Noname manuscript No.
(will be inserted by the editor)

Do Programmers do Change Impact Analysis?

Siyuan Jiang · Collin McMillan · Raul Santelices

Received: date / Accepted: date

Abstract "Change Impact Analysis" is the process of determining the consequences of a modification to software. In theory, change impact analysis should be done during software maintenance, to make sure changes do not introduce new bugs. Many approaches and techniques are proposed to help programmers do change impact analysis automatically. However, it is still an open question whether and how programmers do change impact analysis. In this paper, we conducted two studies, one in-depth study and one breadth study. For the in-depth study, we recorded videos of nine professional programmers repairing two bugs for two hours. For the breadth study, we surveyed 35 professional programmers using an online system. We found that the programmers in our studies did static change impact analysis before they made changes by using IDE navigational functionalities, and they did dynamic change impact analysis after they made changes by running the programs. We also found that they did not use any change impact analysis tools.

Keywords Change impact analysis · Program debugging · Empirical software engineering · Software maintenance · Programmer navigation

This work is supported by the ONR N000141410037, the NSF CCF-1452959 and CNS-1510329 grants. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

S. Jiang
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA 46545
E-mail: sjiang1@nd.edu

C. McMillan
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA 46545
E-mail: cmc@nd.edu

R. Santelices
Dell SecureWorks
Atlanta, GA, USA 30328
E-mail: rsantelices@secureworks.com

1 Introduction

Change impact analysis (IA) is the task of finding the consequences of an alteration to software [6,39]. From a programmer's perspective, those consequences are typically the components of source code that would need to be modified in order to make a change. For example, if function A in a program is modified, and function B depends on function A, then function B may also need to be modified. IA is important because features in programs tend to be implemented across many components in source code [15,34], and a change to any one of the components may affect several of the others.

In theory, programmers will do change impact analysis prior to making any changes to the source code. Consider the example of repairing a bug. The consensus in the literature is that programmers 1) localize the bug behavior to a function or other set of statements, 2) determine a change to that code that will repair the bug, 3) do change impact analysis to determine the effects of that change, and 4) implement and test that change [6,21,10,28]. Several change impact analysis tools have been created based on this consensus, including static [10], dynamic [28], and conceptual [21] solutions.

While these tools have been shown to be effective, in practice no procedure has emerged as a standard accepted by a majority of programmers. This situation is akin to that pointed out for program comprehension tools by Roehm *et al.* at ICSE 2012 [41]. What Roehm *et al.* found was that program comprehension tool support was rich and growing, but that in practice programmers did not use these tools – the reality was that the tools assumed a different usage scenario than the programmers were actually following. Change impact analysis tools may be in a similar situation. Programmers may not do change impact analysis in the way that the literature suggests. For example, it is possible that programmers implement changes as quickly as possible, and repair negative effects of the change as they occur, skipping change impact analysis. Or, it is possible that programmers are simply unaware of the state-of-the-art techniques available.

In this paper, we present an empirical study of change impact analysis by professional programmers. We conduct this study in two phases. In the first phase of our study, we recruited nine programmers to read two bug reports from two open-source systems, and repair the bugs by modifying the systems' source code. We recorded videos of the repair processes. The objective was to observe the programmers' behavior as they fixed the bug. We then analyzed this behavior to determine whether the programmers used any tools for change impact analysis, and if not, whether they performed change impact analysis through manual inspection.

In the second phase, we surveyed 35 programmers who had professional experience at different organizations. The intent of the survey was to determine the knowledge that the programmers had of change impact analysis, and to query the programmers about their current IA procedures. We asked knowledge questions via an online survey, but we also present the programmers with an actual bug report from an open-source system, along with screenshots of an IDE. The programmers clicked on the screenshots to select the components of the IDE that they would use to repair the bug.

We designed three research objectives to answer the question whether programmers do change impact analysis. We will describe the research objectives in Section 6. From the research objectives, we designed several research questions

in each phase. By answering the research questions, we found evidence of a gap between theory of change impact analysis and the practice. We came to this conclusion based on several findings, because the programmers did not know the term "Change Impact Analysis" and they did not use change impact analysis tools. Moreover, we found the programmers did static impact analysis before they made changes and dynamic impact analysis after they made changes.

2 The Problem

The problem we target in this paper is a gap in the current literature regarding how programmers accomplish change impact analysis (IA) in practice. During code change activities (e.g., fixing a bug, adding a feature, etc.), it is generally assumed that programmers conduct IA prior to modifying source code. A rich and diverse set of tools has been created based on this assumption. However, there is little empirical evidence confirming this assumption. While IA procedures are recommended in textbooks and other educational resources, it is possible that programmers follow different strategies. For example, it is possible that programmers make changes first based on intuition, and test the consequences of those changes. This behavior would be in line with current literature surrounding "opportunistic" programming and similar concepts [8,7,18,14], which are part of a broader consensus that programmers only try to comprehend the minimum amount of source code required to make a change [41,24,27,16].

This paper has a direct impact on the design of change impact analysis tools for code change activities, because it improves the understanding of how programmers will use those tools. For example, if a majority of programmers do not do IA prior to making changes, then tool support could potentially be directed to monitoring the effects of changes already made, rather than predicting effects from possible changes. This potential impact is not unprecedented, as similar implications have been discovered in other areas of software engineering, e.g., program comprehension [41] and bug report authorship [20].

3 Background: Change Impact Analysis (IA)

Change impact analysis (IA) is a general process to identify elements that are indirectly or directly affected by a change [30,52,6]. IA has two types of application scenarios. The first application scenario is upfront. In requirement management, given a specific change request, IA is to identify the files and the models that may need to be modified for the requested change. A change request can be a bug report, a request of adding a feature, etc. The second application scenario is in source code change activities, such as debugging, refactoring, feature implementation, etc. When programmers debug, they may do IA to find out whether a change in code fix the bug. In refactoring, programmers may do IA to ensure that the changes will not have any effect on the outputs of the program.

The following subsections will describe three terms related to IA: IA process, IA techniques, and IA activities. We will also discuss the difference among the three terms in Section 3.4.

3.1 Classic IA Process

Petrenko and Rajlich [37] introduced an interactive IA process at the granularity of all components. This process can be used in both application scenarios (in requirement management and code change activities). Based on this interactive IA process, we illustrated the IA process at the granularity of statements in Figure 1. The objective of the IA process is to obtain a *Changed Set*, that is, the statements that need to be changed. Ideally, the *Changed Set* will always contain just the statements that must be changed. However, in practice, the *Changed Set* will not always be accurate.

To begin doing IA, the programmers must identify a location to change. An example of a location would be a statement of code. This statement is called an *initial change location*. First, the programmers put the *initial change location* into the *Changed Set* (Figure 1 Area a).

Second, the programmers follow different procedures to find code that is affected by the *Changed Set* (Figure 1 Area b). For example, they may follow the dependencies from the *initial change location* (we listed four possible strategies in Section 3.2). Then, the programmers add the affected code into the *Possible Changed Set* (Figure 1 Area b). Note that the *Propagation Set* in Figure 1 Area b will be defined later.

Third, the programmers look at every statement in the *Possible Changed Set*. For each statement, if the programmers think the statement need to be updated, they put this statement into the *Changed Set* (Figure 1 Area c). Otherwise, the programmers decide whether the statement propagates the effects of the change, such as changed values. If the statement propagates the effects to other statements, the programmers put it into the *Propagation Set* (Figure 1 Area d).

Fourth, with the new *Propagation Set* and the updated *Changed Set*, the programmers start at Figure 1 Area b again. The programmers stop the IA process when there are no more statements to be added into the *Propagation Set* and the *Changed Set*. The output of IA is the *Changed Set*, that is, the statements that need to be changed because of the *initial change location*.

3.2 IA Techniques

Although the IA process contains any phase described in Figure 1, current IA techniques focus on automating the task in Figure 1 Area b, which is finding the code that may be affected by a given statement. According to Li *et al.* [31], IA techniques can be broadly categorized as: 1) static dependency analysis, 2) dynamic execution information analysis, 3) software repository mining, 4) coupling measurement, or 5) combined approaches.

Static dependency analysis typically does reachability analysis on a graphical representation of a program derived from the source code [9], such as, call graphs, control flow graphs, or dependency graphs. Hattori *et al.* [19] analyzes reachability of a call graph, i.e. whether a method can be called by another method. If a method can be called by another method, the first method can be affected by the later one.

Dynamic execution information analysis uses data generated from the execution of the programs, rather than only the source code [31]. There are different

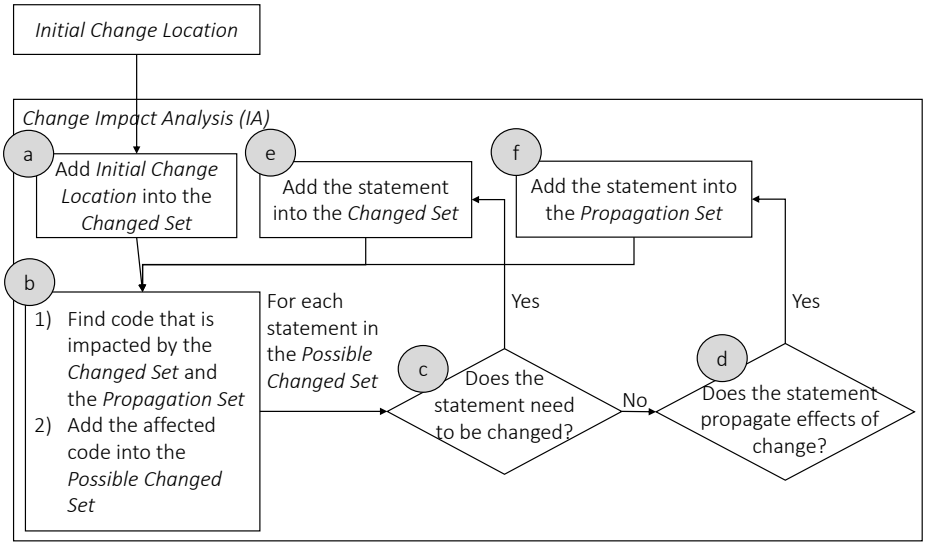

Fig. 1: The Process of Change Impact Analysis in the literature, described in Section 3.1

types of execution data [28, 35]. For example, Law and Rothermel use execution traces of function calls [28]. The intuition is that the function called first may affect the functions called later.

Software repository mining analyzes not only the source code and the execution data, but also the logs of software version control systems. Many kinds of logs can be used for IA [56, 12]. For example, Zimmermann *et al.* [56] apply data mining to changes made in the history to see what methods are usually changed together.

Coupling measurement aims to calculate the degree of dependency between any two of program modules, such as methods. This degree of dependency indicates the possibility of one program module being affected by another module. The majority of coupling metrics are based on interactions between two modules. For example, Beszédes *et al.* [5] use metrics based on Execution-After relations [3]. Different from measuring the interaction between two modules, Poshyvanyk *et al.* [38] proposed conceptual coupling using information retrieval on identifiers and comments from code to measure relationship between two modules.

Combined approaches integrate multiple kinds of analyses to gain better accuracy. For example, Breech *et al.* [9] combined static and dynamic approaches to obtain higher precision. Another example is that Gethers *et al.* [17] integrate dynamic analysis and repository mining, in order to choose an appropriate technique in a specific situation. Similarly, Kagdi *et al.* [21] integrate coupling measurement with software repository mining.

