Measuring Program Comprehension: A Large-Scale Field Study with Professionals

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, Shanping Li

Abstract—During software development and maintenance, developers spend many time on program comprehension is one of the most important activities during the whole life cycle of software development and maintenance. Previous studies show that program comprehension takes up as much as half of a developer's time. However, most of these studies are performed in a controlled setting, or with a small number of participants. Moreover, most prior studies investigate the program comprehension activities only within the IDEs. However, developers' program comprehension activities go well beyond their IDE interactions. Developers all too often produce, consume, and communicate information across multiple applications, e.g., IDEs and web browsers.
In this paper, we extend our our ActivitySpace framework to collect and analyze Human-Computer Interaction (HCI) data across many applications (not just the IDEs). We follow Minelli et al. to assign developers' activities into four categories: navigation, editing, comprehension, and other. We measure comprehension time by calculating the time developers spend in program comprehension, e.g. inspecting console and breakpoint in IDE, reading and understanding tutorials in web browsers, etc. Then we perform a more realistic investigation of program comprehension activities. We perform a field study of program comprehension in practice with a total of 7 real projects, 79 professional developers, and amounting to 3,244 working hours. Our study leverages interaction data that is collected across many applications (not just the IDEs) by the developers. Our study finds that on average, developers spend up to $\sim 58\%$ of their time on program comprehension, and they frequently use web browsers and document editors to perform program comprehension activities. We also investigate the impact of programming language, developers' experience, and project phase to the time spent on program comprehension, e.g., we find senior developers spend significantly less percentages of time on program comprehension than junior developers.

Index Terms—Program Comprehension, Field Study, Inference Model

1 Introduction

Program comprehension (aka., program understanding, or source code comprehension) is a process where developers are actively acquiring knowledge about a software system by exploring and searching software artifacts, and reading relevant source code and/or documentation. Such acquired knowledge helps support other software engineering activities, such as bug fixing, enhancement, reuse, and documentation.

Previous studies show that program comprehension is an essential and time-consuming activity in software maintenance [12], [13], [21], [28], [53]. Zelkowitz et al. claim that program comprehension takes more than half of the time spent on software maintenance [53], which is also confirmed by Fjeldstad and Hamlen [13], and Corbi [12]. Ko et al. perform controlled experiments with two debugging tasks and 10 participants, and they find understanding a program occupies around 35% of the total time [21]. Minelli et al. study the IDE interactions of 18 developers over 700 working hours, and they find that on average developers spend 70% of their time performing program comprehension [28]. However, only seven of the participants are professionals and more than 85% of the studied data is based on the activities of 3 participants who are PhD students. Moreover, the study only investigates program comprehension activities in the IDE.

The current empirical understanding the role of program comprehension for software development has many shortcomings, most notable are: (1) several conclusions are based on anecdotal evidences [12], [13], [53], instead of empirical experiments on developers; (2) most prior studies are performed under controlled experiment with artificial setting, which make it difficult to generalize the results, e.g., [21]; (3) most prior studies involve a small number of participants (e.g., Ko et al.'s study has 10 participants [21], while Minelli et al.'s study has 18 participants [28]), and most of the participants are not professionals; (4) most prior studies only investigate program comprehension activities that occur within IDEs [28], [21]. Our previous study shows that developers use six or more different desktop and web applications in their daily development work [5]. For example, to understand a piece of source code, a developer may not only navigate and search for the related source code inside the IDE, but also search online resources, such as Stack Overflow.

In this paper, we perform a large-scale field study to investigate program comprehension activities in a *realistic setting*, while taking a more holistic approach that examines

- *Xin Xia, Lingfeng Bao and Shanping Li are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. E-mail: xxia@zju.edu.cn, lingfengbao@zju.edu.cn, shan@zju.edu.cn*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Zhenchang Xing is with the Research School of Computer Science, Australian National University, Australia. E-mail: zhenchang.Xing@anu.edu.au*
- *Ahmed E. Hassan is with School of Computing, Queen's University, Canada. E-mail: ahmed@cs.queensu.ca*
- *Lingfeng Bao is the corresponding author.*

activities *across many applications* that are used by developers instead of only using interaction data that is gathered from IDEs. Similar to past studies [12], [13], [21], [28], [53], our study tries to validate a well-known assumption (i.e., program comprehension takes much of developer time) that drives the line of work on improving program comprehension. It is important to evaluate the assumption, because there is a large body of research on improving program comprehension.

Comparing to previous studies, our study need to collect a large number of developers' activity data from their real working environment data. So, we decide to use the methodology of Minelli *et al.* [28] since their approach can automatically infer developers' activities from developers' low-level interaction data. Different from the models of programming understanding used in other studies, e.g. [12], [48], our programming understanding model separates "navigation" from other activities. This is because navigation is one important activity in software development. Ko et al. found that developer usually find the target information by navigating "information scents" [21], e.g. hyperlinks on a web page or graphic icons. Their study leads to a model of program understanding grounded in theories of information forging. But current IDEs cannot support navigation very well. For example, if developers lost track of relevant code in Eclipse as they switch to other tasks, they are forced to find it again. Identifying navigation actions from developers activities can give more insight of developers behavior. For example, we can get the process knowledge from developer development. We not only can know "what a developer code, but also know "how a developer code. However, identifying navigation from other activities, e.g. coding and debugging, is very difficult, since developers interleave navigation and other activities. Furthermore, the data used in some studies has some limitations in real working environment and require manual analysis. For example, Ko et al. use screen capturing technique to record developers working process in which they perform two debugging tasks and three enhancement tasks [21]. Then they transcribe the collected screen-capturing videos into different developer actions (e.g. reading code, editing code, etc.). Their results are based on subjective interpretations of the developers behaviors, but its unrealistic to analyze our collected data manually in our study because we collect developers activity data for a period of time (two weeks in this paper). Furthermore, the storage of screen-capturing videos is very large. Hence, we extend the work of Minelli et al. [28] to investigate programming comprehension.

Following by Minelli et al. [28], we categorize developers' activities into four categories: navigation, editing, comprehension, and other[1]. Navigation time refers to the time developers spend in browsing through software [41], including navigation using IDEs or web browsers, clicking a link, and searching for particular program entities or code, etc. Editing time refers to the time developers spend on editing source code. Comprehension time refers to the time developers spend in program comprehension, including inspection activities such as inspecting console and breakpoint in IDE, reading through a piece of code (identified by e.g., detecting mouse drifting actions), etc. Notice that sometimes developers perform navigation activities to assist program comprehension activities, however, the navigation activities only involve some quick keyboard/mouse activities, such as roll the mouse, or click a link, and in that short time, developers actually do not perform comprehension activities.

