How Do Software Engineers Understand Code Changes? - An Exploratory Study in Industry

Yida Tao[1], Yingnong Dang[2], Tao Xie[3], Dongmei Zhang[2], Sunghun Kim[1]

[1] The Hong Kong University of Science and Technology, Hong Kong, China
{idagoo, hunkim}@cse.ust.hk

[2] Microsoft Research Asia, Beijing, China
{yidang, dongmeiz}@microsoft.com

[3] North Carolina State University, Raleigh, NC, USA
xie@csc.ncsu.edu

ABSTRACT

Software evolves with continuous source-code changes. These code changes usually need to be understood by software engineers when performing their daily development and maintenance tasks. However, despite its high importance, such change-understanding practice has not been systematically studied. Such lack of empirical knowledge hinders attempts to evaluate this fundamental practice and improve the corresponding tool support.

To address this issue, in this paper, we present a large-scale quantitative and qualitative study at Microsoft. The study investigates the role of understanding code changes during software-development process, explores engineers' information needs for understanding changes and their requirements for the corresponding tool support. The study results reinforce our beliefs that understanding code changes is an indispensable task performed by engineers in software-development process. A number of insufficiencies in the current practice also emerge from the study results. For example, it is difficult to acquire important information needs such as a change's completeness, consistency, and especially the risk imposed by it on other software components. In addition, for understanding a composite change, it is valuable to decompose it into sub-changes that are aligned with individual development issues; however, currently such decomposition lacks tool support.

Categories and Subject Descriptors

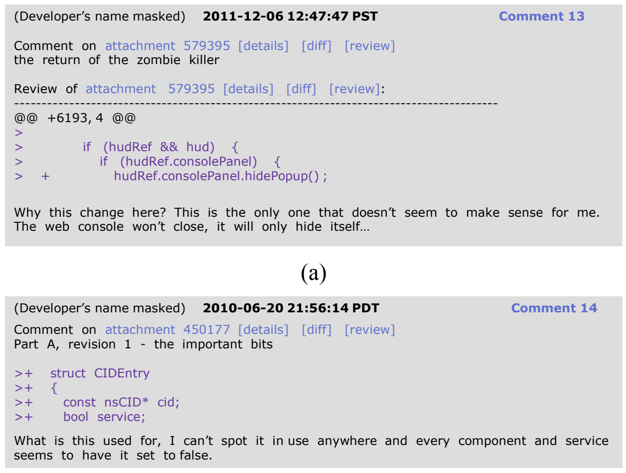
D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Experimentation, Human Factors

Keywords

Code change, code review, information needs, tool support



(Developer's name masked) **2011-12-06 12:47:47 PST** **Comment 13**

```
Comment on attachment 579395 [details] [diff] [review]
the return of the zombie killer

Review of attachment 579395 [details] [diff] [review]:
-----------------------------------------------------------------------------------------
@@ +6193, 4 @@
>
>         if (hudRef && hud) {
>           if (hudRef.consolePanel) {
>   +         hudRef.consolePanel.hidePopup();
```

Why this change here? This is the only one that doesn't seem to make sense for me. The web console won't close, it will only hide itself…

(a)

(Developer's name masked) **2010-06-20 21:56:14 PDT** **Comment 14**

```
Comment on attachment 450177 [details] [diff] [review]
Part A, revision 1 - the important bits

>+  struct CIDEntry
>+  {
>+    const nsCID* cid;
>+    bool service;
```

What is this used for, I can't spot it in use anywhere and every component and service seems to have it set to false.

(b)

Figure 1. Mozilla bug #702707 (a) and #568691 (b). Patch reviewers had difficulties in understanding the associated code changes. See https://bugzilla.mozilla.org/ for details.

1. INTRODUCTION

Software undergoes continuous changes, through which new features are added, bugs are fixed, and performance is improved. These code changes usually need to be understood by software engineers when performing their daily development and maintenance tasks. Previous research has suggested that understanding code changes is the basis of various advanced development tasks, such as troubleshooting unexpected behavior [27] and monitoring maintenance of code clones [40].

To facilitate tasks of code change understanding, various tools and systems have been developed (for simplicity, we refer to "*code change understanding*" as "*change understanding*" in the rest of this paper). For example, the ubiquitous and simple *diff* utility enables engineers to easily inspect code changes. A series of work then emerged to extend the simple *diff* utility by tracking line movement [6], distilling fine-grained change entities [13], and inferring systematic structural differences [27]. Modern Integrated Development Environments (IDEs) and Software Configuration Management systems (SCMs) further empower engineers – the former allow code exploration and manipulation with a simple mouse-click while the latter automatically track and manage code changes.

Although equipped with these advanced tools, engineers sometimes still get stuck when understanding code changes. Figure 1 shows two motivating examples from developers' comments on two Mozilla bug-fixing changes. Neither developers (as change reviewers) fully understood the respective bug-fixing change, and they had to ask the patch submitters for explanation on the bug-fixing changes. In fact, from the comment lists of Mozilla bug reports, we discovered a number of concerns from developers when they reviewed a proposed bug-fixing change. These concerns include the change's implementation detail[1], whether it covers all the necessary files[2], and the impact and risk of the change[3]. The inability to acquire such information may hinder the work of both the change reviewers and the change committers, and slow down the entire development process.

The first step to further assist engineers in understanding code changes is to gain good knowledge of engineers' current practice. However, no systematic study has ever been conducted to this end, and little empirical knowledge has ever been provided. Without such study or empirical knowledge, understanding and improving the current change-understanding practice remains difficult and untargeted. Moreover, any attempt to improve the corresponding tool support might be hindered by this lack of empirical knowledge [42].

To address these issues, we systematically studied software engineers at Microsoft on their change-understanding practice. Our study includes a large-scale online survey and a series of follow-up email interviews – both are designed to address the following research questions:

RQ1: *What is the role of change understanding during software development*? Specifically, we intend to investigate under which development scenario(s) change understanding is required and how often it is required.

RQ2: *What are software engineers' information needs for understanding code changes*? In addition, *what kind of information is difficult to acquire with the existing tool support*?

RQ3: *How to improve the effectiveness and efficiency of the practice in understanding code changes*?

The quantitative data gathered from 180 survey participants reveals the indispensable role of change-understanding practice. Understanding code changes is frequently required – typically several times each day – during major development phases such as implementation and verification. To accomplish a change-understanding task, engineers seek various types of information such as the change's quality (e.g., correctness, completeness, and consistency), its risk (e.g., "*does this change break any code elsewhere?*"), and most importantly the rationale of this change.