3.3 IA Activities

According to the Figure 1, in the scope of this paper, we define IA activities as the actions that programmers do to find and understand the code that may belong to the *Possible Changed Set* (Figure 1, Area *b*), the *Propagation Set* (Figure 1, Area *f*), or the *Changed Set* (Figure 1, Area *e*). By this definition, some examples of IA activities are file navigation, source code reading, and call graphs examining. In fact, any program comprehension activities are seen as IA activities in this paper, because program comprehension is one possible way to do change impact analysis. Note that we do not consider any program comprehension tool as an IA tool, because their purpose is program comprehension in general. Similarly, we do not consider IDE navigational functionalities as IA tools. However, they can be used in IA activities.

An extreme example of IA activities is running programs. By running a program and check its result, programmers may decide to continue the IA process or not. If the result is correct, some programmers may feel confident to commit the changes they made. If the result is incorrect, programmers know there is some code need to be changed. If the value of the result is meaningful, programmers may locate the code to be changed by the incorrect value.

3.4 IA Process, IA Techniques, and IA Activities

In this section, we describe the differences among the IA process, IA techniques and IA activities. The IA process is a general procedure defined in academia that programmers do to find consequences of a source code change. We restrict our discussion of the consequences within the consequences in source code, i.e. the source code locations that need to be changed because of the initial change location. Note that IA process can be started with an initial change location with or without a specific change, which means, programmers can do IA before or after they make a change. We do not make any assumption about whether programmers do IA before they make a change.

IA techniques are designed specifically for a task in the IA process, which is Figure 1 Area *b*. There are IA techniques for post-change, which means that they assume programmers do IA after they make a change. There are also IA techniques for pre-change, which means that they can work without a specific change.

Any activity, whether they occur before or after programmers make a change, if the activity can help programmers in any phase in Figure 1, we consider they are IA activities. IA activities are not necessarily related to IA techniques. IA activities can occur in various tasks, such as debugging, refactoring, reverse engineering, and effort estimation.

4 Background: Debugging

In this section, we describe a general debugging process based on the scientific debugging method introduced by Zeller [55]. The debugging process is shown in Figure 2. Given a bug report, first, programmers reproduce the bug and observe the failure. Then, based on the observation, programmers create a hypothesis about

the cause of the failure. In the next step, programmers test the hypothesis. The methods of testing a hypothesis are various. Sometimes, programmers can print a value at a specific location to determine whether to reject the hypothesis or not. Often, programmers may make a change to the source code and observe whether the program runs as expected. If the observed value is not expected or the program gives unexpected result, it means the hypothesis is rejected. If the hypothesis is rejected, programmers need to make a new hypothesis about the failure cause. If the hypothesis is supported and the bug is fixed, programmers stop the debugging process. If the hypothesis is supported, but the bug is not fixed, programmers refine the hypothesis and test it again.

An example of a hypothesis about a failure cause is "If variable X's value at line 10 is 1, the program should have a correct output." Based on the hypothesis, programmers added a line at line 10 which assigns 1 to X. Then, programmers run the program to test the hypothesis. If the failure is gone, this test supports the hypothesis. If there is a failure, the hypothesis is rejected. The hypothesis being supported does not necessarily mean that the bug is fixed. For example, always assigning value one to X at line 10 may cause errors in other tests, and there may be a deeper cause for the faulty status of X.

The steps in the process of debugging may be skipped by some programmers. For example, programmers may skip reproducing the bug and create a hypothesis based on a bug report. Some hypotheses are not necessarily about what the failure cause is. They can be any hypothesis that is related to the program and help programmers narrow down the problems.

Note that fault localization falls into the step where programmers try to make hypotheses about the failure cause. In this case, the hypotheses are about the locations of faults. Regression testing can be in the beginning and in the end of a debugging process. Regression testing can reveal a bug, which leads to a bug report. The bug report becomes the beginning of a debugging process. Additionally, the results of regression testing may help programmers form hypotheses about the bugs. After programmers fix a bug, regression testing can help programmers ensure the quality of the fix, which occurs in the end of debugging.

Similarly, IA can occur in different phases in the debugging process. Programmers can do IA during fault localization. If a source code location affects the result, programmers may create a hypothesis that this location is a faulty location. Programmers may do IA when they test hypotheses. For example, in order to test the hypothesis "source code line 10 should not affect the output of the program", programmers need to investigate the effects of any change in line 10, which is considered as an IA process.

5 Related Work

The section will cover the empirical studies about IA in general to present the current status of IA and IA techniques. Then, we will discuss the empirical studies in IA practices on which we base our study design. Thirdly, we will introduce the related work in tool adoption because we aim to look for the reason why programmers do or do not do IA and help IA techniques get adopted in practice.

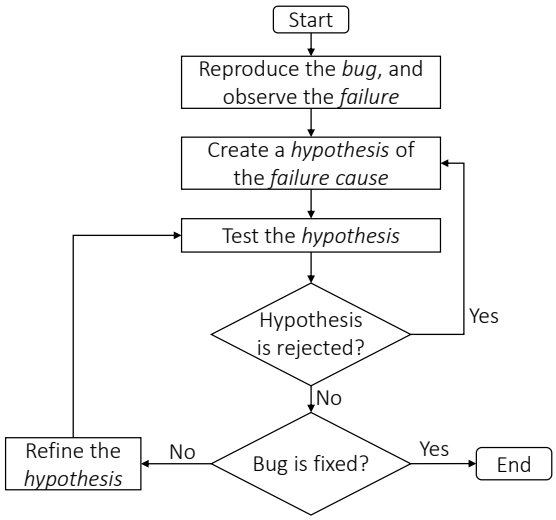

Fig. 2: The process of debugging according to the literature (see Section 4). This is the process that we expect our participants to follow in our studies.

5.1 Empirical Studies in IA

Many empirical studies are conducted to study software changes, their sizes and how to predict changes. Gethers *et al.* [17] created four benchmarks for IA from four open source Java systems. In the total of 277 change requests, half of them were addressed by the changes within five methods. 75% of the change requests were addressed by changing less than 13 methods. Gethers *et al.* introduced an integrated impact analysis approach including textual information analysis, evolutionary information analysis, and execution information analysis. The highest precision reported is 18% and the highest recall reported is 75%. Ye *et al.* [54] created the benchmarks from five Java projects. The number of bug reports they collected for each benchmark ranges from 593 to 6,495. The maximum number of the files fixed for a bug report in each benchmark ranges from 87 to 587. But the median number of the files fixed in any benchmark ranges from one to three. For Eclipse Platform and Tomcat projects, their recommending system includes at least one fixed file in the top ten ranked files in 70% of the bug reports.

For changes not in source code, McIntosh *et al.* [33] mined the relationship between source code changes and build changes in four large systems. They found that only 4% to 26% of the source code changes require build changes. Using code change characteristics, McIntosh *et al.* developed classifiers to explain when build changes are necessary.

There are also many empirical studies evaluating different types of IA techniques. Acharya *et al.* [1] did an empirical study of IA based on static program slicing in ABB. With the high setting defined in their paper, on 147 changes they ran, about 45% had more than 300,000 lines of code, and about 46% had less than 10,000 lines of code. Wu *et al.* [53] explored the dependencies among programs and

other binaries. Without simplifying the dependency graph, a dependency graph for wget has 26 nodes, which means there are 25 binaries interacted with wget.

Rungta *et al.* [44] introduced a change impact analysis technique, iDiSE, and applied this technique to tcas. In the 16 version changes they presented in the paper, at least 57% of the constraints generated in the changed version are impacted by the change. Dam and Ghose [13] implemented an IA technique for agent systems. They did an experiment on two agent systems developed to compete at the Multi-Agent Programming Contest. The precision ranges from 25% to 50% and the highest recall is 65%. Parande and Koru [36] studied five KOffice products and found that dependencies concentrate more on smaller modules.

5.2 Empirical Studies in IA Practice

De Souza [47] went to two large software teams, and studied the actual IA approaches – team strategies that handle software artifact-level dependencies. From this study, De Souza *et al.* introduced *impact management* for team management of dependencies and changes, which models how developers inform impacts to others or prevent impacts from others. The difference between our study and De Souza's study is that our study focused on how programmers do IA individually and their study focused on how programmers communicate IA results with others.

In order to discover important issues in IA, Rovegård *et al.* [42] did an empirical study at Ericsson AB, Sweden. They interviewed 18 people in different organizational levels. Rovegård *et al.* mapped three organizational levels in software engineering – technical, resource and product – to three levels in decision-making – operative, tactical, strategic levels. They found important IA issues from both organizational and personal views. For example, the issue "affected parties are overlooked" is seen as a high priority both in organizational and personal view. They also found that personal views affect organizational views on IA issues. Their study found the important IA questions while our study looks for how programmers do IA. Tao *et al.* [50] did a large-scale online survey and follow-up email interviews in Microsoft. From the study, they also identified a series of questions about understanding changes.

There are several empirical studies on individuals in software engineering. To study how programmers navigate through source code while they debug, Lawrance *et al.* [29] recruited 12 programmers from IBM, and asked the programmers to debug an open-source project, RSSOwl. Lawrance *et al.* required the programmers to "think aloud" when they debugged, and recorded videos and audios of their debugging process. Based on the videos and audios, Lawrance *et al.* investigated how information foraging theory is applied to the navigation behaviors of programmers. Our work has a similar approach with the one of Lawrance *et al.* However, their work is to study the debugging process, and our goal is to study change impact analysis in the context of debugging.

Similar to our study, Wetzlmaier and Ramler [51] did a study on individual behaviors on IA. Their topic is "How well do experienced developers estimate changes?". They found that the estimation of the developers is inaccurate. The difference between their study and our study is that their study finds out the accuracy of IA while we want to finds out whether and how developers do IA.

5.3 Studies in Tool Adoption

Storey [48] reviewed the tools in program comprehension based on cognitive theories. In this review, Storey identified six aspects of tool support for program comprehension: documentation, browsing and navigation support, searching and querying, multiple views, context-driven views, and cognitive support. In 1997, Storey *et al.* [49] did a study of 30 programmers using three tools to solve some high-level program comprehension tasks. Based on their observation, they believe that the tools should support for different comprehension strategies, such as, bottom-up [45] and top-down [11].

Bassil *et al.* [4] did a survey about software visualization tools with more than 100 participants. They found in general the participants were satisfied with the current software visualization tools. In the data collected, the functionality of searching for graphical and/or textual elements is rated as the most useful one in visualization tools. In terms of practical aspects of visualization tools, the reliability and ease of using the tools are rated as the two most important aspects. Kienle and Müller [22] did a literature survey about requirements of visualization tools. They identified seven quality attributes including usability, and seven functional requirements including views and search.

6 Empirical Study Design Overview

The objective of this paper is to begin to answer the question ***do programmers do change impact analysis?*** In particular, we want to study whether programmers do IA when they do code change activities. Towards the problem we stated in Section 2, we designed three research objectives.

$Objective_1$ What knowledge do programmers have of research activities in IA?
$Objective_2$ What technologies do programmers use to do IA if they do any?
$Objective_3$ At what phases of debugging, do programmers do IA if they do any?

The first objective helps us to answer whether there is a gap between research and industry. In studying the second object, if we find technologies used for IA, it is not only the evidence of programmers doing IA, it also provides information about the IA process in practice. In studying the third object, if we find evidence that they do IA before they make changes, this will be the empirical evidence for the assumption of various IA tools.

