He said he talks to colleagues about code approximately once per day. P5 also mentioned that there is only a small number of people who are familiar with most or all of the codebase, and so he tries not to depend on them excessively.

Participants newer to the company and therefore its codebase are more inclined to communicate than their more senior colleagues from our study, but even the most junior participants (P4 and P6) prefer to first make an independent attempt to understand the code. Despite this, P6 will sooner turn to his neighboring colleagues than read through code by himself when he feels overwhelmed by the amount of unfamiliar code he is browsing. He also does not hesitate to ask questions if he thinks doing so will save him time. However, P6 is the lone exception among a participant group that avoids asking code clarification questions.

Coding communication at the company tends to be focused not around code comprehension but on pre- and post- code reviews. A pre-code review is a discussion and feedback elicitation of how a developer plans to implement a given feature, initiated by that developer. Half of our participants (P2, P3, and P5) said they often perform pre-reviews and P2 said he encourages all developers to perform them. All participants indicated that they perform formal post-reviews.

One participant (P1) reported finding a use for ChangeMarkup during formal code post-reviews that was not anticipated. While performing a post-review with a teammate, she found age markers useful not only for locating recently edited code, but also for tracing a change viewed in GitHub back to the editor on a per-line basis. This was made possible by ChangeMarkup because all lines of code edited within recent commits are marked with a unique color.

**Observation 4:** Participants are drawn to unfamiliar visual features but may overlook drilldown features.
Participants were intrigued by the appearance of new visual features within their IDE. The majority of participants who used ChangeMarkup (P1, P2, and P6) were drawn to the markers simply out of curiosity, wanting to learn their meanings. P1 said that every time she saw an unfamiliar marker color, she thought, “What does that mean?” and wanted to investigate. She said she was always curious about any unfamiliar color.

Though unfamiliar colored markers captured participants’ interest, their initial curiosity waned as they came to learn what each color represented. P2 ignored the markers once he knew their meanings. P1 ignored them once they became too predictable; she noted that any edit causes the same purple and orange markers to show up in almost every case, resulting in her loss of interest in them and her automatic dismissal of markers from that point onward.

Even though participants described themselves as naturally curious, some had difficulty discovering some ChangeMarkup features on their own despite the plugin following the same paradigms as plugins familiar to them. For example, ChangeMarkup markers provide more detailed information when clicked, as do those of IntelliJ’s Git plugin, which also appear in the left gutter. However, the majority of participants who used ChangeMarkup (P1, P4, and P6) stated that they never clicked the markers. One of these participants (P6) was surprised to learn during his final interview that the markers can be clicked.

Participants also had trouble understanding the marker meanings or made incorrect assumptions. Those who did not click markers (P4 and P6) would have had limited information from which to draw, but some participants (P3 and P6) did not make full use of the information they did access, whether accessed via clicking or only hovering. P3 read both the hover and click information but said during final interviews, “The orange markers I didn’t really understand,” and described their detailed information as “slightly confusing”. Since he did not understand the orange markers during the study, he thought them redundant and said he therefore did not use them. He said he did not really consider their descriptions while using the plugin but upon rereading the detailed information during the final interview, he said it made sense and that he thought it was a useful marker. His perceived redundancy of the orange markers therefore indicates that he did misunderstand the information. In contrast, two participants (P2 and P4) indicated during final interviews that they did take the time to fully read and understand marker descriptions. This
understanding was demonstrated by P4 in his criticism of the change size markers, described in Observation 2. However, P2 felt that ChangeMarkup’s markers are rendered redundant by IntelliJ’s Git plugin, suggesting he did not fully understand the information provided by ChangeMarkup.

We have thus observed, from analysis of participant interviews and questionnaires, the emergence of three main themes in response to RQ1, *Do developers who are shown information that could potentially help avoid the introduction of bugs behave differently than without that information?*

- Participants are interested in seeing recent code marked in the editor but do not wish to know its exact age.
- Change size warnings did not influence our participants, though there are code characteristics, such as test coverage, that can cause them to code more cautiously.
- Most code discussion by our participants occurs during code reviews rather than during coding tasks.