Our study is conducted within two large IT companies named Insigma Global Service[2] and Hengtian[3] in China, which have more than 500 and 2,000 employees, respectively. In total, we investigated activities of 79 developers across 7 projects over 3,244 working hours in total. Moreover, we interviewed 10 of these developers. Our study finds that: (1) on average, program comprehension takes up to $\sim 58\%$ of the developers' time, (2) besides IDEs, developers frequently use web browsers and document editors during their program comprehension activities, (3) developers in Java projects spend a significantly higher percentage of time on program comprehension than developers in C# projects, (4) senior developers spend significantly less percentages of time on program comprehension than junior developers, and (5) developers working on projects that are in the maintenance phase spend significantly higher percentages of time on program comprehension than those working on projects that are in the development phase.

The following is our list of contributions:

1) We perform a large-scale field study on the role of program comprehension for software development, which include a total of 79 developers across 7 projects over 3,244 working hours. To our best knowledge, it is the largest field study on program comprehension until now. Different from previous studies, our study works on a realistic setting.
2) Our findings are consistent with the previous studies, and we also investigate the impact of programming language, developers experience, and project phase to the time spent on program comprehension.

Paper organization. The remainder of this paper is organized as follows. Section 2 briefly reviews related work. Section 3 elaborates the field study setup and data collection. Section 4 presents our field study results. Section 5 discusses the threats to validity.Section 6 draws the conclusions and mentions future work.

2 Related Work

Measuring Program Comprehension. There have been a number of studies on measuring program comprehension [12], [13], [21], [28], [53]. Zelkowitz et al. [53], Fjeldstad and Hamlen [13], and Corbi [12] all report that program comprehension activities take more than half of the time spent on software maintenance based on anecdotal evidences. Ko et al. perform controlled experiments with two debugging tasks and 10 participants, and and they find that program comprehension occupies around 35% of the total development time [21]. Minelli et al. study the IDE interactions of 18 participants over 700 working hours, and they find developers spend 70% of their time performing

[1]For more details, please refer to Section 2.2.

[2]http://www.insigmaservice.com/

[3]http://www.hengtiansoft.com/en

program comprehension activities [28]. Most of the participants in Ko et al.'s and Minelli et al.'s studies are students rather than professional developers. Their studies also only analyze developer activities in the IDEs.

Extending these previous studies, in this paper, we investigated program comprehension activities performed by *79 professionals* working on *7 industrial projects* in a *realistic setting*. We collected a large amount of interaction data (a total of 3,244 working hours) by monitoring developer activities *across many applications* that they used in their daily work. We also conducted *interviews* to confirm and better interpret our quantitative findings.

Field Study on the Role of Program Comprehension for Software Development. Roehm et al. perform a field study on the role of program comprehension for software development with 28 developers to understand: (1) what strategies developers follow to comprehend programs, (2) what sources of information do developers use, (3) what information is missing, and (4) which tools that developer use and how do they use them [37]. Our field study is different and complement Roehm et al.'s study in several aspects: First, Roehm et al.'s study *do not measure program comprehension time* which is the focus of this study. Second, Roehm et al.'s study observes each participant for 45 minutes, while our study observes each participant for *two weeks*. Third, Roehm et al.'s study is *relatively invasive to developer activities*, with each developer needing to comment on what they are doing in a think-aloud fashion and several researchers observing the developer. This procedure may make developers change their behaviors substantially. Our study involves a less invasive procedure. Fourth, we consider many *different RQs* that Roehm et al.'s study does not consider. Only one of our five RQs (i.e., RQ2: which applications developers use in program comprehension activities) overlaps. Even with this RQ, we consider a different angle by measuring the amount of time that developers spend inside these applications. Our paper also points to web browsers as useful comprehension tools, which was not part of by Roehm et al.'s study.

In a later work, Maalej et al. further extended Roehm et al.'s study to TOSEM by survey 1,477 respondents, and they analyzed the importance of certain types of knowledge for program comprehension, and the way developers typically access and share the knowledge [26]. Different from Maalej et al.'s study, our study did not involve the online survey, and we plan to send out a survey to study practitioners perception on program comprehension to better understand the conclusion of our study in the future.

Identifying Factors Affecting Program Comprehension. There have been a number of studies that investigate the impact of different factors to program comprehension. Siegmund et al. investigate relationships between programming experience and program comprehension by performing short controlled experiments (i.e., 40 minute experiments) using students as participants [40]. Teasley report that naming style has impact on program comprehension [46]. Latoza et al. identify working habit as a factor that impacts program comprehension [23]. In our study (i.e., RQ4), similar to Siegmund et al.'s work [40], we also investigate the impact of programming experience on program comprehension. However, different from their work, our study is performed under a *realistic setting* by monitoring the activities of *professional developers* for *two weeks*. Also, different from the above mentioned studies, we consider additional factors, such as programming language (see RQ3) and project phase (see RQ5).

3 Field Study Setup

In this section, we present our field study setup which includes three parts. We first present the criteria and details of how the participants are selected. Next, we describe the tool used to collect and organize developer interactions across applications. Then, we present the details of our qualitative interviews, which supplement our quantitative findings. Finally, we present the five research questions which would be investigated in our study.

3.1 Participant Selection

One aim of our study is to investigate how professionals (not students) perform program comprehension activities in an realistic setting. We thus select participants in two IT companies in China, named Insigma Global Service, and Hengtian. Insigma Global Service is an outsourcing company which has more than 500 employees, and it mainly does outsourcing projects for Chinese vendors (e.g., Chinese commercial banks, Alibaba, and Baidu). Hengtian is also an outsourcing company which has more than 2,000 employees, and it mainly does outsourcing projects for US and European corporations (e.g., StateStreet Bank, Cisco, and Reuters).

Notice that in these two companies, around 50% of the employees are developers (i.e., around 1,250 developers). Also, a number of projects (around 60%) need to be done onsite (i.e., developers should work in the client's company) and many projects are constrained with strict security policies. Unfortunately, we cannot collect data from these onsite and secure projects. After removing developers that work on these projects, around 830 developers remain as possible participants of our study. Our toolset for collecting developer interactions works on the Windows operating system and not all developers use Windows. Thus, we further remove additional 205 candidate developers from our list of possible participants. As a result, we have 625 developers left. These developers are involved in 25 different projects. Next, we select projects and developers from this pool of 25 projects and 625 developers following these steps:

- To reduce bias due to the project size, the selected projects should have different sizes.
- To reduce bias due to the programming languages used, the selected projects should use different programming languages. We choose projects which use either Java or C# as their main programming language. Java and C# are the two most popular programming languages used inside these two companies. Eight projects use Python, Matlab, or C/C++ as their main languages, and thus we exclude them from our list of projects.
- To reduce bias due to new or inactive projects, we exclude 8 projects that are close to completion and 2 new projects.

At the end, 7 projects remain, and there are a total of 410 developers work on the 7 projects. We send emails to the developers inviting them to join our study. Eighty three developers allow us to install our tool and collect their interactions for two weeks (i.e. 10 working days in total, excluding weekends). Among the 83 developers, 22% (18) have more than 5 years of professional experience, 42% (35) have 3 to 5 years of professional experience, and 36% (30) have less than 3 years of professional experience.