We discovered that engineers face non-trivial challenges when determining the risk of a change. Their current approaches to assess a change's risk mainly include testing and code review. However, the testing approach is considered time-consuming (in rerunning all or even selected existing test cases), and it heavily depends on the capability of the existing test cases to cover the change. The code-review approach is considered human-intensive and error-prone. In addition, code reviewers do not fully benefit from the code-exploration features provided by common IDEs (e.g., "*find all references*" in Visual Studio) when the code reviewers explore the context of the changed code and estimate its risk. The study participants thus call for tools that tackle these two insufficiencies.

We identified an interesting type of check-in practice that makes later change understanding significantly difficult. Basically, engineers sometimes mix multiple bug-fixing changes or changes with other purposes (e.g., new-feature implementation, refactoring) in a single check-in (we refer to this kind of change as a *composite change*). Understanding a composite change requires non-trivial efforts. Participants call for a tool that can automatically decompose a composite change into separate sub-changes, each of which addresses one single development issue (e.g., fixing one bug).

We further found that the rationale of a change is regarded as the most important information for change understanding. Fortunately, it is one of the easiest to acquire if an informative change description (also known as *check-in message* or *commit log*) is provided.

In summary, this paper makes the following main contributions:

- A large-scale exploratory study on industrial practice in understanding code changes. To the best of our knowledge, our study is the first in empirically studying the change-understanding practice.
- An extensive exploration on engineers' information needs for understanding code changes.
- A valuable guideline for future research and tool design that aims at supporting change-understanding tasks.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces our study methodology. Sections 4, 5, and 6 present the study results on the three research questions, respectively, followed by discussion in Section 7. Section 8 presents threats to validity, and Section 9 concludes.

2. RELATED WORK

Various approaches have been proposed to facilitate the change-understanding practice. ChangeDistiller [13] uses Abstract Syntax Tree (AST) differencing to extract fine-grained code changes. LSDiff [27] infers systematic structural code changes and presents changes as logical rules and exceptions. Buse and Weimer [4] used symbolic execution to automatically summarize and document behavioral differences. Holmes and Notkin [21] leveraged static and dynamic dependency graphs to identify inconsistent and subtle behavioral changes. Our work complements these previous approaches by empirically studying the change-understanding practice in industry.

A number of previous empirical studies have explored developers' information needs and work habits. Through interviews, Fritz and Murphy [14] identified 78 questions that developers ask in their daily development tasks. These information needs are classified into a number of categories such as people specific, code-change specific, and work-item progress. LaToza et al. [32] conducted two surveys and several interviews to interpret developers' work habits via their typical tools, activities, practices, and satisfaction. Ko et al. [31] performed a two-month field study involving 17 groups of developers to observe their information-seeking activities. Although closely related to ours, these studies presented a general overview on the entire software-development process instead of addressing one particular development task in depth. Other related empirical studies investigated developers' practices from various perspectives. For example, Buse and Zimmermann [5] discussed engineers' data and analysis needs for decision making. Schröter et al. [48] investigated developers' communication behavior on the submission of a changeset. Developers' practices of

[1] Mozilla bug #422026, comment 20

[2] Mozilla bug #417545, comment 139

[3] Mozilla bug #365992, comment 14

Table 1. Venues for our literature survey

Type	Acronym	Description	Surveyed Articles
Journal	TSE	IEEE Transactions on Software Engineering	25
	TOSEM	ACM Transactions on Software Engineering and Methodology	6
Conference	ICSE	International Conference on Software Engineering	53
	ESEC/FSE	European Software Engineering Conference / Symposium on the Foundations of Software Engineering	19
	OOPSLA	Conference on Object-Oriented Programming Systems, Languages, and Applications	6
	ISSTA	International Symposium on Software Testing and Analysis	9
	ASE	International Conference on Automated Software Engineering	27
	ECOOP	European Conference on Object-Oriented Programming	5
	FASE	International Conference on Fundamental Approaches to Software Engineering	7
	ICSM	International Conference on Software Maintenance	26

managing change impact [50] and using object-oriented concepts [16] were also empirically evaluated through field studies and surveys. Our study differs from these previous studies in that we empirically studied in-depth a specific development practice – understanding code changes.

Another line of related work focuses on program comprehension. Cornelissen et al. [9] identified 176 research articles published in the past decade for addressing program comprehension through dynamic analysis. LaToza and Myers [33] combined a field study and user survey to gain the knowledge of how developers understand large and complex code bases. They found that reachability questions are common and often time-consuming to be answered. In the study conducted by Ko et al. [30], developers were given unfamiliar programs and asked to perform debugging and enhancement tasks. By observing their IDE usage, Ko et al. proposed a program-comprehension model as a process of searching, relating, and collecting relevant information. The change-understanding practice studied in our work shares certain characteristics with program comprehension. However, these previous studies provide no or little insight on how code changes affect developers' understanding of the source code.

Tool design and support for software-evolution tasks have been evaluated. Sillito et al. [49] analyzed a wide range of industry and research tools that support for answering questions asked during code-change tasks. They concluded that better support is needed for maintaining context and piecing information together. Ko et al. [30] observed that navigation tools caused significant overhead during software-maintenance tasks. Instead of simply evaluating the current tool support, our study explored the potential feature enhancement expected by engineers for their change-understanding tasks.

3. METHODOLOGY

Our exploratory study on change understanding consists of a large-scale online survey and a series of follow-up email interviews. In this section, we introduce the design of our exploratory study in terms of the three research questions presented in Section 1. We also report the subject-selection process and the distribution of study participants.

3.1 Online Survey

Our online survey consists of 12 questions that can be divided into three parts. The first part of the survey includes simple demographic questions about participants. We investigate the role of change-understanding practice (**RQ1**) in the second part of the survey. We generalize the development process and its phases in a waterfall fashion [46], and propose seven scenarios within the development phases that frequently involve code manipulation (Figure 2). Participants were asked to select their most-often encountered scenarios that require understanding code changes. Additionally, participants were asked to report their frequency of understanding code changes in this second part of the survey.

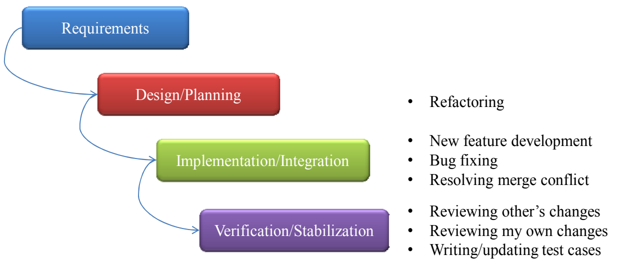

Requirements
Design/Planning
Implementation/Integration
Verification/Stabilization
• Refactoring
• New feature development
• Bug fixing
• Resolving merge conflict
• Reviewing other's changes
• Reviewing my own changes
• Writing/updating test cases

Figure 2. Seven development scenarios and their corresponding development phases used in our survey.