We have also observed the following regarding RQ2, *How do developers notice and interact with new editor markers about potential bugs?*

- Participants are naturally curious and drawn to new visual features in their IDE, but may neglect to fully interact with them and thus can miss out on their informative potential or misunderstand their meanings.

### 5.4 Analysis of Logs and Developer Communication Diaries

One participant (P4) both submitted a communication diary and had logged data available from his machine. The communication events fall within the timespan of P4’s log and so we were able to perform an analysis of P4’s data that was not possible with the other participants. Two other participants (P1 and P6) also provided ample plugin log data to analyze. Figure 5.1 shows the data for these three participants. All tooltip events represent a hover action that likely produced a tooltip; no click actions were captured in the log, consistent with the minimal clicking reported by participants during final interviews. For each log, the timespan covered is continuous but does
not necessarily begin and end at the start and end of the study, respectively, as represented by the gradient shapes. There are no end gradients for P4 or P6 as their data spanned beyond the end of the study. Data over a timespan refers to any contiguous IDE logging from the first logged event to the last available or the end of the study, not just logging specific to ChangeMarkup. This gives us more information about times participants were intentionally not using ChangeMarkup. For example, P4 was probably using his IDE between 11:30 AM and 2:30 PM of Thursday November 20 without interacting with any ChangeMarkup markers, but we cannot tell whether or not he interacted with markers prior to 11:00 AM of that day. P4 communication events are represented with 30 minutes of uncertainty because he recorded all his diary entries as occurring on the hour or half hour. It should also be noted that Figure 5.1 shows 9:00 AM to 8:00 PM every day, but we do not know what times our participants arrived at and left work. We cannot assume that an IDE log entry means the participant was at work at the time; logging can occur when the machine is idle. It is also unlikely that participants were in the office over weekends.
Based on Figure 5.1, we make a few observations. P4 viewed far more tooltips on the days he did not talk to anybody. P4 either communicated or viewed tooltips every day; there are no days besides weekends on which he did neither. P4’s communications are infrequent; on days that he does communicate, he never has more than three conversations. This seems to align with his preference, indicated during initial interviews, for making a solid effort to understand code rather
than having to talk to somebody. If so, the increased tooltip viewing on communication-free days could suggest he is using ChangeMarkup, in place of communication, to learn about code.

The bursty tooltip viewing activity of two participants (P4 and P6) may be evidence of learning. We learned from final interviews that some participants came to associate visual properties of the markers with their meanings; P4 knew a marker’s meaning based on its position, and P6 knew what the colors meant. Since we also learned that developers are intrigued by new visual features, we would expect tooltip viewings to be numerous at the start of the study and occur less frequently as the participant relies more on visual features rather than tooltips. Though it occurs relatively late in the study, the bursty activity seen on Friday 21, Monday 24, and Tuesday 25, could possibly be the result of those participants trying to learn about the plugin at those times. P6 did not realize that multiple code ages are represented; if bursty activity is indeed indicative of learning, it is possible that he relied solely on the purple color of age markers from that point forward, no longer needing to consult tooltips.

Analyzing the raw log data, we noticed that all three participants hovered over age and change size markers with relatively even temporal distribution; there was no discernible pattern in which type of tooltip is viewed. We analyzed the times of P4’s tooltip viewings and communications and determined the average amount of time that passes between either a tooltip viewing or a communication and the next event, shown in Table 5.4. We calculated averages by measuring the time from each activity to the next, adding the times corresponding to each event pairing and dividing by the number of event pairs. For example, there were 6 instances in which the first event P4 did after viewing a tooltip was communicate with somebody, and when this did happen, the time between the two events was approximately 35 minutes. The calculated times include nights and weekends.
<table>
<thead>
<tr>
<th>Event type</th>
<th>Type of next event</th>
<th>Average time until next activity</th>
<th>Total event pairings of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tooltip viewing</td>
<td>Tooltip viewing</td>
<td>7 minutes</td>
<td>20</td>
</tr>
<tr>
<td>Tooltip viewing</td>
<td>Communication</td>
<td>35 minutes</td>
<td>6</td>
</tr>
<tr>
<td>Communication</td>
<td>Tooltip viewing</td>
<td>2 minutes</td>
<td>6</td>
</tr>
<tr>
<td>Communication</td>
<td>Communication</td>
<td>12 minutes</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.4 Average time passed between P4 events