For each developer, we compute his/her effective working hours across the two weeks. Effective working hours refer to the time when a developer stays in front of the computer, doing things which are related to the project. We exclude the time the developer spends on personal activities (e.g., eating lunch/dinner), or meetings. Figure 1 presents the distribution of the effective working hours. The median effective working hours recorded by our tool is 37.4 hours, the minimum working hours is 1.4 hours, and the maximum working hours is 96 hours. We found 4 participants worked less than 5 hours during the two weeks, two of them were project managers and they needed to attend many meetings at that time, one of them moved to the client's site to work, and another needed to fly to another country to attend a industrial conference. We removed the data collected from these 4 participants to reduce the noise, and in total we analyze the data from 79 participants. Also, we notice 9 participants worked for more than 80 hours during two weeks. Among them, seven were from project A, since they have a minor release at that time. Another two were from project G, they told us that they are new to the project team, thus they worked many hours per day to be familiar with project.

Table 1 presents the statistics of the 7 projects[4] that we investigate in our study. The columns correspond to the name of the projects (Project), the start time of the projects (Start.), the number of the developers (# D.), the number of developers who participate in our study (# S.), the number of lines of code (LOC), the main programming language (Pro.), the size of the projects (Size) (L=large, M=medium, and S=small), and the project phase (P.) (M=Maintenance and D=Development).

Among the seven projects, projects A, E, and G contain more than 5M LOCs, and more than 50 developers, considering the size of LOCs, number of developers, and also developer input and the two companies definition, in this study, we consider these 3 projects as the large-size projects. Also, projects B, C, and F contain 1M to 3M LOCs, and 12 - 45 developers, we label them as medium-size projects. Moreover, project D only has 0.3M LOCs, and 10 developers, and we label it as a small-size project. We also ask developers to categorize each project into either maintenance or development phase depending on whether the corresponding software product has been released or not. Among the 7 projects, three are large-sized projects (A, E, and G), three are medium-sized projects (B, C, and F), and one is a small-sized project (D). Four projects use Java (A, C, D, and F), and three use C# (B, E, and G) as the main programming language. Four projects are in the development phase (C, D, E, and G), and three projects are in the maintenance phases (A, B, and F).

[4]Due to the security polices in these two companies, we anonymize the project name.

TABLE 1: Statistics of the studied projects.

Project	Start.	# D.	# S.	LOC	Pro.	Size	Phase
A	2010.10	118	**18**	10M	Java	L	M
B	2011.08	12	**4**	2M	C#	M	M
C	2013.07	30	**5**	1M	Java	M	D
D	2014.12	10	**4**	0.3M	Java	S	D
E	2012.04	80	**17**	5M	C#	L	D
F	2015.04	45	**10**	3M	Java	M	M
G	2014.08	115	**21**	11M	C#	L	D

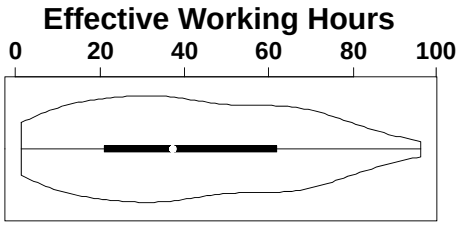

Fig. 1: Distribution of effective working hours.

3.2 HCI Data Collection and Analysis

In this study, we extend our *ActivitySpace* framework [5], [6] to collect and analyze Human-Computer Interaction (HCI) data in developers' daily work. Figure 2 shows the process of data collection and analysis: First, we use *ActivitySpace* framework to collect time-ordered events while a developer is interacting with applications. Then we divide a sequence of time-ordered events into some working sessions by identifying idle periods and divide a working session into some spree by the reaction time. Next, we will classify these sprees by the information provided by the collected events. Finally, we compute the time of different activities.

3.2.1 Tracking Event

As the developer is interacting with an application, *ActivitySpace* generates time-ordered events (see Figure 3 for an example). Each event has a time stamp down to millisecond precision. Each event is composed of an event type, basic window information collected using OS window APIs, and focused UI information that the application exposes to the operating system through accessibility APIs (for mouse click event only). *ActivitySpace* monitors three types of mouse events (including mouse move, mouse wheel, and mouse click) and two types of keyboard events (including normal keystrokes like alphabetic and numeric keys and shortcut keystrokes like "Ctrl+F" (Search or Find) and "Ctrl+O" (Eclipse shortcut for quick outline).

Basic window information includes the position of mouse or cursor, the title and boundary of the focused application window, the title of the root parent window of the focused application window, and the process name of the application. If the event type is mouse click, *ActivitySpace* uses accessibility APIs to extract the following focused UI information: *UI Name*, *UI Type*, *UI Value* and *UI Boundary* of the focused UI component, and the *UI Name* and *UI Type* of *the root parent UI component*. The accessibility information is very helpful to infer the application context of the

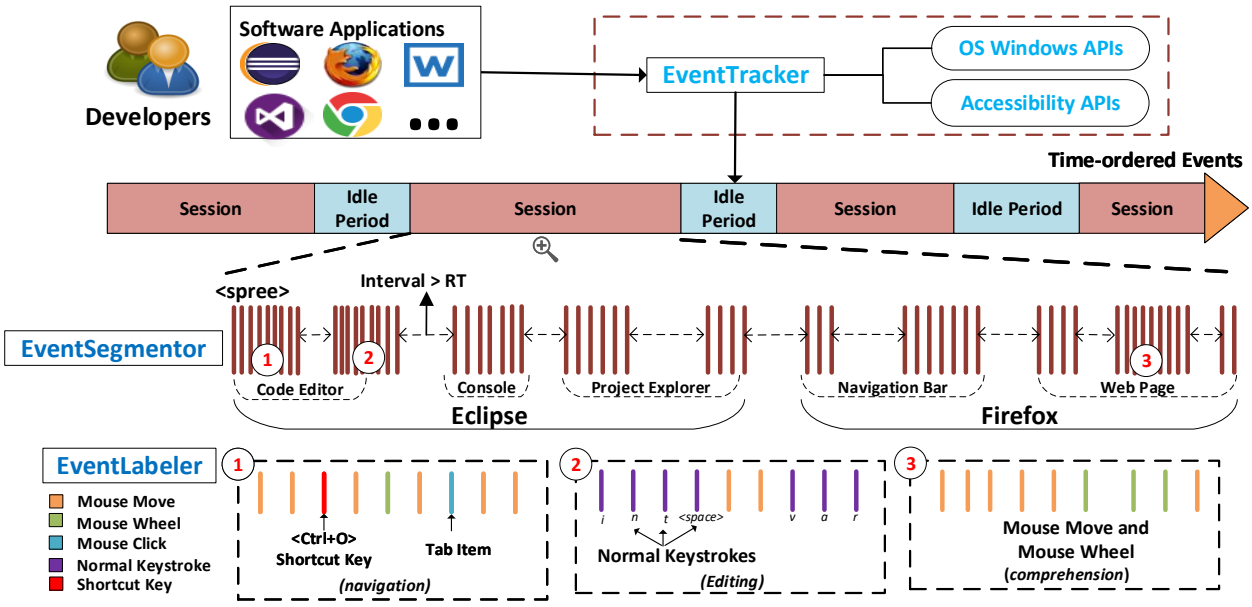

Fig. 2: The process of HCI data collection and analysis.

developer's action. For example, if the developer selects an item in "Project Explorer" of Eclipse or "Solution Explorer" of Visual Studio, *ActivitySpace* will record both the selected item and its root parent UI component ("Project Explorer" or "Solution Explorer"). This contextual information allows us to classify the event as a navigation event.