The third part of the survey explores participants' information needs for understanding code changes (**RQ2**). Instead of directly asking participants about their information needs, which might be vague and difficult to respond, we presented them a list of potential information needs explicitly in the survey question. To prepare these potential information needs and ensure their relevance to the change-understanding practice, we performed a preliminary literature survey on the state-of-the-art research related to code-change analysis and management. In this literature survey, we browsed over 180 articles published in 10 major software-engineering venues within the past decade (Table 1). From these research articles, we extracted 15 information needs for understanding code changes (Table 2). Participants were asked to rate the importance of each information need and estimate the difficulty of acquiring such information in their current change-understanding practice. Participants could also write down their additional information needs in a free-text form.

Before distributing our survey, we conducted pilot interviews with three experienced engineers at Microsoft. We ran the survey with them and made notes of their comments. According to their feedback, we refined the survey questions and adjusted the wording to make sure that the questions are relevant and clear[4].

[4] The survey questions are available at http://research.microsoft.com/en-us/projects/softwarechange/.

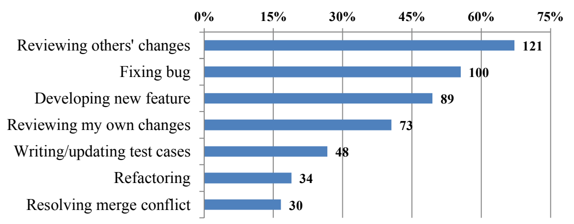


Figure 3. Participant distribution across different development scenarios that require understanding code changes. (The value beside each bar is the absolute number of responses.)

3.2 Participants

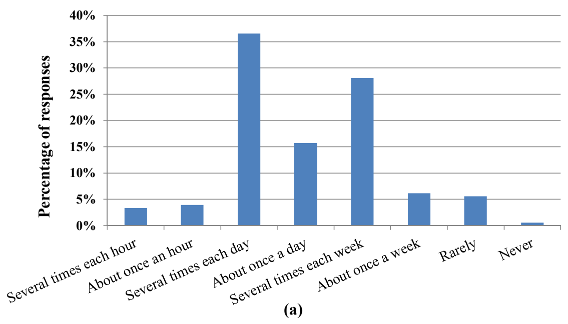
The population that we selected for the online survey included Software Design Engineers (SDEs), Software Design Engineers in Test (SDETs), and Program Managers (PMs). At Microsoft, SDEs, SDETs, and PMs are three core roles who closely collaborate throughout the entire software development. According to the feature specification written by PMs, SDEs implement new features and fix bugs; SDETs write test cases and test the software. These three roles could offer insightful opinions towards the change-understanding practice since they have frequent exposure to source code and changes.

The survey was conducted in early December 2011, and its invitation was sent to 1,279 subjects randomly selected from the mailing list of Microsoft. Within two weeks, we received 180 responses – 99 from SDEs, 56 from SDETs, and 25 from PMs. Note that we also received 172 out-of-office auto-replies. If we excluded these subjects, our survey-response rate is about 16%, comparable to the rate of other similar previous studies [2, 32]. On average, the respondents have 9.1 (±4.9) years of experience at Microsoft. Of all the respondents, 135 were individual contributors (i.e., engineers that are responsible for only tasks completed by themselves), 9 were architects, 28 were lead, and 8 were managers. This population works on various types of products including operating systems (27%), standalone desktop applications (37%), web applications (25%), mobile applications (11%), software/web services (42%), and others (6%).

3.3 Follow-up Interviews

After analyzing the online-survey responses, we performed follow-up email interviews in order for participants to further elaborate on two main findings. One finding is that participants considered it difficult to satisfy some important information needs for understanding code changes. We included one such information need, namely, a change's risk of breaking other code, in follow-up interviews. We hope that the participants could explain the insufficiencies in the current practice and suggest potential improvements for the corresponding tool support. The other finding is that participants added several other information needs for understanding code changes as the free-text-form answer. We included one such newly emerged information need –"*can this change be broken into smaller discreet changes?*" – in the interviews and asked for further elaboration. Section 6 presents the collected feedback and discusses potential improvement for the change-understanding practice (**RQ3**).

In addition, we investigated some unexpected survey findings in the follow-up interviews. For example, while participants generally considered knowing the rationale of a change as the top priority in change-understanding tasks, they claimed that the rationale information is actually the easiest to acquire in current practice. Section 7 discusses the interview feedback on two such findings.



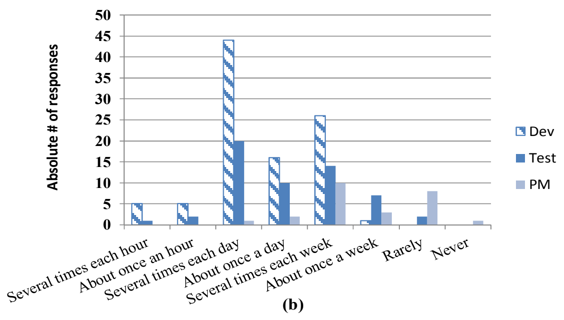


Figure 4. Frequency of understanding code changes

4. ROLE OF CHANGE UNDERSTANDING

In this section, we describe common development scenarios that require understanding code changes. We also report how frequently participants are involved in each scenario. These two findings are used to characterize the role of change-understanding practice in software development process (**RQ1**).

4.1 Scenarios

We characterize the software-development process as a four-phase waterfall-style model, including *requirements*, *design/planning*, *implementation/integration*, and *verification/stabilization* as shown in Figure 2. While the first phase focuses on analyzing user requirements, the subsequent three phases generally involve code manipulation and thus may require engineers to understand code changes. Based on common development practice, we proposed seven potential development scenarios that require understanding code changes and asked participants to choose the one(s) that they most often encounter.

Figure 3 shows the results. "Reviewing other's code changes (to give comments or approval for check-in)" received 121 votes (67.2%) and ranked the first. "Reviewing my own code changes (to ensure its quality before check-in)" received 73 votes (40.6%) and ranked the fourth. In other words, the code-review process requires engineers to understand code changes. Thus, a better understanding of changes might smooth the code-review process and help reviewers spot bugs more effectively.

Another frequent scenario is bug fixing (55.6%), during which engineers are often required to understand why a previous change introduces a bug and how it can be fixed. Surprisingly, about half of the participants (49.4%) need to understand code changes in their task of new-feature implementation. According to the participants' comments from the subsequent open-ended question, we

Table 2. Potential information needs for understanding code changes. The third column briefly describes where these information needs are derived from.