It appears that regardless of the event type, the time elapsed before viewing a tooltip is much less than the time elapsed before communicating. It also seems that if P4 views a tooltip, much more time passes before he has a conversation (35 minutes) than the amount that passes if he has a conversation followed by another conversation (12 minutes). This might suggest that tooltips are informative enough to wait longer before needing a conversation, or that conversations lead to follow up conversations, but due to the small sample size of events we cannot know for sure.

Due to the sparsity of overlapping log and diary data, we can only present possible interpretations of participant behavior; further research would be needed to verify these ideas. As only one participant (P4) provided overlapping log and communication diary data at all, our observations about his behavior cannot be generalized.

5.5 Threats to Validity

We have identified potential threats to our study methodology and data analysis techniques. Our field study may have been too short for us to expect a significant response to ChangeMarkup, the usage of which we did not formally teach our participants. We also encountered difficulties with compatibility of ChangeMarkup on participant machines, which may have reduced the number of IDE logs we could collect. API limitations prevented us from detecting with certainty which hover actions actually produced a tooltip. Finally, our participant pool was too small to produce generalizable results.
Final interviews and participant data indicate that some participants misunderstood the marker meanings or overlooked the drilldown features, but we intentionally did not teach participants formally about the data or usage of ChangeMarkup. It is possible that this limited our ability to see changes in developer behavior. We made our choice to more closely simulate what we believe is the more realistic industry scenario of not receiving an accompanying tutorial on installation of a new plugin.

Two weeks, the length of our user study, may not be a long enough duration over which to expect any changes in developer workflow or communication behavior. We designed and distributed the questionnaires with the intention of capturing such changes, hopefully as a result of using ChangeMarkup, which would be indicated by any variations in a participant’s responses from one questionnaire to the next. We did not observe any such variations but it is possible that this is due to the short duration of the study.

The plugin generated logs and participant maintained diaries were sparse, significantly reducing the amount of study data we could analyze. Plugin logs were missing due to compatibility issues between ChangeMarkup and the variety of participant work computers, or the different IDE log configurations of those machines. This shortage of recorded data is compounded by the fact that some participants produced only a log, since logs and diaries are generally inconclusive if interpreted independently.

Due to API limitations encountered during implementation of ChangeMarkup, we were unable to programmatically capture which marker hover actions actually produce a tooltip; we know only that hover actions are repeatedly logged from the time a mouse enters a marker until the time it leaves, and that a series of sequential logged hover actions for a single marker means the mouse may have been over the marker long enough for its tooltip to appear. To decide how many logged hover actions probably indicate a tooltip, we performed 20 trials on our own machine in which we hovered over a marker until a tooltip appeared, then immediately moved the mouse away to cease logging of the hover action. We found the minimum number of log entries in which we obtained a tooltip to be 4. We used the minimum value to allow for our reaction time delay in moving the mouse away from the marker. Since participants ran ChangeMarkup on different machines than the one on which we performed this trial, the tooltip viewing data
presented in Section 5.4 could include some hovers that did not produce tooltips or be missing others that did. A related potential source of error is that when a marker hover action is logged, the only information identifying the marker is its tooltip, but markers can have identical tooltips. This means a series of sequential logged hover actions with the same tooltip could be not due to the mouse resting on a marker, but rather being moved across several markers having the same tooltip.