We target the three research objectives with a "depth" study and a "breadth" study of programmer behavior. In the "depth" study we recorded the behavior of nine programmers repairing actual bugs in source code. The "breadth" study is a survey of 35 programmers. We designed four research questions in the in-depth study and five research questions in the breadth study. We illustrate the overview of the objectives and research questions in Figure 3.

7 In-depth Study Design

This section will describe our in-depth study, in which we hired programmers to solve actual bugs in software. Following, we will cover our research questions, methodology, subject applications, participants, and threats to validity.

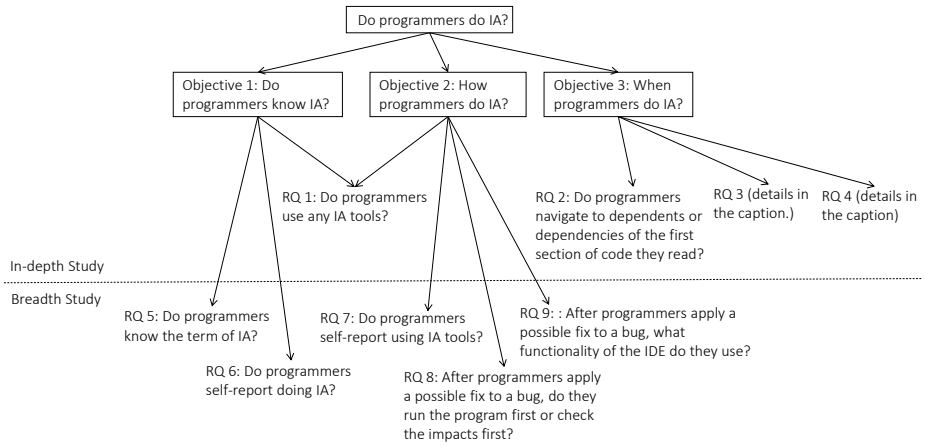

Fig. 3: All the research questions are introduced in Sections 7.1 and 10.1. In the in-depth study, RQ_1 helps to answer $Objective_1$ and $Objective_2$. RQ_2, RQ_3, and RQ_4 helps to answer $Objective_3$. In the breadth study, RQ_5 and RQ_6 helps to answer $Objective_1$. RQ_7, RQ_8, and RQ_9 helps to answer $Objective_2$.
RQ_3: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the first time they read a change location and the first time they alter the change location? RQ_4: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the first time they alter code and the first time they run the altered code?

7.1 Research Questions

In this section, we have specified research questions to study the specific measurable behaviors related to the research objectives. We pose the following Research Questions (RQs) towards our objective of determining whether programmers do IA. We formulate these RQs with the idea of recording the actual behavior of programmers during debugging.

RQ_1 Do programmers use any IA tools?
RQ_2 Do programmers navigate to dependents or dependencies of the first section of code they read?
RQ_3 How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities to navigate dependencies between the first time they read a change location and the first time they alter the change location?
RQ_4 How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities to navigate dependencies between the first time they alter code and the first time they run the altered code?

RQ_1 is for $Objective_1$ and $Objective_2$. If programmers use IA tools, this indicates that programmers know the concept of IA. Using tool or not is an evidence of how programmers do IA. The rationale behind RQ_2 is that the first section of code the programmers read is the beginning of the debugging process. If a pro-

grammer reads code along the dependencies of that code, we think this indicates programmers doing IA in the beginning of debugging. So RQ_2 helps to answer $Objective_3$. Similarly, RQ_3 and RQ_4 are for $Objective_3$, too. The more activities programmers do before or after they make a change, the stronger is the evidence of programmers doing IA during the corresponding period.

7.2 Methodology

We followed an observational study methodology [2], under guidelines for case studies recommended by Runeson *et al.* [43] and Robillard *et al.* [40]. Our overall procedure is as follows.

1. **Identify bug reports.** We identified two bug reports in two different open source programs.
2. **Set up development environment in a virtual machine (VM).** For each bug report, we built a VM with Ubuntu 14.04 as the guest operating system. Inside the VM, we installed Eclipse and set up the open source program inside Eclipse. We set up the development environment inside a VM so that we can send the environment to the participants who cannot be in our lab.
3. **Install recording software in the VM.** We installed SimpleScreenRecorder[1] inside the VMs to record videos of the screens.
4. **Conduct study with participants.** For each participant, we gave her the two bug reports. In total, we obtained two videos from each participant. In this study, we do not introduce IA concept to the participants to minimize the bias. There are ten participants. For seven participants, we hired them for one hour per bug. If they are not able to fix one bug in one hour, they can give up or continue at will. For the remaining three participants, we hired them to fix the bugs, so there is no time limitation. To minimize the bias of our working setting, we explicitly asked all the ten participants to install any plugin or tools that they used in their normal working environment.
5. **Complete study.** We repeated step 4 for ten participants. For four participants that did experiments in our lab, one author was present during the study to assist with technical problems, but that author did not communicate with the participants about the bug or code. The author sat away from the participants and did not watch them, to avoid observer bias. For the other participants who did the experiments remotely, the author was available online for potential technical problems.

7.3 Subject Applications

The two subject applications we used in our study are "*PDF Split and Merge*" (PdfSam)[2] and *Raptor*[3]. In each Java program, we chose one bug report for our study. The sizes of the programs are listed in Table 1. We chose these projects because they are real, of non-trivial size, and the purposes of the projects are clear

[1] `http://www.maartenbaert.be/simplescreenrecorder/`

[2] `http://www.pdfsam.org/`

[3] `https://code.google.com/p/raptor-chess-interface/`

Table 1: two open-source programs used in our study. KLOC reported with all comments removed.

	Methods	KLOC	Java Files
PdfSam	2686	31.2	316
Raptor	1136	13.1	123

and easy to understand. We chose the two bug reports because they were fixed and can be easily understood by the programmers who do not know the programs.

With each bug report, we presented to the participants some information to let them know the program and the bug faster and easier. The information includes the purpose of the program, the entry point of the program, an input of the program that reveals the bug. For the seven participants who only have one hour, we also provided a suggested fix because of two reasons. One reason is that it took us about 2 hours to understand the bug and locate related source code. We do not want participants spend all the time in understanding the application and have not time changing anything in one hour. The second reason is that when programmers do their daily job, they often have some ideas of where might be the problem. Therefore, giving a suggested fix matches this scenario. Note that we did not provide the suggested fix for the other three participants who were required to fix the bugs no matter how much time they spend.

7.3.1 PdfSam and the Subject Bug

PdfSam is a PDF file editor. The bug we chose (id: 100) was reported in June 2014. A user reported that *PdfSam* failed to rotate a PDF file. The actual fix is shown in Figures 4 and 5. The cause of the bug was that the developer used $RandomAccessFileOrArray$ to load pdf files into memory. In this API function, some information about pdf file is lost. This lost information caused the rotation not being applied to a new pdf file. $RandomAccessFileOrArray$ is called in method $readerFor$. To fix the bug, the developer added another method called $fullReaderFor$. The $fullReaderFor$ is the same with $readerFor$ except $fullReaderFor$ calls $FileInputStream$ instead of $RandomAccessFileOrArray$. Then, inside the method that handles rotation, the developer replaced $readerFor$ with $fullReaderFor$. The fix will only affect *PdfSam* when the pdf files are requested to be rotated. Our suggested fix is directly replacing $RandomAccessFileOrArray$ with $FileInputStream$ inside method $readerFor$. The suggested fix potentially affects all the methods that calls $readerFor$. Additionally, the participants do not know whether the suggested fix works.

7.3.2 Raptor and the Subject Bug

Raptor is a graphical user interface for a Chess Server (*FICS*[4]). The bug report we used (id: 1) was reported by a developer of this program in September 2009. The developer reported that the clocks did not tick down when they should. The actual fix is shown in Figures 6 and 7. The reason for the bug is that

[4] `http://www.freechess.org/`

```
public class RotateCmdExecutor{
  ...
  public void execute(AbstractParsedCommand parsedCommand){
    pdfReader = PdfUtility.readerFor(fileList[i]);
}}
public final class PdfUtility {
  ...
  public static PdfReader readerFor(PdfFile file){
    PdfReader reader = new PdfReader(
           new RandomAccessFileOrArray(file.getFile().getAbsolutePath()), …);
    [code block 1]
    return reader;}
}
```

Fig. 4: The original code before the fix in PdfSam.

```
public class RotateCmdExecutor{
  ...
  public void execute(AbstractParsedCommand parsedCommand){
    pdfReader = PdfUtility.fullReaderFor(fileList[i]);
}}

public final class PdfUtility {
  ...
  public static PdfReader readerFor(PdfFile file){
    PdfReader reader = new PdfReader(
           new RandomAccessFileOrArray(file.getFile().getAbsolutePath()), …);
    unethical(reader);
    return reader;}

  public static PdfReader fullReaderFor(PdfFile file) ... {
    PdfReader reader = new PdfReader(new FileInputStream(file.getFile()), …);
    unethical(reader);
    return reader;}

  private static void unethical(PdfReader reader) ... {
    [code block 1]}
}
```

Fig. 5: The code after the actual fix in PdfSam.

the states of the game were not correctly updated. The cause is in the method *updateNonPositionFields*, which is supposed to update the states of the game. The problem is that the game's state is not updated with $IS\_CLOCK\_TICKING\_STATE$ when the game is at $EXAMINING\_STATE$.

To fix this bug, a developer added a patch in the method *updateNonPositionFields*. By adding an if statement in *updateNonPositionFields*, *Raptor* is forced to up-

```
public class FicsUtils implements GameConstants {
  public static void updateNonPositionFields(Game game, Style12Message message) {
    switch (message.relation) {
      case Style12Message.EXAMINING_GAME_RELATION:
        game.setState(Game.EXAMINING_STATE);
        break;
      case …
      break;
    }
  }
}
```

Fig. 6: The original code before the fix in Raptor.

```
public class FicsUtils implements GameConstants {
  public static void updateNonPositionFields(Game game, Style12Message message) {
    switch (message.relation) {
      case Style12Message.EXAMINING_GAME_RELATION:
        game.setState(Game.EXAMINING_STATE);
        break;
      case …
        break;}
    if (message.isClockTicking) {
    game.addState(Game.IS_CLOCK_TICKING_STATE);
    } else {
        game.clearState(Game.IS_CLOCK_TICKING_STATE);
    }
  }
}
```

Fig. 7: The code after the actual fix in Raptor.

dated the $IS\_CLOCK\_TICKING\_STATE$ every time $updateNonPositionFields$ is called. The suggested fix we provided is a change that committed in the repository with this fix. The change is related to updating states at $EXAMINING\_STATE$, but the change does not fix the bug. The change is in Class $ExamineController$, which is the chess board controller when the game is in $EXAMINING\_STATE$.