Information Needs	Abbreviation	Source
Reasoning and Assessing the Change		
I-1: What is the rationale behind this code change?	Rationale	The necessity of check-in message [4, 27]
I-2: Is this change complete? Did it miss any place that should also be changed at the same time?	Completeness	Co-change prediction and change propagation [36, 57, 58]
I-3: Is this change correct? Does it work as expected?	Correctness	Bug-introducing change and bad-fix [17, 29, 56]
I-4: Does this change introduce poor design, or break the current design? How?	Design	Modularity violation and code decay [12, 54]
I-5: Does this change introduce code clones? How?	Clones	Code clones might be harmful [24, 34]
I-6: How does this change alter the program's dynamic behavior?	Behavior	Behavioral and latent change [8, 21]
Exploring the Context and Impact of the Change		
I-7: Who references the changed classes/methods/fields?	References	Change impact analysis [3, 44]
I-8: How does the caller method adapt to the change of its callees?	Caller	API change adaptation [10, 11, 55]
I-9: Does this change break any code elsewhere? How?	Risk	Risk and impact analysis [20, 37, 48]
I-10: Are there any other places that need similar changes?	Consistency	Consistent editing [38, 40, 41]
I-11: Which test cases should be run to verify this change?	Tests	Change impact and regression test [44, 53]
I-12: Is any additional test case needed to cover this change?	New tests	Test augmentation [43, 47]
I-13: Which part of the change may cause the test case(s) to fail?	Failing tests	Failure localization [44, 45, 51]
Evaluating the Change History		
I-14: Is this changed location a hotspot for past changes? How many times has this location been changed?	Change-proneness	Frequently modified code might be defect-prone [18]
I-15: Is this changed location a hotspot for past bug-fixes? How many times has this location been fixed?	Defect-proneness	Frequently fixed code might be defect-prone [26, 28]

found that one possible reason is for learning purposes. The following excerpts from two participants' comments indicate that engineers want to learn from previous changes and apply the acquired knowledge for their future development needs.

"*...understand how a feature/product/component works*"

"*...understand others' code changes for my sample application development needs*"

We found that among the 56 SDET participants, 41 of them (73.2%) selected "writing/updating test cases." This result shows that, for testers, writing/updating test cases is their primary scenario that requires understanding code changes. In addition, more than 15% of participants indicated that they needed to understand code changes in the remaining two scenarios – refactoring and resolving merge conflict.

4.2 Frequency

As shown in Figure 4(a), more than 36% participants need to understand code changes several times each day. In addition, a high percentage (43.8%) of participants is involved in this practice once a day or several times each week. In general, the majority of participants need to understand code changes on a daily basis.

A drill-down with respect to participants' work role further explains the extreme cases on both ends of the frequency distribution. As shown in Figure 4(b), developers need to understand code changes quite frequently; some of them even perform such practice several times each hour. PMs, on the other hand, need to understand code changes much less frequently – this observation may result from their work responsibilities described in Section 3.2. Note that testers' frequency of understanding code changes is similar to that of developers. While one might expect that testers' duties mainly involve writing and updating test cases, we have shown that even these tasks require understanding code changes frequently.

Understanding code changes is an indispensable practice in software development, especially for developers and testers. It is frequently required in major development phases, in particular during the code-review process.

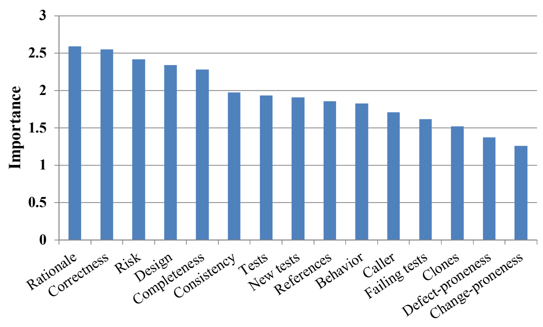
5. INFORMATION NEEDS

As described in Section 3.1, we extracted 15 potential information needs from the state-of-the-art research. Table 2 lists these information needs (in the form of questions), their abbreviations, and part of their literature source. In this section, we report participants' rating of these information needs in terms of their importance for understanding code changes and the difficulty of acquiring the corresponding information in the current practice (**RQ2**). We also summarize additional information needs extracted from participants' free-text answers in Section 5.3.

5.1 Rating

Participants were first asked to rate the importance of each information need on a 4-point scale (3-very important; 2-important; 1-somewhat important; 0-not important). Figure 5(a) shows the information needs ordered by their average importance score. I-1~I-4 together with I-9 are considered very important (score > 2) for understanding code changes. These information needs are about the rationale behind a change (I-1: "*what is the rationale behind this change?*"), assessing the change quality (I-2~I-4: completeness, correctness, and design issue of the change) and its risk (I-9: "*does this change break any code elsewhere?*"). Information concerning a change's historical metrics (I-14~I-15: change-proneness and defect-proneness of the changed location) are considered less important for understanding the change.

We further explored whether these information needs are fulfilled in the current practice by asking participants to rate the difficulty of acquiring the corresponding information on a 4-point scale (3-difficult; 2-somewhat difficult; 1-relatively easy; 0-straightforward). Figure 5(b) shows the results. Whether a change is consistent or not (I-10: "*are there any other places that need similar changes*") is considered the most hard-to-acquire information. The need to assess a change's consistency mainly originates from the existence of homologous code across a software system. Homologous code refers to code fragments that are similar in textual content (e.g., code clones), share common dependence conditions [52], and play similar roles or perform similar interactions with other objects [41]. Previous studies [15, 23, 24] have shown that inconsistent modification to these similar code fragments could cause unexpected software behavior and introduce



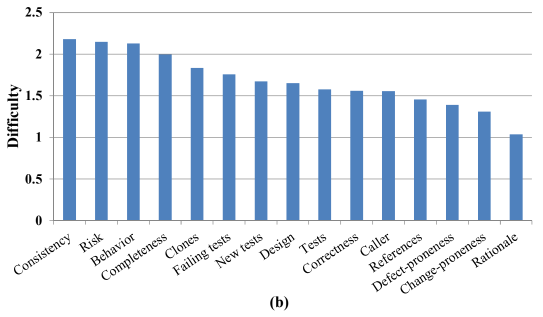


Figure 5. The average importance score (a) and difficulty score (b) for each information need

faults. Although engineers are aware of the importance of a change's consistency (the importance score for I-10 is 1.94 as shown in Figure 5(a)), they have a difficult time acquiring such information.

The second most hard-to-acquire information is I-9 ("*does this change break any code elsewhere?*"), followed by I-6 ("*how does this change alter the program's dynamic behavior?*") and I-2 ("*is this change complete? Did it miss any place that should also be changed at the same time?*"). Answering I-6 typically requires advanced techniques of program analysis such as semantic [1, 22] or behavioral differencing [21] in addition to simple textual differencing. Answering I-2 may require change reviewers to understand the logical coupling (determined by program analysis) and the evolutionary coupling of the change entities (e.g., entities that are often changed together or subsequently [7, 58]) – both types of information are not directly available in the current practice.