In Figure 3, the first three events occur in an Eclipse application window, and the last two events occur in a Firefox application window. Note that each event has its own window information. However, due to space limitation, we show only window title, window boundary, root parent window title, and process name for one of the first three events and one of the last two events. The focused UI information collected for the two mouse click events shows that the developer selects a file in "Project Explorer" in Eclipse, and searches *java calendar* on Google in Firefox.

In this study, we configure *ActivitySpace* to monitor applications that are commonly used in developers' daily work, including web browsers (e.g., *Firefox, Chrome, Internet Explorer*), document editors (and/or readers) (e.g., *Word, Excel, PowerPoint, Adobe Reader, Foxit Reader, Notepad, Notepad++*), and IDEs (e.g., *Eclipse, Visual Studio*). We have validated the list of applications monitored with the developers and they confirm that these are the ones that they typically use. We did not monitor command line tools since we were informed that developers in the two companies rarely use them when they worked in the Windows environment[5]. *ActivitySpace* generates a placeholder event of "unknown" event type when a mouse or keyboard event occurs in all other applications. We analyse the proportion of time developers spend on such "unknown" applications, and it is typically less than 2%.

[5]The developers also informed us that they would frequently use command line when they worked in Linux environment, however our current tool can only capture the interaction data in Windows.

3.2.2 Identifying Effective Working Sessions

Given a sequence of time-ordered events, *ActivitySpace* first removes all the "unknown" events. That is, we do not consider activities in unmonitored applications in the subsequent analysis. Then, *ActivitySpace* identifies *idle periods* during which no mouse or keyboard events occur. In this study, we set the threshold of idle period at 1 hour. We have checked with the developers who agree that if a developer has no mouse or keyboard events in 1 hour, he/she has a high probability to do some program comprehension unrelated activities, such as taking a lunch or joining a meeting. Idle periods split a sequence of time-ordered events into effective working sessions. For a developer, his effective work hours is the sum of the time duration of all the effective sessions.

3.2.3 EventAnalyzer

Given an effective working session, the event segmentation component (*EventSegmentator*) of *ActivitySpace* first splits the sequence of events into application-window segments by *Process Name* of basic window information, for example Eclipse or Firefox. Then, for each application-window segment, *EventSegmentator* further splits the sequence of events into view segments by *Window Title* of basic window information or *Parent UI Name* or *Parent UI Type* of accessibility information, for example, Project Explorer, Console and Code Editor in Eclipse window, and Navigation Bar and Web Page area in Firefox window.

Finally, for each view segment, *EventSegmentator* splits the sequence of events into a sequence of *sprees* by the *reaction time (RT)*, which is defined as follow:

Definition 1 (Spree). A spree is a sequence of mouse/keyboard events in which the interval between each pair of events is less than *reaction time (RT)*.

The time interval between the two consecutive sprees must be larger than the *RT*, while the time interval between

Event Type	Mouse Move	Mouse Click	Mouse Wheel	Mouse Click	KeyInput: "Ctrl+V"
Cursor Position	(28, 439)	(143, 254)	(193, 397)	(595, 262)	(595, 262)
Window Title	N/A	N/A	N/A	N/A	N/A
Window Boundary	(6, 105, 495, 1008)	(6, 105, 495, 1008)	(6, 105, 495, 1008)	(0, 0, 1920, 1040)	(0, 0, 1920, 1040)
Parent Window Title	Java – Project/package/TimelineExample.java - Eclipse	Java – Project/package/TimelineExample.java - Eclipse	Java – Project/package/TimelineExample.java - Eclipse	java calendar - Google Search - Mozilla Firefox	java calendar - Google Search - Mozilla Firefox
Process Name	eclipse.exe	eclipse.exe	eclipse.exe	firefox.exe	firefox.exe
UI Name		JSTreeDao.java		Search	
UI Type		tree item		combo box	
UI Value		N/A		java calendar	
UI Boundary		(123, 249, 205, 267)		(136, 121, 706, 140)	
Parent UI Name		Project Explorer		java calendar - Google Search - Mozilla Firefox	
Parent UI Type		pane		window	

Fig. 3: An example of low level events.

the two consecutive events in a spree must be smaller than or equal to the *RT*. The *reaction time* is the time that elapses between the end of a physical action sequence (e.g., typing, moving the mouse, etc.) and the beginning of concrete mental processes (e.g., reflecting, planning, etc.), which represent the basic moments of program understanding. The *RT* is also known as "Psychological Refractory Period", which has been used in many psychology studies (e.g., personality, driving, and level of alcohol or caffeine). The term "psychological refractory period" refers to the period of time during which the response to a second stimulus is significantly slowed because a first stimulus is still being processed [33]. According to this theory, developers cannot perform different activities (i.e., programming comprehension, navigation, editing) in the same time. So, we use *RT* to split the event sequence into sprees. For example, a developer is typing a piece of code in an editor. After some typing, the developer pauses and thinks about the code he just wrote and plans the next steps. Such pauses will split the event sequence in a view segment into sprees. Note that a spree might only contain a single action when an action happens very slowly, for example, a slow nivation action (a pause, a scroll or an open caller). In such cases, the intervals among actions are usually larger than *RT*, which can be considered as the moment of program comprehension. The *RT* might vary from human factors (e.g. personality, age, level of alcohol or caffeine, etc.) and the task at the hand. Different settings of *RT* might generate different results, but Minelli *et al.* [28] reported that the different *RT* values did not affect their findings. So, in this study, we set *RT* at 1 second, following their *RT* setting. We also discuss the effect of different *RT* values in section 5.1.

3.2.4 Classifying Sprees

Given a spree, the event labeling component *EventLabeler* of *ActivitySpace* classifies the spree as *navigation*, *comprehension* or *editing*. Our classification scheme follows Minelli et al.'s work [28]. Minelli et al. assign *inspection* activities (e.g. inspecting stacktrace in Eclipse Console) as *comprehension* category, and *Browsing* (e.g. selecting a package, method, or class in Project Explorer of Eclipse) and *Searching* (e.g. Starting a search in a Finder UI) activities to *navigation* category. Figure 4 presents the process of spree categorization of *EventLabeler*.

First, *EventLabeler* checks the window context (Window Title, Parent UI component, sub-window) which usually reflect the developers' activities directly to classify the spree as navigation or comprehension. We identify the most common used UI components, sub-windows in our collected data which are listed in upper part of Figure 4. For Eclipse and Visual Studio which are used as main IDEs in our study, if the developer is performing *inspection* activities (e.g. inspecting **Console** in Eclipse window or **Output** in Visual Studio window), the spree is classified as *comprehension*; if the developer is performing *browsing* or *searching* activities (e.g. using **Project Explorer** in Eclipse window or **Solution Explorer** in Visual Studio window), the spree is classified as *navigation*. For browser, if the spree is in **Navigation Bar** or in a search engine's web page, we regard this spree as *navigation*. For all other applications, sprees in **Search/Find** windows are classified as *navigation*.