7.4 Participants

We recruited seven programmers for one hour per bug in our study. One participant was not familiar with Eclipse, so we discarded the videos of the participant. The remaining six participants are listed as Participants 1 to 6 in Table 2. Additionally, we hired another three programmers to fix the bugs with no time limit and no suggested fixes, who are listed as Participants 7 to 9 in Table 2. Most of the participants we hired are experienced programmers. The average experience level of the nine programmers is 10 years on paid positions. Due to time and resource

Table 2: Participants in In-depth study

	Java Experience	Years Exper.	Organization
Participant 1	2	4	Ph.D. program University of Notre Dame
Participant 2	0 (5 years in OO)	5	Computing resource center University of Notre Dame
Participant 3	12+	12+	withheld for privacy
Participant 4	10	5	Ph.D. program University of Notre Dame
Participant 5	15	30	withheld for privacy
Participant 6	7	9	Ph.D. program Peking University
Participant 7	5	4	A Bank IT department
Participant 8	5	5	withheld for privacy
Participant 9	12	12	A financial service software company

limits, we decided not to hire more programmers. Table 2 list the participants and their experience.

7.5 Threats to Validity

Like any study, our study has threats to validity. First, the two programs in the study may not be representative. We mitigate this threat by choosing two types of programs. One program is an online application, and the other is offline. Therefore, our results are not limited in one program type. Second, the two bug reports may not be representative. We mitigate this threat by choosing real bugs from the real repositories, so that the bugs are at least realistic and may happen in practice. Third, we chose Eclipse as the working environment, which may affect the behavior of the programmers in debugging. However, we think Eclipse is very widely used, and can represent general IDEs.

Fourth, the participants are not the developers of the selected programs. Therefore, their behavior may differ from the real developers of the applications. We mitigate this threat by offering some information about the programs and the bugs, see Sections 7.2. Although the participants have some information about the application and the bug, the participants still may do more activities for program comprehension than the real developers of the applications. So there may be more IA activities done in the study than in the real world. Also, three participants are Ph.D. students, whose behavior may be different from the programmers in industry.

7.6 Reproducibility

To ensure reproducibility by independent researchers, we put all the data via an online appendix, including the VM images, the videos, the logged activity sequences, the recognized change locations, the scripts for data analysis, and the results:

`http://nd.edu/~sjiang1/IA-study.htm`

8 In-Depth Study Data Collection

Before we processed the videos, we discarded the two videos of one participant because he was not familiar with Eclipse. Then, we processed the remaining 18 videos as follows.

8.1 Recorded activities

We watched the videos and recorded all activities with time-stamps, such as "open a file", "edit a line of source code", and other activities in IDE. Additionally, if an activity modifies source code in any way, we also recorded the file name and the line number that the activity modifies. For a complete list of the activities, please refer to the online appendix in Section 7.6. All the activities we recorded in a video formed one sequence of activities. For 18 videos, we obtained 18 sequences.

8.2 Found change locations

We found the change locations in the sequences. To find change locations, we first found all the activities that modify source code in the sequences. We removed the modifications that do not have any impact. These modification includes: adding/deleting comments, formatting (adding/deleting spaces and lines), modifications that were undid immediately (for example, in the middle of the modification, the programmer realized that this modification would not work and undid the whole modification.) It is possible that there are some changes that may seem to have impacts, but actually they do not have any impact. We took these changes into account, because without inspection, programmers do not know that the changes have no impact either.

Besides the changes that have no impact, we also excluded a special case of the modifications, which is undo. Often the programmers undid changes they made because of the two reasons: 1) they ran the program and the change did not have expected impact on the output of the program, and 2) the changes are made accidentally. We ignored these "undo" changes because their impacts are discarding the impacts of the changes, which programmers may know about.

There are another two special changes that we did not take into account. One programmer created a new test project. The other programmer created a new test class. Because our scope of the changes is limited to the changes to the existing code, we excluded these two changes in our results.

From the remaining modifications, we collected the file names and the line numbers. If some lines of code are successive in the same file, we grouped these lines into one "change location". Even the lines of code that were modified at different times, we grouped them into one "change location" because these changes are related to each other. In one video, a programmer may have multiple change locations.

Table 3: The Time Labels

Label	Description
$FReT$	The first time that a programmer reads a change location.*
FMT	The first time that a programmer modifies a change location.
FRT	The first time that a programmer runs the program after FMT.
LMT	The last time that a programmer modifies a change location.
LRT	The first time that a programmer runs the program after LMT.

* Note that it is impossible to know the change location before the location is altered. So we found FMT first, then we found $FReT$.

Table 4: Eclipse navigational functionalities

Category Name	Eclipse Actions	Description
Call hierarchy	open call hierarchy	show the callers of a method
Type hierarchy	open type hierarchy	show the supertype/subtype of a class
Declaration	open declaration	open the definition of a class/method/variable
Implementation	open implementation	open the implementation of a method
	open super implementation	open the super implementation of a method
References	find references	show all the references in Workspace/Project of a class/method/variable.
Search	text search	search the exact text in Workspace/Project
	java search	search all the occurances of a java element in Workspace/Project
	text find	find the exact text in the current file

8.3 Found important times

First, we logged the first times programmers read a change location ($FReT$). Then, we logged the times when each change location was edited. Programmers may modify the location multiple times and the number of modifications varies. For our research questions, we only logged the first modification time (FMT) and the last modification time (LMT). Furthermore, for each modification time, we logged the first time that the program was executed after that modification (FRT and LRT). In summary, Table 3 lists all the time labels we marked.

8.4 Measured distances

We measured elapsed time in seconds, the number of files visited, and the number of times that Eclipse functionality is used between three periods of time, which are from $FReT$ to FMT, FMT to FRT, and LMT to LRT. We counted all the IDE functionalities that help programmers navigate dependencies of a code element, including "open call hierarchy of" a method, "open declaration of" a class, and so on. We listed all the navigational functionalities in Table 4. Note that although "search" is listed in Table 4, "search" is not counted as the IDE functionality that help programmers navigate dependencies, so we did not count "search" in answering our research questions.

Table 5: Participants in In-depth study

	PdfSam	Raptor
Participant 1	fixed	not fixed with a major progress
Participant 2	fixed	not fixed with a major progress
Participant 3	fixed	not fixed with a major progress
Participant 4	fixed	not fixed
Participant 5	fixed	not fixed
Participant 6	fixed	not fixed with a major progress
Participant 7	fixed	fixed
Participant 8	fixed + refactored	fixed
Participant 9	fixed	fixed

9 In-Depth Study Results

In this section, we will describe the quality of the patches that our participants created. Then, we will begin to answer the question *do programmers do change impact analysis?* From a high level, we found the evidence that the programmers did IA but did not use IA tools: 1) the programmers did not use IA tools; 2) the programmers did IA before they made changes; 2) the programmers **ran** the programs after they made changes. In the rest of the sections, we will explain the statistical details of how we came to this conclusion.

9.1 Quality of Patches

For PdfSam, Participants 1 to 6 applied the exact patch of the suggested fix to the program. Participant 7 figured out a way to fix the problem in a different method. Participant 8 made a similar patch to the actual fix, so this patch has less impact than the other patches. Participant 9 applied the exact patch of the suggested fix.

For Raptor, Participant 1 to 6 did not fix the problem. However, Participants 1, 2, 3, and 6 made a major progress where the clocks began to tick down but at a wrong time. Participants 7 and 8 made similar patches to the actual fix. Participant 9 also made a similar patch, but s/he made sure the new code is only accessed when the game is under $EXAMINING\_STATE$. So the last patch has less impact than the other patches.

9.2 Example Result

This section will explain the result for a change location in PdfSam as an example, see Table 6. The change location is at line 94-95 in the file *PdfUtility.java*.

From Table 6, the participant read only one file between the first time s/he read the change location and the first time s/he altered the change location ("$FReT$ to FMT" in Table 6). This file is the file containing the change location. This indicates he did not do IA across different files before s/he made the change. Additionally, there is no Eclipse functionality used to navigate source code between $FReT$ and FMT, which further indicates the programmer did not do much IA in this period within the file. For FMT to FRT and LMT to LRT, the numbers are

Table 6: The Measured Distances for One Change Location in *PdfSam*. See Section 9.2 for the description of this example.

	FReT to *FMT*	*FMT* to *FRT*	*LMT* to *LRT*
Time in seconds	4	14	16
# of Files	1	1	1
# of Dep. Navs.	0	0	0

Table 7: The overall time in seconds, the number of visited files, and the number of uses of IDE functionalities for 18 debugging session

Participant Id		1	2	3	4	5	6	7	8	9
Pdfsam	Time in Sec.	7800	3051	3314	453	3544	1555	4295	13367	6168
	# of Files	33	15	6	2	13	9	12	34	24
	# of Func.	15	21	4	0	36	21	38	46	37
Raptor	Time in Sec.	3119	4537	3352	3543	4941	4081	2809	6546	5343
	# of Files	15	24	6	6	23	9	15	21	24
	# of Func.	18	30	1	1	46	60	23	34	33

same, which indicates the programmer did not do much IA before s/he ran the program.

9.3 Aggregate Result

From the 18 videos, we found 31 change locations. Table 7 listed the total time for each video, the number of visited files, and the number of the times that IDE functionalities were used. The average distances of the 31 changes in three time periods are presented in Table 8. We also presented the average results for each bug in Tables 9 and 10.

A big difference between Tables 9 and 10, the participants without the suggested fixes spent more time in *FReT* and *FMT* in PdfSam than in Raptor. However, in the same period of time, the number of files visited and the number of uses of functionalities are similar in PdfSam and Raptor. This indicates that the participants without the suggested fixes read similar amount of code before they made the changes, but they spent more time in reading the code in PdfSam.

For Figures 8, 9 and 10, each figure corresponds to a metric we measured, i.e., time length in seconds, the number of files visited, and the number of IDE functionality uses. There are three boxplots in each figure. Each boxplot represent a time period, that is *FReT* to *FMT*, *FMT* to *FRT* and *LMT* to *LRT*. In general, we can see the programmers put the most effort before they make any modification for a change location, and they put the least effort after they make changes and before they run the programs.

Table 8: The Average Distances for the 31 Changes. The median distances are in parentheses.

		FReT to *FMT*	*FMT* to *FRT*	*LMT* to *LRT*
Time in Sec.	With Suggested Fix	472 (137)	119 (48)	29 (17)
	No Suggested Fix	1729 (1247)	53 (46)	19 (19)
	All	877 (416)	97 (46)	27 (17)
# of Files	With Suggested Fix	4 (3)	2 (2)	2 (2)
	No Suggested Fix	10 (9)	1 (1)	2 (2)
	All	6 (4)	2 (1)	2 (2)
# of Func.	With Suggested Fix	4 (0)	2 (0)	0 (0)
	No Suggested Fix	11 (9)	0 (0)	0 (0)
	All	6 (1)	1 (0)	0 (0)

Table 9: The Average Distances for the 17 Changes for PdfSam. The median distances are in parentheses.

		FReT to *FMT*	*FMT* to *FRT*	*LMT* to *LRT*
Time in Sec.	With Suggested Fix	316 (226)	88 (56)	40 (20)
	No Suggested Fix	1979 (1633)	39 (40)	19 (19)
	All	903 (416)	71 (40)	31 (20)
# of Files	With Suggested Fix	4 (4)	2 (2)	2 (2)
	No Suggested Fix	10 (7)	1 (1)	2 (2)
	All	6 (4)	2 (1)	2 (2)
# of Func.	With Suggested Fix	2 (1)	1 (0)	0 (0)
	No Suggested Fix	11 (7)	0 (0)	0 (0)
	All	5 (1)	1 (0)	0 (0)

Table 10: The Average Distances for the 14 Changes for Raptor. The median distances are in parentheses.