5.2 Importance vs. Difficulty

Figure 6 shows the information-need distribution in terms of their importance and difficulty scores. We mainly care about the information needs located on the top-right corner, since they are considered important but the corresponding information is difficult to acquire. I-9 ("*does this change break any code elsewhere?*") appears to be the most needed information that is hard to acquire. It is followed by I-2 that concerns a change's completeness. Other concerned information includes I-4 (design), I-10 (consistency), and I-3 (correctness). While all of this information deserves attention, we explicitly investigated the current practice in answering I-9 in the follow-up email interviews since I-9 is the only information whose importance and difficulty scores both exceed 2. The interview result is presented in Section 6.1.

On the other hand, information about historical change metrics (I-14 and I-15) is considered less important and easier to acquire.

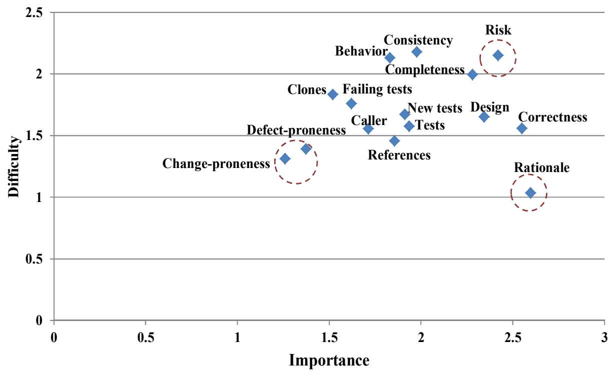


Figure 6. Difficulty vs. importance for 15 information needs

We discuss possible explanations in Section 7. Note that I-1 ("*what is the rationale behind this code change?*"), the most important information need, turns out to be the most straightforward question for engineers to answer. This finding is contrary to our intuition as well as those suggested by the literature [14, 32, 35, 49]. Based on the follow-up interviews, we found that the difficulty of answering I-1 heavily relies on the availability and quality of the change description. We discuss this finding in Section 7.

5.3 Other Information Needs

The information needs investigated so far are proposed based on our literature survey on the state-of-the-art research (as discussed in Section 3.1). Although we expect this literature survey to be comprehensive, it is still possible that engineers' certain information needs for understanding code change are not covered in this survey. We mitigated this issue by providing an open-ended question right after the rating questions and allowing participants to describe their additional information needs in a free-text form.

We received 33 answers to this open-ended question. We read through these answers and summarized them into two categories. Answers in the first category contain information needs different from our proposed ones. We grouped answers in this category if they implied similar information needs. For example, "*what is the ideal solution to the problem?*" and "*what were the other potential solutions to the issue?*" both imply similar information need. In this case, we grouped them together and generalized them as one information need: "*does this change provide the ideal/optimal solution?*" Answers in the second category do not exactly provide new information needs – they are more like elaborations on our listed ones. For example, some participants responded that they want to know "*why the code change is needed?*" and "*are there non-obvious assumptions made by the change?*" We treated these questions as elaborations on the already listed information need "*what is the rationale of the change?*"

In total, 24 new information needs were identified. We present some of them in Table 3. While most of the newly emerged information needs are self-explanatory, there exist a few whose intention is not clear. We selected one such information need – "*can this change be broken into smaller discreet changes*" – for further investigation. The participant who raised this question did not provide further explanation, and it was not straightforward for us to infer his/her reason for asking such question during a change-understanding task. Therefore, we included this information need, which we referred to as *change decomposition*, in the follow-up interviews. The responses from the follow-up interviews are discussed in Section 6.2.

Table 3. Examples of newly emerged information needs

No.	Newly emerged information need
1	Can this change be broken into smaller discreet changes?
2	Is this change a (major) design/architectural change?
3	How does this change affect the code base's stability and maintainability?
4	How does this change affect the program's performance?
5	How does this change affect end user experience?
6	Is this change safe given the current phase in product lifecycle?
7	Does this change introduce security bugs by over/under encoding?
8	Which documentation is linked to this code change?

Information regarding a change's quality (e.g., its completeness and consistency) and risk (e.g., whether it breaks any code elsewhere) is important for understanding the change, but such information is difficult to acquire in the current practice.

6. IMPROVING CURRENT PRACTICE

The online survey revealed important information needs for understanding code changes. We included two information needs – determining a change's risk and decomposing a change – in the follow-up email interviews for further investigation. Based on participants' feedback, we suggest how to improve the effectiveness and efficiency of the change-understanding practice (**RQ3**).

6.1 Determining a Change's Risk

The quantitative data collected from the online survey reveals that I-9 ("*does this change break any code elsewhere?*") is an important concern (importance score > 2) during a change-understanding task, but this information is difficult to acquire (difficulty score > 2) in the current practice. We included this finding in the follow-up interviews and asked participants whether they agree with this finding or not. Additionally, we asked them to briefly describe their current practice (e.g., approaches and tool usage) in determining the risk of a change.

Among 23 participants who proceeded to the follow-up interviews, 15 totally agreed with our finding; 3 agreed that I-9 is important, but they pointed out that acquiring such information could be easy in some circumstances; 4 participants only described their current approaches to handle I-9, and their opinions towards our finding were not clear; 1 participant did not input anything on this particular finding. In general, the majority of participants confirmed that I-9 is important but difficult to answer in the current practice.

According to the participants' description, two approaches are typically used to check whether a change breaks anything (I-9). The primary approach is unit/regression testing, as one participant explicitly stated "*...does this change break anything else' is the driving force behind practices like unit tests and other regression tests.*" Typically, "*most check-ins have some sort of test to verify the code*", and "*running all appropriate unit tests for the code change could test the immediate impact of the change.*" However, the testing approach is limited as its ability to answer I-9 heavily depends on the adequacy of the test suite; in addition, testing itself is time consuming:

"*...it's not a full solution in and of itself as it's heavily dependent on how thorough the unit tests are...*"

"*...the difficulty lies in determining all inputs to the changed location...*"

"*...assuming a well-designed test suite with good code coverage. However, this is still limited as functional and integration tests are too time consuming to set up and run...*"

Another commonly used approach to answer I-9 is manual code review/inspection. During this process, reviewers might have to check all the dependencies of the changed parts; such checking typically requires significant manual efforts as well as the support of compiler, debugger, and static-analysis tools:

"*I usually would need to manually find portions of the code which are using changed portion and figure out how this change affects callers. Sometimes it's not obvious from the code itself, I have to actually step through the code with debugger to understand it.*"