If *EventLabeler* cannot determine the category of the spree based on its window context, it will then try to label the events in the spree in order to determine its category. The lower part of Figure 4 presents how *EventLabeler* labels an event. For mouse click event, *EventLabeler* classifies the event as *navigation* or *comprehension* event based on the UI type of the focused UI component where the mouse click occurs, as summarized in part *Navigation UI Type* of Figure 4. *UI Type* may indicate the type of activities developers perform, for example, if the *UI Type* is *tree item* or *scroll bar*, the developers usually perform *Browsing* activities, then *EventLabeler* classifies the event as *navigation* event. If the mouse click event occurs in a non-navigation UI Type, the event is classified as *comprehension* event. For shortcut key event, *EventLabeler* labels the event according to its function. For example, "Ctrl+F" is classified as a *navigation* event, while "F6" (step over in Eclipse) is classified as *comprehension*. We identify the most common used shortcut keys in our collected data, as summarized in part *Navigation Shortcut Key* and *Comprehension Shortcut Key* of Figure 4. *EventLabeler* labels normal keystroke events as *editing*.

If all the events in the spree are mouse move and/or mouse wheel events (aka. *Mouse Drifting* in Minelli et al.'s work), *EventLabeler* classifies the spree as *comprehension*, for example, the spree (3) in Figure 2 in which the developer is browsing a web page using a mouse. If the number of editing events are more than 50% of the sum of editing, navigation and comprehension events, *EventLabeler* classifies the spree as *editing*, for example, spree (2) in Fig 2. Finally, if the number of *navigation* events is greater than that of *comprehension* events, *EventLabeler* classifies the spree as *navigation*, otherwise as *comprehension*. For example, spree (1) in Fig 2 has two *navigation* events (Ctrl+O to show quick outline and selecting another editor in the tab), but no *comprehension* events. Thus, the spree is classified as *navigation*.

3.2.5 Computing Activity Statistics

The comprehension time is the sum of the duration of all the *comprehension* sprees and all the time intervals between sprees that are longer than the *RT* (1 second in this study) and shorter than a threshold (5 minutes in this study). Based on our observation and interview, the time intervals longer than 5 minutes usually represent the time period during which the developers have a short break or chat with their colleagues. We do not consider these time intervals as idle period because the developer is still in a working mode on the computer, unlike a long meeting or lunch break. The *navigation* and *editing* time are the sum of the duration of all the *navigation* and *editing* sprees respectively.

We aggregate the statistics of developers' activities according to different types of applications. In this study, we classify the monitored applications into three types: IDEs (Eclipse, Visual Studio), web browsers (e.g., Firefox, Chrome, IE), and document editors (Word, Excel, PowerPoint, PDF reader, Notepad, Notepad++, etc.).

We filter activities in web browsers that are unlikely related to software development tasks (e.g. visiting news or shopping websites) using the keywords in the title of web pages visited (for example, "Sina", one of the most popular news websites in China, or "taobao", the most popular online shopping website in China). We observe the collected data and identify a set of keywords to filter non-software-development activities in web browsers. We use a long list of filters that were empirically determined and fine tuned to ignore websites that are unrelated to software development. Table 2 shows some example keywords of our used website filters. We divide the websites that are unrelated to software development into seven categories: *News, Sports, Social Network, Shopping, Game, Video, Money*. Noted that most of example keywords in Table 2 are translated from Chinese. This results in more accurate statistics of developers' work.

To understand program comprehension across different applications, *ActivitySpace* identifies all application switchings by the difference between the process names of the two consecutive events. Then *ActivitySpace* can find all continuous switching sequences, i.e. a sequence that contains some kinds of switchings, such as IDE ⇒ Web Browser, IDE ⇒ Web Browser ⇒ IDE, of length 2 to 4. For each sequence, *ActivitySpace* counts instances of such switching and computes the total time spent. The total time carefully considers overlaps; for example, IDE ⇒ Web Browser ⇒ IDE ⇒ Web Browser ⇒ IDE has two instances of IDE ⇒ Web Browser ⇒ IDE, but the two instances are overlapping. This overlapping part is only considered once in the computation of total time.

TABLE 2: Examples of Website Filters

Website Category	Example Keywords
News	*Sina, NetEase, Sohu, Tencent*
Sports	*NBA, Basketball, Football*
Social Network	*weibo, weixin, QQ*
Shopping	*Taobao, Tmall, Jingdong*
Game	*Game, Dota, LOL*
Video	*Iqiyi, Youku, AcFun, Bilibili*
Money	*stock, real estate*

TABLE 3: Percentage of time two developers spend on comprehension (Compre.), navigation, editing, and others as computed by our data collection tool and manually labeled by developers.

Dev	Tool	Compre.	Navigation	Editing	Other
Dev 1	Our	58.26%	18.38%	15.25%	8.15%
	Manual	58.31%	18.41%	15.11%	8.17%
Dev 2	Our	62.38%	22.45%	13.88%	1.29%
	Manual	62.30%	22.47%	13.71%	1.52%

3.2.6 Accuracy of Our Data Collection Tool

To investigate the accuracy of our data collection tool on identifying program comprehension activities, we perform a preliminary study on two developers. To do so, we install our data collection tool and a video recording tool on these two developers' desktop. Next, we record 4 working hours for each of these two developers by using both our data collection tool and the video recording tool. Then, we invite these two developers to join us to review the videos we collected. We ask the developers to tell us what they did in every 1 second, and categorize what they did into one of the four activities (i.e., navigation, editing, comprehension, and others). We also use our data collection tool to automatically compute the time developers spend on these four activities.

Table 3 presents the percentage of time the two developers spend on comprehension, navigation, editing, and others computed by our data collection tool and developer manual labeling. We notice the difference between our data collection tool and manual labeling is relatively small (less than 0.23%), thus our proposed tool could achieve an acceptable accuracy.

3.3 Interview

In addition to analyzing the collected data, we interview 10 out of the 79 participants, to confirm and better interpret our findings. We perform the interviews at the end of the monitoring process. Table 4 presents the working experience, programming languages, and project teams of the 10 interviewees. We select 5 Java developers and another 5 C# developers. Notice these interviewees are selected based on their professional experience and availability.

The interviews are semi-structured and are divided into three parts. In the first part, we ask each developer some demographic questions, such as the working experience of the interviewees. In the second part, we ask some open-ended questions, such as the importance, challenges, and difficulties met during the program comprehension process.