		FReT to *FMT*	*FMT* to *FRT*	*LMT* to *LRT*
Time in Sec.	With Suggested Fix	643 (133)	156 (40)	21 (9)
	No Suggested Fix	1354 (1234)	74 (67)	n/a
	All	846 (275)	131 (64)	21 (9)
# of Files	With Suggested Fix	5 (1)	2 (1)	1 (1)
	No Suggested Fix	9 (9)	1 (1)	n/a
	All	6 (3)	2 (1)	1 (1)
# of Func.	With Suggested Fix	7 (0)	2 (0)	0 (0)
	No Suggested Fix	10 (10)	0 (0)	n/a
	All	8 (1)	2 (0)	0 (0)

9.4 Difference between the study with the suggested fixes and the study without the suggested fixes

In Table 5, Participants 1 to 6 fixed the PdfSam's bug by using our suggested fix. But for Raptor, all the participants fail to fix the bug. This indicates that the quality of the suggested fixes affects whether the participants could fix the bugs. If the suggested fix is similar to the actual fix, the participants can easily fix the bug without extra effort to refactor the program for better code quality. If the

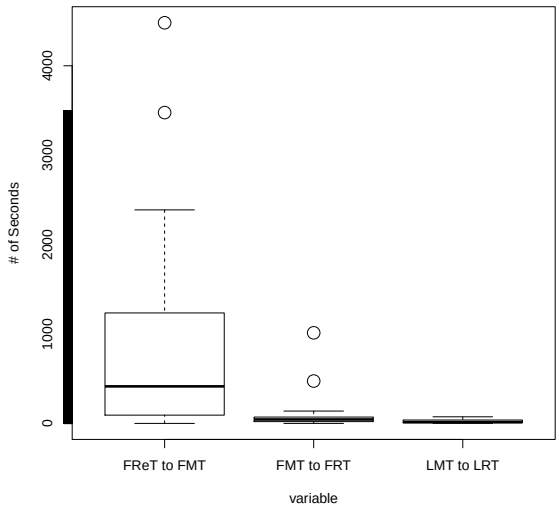

Fig. 8: The distribution of the time lengths

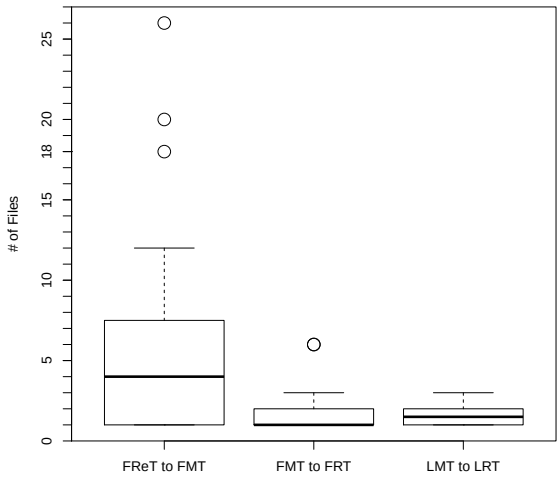

Fig. 9: The distribution of the number of visited files

suggested fix is not in the code where the actual fix is, the participants cannot fix the bug in a limited time.

Participants 7, 8 and 9 fixed both the bugs in PdfSam and Raptor, because they were asked to fix the bugs to finish the task without a time limit. Note that only Participant 8 refactored PdfSam's code like the actual fix in the repository.

In Tables 8, 9, and 10, the biggest difference between Participants 1-6 and Participants 7-9 is during $FReT$ to FMT. Participants 7-9 spent more time, visited more files and used Eclipse functionality more times before they modify a location. The difference is understandable because the participants have less information (no suggested fixes) than Participants 1 to 6.

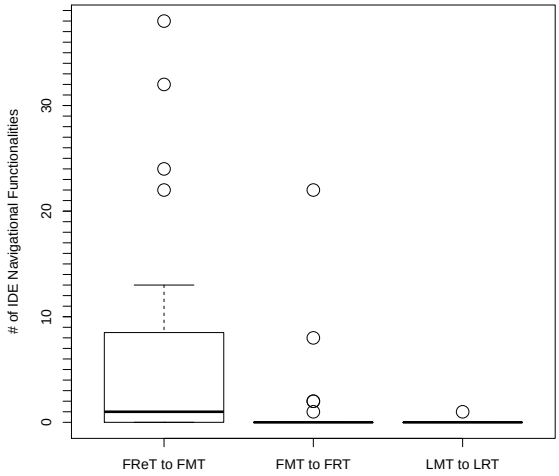

Fig. 10: The distribution of the number of IDE functionality uses.

9.5 RQ_1: Do programmers use any IA tools?

No programmer used any IA tools except code navigation functionalities in Eclipse. Note that we explicitly told participants that they can install any plugin. Our interpretation is that the programmers tend not to use IA tools. One possible reason is that for fine-grained IA, such as statement-level IA, programmers often do it by comprehending code themselves. For coarse-level IA, such as method-level IA, the code navigation functionalities, such as "open call hierarchy", are enough. Not using IA tools indicate that programmers are not familiar with the techniques proposed for IA tasks.

9.6 RQ_2: Do programmers navigate to the dependencies or the dependents of the first section of code they read?

In the 11 out of 18 debugging sessions, the programmers in our study did not navigate to dependents and dependencies of the first section of code that they read. Therefore, we think the programmers do not prioritize IA at the beginning of the debugging process, which should be fault localization. In 11 of the 18 videos, the programmers jumped from the first file they read to the second file.

In the remaining seven videos, we found evidence that the programmers do IA. Most of them do dynamic IA by following the execution. They ran the program with a debugger step by step. Two of them did static IA by using "open declaration of" a method call in the first file.

9.7 RQ_3: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the times of the first read and the first modification?

Between the first read ($FReT$) and the first modification (FMT) of a change location, the programmers had more code navigations than they did in the other periods, see Table 8. We think this indicates programmers doing IA in a static way. The main task in this period is to decide whether or how the change location should be changed. To do this task, the programmers need to understand what the change location affects.

When we did not provide the suggested fixes, the programmers visited an average of ten files. However, in the videos that we provided the suggested fixes, the number of visited files in this period is small. On average, the programmers only navigated less than four files. By comparing these two results, our interpretation is that the programmers tend to try changing the code as soon as possible with the minimum amount of program comprehension.

In Figure 9, most of the programmers read fewer than twelve files. Our interpretation is that reading twelve files should be sufficient for programmers figure out an initial change for a location. The outliers at $FReT$ to FMT in Figures 8, 9 and 10 correspond to four change locations.

For two change locations, the programmers altered other change locations first after they read the change location. For another outlier, the programmer comprehended the entire program before she made any modification. The activities between the first read and the first modification include the tasks to comprehend the overall structure of the program. These two outliers are similar to the cases of inattentional blindness reported by Robillard *et al.* [40], which is the situation where programmers do not intend to change a location when they read the location at the first time.

For the fourth outlier, we believe this is the case where the programmer did extensive IA (18 files visited, 22 times of using IDE functionalities) after s/he locate this line and before s/he make a change to the location.

9.8 RQ_4: How long do programmers take, how many files do they visit, and how many times do they use IDE functionalities between the times of modification and the first run of the altered code?

The programmers in our study almost always ran the programs immediately after they made the last modification of a change location. In 75% of the cases, the programmers navigated to no more than two files (including the files that contain the change locations) in Figure 9 during both LMT to LRT and FMT to FRT.

The numbers of uses of the Eclipse navigational functionalities are similar in the two periods, too. FMT to FRT on average have one navigational functionality used. And in LMT to LRT, there is almost no functionality used. Our interpretation is that the programmers tend to run programs immediately once they made the changes.

The average time between FMT and FRT is overall 97 seconds. Because the time we recorded for modifications is the beginning time of the modifications, the

period between FMT and FRT includes the time that the programmers spent on actually changing code.

The outliers at FMT to FRT in Figures 8 and 9 correspond to two change locations. For one change location, the first edit of the location is not intended to change the program, but to help the programmer to "open declaration of" a method. The call of the method was commented out, in order to use Eclipse navigational functionality, the programmer uncommentted this code, so that s/he could click to the declaration of the method. For the other outlier, the programmer navigated to other places and inserted some log function calls, so that when the program runs, it would output more useful information. This case shows that the programmer did IA after s/he made the change.

9.9 Qualitative Results

For the last three participants that we hired to fix the bugs, we also did interviews after they finished their jobs. The interviews were conducted in an online chatting tool, which is provided by the online hiring platform we used.

9.9.1 knowledge of IA

One participant reported that he knew the term, "but have not seen anything related to that". In his understanding, he thought IA is the question about "what would be the impact of some change that I want to make". The other two participants do not know IA, but said they did IA after we introduced IA to them. IA is introduced as "Change-Impact Analysis is a task of finding the source code which is affected by the source-code change that you are going to make."

When the programmers were asked about IA in their daily work, they often refers to post-change IA. "I always try to understand how I can influence the code. If I'm uncertain about my changes I can make a list of influenced part and give it to our QA[Quality Assurance] engineers. They are checking all cases." (Participant 8) "I have to make sure that my change will not cause bugs or other problems for other parts of the project or system's components ... " (Participant 9) "... I get that a lot in my work - as the systems quite often use global variables, that are a mess to track." (Participant 7)

9.9.2 practice of IA

From what the participants said, the need of doing IA is different for different fixes. "In Raptor project - not much, because the fix seemed to be fairly isolated. In PDF project - yeah - as I still had some doubts about what's the lifecycle of PdfDictionary and what uses it" (Participant 7) "in pdfsam I understood that changing Pdfutils class can lead to bugs in other ways of manipulating pdfs. .. And I've tryed to fix the bug not introducing any changes in other components ... In raptor it seems clear that the input params are not processed correctly. I've checked places where this state of game is used and it seems that my changes can't influence the other code." (Participant 8)

The participants also mentioned to do IA by exploring the methods being called. "Example: I change a method of a class. After my changes I have to find

direct and indirect calls of this method and make sure that system will be ok after my changes." (Participant 9) Note that the programmers also mentioned to rely on quality assurance team to do IA. "If I'm uncertain about my changes I can make a list of influenced part and give it to our QA engineers. They are checking all cases." (Participant 8)

9.10 Comparison to Other Results of Code Changing Tasks

Many research projects were conducted on the topic of code changing [46,25,23]. Sillito *et al.* [46] studies three research questions: 1) What knowledge programmers need when they change code? 2) How programmers get the relevant information? 3) How well the existing tools help programmers get the knowledge? They observed 27 sessions of programmers doing code changing tasks. They identified four categories of questions that programmers asked when they did the tasks. In the four categories, there are two categories that are related to our results.

The first category is "finding focus points". In this category, Sillito *et al.* noted five questions, like "Which type represents this domain concept or this UI element or action?" [46]. In our in-depth study specifically, the participants asked the question "where is the code that represents this action?". Particularly for *PdfSam*, all the three participants in the interview said they wanted to locate the code that rotating the pdf file. "The first step is select project modules which may contain the bug. E.g. for PdfSam project I looked for the "Rotation" related modules." (Participant 7) "I knew from description that rotation failed and I was finding places where rotation is using." (Participant 8) "... I tried to find the code that handles rotation first" (Participant 9)

The second category is "expanding focus points". This category has 15 questions, such as "Where is this method called or type referenced?" and "What does the declaration or definition of this look like? " [46]. In the videos, we recorded the uses of Eclipse navigational functionalities. To compare our results to Sillito *et al.*'s report, we generated a snapshot of the uses of Eclipse navigation functionalities. [5] We listed the navigational functionalities in Table 4. In the Tables 11 and 12, we listed the number of the uses of each Eclipse navigational functionality. Note that we count only the action of a functionality. For example, for Participant 1, s/he opened type hierarchy once, but s/he might check the subtypes/supertypes multiple times as long as s/he does not close the result window. Among the six functionalities, opening declaration and searching are two most often used functionalities.