"*…rebuilding the change and all its dependencies. This allows the compiler, linker and static analysis tools to evaluate any gross errors.*"

Participants who partially agreed with our findings mentioned that static-analysis features (e.g., finding references) come in handy for code exploration, which is typically involved when determining the risk of a change. However, sometimes the large volume of returned references (e.g., method callers) might overwhelm the reviewers and reduce their efficiency in completing the task ("*…this is an error-prone process if left to code inspection as it would require evaluating all possible code paths through all calling functions...*"). It is even harder for reviewers to decide a change's risk this way if the affected component is beyond the reviewers' control or knowledge:

"*In particular when dealing with cross-component or cross-binary changes, finding the associations in the code, which allow you to make these determinations, is difficult without component or binary specific knowledge.*"

"*It is hard to evaluate impacts on other components, unless there is clear interface between this component and others. Very frequently, other components have some assumptions on this component, while these assumptions are not documented.*"

In general, testing and code review/inspection are two major approaches to determine the risk of a change, and there exist a number of tools (e.g., static analysis) to assist with this task. However, as one participant stated, "*none of those are really very satisfying, though, as my confidence level in the change is not as high as I would like.*" From participants' feedback, we identified two potential tool improvements with respect to these two approaches (Table 4). First, participants call for a feature that detects code portions impacted by the change and the affected test cases. This feature is also expected to automatically identify the corresponding test owner(s) and inform them of the retesting task. In fact, there are a number of previous studies [44, 53] that offer similar solutions. However, they focus on the fault-localization context and have not been applied to the change-understanding context. Second, participants call for a feature that performs IDE static-analysis functionalities (e.g., go to definition, find references) specifically on the changed code under the diff view. The participants claimed that although the diff tool works fine for visualizing code changes, it "*misses a level of understanding object relationships*" (see the full quote in Table 4). The static-analysis features, on the other hand, allow engineers to view object dependencies easily but operate only on the complete code base. As participants suggested,

Table 4. Desired tool support for determining a change's risk

Desired feature	Quote
A feature that detects the code portion impacted by the change and the affected test cases. The feature should also be able to automatically notify the test owner of the retesting task.	"*A feature that tells me that based on the changed code, which code must be retested as it is dependent upon the change, who owns testing that dependency, notifies the contact that the dependency must be retested, and tells the contact which tests must be run.*"
	"*A feature that flags any existing test cases that are related to the code change based on coverage data. An "Intelli-sense" for updating these tests would be nice as well.*"
A feature that performs IDE static-analysis functionalities (e.g., go to definition, find reference, caller/callee tree, etc.) on the changed code portion under the diff view.	"*I think from mostly looking at diff tools and seeing code changes, the current tool set is missing a level of understanding object relationships. VS has tools that let you view object references and dependencies. It would be nice to be able to select a function with a code change and easily ask where that function is being used. I can do that using a search index and a diff tool, but they are not currently integrated as far as I know.*"
	"*Most needed is the ability to use Visual Studio code analysis tools (go to definition, find all references, caller/callee tree, find in files, object browser) on the code change.*"
	"*A feature which allows for navigation in the diff, e.g. "go to definition" or "find caller".*"

an integration of the diff utility and the static-analysis features can improve engineers' efficiency in determining the risk of a change.

6.2 Decomposing a Change

One survey participant added another information need – "*can this change be broken into smaller discreet changes?*" – without explaining why he/she raised this question during a change-understanding task. We included this question in the follow-up email interviews and asked the participants for elaboration. According to their description, the need of change decomposition stems from the existence of changes that share certain characteristics. These changes usually involve a large number of files ("*anyone that checks in 50 files (outside of a simple name change done via an automated tool and that's the ONLY change) ...*"), spans a lot of features ("*…code changes spanning too many features…*") or address multiple issues (e.g., multiple bug fixes). We refer to such change as a *composite* change.

A typical example of a composite change is a single change that involves multiple bug fixes. Such changes are sometimes pointed out by the committers, who explicitly mention that "*this change is for bug #1, #2, and #3...*" in the check-in message. A less direct way to identify such a change is through issue-tracking systems, in which a single commit is linked to more than one bug[5]. Another example of a composite change is floss refactoring, during which a developer intersperses other kinds of code changes (e.g., adding a feature, fixing a bug) with refactorings [25, 39]. Murphy-Hill et al. [39] provide evidence that floss refactoring happens frequently in the evolution of Eclipse and Mylyn.

A composite change, which is commonly observed across software evolution history, appears to be difficult to understand. In particular, figuring out the part of a change that relates to each individual constituent issue (e.g., each bug fix) is considered difficult and error-prone:

"*…sorting out which changes goes with which bug is difficult and error-prone.*"

"*…in cases where many sources are checked-in, it can be difficult to tell since many bug fixes are checked-in together.*"

"*…some changes are combined with other changes (e.g. multiple bug fixes), so it is hard to figure out whether a specific change was related to one bug or another.*"

Although this understanding obstacle directly results from engineers' check-in practice, e.g., whether they check in each bug fix individually or wait until a bunch of bugs are fixed and then check in them all together, participants do not really question the practice itself. One participant mentioned that "*people mix multiple bug fixes in one check-in for better productivity.*" Murphy-Hill et al. [39] also suggest that floss refactoring is typically performed to keep the code healthy. Instead, participants call for a feature that could automatically decompose a composite change into separate sub-changes, each of which addresses a single development issue individually (e.g., fixing one bug):

"*... some change lists for review may contain multiple fixes spread across overlapping sets of files. It would be useful to be able to analyze groups of functional changes (i.e., each bug fix) instead of having to go file by file and context switch between changes.*"

"*(I want) a feature that allows me to follow on particular variable or other change through the changelist. If the dev has grouped a bunch of fixes, I want to be able to see all the changes that related to one bug, or possibly one variable within the change list.*"

Independently from us, Herzig and Zeller [19] also pointed out this composite-change problem (they refer to it as "tangled change") and investigated the distribution of such changes in five open source projects. They proposed an approach to untangle such changes to reduce noise and bias when mining software repositories. Our work differs from theirs in that we investigate the effect of composite changes in the context of change understanding. Based on industrial practitioners' feedback, we provide evidence that understanding such changes requires non-trivial efforts, and a tool feature for change decomposition is desired. To the best of our knowledge, no previous study has ever addressed this problem. We expect that the discussion here could shed lights on inspiring future work towards this end.

To accomplish a change-understanding task, engineers need better support for determining the risk of a change and decomposing a composite change.

[5]See revision 990792 for Apache Commons Math. This single revision addresses three issues (MATH-394, MATH-397, and MATH-404), one of which is a bug fix and the other two are minor improvements. http://svn.apache.org/viewvc?view=revision&revision=990792

7. DISCUSSION

As described in Section 5.2, two findings on engineers' information needs for change understanding are inconsistent with our intuition and the existing research. In this section, we discuss these two findings based on the feedback collected from the follow-up email interviews.