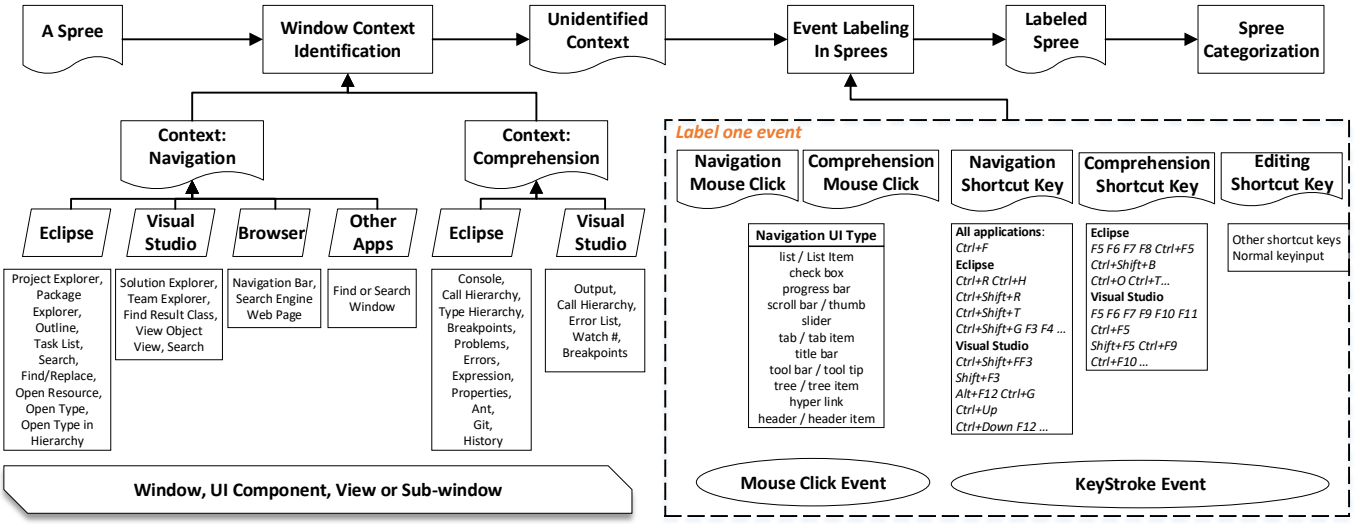

Fig. 4: The Process of Spree Categorization of *EventLabeler*

TABLE 4: Statistics of the 10 interviewees.

Interviewee	Professional Exp.	Program Lang.	Project
P1	> 5 Years	Java	A
P2	>5 Years	Java	A
P3	2 - 5 Years	Java	A
P4	2 - 5 Years	Java	C
P5	< 2 Years	Java	D
P6	> 5 Years	C#	E
P7	2 - 5 Years	C#	E
P8	2 - 5 Years	C#	E
P9	< 2 Years	C#	E
P10	< 2 Years	C#	E

We also ask interviewees to recall some situations when they find program comprehension particularly challenging. The purpose of this part is to allow the interviewees to speak freely about their program comprehension experience.

In the third part, we consider a list of topics related to program comprehension, and asked the interviewees to discuss these topics, especially those that they have not been discussed during the second part of the interview. The topics include the impact of different programming languages on program comprehension, the impact of project phase (development phase or maintenance phase) on program comprehension, etc.

After the interviews, we used a transcription service named LuyinBao [6] provided by a famous speech recognition comapny iFlyTek in China to transcribe the audio into text. We then read the text, and performed open card sorting [42] to group statements from the 10 interviewees into different categories. To do so, we first removed the statements which are not related to program comprehension, e.g., "*I have experiences on legacy system reengineering*". Then, we created one card for each of the statements, and the first two authors worked together to group the statements into different categories. For each statement, they first manually extracted key phrases from it. And then they grouped the statements with similar key phrases into the same category. The process is repeated until all statements made by the interviewees are mapped to at least one category. Furthermore, since all of the 10 interviewees were Chinese, we use Chinese as the main language to discuss with them. In the paper, we translate Chinese into English.

[6]http://luyin.voicecloud.cn/

3.4 Research Questions

(RQ1) How much of developers' time is spent on program comprehension? What are some common factors that increase program comprehension time?

Previous studies show program comprehension takes up half of a developer's time [12], [13], [21], [28], [53]. However, some conclusions are based on anecdotal evidence [12], [13], [53], instead of being derived from empirical studies. Some studies are performed under controlled experiment instead of real project settings [21]. Furthermore, some studies only involve a small number of participants [21], [28], and most of them are not professionals. To address the limitations of prior works, in this work, we revisit the same question by monitoring 79 developers working on 7 real world projects of 0.3-10 millions lines of code over a period totalling of 3,244 working hours.

(RQ2) Which applications do developers use in their program comprehension activities? How much time do they spend inside these applications during their program comprehension activities?

Previous studies only investigate program comprehension activities performed inside IDEs [21], [28]. We also notice that to understand a piece of source code, a developer may not only navigate and search for the related source code inside the IDE, but also search online resources, such as Stack Overflow. Investigating program comprehension activities across multiple applications can help better understand how developers perform program comprehension in practice.

(RQ3) Do different programming languages affect the percentage of time spent on program comprehension?

Different programming languages, such as Java and C#, may affect the percentage of time spent on program

TABLE 5: The average percentage of time developers spend on comprehension (Compre.), navigation, editing, and others.

Project	Compre.	Navigation	Editing	Others
ALL	**57.62%**	**23.96%**	**5.02%**	**13.40%**
A	63.37%	19.31%	5.02%	12.30%
B	55.80%	24.83%	6.36%	13.02%
C	58.86%	27.62%	3.90%	9.62%
D	53.32%	28.36%	5.31%	13.01%
E	56.15%	23.59%	5.53%	14.73%
F	64.05%	20.30%	4.66%	10.99%
G	51.80%	28.02%	4.59%	15.41%

comprehension. A number of factors (e.g., programming languages, developer experience, and project phase, etc) would affect the time spent on program comprehension, and investigating the impact of programming languages on the percentage of time spent on program comprehension could help developers understand their program comprehension activities better.

(RQ4) Do senior developers spend less percentages of their time on program comprehension?

The working experience of a developer may impact the needed time for program comprehension activities. Senior developers' behaviors are different from junior developers' behaviors, which might cause different time spent on program comprehension activities. In this research question, we investigate whether senior developers spend less time on program comprehension. The answer of this RQ can help identify the target beneficiary (e.g., senior or junior developers) for the automated program comprehension tools.

(RQ5) Do different project phases affect the percentage of time spend on program comprehension?

Different project phases, such as development and maintenance, may affect the time needed for program comprehension activities. In this research question, we investigate whether projects at different phases require different amount of program comprehension effort. Similar to RQ4, the answer of this RQ provide suggestions and references to tool vendor to design program comprehension tools when considering project phases.

4 Field Study Results

In this section, we present the results of our case study with respect to our five research questions.

(RQ1) How much of developers' time is spent on program comprehension? What are some common factors that increase program comprehension time?