Ko *et al.* [23] performed a study where the participants were asked to do two debugging tasks and three enhancement tasks. The study found three activities. The first activity is searching. We also found searching activities in the participants. The number of searches is reported in Table 11. Ko *et al.* also found that the participants lost track of relevant code because the navigational tools were used for various purposes. We found one case that is consistent with this finding. Participant 9 opened the declaration of "getState" seven times, and "isInState" three times in a debugging session. This shows that although the participant had

[5] This data is preliminary and not included when we answer our research questions.

Table 11: IDE Activities in Debugging *PdfSam*

Participant Id	Call Hierarchy	Type Hierarchy	Declaration	Implementation	References	Search
1	5	0	10	0	0	0
2	0	0	20	1	0	0
3	0	0	4	0	0	0
4	0	0	0	0	0	0
5	0	0	33	0	3	4
6	0	0	20	0	1	3
7	0	0	21	15	2	2
8	0	0	24	21	1	47
9	16	0	21	0	0	28

Table 12: IDE Activities in Debugging *Raptor*

Participant Id	Call Hierarchy	Type Hierarchy	Declaration	Implementation	References	Search
1	7	1	10	0	0	6
2	0	0	30	0	0	16
3	0	1	11	0	0	5
4	1	0	0	0	0	17
5	0	0	38	0	8	9
6	0	0	55	0	5	4
7	0	0	14	5	4	20
8	0	0	31	3	0	2
9	11	0	22	0	0	7

visited the declaration of "getState", s/he still had the need to find the information again.

LaToza *et al.* [25] studied both novices and experts and observed their behaviors when they were asked to improve the design of the programs. LaToza *et al.* reported that the participants made "path choice decisions", which is choosing to explore the locations that may have useful information. We have similar notes from our participants. "Firstly I found an exception in pdfsam and it misleads me. I spend time to fix this exception." (Participant 8) "there seems to be a swt gui thread, the code hits this thread and pauses on a lock it is like it is waiting for a change of state so the thread can be unlocked ... investigating Game.IS_CLOCK_TICKING_STATE with the debugger"(Participant 3, Raptor, 33:06, in the notepad shown in the video. We did not require the participants to take notes. This is what we observes from the video.)

We also found the evidence for the participants confirming or disconfirming hypotheses, which is also reported by Latoza *et al.*. "For the Raptor project I initially had a hypothesis that there should be some event handler that does not get activated once you unpause, ... and tried to confirm/deny that first." (Participant 7) This is also consistent with the debugging model we introduced in Section 4.

10 Breadth Study Design

In this section, we will describe our breadth study, in which 35 professional programmers filled out our online survey. We will cover our research questions, online survey format, methodology, survey software, participants, and threats to validity.

10.1 Research Questions

In this section, we designed five research questions to study the measurable factors related to the research objectives.

RQ_5 Do programmers know the term of IA?
RQ_6 Do programmers self-report doing IA?
RQ_7 Do programmers self-report using IA tools?
RQ_8 After programmers apply a possible fix to a bug, do they run the program first or check the impacts of the fix first?
RQ_9 After programmers apply a possible fix to a bug, what functionality of the IDE do they use?

The rationale behind RQ_5 is that programmers may not know the term of "change impact analysis", because it is a term heavily used in academia but perhaps not in industry. If the programmers do not know the term, they may not do IA intentionally. The rationale behind RQ_6 is to determine the self-reported level of IA. The rationale behind RQ_7 is that if programmers use IA tools, we know that programmers do IA. If programmers do not use the tools, it is necessary to further investigate what prevents IA techniques from practice.

RQ_5 to RQ_7 are self-reported questions. In addition to these self-reported questions, we have RQ_8 and RQ_9, which focus on the behavior of programmers. The rationale of RQ_8 is that if programmers run a program immediately after they apply a possible fix, they prioritize dynamic IA after they make changes. RQ_9's rationale is that the different functionalities of the IDE that programmers use is an evidence of whether or how they do IA. For example, if programmers click "call hierarchy" of a method, it indicates that the programmers do IA by inspecting call graphs.

10.2 Methodology

We used a survey study methodology, and followed the steps done by LaToza and Myers [26]. Our procedure was as follows:

1. **Formulate research questions.** We based our research questions on related literature on IA, such as the study done by Rovegård et al. [42]
2. **Design survey questions.** We aim to rely as little on self-reported level as we can. Therefore, we put RQ_8 and RQ_9 first. For these two questions, we provide a real situation to the participants: a real project and a real bug report. Then, we provide a suggested fix, so that the participants have an initial change location. Under this scenario, we ask the participants what they will do next (see Section 10.3 Item 3). This question corresponds to RQ_8. Furthermore,

we give an image of an Eclipse IDE. And we ask the participants to point out where they will go next after they make a change in source code (see Section 10.3 Item 4). This question corresponds to RQ_9. After these questions, we ask participants about IA, including whether they know about it (RQ_5), whether they do it (RQ_6), and whether they use IA tools (RQ_7). With this order of the questions, we mitigated the biases that may occur in the previous questions.

3. **Distribute the survey.** We recruited participants by *convenience sampling* [42]. We selected participants based on their programming experience and their availability. We targeted professional programmers in industry, especially in large software companies, see Section 10.4. We obtained more than 30 participants, because we want to have statistically significant results.
4. **Collect and analyze the results.** We used an automated process of collecting responses provided by Qualtrics Survey Software. In this way, we ensured maximum accuracy and maximum response rate. We have one hypothesis based on our results in Sections 11.4. For this hypothesis, we used Pearson's Chi-square test and Fisher exact test [32]. We used Fisher exact test because some numbers in our data are smaller than five. In such cases, Fisher exact is more accurate than Pearson's Chi-square test.

10.3 Online Survey Format

We built our online survey by Qualtrics Survey Software[6]. This online survey has six web pages. A deactivated survey is available at the online appendix in Section 10.6. The format of this survey is as follows:

1. **The first page** asks about programmers' professional experience.
2. **The second page** shows a bug report, which is the bug report of *Raptor* described in Section 7.3.
3. **The third page** suggests a fix to that bug, and asks programmers whether they will run the fix or check the impacts of the fix first (RQ_8).
4. **The fourth page** shows an image of an Eclipse IDE. In the image, there is an IDE, where the suggested fix is applied, see Figure 11. There are 34 clickable areas in the image and each area corresponds to a functionality of the IDE. We asked programmers to click on the area that they will go next after they apply the fix (RQ_9).
5. **The fifth page** asks programmers whether they know about IA (RQ_5), whether they do IA (RQ_6), and whether they use any automated tools for IA (RQ_7). If the programmers do not know about IA, we used the following exact words for defining IA:
   "Change-Impact Analysis is a task of finding the source code which is affected by the source-code change that you are going to make."

[6] `http://www.qualtrics.com/`

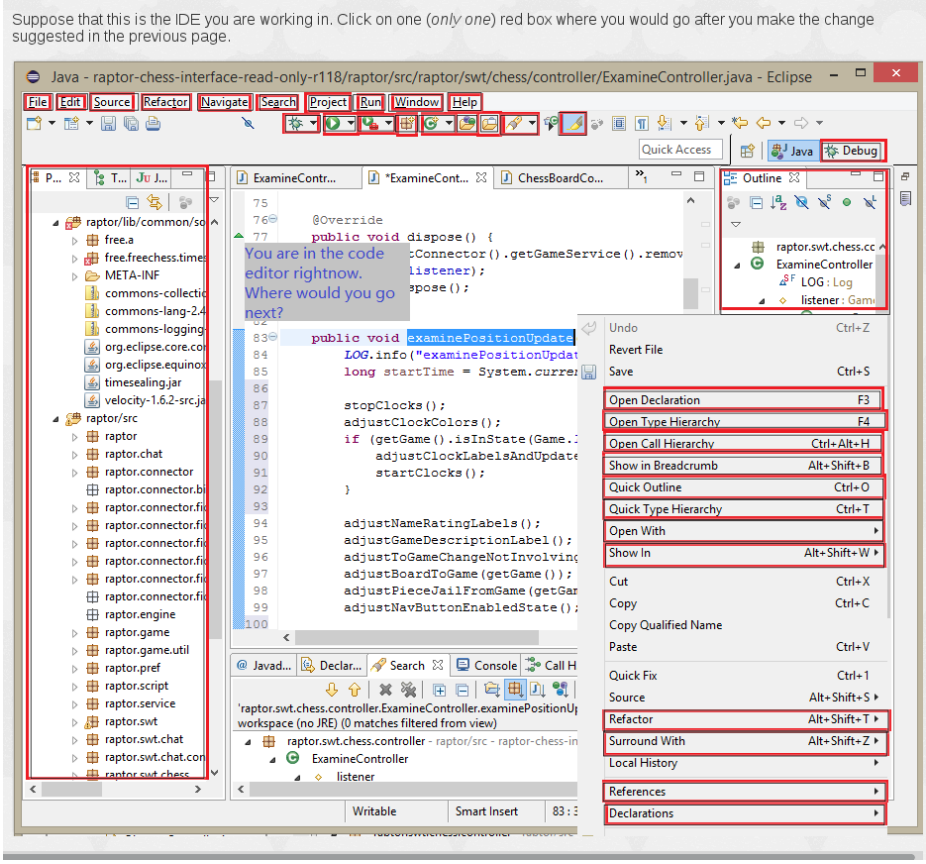

Fig. 11: Page 4 of the survey for our breadth study in Section 10. This page shows an image of an Eclipse IDE. The red boxes are the areas that participants can click. The areas include the window of Package Explorer, the button of Open Declaration in the context menu, and other menu options and Eclipse windows. We asked each participant to click on the area that she will go next after she makes a fix. The reason for this page is to answer RQ_9.

10.4 Participants

There are 35 participants. All of them self-reported that they program in Java comfortably. Seventeen of them have more than four years of professional experience in industry. Fourteen of them have one to four years of professional experience. Four of them have less than one year of professional experience. Twenty-four of them are working or worked in industry. Ten of them are graduate students. One of them programs as a hobby. In ten graduate students, six have intern experience as a programmer. The professional programmers are from the Computing Research Center (CRC) at the University of Notre Dame, and other various large IT companies.

10.5 Threats to Validity

As with any study, our work has threats to validity. The project and the bug we used in the survey may not represent a general bug scenario. Also, the participants may not understand the project or the bug, so their responses may not represent their usual behavior in their own projects. However, in the survey, we do not ask the participants to actually fix the bug. Therefore, the participants do not need to consider the details of the project and the bug. Additionally, we used an image of Eclipse, which may not be the usual IDE that the participants use. We mitigate this threat by choosing Eclipse, which is one of the most often used IDE for Java programs.