Why is understanding the rationale of a change fairly easy?

The rationale of a change (I-1) is considered the most important information need for understanding the change (Figure 5(a)). Intuitively, acquiring such information takes non-trivial efforts, and a number of previous studies [14, 32, 35, 49] also suggest the same. Nevertheless, our quantitative data shows that understanding a change's rationale is in fact easy (I-1's difficulty score ≈ 1 as shown in Figure 5(b)) in the current practice. We asked participants for their opinions on this finding: 19 participants basically agreed that change-rationale information is easy to acquire if (1) an informative change description (also known as *check-in message* or *commit log*) is available (17 participants) and (2) the code is well-written for readability (2 participants):

"*Yes, it is generally easy, when given a bug description or a checkin description.*"

"*Good comments are usually the best way for indicating intent in code, particularly if the changes are for reasons that cannot be inferred in the local code context.*"

"*In well written code this task is relatively easy. This relates to code readability issue.*"

Three other participants held different opinions: they thought that figuring out the rationale of a change could be quite difficult. But their concerns were also about the availability and quality of the change description. One participant stated that a change's rationale can be inferred from "*collateral metadata such as bugs, changelist descriptions, review comments, etc.*", but these data "*tends not to accompany the code itself over time as it goes into maintenance by other owners*". Another participant described his experience on reviewing a change with comment "refactor code", but this change was in fact much more than just refactoring. In such case, he was really confused about the rationale behind the change, wondering "*was the refactor to improve CPU cache locality, load time, build time, political divisions within the product team, code readability, prep for new feature development, or simply idle hands?*"

In general, a well-written change description is the key to understand a change ("*we rely on check-in comments to accomplish a lot of the above (information seeking)*"). However, change descriptions are not always informative; sometimes they are not even available. This problem is also pointed out by a number of previous studies [4, 39]. Buse and Weimer [4] reported that, for five large open source projects, only 67% of changes on average are accompanied by a commit message that accurately describes what happened in the change. Murphy-Hill et al. [39] observed that check-in messages for a change do not reliably indicate the presence of refactoring in the change.

Tools could provide partial support to alleviate this issue. For example, DeltaDoc [4] uses symbolic execution and code summarization to automatically generate succinct human-readable documentation for arbitrary code changes. Zimmermann et al. [59] proposed the CUEZILLA tool, which is applied to an analogous scenario – when users encounter a bug and file a bug report. This tool is trained to measure the quality of a new bug report and recommend elements to be added to improve its quality. However, these tools cannot entirely replace human in the documentation-generation process. DeltaDoc can summarize only **what** is changed (e.g., the conditions under which program behavior changes and what the new behavior is) instead of explaining **why** the change is made and **why** the change is made this way (i.e., the rationale of the change) [4]. For bug-report generation, manual work is still required especially in the presence of rarely occurring or non-crashing bugs [59]. Hence, instead of purely relying on tools, engineers should try to provide informative change description in the first place. They should be better aware of the change description's importance, and the fact that its quality significantly impacts the later change-understanding efforts. After all, as one participant stated, "*it's entirely up to the dev making the change as to how hard or easy it is for someone else to figure out why the change was made.*"

Why are historical metrics not considered as good sources of information?

Previous studies used historical metrics such as change-proneness or defect-proneness to assess code quality [18, 26, 28] or characterize code changes [29]. However, we found that such metrics (I-14 and I-15) are not that important in the context of understanding code changes (see Figure 5(a)). The follow-up participants generally agreed with this finding. They stated that engineers, especially developers, typically care about the specific task at hand rather than past issues:

"*As a dev, I agree that change history isn't very important for me mainly because the most important thing is the status of the current codes. It's useful only when re-designing, re-factoring or optimizing the codes, we will consider how to avoid hotspots.*"

"*I-14, 15 are very interesting for people who either have to deal with the risky code (test), or those who need to plan future work in specific areas (PM). Devs seem to be most often concerned with the here and now, and interested mostly in the short-term issues around a check-in.*"

In addition, developers mostly rely on their own knowledge to understand a code change. On the other hand, testers or program managers are not that entrenched in the code, so they might resort to historical metrics for understanding changes:

"*While historical frequency of changes can inform test/pm as to the relative bugginess/complexity of code area, I think most devs get a sense of that much more directly already since they are reading the code and do not rely on metrics for the same purpose.*"

"*Identifying hotspots are interesting to flag areas of problems and raise some concerns to why this area is changed so often, but that I can live without as a developer. Because even the fact that I do multiple code reviews in one area (over time) is going to tell me that is potentially a problem area (from a design (i.e. poor design) or bug (i.e. poor implementation) perspective.*"

Previous work leverages historical metrics to detect or predict fault location [18, 26, 28]. According to the interview feedback, such heuristics for prediction does not really reflect the type of information that developers actually used for assessing code quality and locating problematic areas. Our interview suggests that compared to developers, testers and program managers might be more proper target users of the historical metrics.

8. THREATS TO VALIDITY

From an internal-validity point of view, the generality of the survey questions may be affected by researcher bias stemming from personal assumptions. We believe that this threat is mitigated by the pilot interview, the open-ended survey questions, and the follow-up email interviews. The pilot interview, which was conducted with one lead SDE, one SDET, and one research SDE, helped us refine the survey questions and ensure their relevance. An open-ended question was provided after the scenario and information-need questions, enabling participants to add or elaborate their own opinions. Furthermore, we confirmed some of our survey findings in the follow-up email interviews.

From an external-validity point of view, this study was conducted exclusively with software engineers (developers, testers, and program managers) at Microsoft. This threat is mitigated by the fact that the survey received 180 responses from engineers from various product teams including operating systems, desktop applications, mobile and web services. We also mitigate the survey bias by randomly sampling the survey participants and keeping the survey anonymous.

9. CONCLUSIONS

Software evolves with a series of source code changes. These changes need to be understood, assessed, and verified with significant development effort. To cope with such significant efforts, there is a strong need to understand and improve the change-understanding practice. Empirical knowledge of the current change-understanding practice can provide valuable insights for software-engineering practitioners and researchers to contribute to the improvement of this practice, e.g., by inspiring them to design better tool support for the change-understanding task.