Results. Table 5 presents the average percentage of time developers spend on comprehension, navigation, editing, and others for the 5 projects. We notice that **on average across the 5 projects, developers spend 57.62% of their time on program comprehension activities, followed by navigation (23.96%), others (13.40%), and editing (5.02%)**. Figure 5 presents the percentage of program comprehension time for the 5 projects. From the figure, we notice developers in different projects spend various time on program comprehension activities, which vary from 51.80% (G) - 64.05% (F). Our finding is consistent with the results reported by previous studies [12], [13], [21], [28], [53].

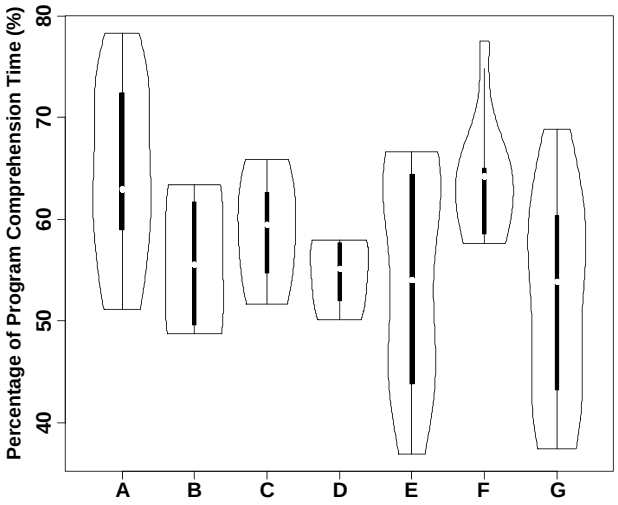

Fig. 5: Percentage of program comprehension time.

To further investigate why developers spent so much time on program comprehension, we randomly choose 200 sessions where developers spent more than 20 minutes on program comprehension from our collected data. By using the snapshots and the events provided by *ActivitySpace*, the first two authors can trace back and replay on what developers did during these sessions. We found that in the 200 long program comprehension sessions, developers used IDEs, web browsers, and text editors in 144, 171, and 120 sessions, respectively. Next, the first two authors manually concluded the root causes on why these sessions have a long program comprehension time. To do so, for each session, the two authors use one or two sentences to describe the root causes independently. Then, they worked together to discuss the root causes to make agreements. The first two authors agreed on the root causes of 170 sessions (i.e., 85% agreement), and for the sessions they cannot make the agreements, we invited another Ph.D students who has 5 years of professional experience to make the judgement. Finally, we made agreement on all of the 200 sessions.

After we concluded the root causes for each session, we grouped the sessions with the same root causes together. To do so, the first two authors manually extracted key phrases from the sentences of root causes in each session, and constructed the categories based on the key phrases. Then, we grouped sentences in the same category. Some sentences can be assigned to multiple categories since they say more than one root causes, we also broke them down into multiple categories. Table 6 presents the root causes for the 200 long program comprehension sessions. Since there might be more than one root causes from the long program comprehension time, the sum of the number of sessions for the 8 root causes is more than 200. We discussed with the 8 root causes with 10 interviewees, and we also invited them speak freely on other root causes of long program comprehension time.

1. None or insufficient comments. A lot of code we inspected has no comment or insufficient comments among the 200 sessions. For example, in Java projects, we notice a lot of comments are "TODO Auto-generated method stub" (the default comments when automatically generating a

class/method in Eclipse), or "To be added". Moreover, we also noticed that in 30 sessions, developers finally added the comments to the class/method after they spent a long time to comprehend a piece of source code.

In our interview, all of the ten interviewees agree that insufficient comments cause program comprehension difficulties. Developers "*cannot understand the source code if there are insufficient comments, especially when the source code is a bit complex*" (P6). In practice, without comments, developers have to look at the code and use bottom-up comprehension, which will cause difficulties in program comprehension. Also, sometimes comments are not updated along with the code, which in turn cause the difficulties on program comprehension. This is especially true for projects with high turnover rates; P1 stated: "*My project is in the maintenance phase, developers always leave the team to work in other new projects. Due to the lack of comments, whenever we are asked to implement a new function or fix a bug, we have to read and understand the relevant source code, which may take a long time*". Previous studies showed the turnover rate in IT companies varied from 20% - 35% [14], [50], [18], which increase the maintenance cost on program comprehension due to insufficient comments.

2. Meaningless classes/methods/variables names. Developers might need to spend more time to understand the source code if there are many meaningless classes/methods/variable names. For example, in the 200 sessions we analyzed, we notice one method "readHistory" needs to open 5 files, and the code simply name 5 " BufferedReader" instances as "br1" to "br5". When a developer comprehended this method, we noticed he frequently traced back to the definition statement of "br1" to "br5" whenever he saw the operation on these 5 files.

In our interview, nine out of the ten interviewees agree that meaningless classes or methods or variables will cause program comprehension difficulties, since it will cause the difficulty to understand the semantic meanings of classes/methods/variables, as P8 stated "*Some developers name a variable casually, such as int a, double b, which make the program hard to understand and maintain*".

3. Large number of LOC in a class/method. Some classes/methods are extremely long, e.g., more than 500 LOCs. In our interview, four out of ten interviewees mentioned large classes or methods would cause the difficulty to understand the logic since the code is complex. For example, in the 200 sessions we analyzed, one class named "StockMarketOperation", which provided the stock buying, selling, buying on margin, and selling shortly functionalities, even has more than 2,000 LOCs. A developer spent 30 minutes to comprehend this class when he was trying to locate a bug in this file.

4. Inconsistent coding styles. Due to the evolution of a software system and lack of strict style guideline, the coding styles of a project, a class, and even a method can be different. Among the 200 sessions we analyzed, 21% sessions which needed long program comprehension time are due to inconsistent coding styles. For example, class "EmailSending" has been revised by different developers to add more functionalities, and different developers have different coding styles, which cause a number of similar variables, e.g., "user_name", "UserName", "userName", and "User_Name". Some of these variables are defined as public variables, and some are defined inside a method. A developer needed to trace back multiple times to understand the meaning of the variable user name.

In our interviewee, nine out of the ten interviewees agree that inconsistent coding styles (e.g., camelCase or under_score) [8], [39] will cause program comprehension difficulties. A number of project teams do not have strict coding styles nor naming conventions; for example, a developer can name a method in the format of "helloWorld()", while others use the following formats: "Hello_World()" or "HelloWorld()". *If the source code follows multiple naming conventions, the source code is hard to understand* (P4). Some studies on program comprehension also argued whether camelCase is superior to under_score in practice [8], [39]. For example, Binkley et al. performed an eye tracking study on 135 programmers and non-programmers to understand the impact of identifier style on code readability, and they found camelCase is superior to under_score [8]. Later, Sharif and Maletic performed a replication study on Binkley et al.'s eye tracking study, and the difference between these two studies were that the participants were trained mainly in the underscore style and were all programmers [39]. They found there is no difference in accuracy between the two styles, participants recognize identifiers in the under_score style more quickly. Thus, in practice, we recommend project teams to strictly follow only one type of coding styles and naming conventions.