Besides these threats, we also have response bias in the survey. To mitigate the bias, we introduce the term, change impact analysis, at the end of the survey. Additionally, the population of the participants may not be representative. We attempt to mitigate this by having a sample of 35 programmers who have various programming experience. From this sample size, we are able to obtain statistically significant results.

10.6 Reproducibility

A deactivated survey, the result, and the scripts for data analysis are available in the online appendix:
`http://nd.edu/~sjiang1/IA-study.htm`

11 Breadth Study Results

In general, our breadth study shows that programmers do not use IA tools but do dynamic IA after they make changes. In the rest of the sections, we will explain how we came to this conclusion. The overall results are shown in Figures 12 and 13.

11.1 RQ_5: Knowledge of Term of Change Impact Analysis

We found evidence that the majority of the programmers do not know what IA is. In our 35 participants, nine of them reported that they knew the term of IA. Twenty-six (77%) of the programmers did not know the term IA before this study. Our interpretation is that the academic theory of IA has not penetrated education such that the programmers leave school knowing what IA is.

11.2 RQ_6: Self-reported Level of Change Impact Analysis

According to our results, the majority of the programmers reported they did IA. Twenty-eight (80%) of the 35 participants reported that they did IA. We think the high self-reported level indicates that the programmers think IA is an important and a necessary process in programming.

However, we found evidence showing that programmers who know IA are less likely to think they do IA. In the nine programmers who knew IA, four (44%) programmers reported that they did IA. In the 26 programmers who did not know IA, 24 (92%) programmers reported that they did IA. We think whether programmers report they do IA is related to whether they know the IA concept. We tested the hypothesis in Table 13. The null hypothesis is that reporting doing IA and having heard of IA are independent. With a critical alpha level of 0.05, the null hypothesis is disproved using Pearson's chi-square test and Fisher exact test. Therefore, whether the programmers report they do IA or not is dependent on whether they knew the concept of IA before the study. Our interpretation is that the theory of IA may be different from the industry practice. When programmers know the academic term of IA, programmers may refer IA to the IA process in the literature that is described in Section 3.1. They do not report they do IA because they do not follow the IA process. For those who do not know the academic term, they report that they do IA because they consider IA as a general concept.

11.3 RQ_7: Use and Knowledge of IA Tools

Our results show that the majority of the programmers do not use IA tools, which matches our finding in Section 9.7. Of the 28 programmers who reported that they did IA, 23 programmers (82%) reported that they did not use IA tools. Three programmers (11%) reported that they were not sure whether they used tools or not. One possible explanation is that the three programmers use IDE functionalities to do IA, such as "open call hierarchy of" a method, and they are not sure whether these functionalities are IA tools or not. Overall, the result shows that most of the programmers do not use IA tools.

11.4 RQ_8: First Step after Having a Possible Fix

In this research question, we asked what the programmers do directly *after they apply a possible fix*. Of the 35 participants, 17 (49%) programmers chose to run the program to see whether the fix works; 17 (49%) programmers chose to check the effects of the change; One programmer chose neither.

An explanation for running the program first is that programmers can find out whether the fixes work or not by running the program. If the fixes do not work, the programmers may undo the fixes. In this case, the programmers may not need to further investigate these changes. So programmers may want to run the program first.

11.5 RQ_9: First IDE Functionality Used after a Possible Fix

To process the result, we divided the functionalities listed in the Figure 13 into three categories. The first category is "running" the program. There are 23% programmers who chose to run the program. By running the program, programmers can observe the impacts of a change on the outputs. So running the program is one of the simplest dynamic IA methods.

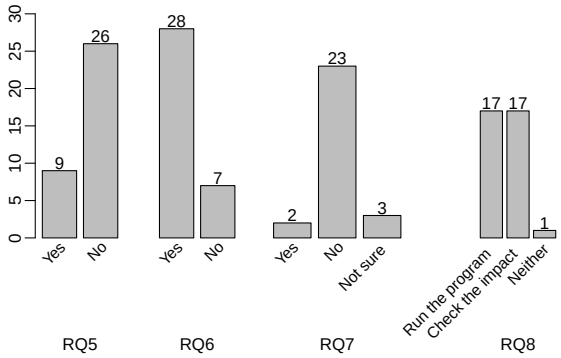

Fig. 12: The results for RQ_5 to RQ_8 in our breadth study in Section 10. RQ_5: do programmers know the term of change impact analysis? RQ_6: do programmers think they do change impact analysis? RQ_7: do programmers use change impact analysis tools? RQ_8: after programmers make a possible fix, do they run the program first or check the impacts of the fix first?

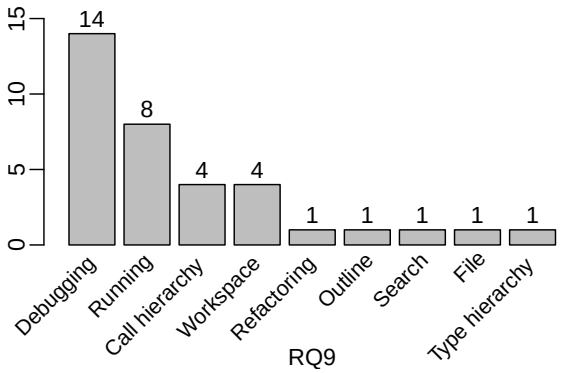

Fig. 13: The result of RQ_9 in our breadth study in Section 10. RQ_9: after programmers make a possible fix, what IDE functionality do they use?

The second category is "debugging" the program, which is running the program with a debugger. We believe debugging is a type of dynamic IA because programmers can follow dependencies from the change location in execution. Figure 13 shows that 40% of the programmers run in debug mode directly after they make a change. The first and second categories cover 63% of the participants, which indicates that most programmers use dynamic approaches to do IA after they make changes.

The third category includes the functionalities that may help the programmers check the effects of the change for multiple places. These functionalities are opening "call hierarchy", navigating "workspace", "navigating outline", "searching", and "opening type hierarchy". Our results show that there are 31% of the participants do IA by these static methods.

In summary, most programmers do IA after they make a change. Among the programmers who do IA, more than half of them do IA dynamically.

Table 13: The counts of the programmers classified by self-reported level of IA and their knowledge of IA.

	Heard of IA	Not heard of IA	All
Do IA	4	24	28
Do not do IA	5	2	7
All	9	26	35

The null hypothesis: "Do/Do not do IA" and "Heard/Not heard of IA" are independent.
Pearson's chi-square statistic = 9.573, p-value = 0.002
Fisher exact test, p-value = 0.006

11.6 Qualitative Results

We received nine comments related to IA and software engineering in our breadth study. Three comments expressed the needs for IA tools. For example, one programmer commented "Repeating the specific bug in a large project is time consuming. I would appreciate if Eclipse community can offer a tool to help me figure out which variables will be changed before tracing the code in debug mode." Three programmers described their IA practice in the comments. One practice is single-step debugging. Another practice is "in a manual trial and error manner". The third comment said "I would compile the fix just to see what happens first (see if it addresses the bug). If so, I would then do change impact analysis to determine if this produces new bugs."

12 Discussion

In this section, we will discuss our three research objectives based on our results of the research questions in the two studies.

12.1 $Objective_1$ What knowledge do programmers have of research activities in IA?

Programmers do not know the term Change Impact Analysis. In the survey, 26 out of 35 programmers do not know the term. In the interviews in the in-depth study, two of the three programmers do not know the term. 23 out of 35 programmers in the survey do not use any IA tools, and the nine programmers in the in-depth study did not use any IA tools in the video. In the three interviews, the programmers all reported that they do not use any IA tools.

However, **programmers often recognized IA once the term was introduced.** Most programmers in the survey reported that they do IA. In the survey, 28 out of 35 programmers reported they do IA. Three programmers in the interviews all recognized IA and reported that they did IA. This implies that IA exists in the industry practice and is important to the programmers. However, the IA tools are not known to the programmers. This shows that the industry has not been benefited from the research work of IA.

In the interviews, we found IA often is recognized as a process after programmers have a possible fix. The importance of IA is mentioned because programmers do not want their patches causing new bugs.

"I always try to understand how I can influence the code. If Im uncertain about my changes I can make a list of influenced part and give it to our QA[Quality Assurance] engineers. They are checking all cases." (Participant 8)

"I have to make sure that my change will not cause bugs or other problems for other parts of the project or systems components ... " (Participant 9)

"... change impact analysis to determine if this produces new bugs." (a comment in the breadth study)

Three possible reasons can explain why we do not have evidence for much post-change impact analysis. First, in the study, the quality of the code is not a priority, because the participants know that their changes are for research purposes and the code will not be used by others. Second, they may not see post-change impact analysis as a part of the debugging process. It is possible that programmers think that the bug is fixed once the program outputs the correct output. As Participant 8 mentioned, IA tasks can be assigned to quality assurance engineers. Quality assurance engineers are those programmers who ensure that software meets specific quality standards. For example, one of the most common practice of quality assurance is testing. Note that in this paper, we did not study the practice of quality assurance engineers. However, IA tools may be used in such practice.

Third, in our study, we did not count the activities after the participants ran the program, which is after FRT and LRT in Section 3. The participants might do post-change impact analysis after FRT and LRT, but we cannot distinguish them from pre-change impact analysis for the next change.

Interestingly, we found evidence of programmers doing IA where IA is not recognized—in the middle of debugging processes. In the in-depth study, programmers ran programs and used debuggers very often after they made changes. This indicates that IA tools may be useful during debugging process, especially after programmers try a change which does not work.

12.2 $Objective_2$ What technologies do programmers use to do IA if they do any?

The plainest static method of IA is reading code by hand. **Programmers use IDE navigational features to navigate through the dependencies.** In the in-depth study, on average each programmer visited 16 files and used 28 times of IDE navigational functionalities during debugging.

The plainest dynamic method of IA is running program and checking the outputs. Instead of running program directly, **programmers often use debuggers so that they can check the values in the middle of an execution.** In the breadth study, 14 of the 35 programmers chose to use debuggers and 8 of the 35 programmers chose to run programs after they make a change. In the survey, one programmer commented "I understand change impacts mostly via single-step debugging."

In the interviews, the programmers expressed their satisfaction about the current toolset for debugging. "VM and modern IDEs have enough tools count for app's debugging" (Participant 9) "the default debugging functionality was sufficient for me" (Participant 7)

In the survey, however, the programmers commented about the need for IA tools. "There is still a lack of automatic supports of impact analysis." "Repeating the specific bug in a large project is time consuming. I would appreciate if Eclipse community can offer a tool to help me figure out which variables will be changed before tracing the code in debug mode."

These results show IA may be helpful for debugging. For example, programmers read at most 26 files before they made changes in our study. However, neither navigational features nor debuggers keep track of the dependencies discovered among the files. If IA tools can store the dependencies of the visited files, programmers may navigate the code more efficiently.

12.3 $Objective_3$ At what phases of debugging, do programmers do IA if they do any?

Programmers do static IA before they make a change. In the videos of the in-depth study, before the programmers made changes, they used an average of six times of IDE navigational functionalities. For the programmers without suggested fixes, they used IDE navigational functionalities for an average of eleven times before they made changes. In the interviews, the programmers also reported doing static IA in the end of debugging. "After my changes I have to find direct and indirect calls of this method and make sure that system will be ok after my changes." (Participant 9) However, whether and how much programmers actually do IA in the end of debugging need to be further studied.