Our exploratory study conducted at Microsoft serves as a first attempt to this end. The quantitative data from the online survey confirms that understanding code changes is a fundamental practice that happens frequently in major development phases. The qualitative data from open-ended questions and follow-up interviews further reveals a number of non-neglectable gaps between the current practice and the actual needs from developers. First, determining a change's risk is very important when understanding the change, but it is difficult even when existing tools provide a certain degree of support. Second, the crucial need to assess a change's quality (e.g., its completeness and consistency) also lacks tool support. Another commonly demanded but unavailable support is change decomposition. Engineers often address multiple issues within a single composite change, and such composite change would be easier to understand if it is decomposed into sub-changes that are aligned with each individual constituent development issue. We further found that the quality of the change description significantly affects the efforts of later change understanding. Engineers should be aware of the change description's importance and be responsible for providing informative descriptions upon check-in.

Acknowledgments. We thank all the developers, testers, and managers at Microsoft who participated in our survey and interviews, and offered their valuable feedback. We thank Jacek Czerwonka, Miryung Kim, Robin Moeur, and Thomas Zimmermann for their helpful comments on our early draft of the survey questionnaire. We also thank Kathryn Mckinley and the anonymous reviewers for their valuable comments and suggestions on an earlier version of this paper.

10. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Eng.*, vol. 14, no. 1, pp. 3-36, Mar. 2007.

[2] A. Begel and N. Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. In *ESEM'07*, pp.255-264, 2007.

[3] D. Binkley, R. Capellini, L. R. Raszewski, and C. Smith. An implementation of and experiment with semantic differencing. In *ICSM'01*, pp.82-91, 2001.

[4] R. P.L. Buse and W. R. Weimer. Automatically documenting program changes. In *ASE'10*, pp. 33-42, 2010.

[5] R. P.L. Buse and T. Zimmermann. Information needs for software development analytics. In *ICSE'12*, pp. 987-996, 2012.

[6] G. Canfora, L. Cerulo, and M. D. Penta. Ldiff: An enhanced line differencing tool. In *ICSE'09*, pp. 595-598, 2009.

[7] M. Ceccarelli, L. Cerulo, G. Canfora, and M. D. Penta. An eclectic approach for change impact analysis. In *ICSE'10*, pp. 163-166, 2010.

[8] R. Chern, and K. De Volder. The impact of static-dynamic coupling on remodularization. In *OOPSLA'08*, pp. 261-276, 2008.

[9] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, vol.35, no.5, pp.684-702, Sept.-Oct. 2009.

[10] B. Dagenais and M. P. Robillard. SemDiff: Analysis and recommendation support for API. In *ICSE'09*, pp.599-602, 2009.

[11] B. Dagenais and M. P. Robillard. Recommending Adaptive Changes for Framework Evolution. ACM Transactions on Software Engineering and Methodology, vol.20, no.4, September 2011.

[12] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, vol.27, no.1, pp.1-12, Jan 2001.

[13] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, vol.33, no.11, pp.725-743, Nov 2007.

[14] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE'10*, pp. 175-184, 2010.

[15] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA'10*, pp. 175-190, 2010.

[16] T. Gorschek, E. Tempero and L. Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In *ICSE'10*, pp.115-124, 2010.

[17] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed?. In *ICSE'10*, pp. 55-64, 2010.

[18] A. E. Hassan and R. C. Holt, The top ten list: dynamic fault prediction. In *ICSM'05*, pp. 263- 272, 2005.

[19] K. Herzig and A. Zeller. Untangling Changes. Unpublished manuscript, September 2011. http://www.st.cs.uni-saarland.de/publications/details/herzig-tmp-2011/

[20] R. Holmes and R. J. Walker. Customized awareness: Recommending relevant external change events. In *ICSE'10*, pp.465-474, 2010.

[21] R. Holmes and D. Notkin. Identifying program, test, and environmental changes that affect behavior. In *ICSE'11*, pp. 371-380, 2011.

[22] D. Jackson and D. A. Ladd. Semantic Diff: a tool for summarizing the effects of modifications. In *ICSM'94*, pp.243-252, 1994.

[23] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC/FSE'07*, pp. 55-64, 2007.

[24] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, Do code clones matter?. In *ICSE'09*, pp. 485-495, 2009.

[25] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE'11*, pp.351-360, 2011.

[26] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE'96*, pp. 364-371, 1996.

[27] M. Kim and D. Notkin. Discovering and representing systematic code changes. *In ICSE'09*, pp. 309-319, 2009.

[28] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pp. 489-498, 2007.

[29] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying Software Changes: Clean or Buggy?. *IEEE Transactions on Software Engineering*, vol.34, no.2, pp.181-196, March-April 2008.

[30] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, vol.32, no.12, pp.971-987, Dec. 2006.

[31] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE'07*, pp. 344-353, 2007.

[32] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE'06*, pp. 492-501, 2006.

[33] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *ICSE'10*, pp.185-194, 2010.

[34] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, vol.32, no.3, pp. 176- 192, March 2006.

[35] D. R. Licata, C. D. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. In *ASE'03*, pp. 281- 285, 2003.

[36] H. Malik and A. E. Hassan. Supporting software evolution using adaptive change propagation heuristics. In *ICSM'08*, pp.177-186, 2008.

[37] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE'03*, pp. 287-296, 2003.

[38] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI'11*, pp. 329-342, 2011.

[39] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE'09*, pp. 287-297, 2009.

[40] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *ASE'09*, pp.123-134, 2009.

[41] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE'10*, pp. 315-324, 2010.

[42] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE'00*, pp. 35-46, 2000.

[43] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *ASE'10*, pp. 397-406, 2010.

[44] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA'04*, pp. 432-448, 2004.

[45] X. Ren and B. G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA'07*, pp. 239-249, 2007.

[46] W. W. Royce. Managing the development of large software systems. In WESCON, 1970.

[47] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test suite augmentation for evolving software, In *ASE'08*, pp.218-227, 2008.

[48] A. Schröter, J. Aranda, D. Damian and I. Kwan. To talk or not to talk: factors that influence communication around changesets. In *CSCW'12*, pp. 1317-1326, 2012.

[49] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, vol.34, no.4, pp.434-451, July-Aug. 2008.

[50] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *ICSE'08*, pp. 241-250, 2008.

[51] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *ESEC/FSE'06*, pp. 57-68, 2006.

[52] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *ASE'10*, pp. 457-466, 2010.

[53] J. Wloka, B. G. Ryder, and F. Tip. JUnitMX - A change-aware unit testing tool. In *ICSE'09*, pp.567-570, 2009.

[54] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *ICSE'11*, pp. 411-420, 2011.

[55] W. Wu, YG. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *ICSE'10*, pp. 325-334, 2010.

[56] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, How do fixes become bugs?. In *ESEC/FSE'11*, pp. 26-36, 2011.

[57] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, vol.30, no.9, pp. 574- 586, Sept. 2004.

[58] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE'04*, pp. 563- 572, 2004.

[59] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report?. *IEEE Transactions on Software Engineering*, vol.36, no.5, pp.618-643, Sept.-Oct. 2010.