We were fortunate to have a participant pool spanning a wide range of experience levels, but our sample size was small, effectively only five developers, so findings are not generalizable.
Chapter 6: Discussion

When presented with file-level bug prediction information at code review time, Google developers did not find the information actionable [11]. When ChangeMarkup was used to display line-level code stability information in five industrial developers’ editors, only a small subset of the developers found the information actionable. Clearly, more research is needed to understand appropriate points in the software development process to present information about bug proneness. The fact that the participants in our study mentioned code age as being an indicator for when they take more care suggests that the presentation of certain code characteristics may be of value to the developers. Determining which code characteristics are most valuable requires interviews or other studies with a broader set of industrial developers.

Initially, developers participating in the study showed interest in the new information available in the editor. However, quickly they returned to prior habits of focusing on other information in their editor. It is possible that an adaptive plugin that is better able to anticipate when a developer may need code age information and show the information only in limited circumstances may help reinterest a developer in investigating the information. For example, an adaptive form of ChangeMarkup might only show code age and commit size information as part of the process of submitting a commit. At this point in time, a developer could reconsider if a commit should proceed prior to more review. Or, as another example, when a developer starts reading or investigating a section of the code not recently visited, an adaptive form of ChangeMarkup could turn on code age information and make the developer temporarily aware of potential points of instability.

6.1 Future Changes and Uses of ChangeMarkup

We have ideas for such a variation of ChangeMarkup specifically tailored to the code review task described by one participant (P1) of tracing a commit as viewed in GitHub back to the editor. P1 found ChangeMarkup helpful during this activity because it marked the lines of each commit she viewed with a unique color in her editor. Based on this feedback, an adaptive version
of ChangeMarkup, which enables itself only during such a commit comparison task, might be useful during code reviews. Since it would be disabled whenever the developer is actively writing code, we could give less consideration to conflicts with other colors in the IDE, such as those of compiler warnings, freeing up a wide range of colors by which to represent various code ages. This would allow the viewer to more easily distinguish different commit ages than is currently possible using the shades of purple shown by ChangeMarkup. Since it would disable itself during most tasks, we would also be less concerned about intrusiveness of visualization, allowing for more easily seen highlights. For example, the plugin could highlight entire lines in subdued colors, rather than being limited to high salience coloring in the gutter only.

Another possibility for future work is a simplified version of ChangeMarkup that shows only the latest commit highlighted in the editor. One participant (P6) directly stated that he would like a plugin that does exactly this. Of the concepts we studied, participants showed greatest interest in being able to see the most recent commit marked in the editor, indicating in interviews that such is useful for drawing attention to recently edited, and therefore potentially unstable, code (P3) or for tracking personal progress (P4, P6). We speculate that not showing older commits would not be a drawback of this simplified approach; most participants were unaware during our study that multiple ages were presented at all.

One participant (P4) suggested another possible variation of ChangeMarkup that shows only the purple code age markers, augmented with the name of the author of the related commit. A full history of commit ages and authors for the line could be listed upon clicking the marker. When asked what additional plugin features he would like to see, P4 proposed adding this full author history to the already presented code age.
Chapter 7: Conclusion

A growing number of techniques are being proposed to help draw a developer’s attention to code that might cause bugs earlier in the software development lifecycle. Much of this work is focused on development of techniques with the evaluation of the techniques based on retrospective studies of changes to source repositories. In this thesis, we consider a simple hypothesis: can the provision of code age and change size information in a developer’s editor help a developer determine when to more carefully consider code as part of a change? We developed a plugin, ChangeMarkup, to display this information in the editor used by five JavaScript industrial developers. Although several of the developers describe using code age as a sign of where code might be brittle and special care might need to be taken, it was challenging to engage the developers to pay attention to simple visual cues of the information in the editor. Through the use of the plugin and interviews of the developers, we learned that these developers are interested in even simpler cues than we provided: is code new or old (as opposed to age gradations). Developers did not find change size information to be helpful despite previous studies that have shown the number of lines added and modified is an indicator of unstable code. Although small, our tool and study provide insights into the complexities of changing developer behaviour with information related to bug likelihood.
Bibliography