Programmers do dynamic IA almost immediately after programmers make changes. In the in-depth study, the programmers on average visited two files and used one time of navigational functionality before they run the programs (with or without debuggers). In more than the half of the changes, the programmers visited only one file and did not use any navigational functionality. Especially, for the last modification of a change location, the programmers spent an average of 27 seconds before they ran the programs.

In the debugging process in literature (described in Section 4), programmers create and test hypotheses. In testing hypotheses, programmers are doing some form of IA. From the breadth study, one programmer commented "I hadn't heard of the term Change-Impact Analysis but I do some form of it, usually in a manual trial and error manner." Also, we have similar comments from the interviews in the in-depth study. "first locate the issue and then confirm/deny hypothesis what is wrong" (Participant 7)

These findings imply that IA processes in practice differ in the two phases: pre-change and post-change. Likewise, IA tools may need different types of techniques in the two phases. For pre-change, static IA techniques may be preferred because they have lower cost than dynamic techniques.

For the post-change phase, dynamic IA techniques may be used because programmers often run programs and check dynamic information after they make changes. In this case, the cost of dynamic IA techniques can be much lowered. First, dynamic IA techniques are applied on only one execution, which is prepared by programmers already. Second, dynamic IA techniques are applied on one specific change, which can be identified automatically by comparing the current code and the code ran in the previous execution.

13 Conclusion

In this paper, we did two studies to find out whether programmers do change impact analysis (IA). In our in-depth study, we hired nine professional programmers repairing two real bugs in two open source systems. We recorded and analyzed the videos of their debugging processes. In our breadth study, we hired 35 professional programmers to fill out our online survey. In the online survey, we asked them about what they know about IA and what they do to fix a bug.

From our two studies, we discovered the evidence of programmers doing IA. In the in-depth study, we found the evidence indicating programmers doing IA, and in the breadth study, most programmers reported that they did IA. Second, we found the evidence that the practice of IA is different from the process of IA described in the literature. No programmer in our in-depth study used any IA tools, even though they did IA to fix the bugs. In the breadth study, most of the programmers reported they did not use IA tools and most of the programmers did not know the term "change impact analysis". The purposes of IA can be various. We found programmers tend to think IA as a process that should be done after debugging. However, during debugging, programmers also do IA to figure out how to fix the bugs.

14 Acknowledgment

We thank the programmers who participated in our studies for their time, effort, and meaning comments. We also thank the developers of *Raptor* and *PDF Split and Merge* for providing the repositories.

References

1. Acharya, M., Robinson, B.: Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 746–755. ACM, New York, NY, USA (2011)
2. Altmann, J.: Observational study of behavior: sampling methods. Behaviour pp. 227–267 (1974)
3. Apiwattanapong, T., Orso, A., Harrold, M.J.: Efficient and precise dynamic impact analysis using execute-after sequences. In: Proc. of the 27th Int'l Conf. on Software Engineering, 10, pp. 432–441. ACM, New York, NY, USA (2005)
4. Bassil, S., Keller, R.: Software visualization tools: survey and analysis. In: Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on, pp. 7–17 (2001)
5. Beszedes, A., Gergely, T., Farago, S., Gyimothy, T., Fischer, F.: The dynamic function coupling metric and its use in software evolution. In: Software Maintenance and Reengineering, 2007. 11th European Conf. on, pp. 103–112 (2007)
6. Bohner, S.A.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA (1996)
7. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems, pp. 1589–1598. ACM (2009)
8. Brandt, J., Guo, P.J., Lewenstein, J., Klemmer, S.R.: Opportunistic programming: How rapid ideation and prototyping occur in practice. In: Proc. of the 4th Int'l Workshop on End-user Software Engineering, pp. 1–5. ACM, New York, NY, USA (2008)

9. Breech, B., Tegtmeyer, M., Pollock, L.: Integrating influence mechanisms into impact analysis for increased precision. In: Software Maintenance, 2006. 22nd IEEE Int'l Conf. on, pp. 55–65 (2006)
10. Briand, L.C., Wust, J., Lounis, H.: Using coupling measurement for impact analysis in object-oriented systems. In: Proc. of the 1999 IEEE Int'l Conf. on Software Maintenance, pp. 475–482. IEEE (1999)
11. Brooks, R.: Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies **18**(6), 543 – 554 (1983)
12. Canfora, G., Cerulo, L.: Fine grained indexing of software repositories to support impact analysis. In: Proc. of the seventh Int'l Workshop on Mining Softw. Repositories, pp. 105–111. ACM, New York, NY, USA (2006)
13. Dam, H.K., Ghose, A.: Automated change impact analysis for agent systems. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, pp. 33–42 (2011)
14. Dorn, B., Guzdial, M.: Learning on the job: characterizing the programming knowledge and learning strategies of web designers. In: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems, pp. 703–712. ACM (2010)
15. Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do crosscutting concerns cause defects? IEEE Trans. on Softw. Eng. **34**(4), 497–515 (2008)
16. Fjeldstad, R.K., Hamlen, W.T.: Application Program Maintenance Study: Report to Our Respondents. In: Proc. GUIDE 48 (1983)
17. Gethers, M., Dit, B., Kagdi, H., Poshyvanyk, D.: Integrated impact analysis for managing software changes. In: Software Engineering, 2012 34th Int'l Conf. on, pp. 430–440 (2012)
18. Hartmann, B., Doorley, S., Klemmer, S.R.: Hacking, mashing, gluing: Understanding opportunistic design. Pervasive Computing, IEEE **7**(3), 46–54 (2008)
19. Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J., Damasio, J.: On the precision and accuracy of impact analysis techniques. In: Seventh IEEE/ACIS Int'l Conf. on Computer and Information Science, pp. 513–518 (2008)
20. Huo, D., Ding, T., McMillan, C., Gethers, M.: An empirical study of the effects of expert knowledge on bug reports. In: IEEE Int'l Conf. on Software Maintenance and Evolution, pp. 157–166 (2014)
21. Kagdi, H., Gethers, M., Poshyvanyk, D.: Integrating conceptual and logical couplings for change impact analysis in software. Empirical Software Engineering **18**(5), 933–969 (2013)
22. Kienle, H., Müller, H.: Requirements of software visualization tools: A literature survey. In: Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on, pp. 2–9 (2007)
23. Ko, A., Myers, B., Coblenz, M., Aung, H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. Software Engineering, IEEE Transactions on **32**(12), 971–987 (2006)
24. Lakhotia, A.: Understanding someone else's code: analysis of experiences. J. Syst. Softw. **23**(3), 269–275 (1993)
25. LaToza, T.D., Garlan, D., Herbsleb, J.D., Myers, B.A.: Program comprehension as fact finding. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 361–370. ACM, New York, NY, USA (2007)
26. LaToza, T.D., Myers, B.A.: Developers ask reachability questions. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering - Volume 1, 10, pp. 185–194. ACM, New York, NY, USA (2010)
27. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: Proc. of the 28th Int'l Conf. on Software engineering, pp. 492–501. ACM, New York, NY, USA (2006)
28. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: Proc. of the 2003 Int'l Conf. on Software Engineering, pp. 308–318. IEEE (2003)
29. Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., Fleming, S.: How programmers debug, revisited: An information foraging theory perspective. Software Engineering, IEEE Trans. on **39**(2), 197–215 (2013)
30. Lehnert, S.: A taxonomy for software change impact analysis. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, IWPSE-EVOL '11, pp. 41–50. ACM, New York, NY, USA (2011)

31. Li, B., Sun, X., Leung, H., Zhang, S.: A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability **23**(8), 613–646 (2013)
32. McDonald, J.H.: Handbook of biological statistics, vol. 2. Sparky House Publishing Baltimore, MD (2009)
33. McIntosh, S., Adams, B., Nagappan, M., Hassan, A.: Mining co-change information to understand when build changes are necessary. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pp. 241–250 (2014)
34. McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: 2011 33rd Int'l Conf. on Software Engineering, pp. 111–120. IEEE (2011)
35. Orso, A., Apiwattanapong, T., Harrold, M.J.: Leveraging field data for impact analysis and regression testing. In: Proc. of the 9th European Software Engineering Conf. Held Jointly with 11th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering, pp. 128–137. ACM, New York, NY, USA (2003)
36. Parande, M., Koru, G.: A longitudinal analysis of the dependency concentration in smaller modules for open-source software products. In: Software Maintenance (ICSM), 2010 IEEE International Conference on, pp. 1–5 (2010)
37. Petrenko, M., Rajlich, V.: Variable granularity for improving precision of impact analysis. In: Program Comprehension, 2009. IEEE 17th Int'l Conf. on, pp. 10–19 (2009)
38. Poshyvanyk, D., Marcus, A., Ferenc, R., Gyimthy, T.: Using information retrieval based coupling measures for impact analysis. Empirical Software Engineering **14**(1), 5–32 (2009)
39. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. In: ACM Sigplan Notices, vol. 39, pp. 432–448. ACM (2004)
40. Robillard, M., Coelho, W., Murphy, G.: How effective developers investigate source code: an exploratory study. Software Engineering, IEEE Transactions on **30**(12), 889–903 (2004)
41. Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do professional developers comprehend software? In: Proc. of the 34th Int'l Conf. on Softw. Eng., pp. 255–265. IEEE Press, Piscataway, NJ, USA (2012)
42. Rovegard, P., Angelis, L., Wohlin, C.: An empirical study on views of importance of change impact analysis issues. Software Engineering, IEEE Trans. on **34**(4), 516–530 (2008)
43. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Eng. **14**(2), 131–164 (2009)
44. Rungta, N., Person, S., Branchaud, J.: A change impact analysis to characterize evolving program behaviors. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 109–118 (2012)
45. Shneiderman, B.: Software Psychology: Human Factors in Computer and Information Systems (Winthrop Computer Systems Series). Winthrop Publishers (1980)
46. Sillito, J., Murphy, G., De Volder, K.: Asking and answering questions during a programming change task. Software Engineering, IEEE Transactions on **34**(4), 434–451 (2008)
47. de Souza, C.R.B., Redmiles, D.F.: An empirical study of software developers' management of dependencies and changes. In: Proc. of the 30th Int'l Conf. on Software Engineering, pp. 241–250 (2008)
48. Storey, M.A.: Theories, tools and research methods in program comprehension: past, present and future. Software Quality Journal **14**(3), 187–208
49. Storey, M.A., Wong, K., Muller, H.: How do program understanding tools affect how programmers understand programs? In: Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on, pp. 12–21 (1997)
50. Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S.: How do software engineers understand code changes?: An exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 51:1–51:11. ACM, New York, NY, USA (2012)
51. Wetzlmaier, T., Ramler, R.: Improving manual change impact analysis with tool support: A study in an industrial project. In: Software Quality. Software and Systems Quality in Distributed and Mobile Environments, *Lecture Notes in Business Information Processing*, vol. 200, pp. 47–66. Springer Int'l Publishing (2015)
52. Wilkerson, J.: A software change impact analysis taxonomy. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 625–628 (2012)
53. Wu, Y., Yap, R., Ramnath, R.: Comprehending module dependencies and sharing. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol. 2, pp. 89–98 (2010)

54. Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 689–699. ACM, New York, NY, USA (2014)
55. Zeller, A.: Why programs fail: a guide to systematic debugging. Elsevier (2009)
56. Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S.: Mining version histories to guide software changes. Software Engineering, IEEE Trans. on **31**(6), 429–445 (2005)