5. Navigating multiple times. Abstraction is one of the most important features for object-oriented programming languages. Sometimes abstraction causes additional program comprehension time since developers might navigate multiple times to find the relevant source code. For example, in our collected data, there is an abstract class named "StockExchange", and a number of classes inherit this abstract class, such as "StockExchangeChina", "StockExchangeUS", "StockExchangeIndia", and "StockExchangeSingapore". Since the project used factory design pattern to wrap the implementation of detailed classes, to locate the buggy method in the one of inherited classes, a developer needed to comprehend the method in abstract class, and navigate and comprehend to each of the inherited methods in inherited classes, and finally located the buggy method.

In our interview, seven out of the ten interviewees mention that the high-level abstractions in source code might cause the multiple navigation times. P7 stated: "*Abstraction can help to reuse the APIs in the source code, but it will also lead to difficulties in understanding the behavior of source code. For example, if class A and B are both inherited from the abstraction class C. When we are asked to write a new class D which is also inherited from C, we need to read the source code in A, B, and C to get hints on how to write class D. The process can be extremely difficult if there are a number of abstractions in the source code*". We note that all of the seven developers who share this difficulty have only worked less than 5 years. Experienced developers among our interviewees (P1, P2, and P6) however mention that they do not have this problem in understanding source code. Different from the first 3 reasons which are common to all the types of developers, the program comprehension

difficulty due to high-level abstraction can be relieved when developers gain more experience, and involve more design and development activities.

6. Query Refinement, and browse a number of search results/links. When developers search online to comprehend an exception/error/bug or an API, they might need to refine the queries multiple times to find a desired results. For example, in our collected data, a developer needed to comprehend an exception on database connection, and since he has limited experience on database connection, he input this query"how to connect database using Java" in Google. After reading and comprehending top several results, none of them are relevant. But he noticed a new word "JDBC", and refine the query as "Java Database Connection JDBC", but after reading comprehending the source code provided in the top results, none of them are relevant. Finally, he refined the query as "Java Database Connection JDBC pool", and found a relevant answer in Stack Overflow.

In our interview, only three out of ten interviewees agreed that query refinement is one of the root cause on long program comprehension time. All of these three interviewees are junior developers who worked less than 2 years. As P1 stated: "*I think with the increase of experience, it would be easy to find the suitable queries when search online*".

7. Lack of documents, and ambiguous/Incomplete document content. From our study, we noticed the some document contents are either ambiguous or incomplete, which cause developers spend long time to comprehend these documents. For example, in one design documents, description of one key functionality on the logic funding transfer rules is too short and not clear, a developer spent a long time to comprehend this functionality, and finally he decided to send emails to other colleagues.

In our interview, nine out of the ten interviewees agreed that lack of documentation, and ambiguous/Incomplete document content often leads to long program comprehension time. In our study, documentation refers to the requirement, design, and API documents. P1 and P2 who have led a project on reengineering of legacy systems told us that "*legacy systems always have no or limited documents; the first step is to manually read and understand the source code to generate documentations. We find that this process is extremely hard for the developers, and they need to spend more than 90% of their time on program comprehension*".

Nowadays, agile software development methodology is one of the most popular development methods. Paetsch et al. found in agile software development, it is infeasible to create complete and consistent requirements documents, which might cause long-term problems for agile teams [32]. And the Agile manifesto [7] also pointed out: Working software [is valued] over comprehensive documentation. Unfortunately, a limited focus on documentation in agile development will increase the program comprehension cost. P5 stated: "*Agile can increase the productivity of a developer, however, it will increase the program comprehension time when some new developers join the project team since there are limited documents to which they can refer.*"

TABLE 6: Root causes for the 200 long program comprehension sessions.

Application	Root Cause	# Sessions
IDE	None or insufficient comments	92 (46%)
	Meaningless classes/methods/variables names	75 (38%)
	Large number of LOC in a class/method	63 (32%)
	Inconsistent coding styles	42 (21%)
	Navigating inheritance hierarchy	38 (19%)
	Unfamiliarity with business logic	0 (0%)
Web Browser	Query refinement, and browse a number of search results/links	83 (42%)
Text Editor	Ambiguous/Incomplete document description	79 (40%)
	Search for the relevant documents	12 (6%)

In practice, developers love to write code more than documents[7], thus lack of documentation is problematic in every development process, which cause the difficulties in program comprehension.

8. Search for the relevant documents. In project C, we noticed they have different types of documents, e.g., requirement documents, design documents, API usage documents, and test case documents. And each type of documents have multiple versions. We found that in 12 sessions, developers spent long comprehension time on documentations since they needed to browse multiple versions of documents to find the description of a specific function implementation or a specific test case. In our interview, only one interviewee (P4) mentioned that too much documents will cause difficulty to program comprehension, since he is from project C.

Besides the 8 root causes, during the interview, we also found one more root cause on long program comprehension time, i.e., unfamiliarity with Business Logic.

9. Unfamiliarity with Business Logic. Five out of the ten interviewees mention that unfamiliarity with business logic also causes program comprehension difficulties. P1 stated: *"unfamiliarity with the business logic is very common for developers who just join a new project. For these developers, they need to read the source code and relevant documents first to understand the whole project"*. Program comprehension difficulty due to unfamiliarity with business logic is one of the common problem that a newcomer meet, and it can be relieved when the newcomer stay longer in the project team, or he/she gain more experience on software development.

Implications. In RQ1, we find that developers spend $\sim 58\%$ of time on program comprehension, which validates the well-known assumption (i.e., program comprehension takes much of developer time) that drives the line of work on improving program comprehension [12], [13], [21], [28], [53]. Our results also show that the efforts of previous studies on program comprehension are necessary, and we should still need to develop more advanced program comprehension tools to improve the performance on program comprehension. Here, we list some potential tools by mining the logs from our collected data, and interviews:

[7]http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=35336

Code and Documentation Quality Control. From our study and interviews, we found none or insufficient comments, meaningless classes/methods/variables names, large number of LOC in a class/method, inconsistent coding styles, lack of documentation, and ambiguous/Incomplete document content are all important root causes which lead to more time spent on program comprehension activities. However, an automated tool which assess the quality of code and documentation in a project could help reduce the effort on program comprehension. In our interview, four out of ten interviewees pointed out the need to assess the quality of code and its documentation. P1 stated: "*I spend a long time on program comprehension just because the code quality is low. I think if we have better code control, such as strict code review, then I can save more time on program comprehension*". Currently, to catch deadlines, project teams often do not pay much attention to documentation. There is a need for tools that can automatically extract useful documentation, beyond simple UML diagrams or Javadocs, from source code, to substantially reduce program comprehension effort.

Comments and Documentation Generation. In our study, we found none or insufficient comments, and lack of documentation are two root causes which lead to more time spent on program comprehension activities. In software engineering community, many studies are proposed to automatically generate comments [51], [44], [43], [30], and automatically generate documents [27], [20]. Our findings support these existing research studies, it would be interesting to deploy these tools into practice to improve the efficiency of program comprehension.

Automated Generation and Refinement of Search Queries. From our study, we noticed sometimes developers need to refine their queries multiple times and browse a number of search results/links to find the relevant results, which leads to more time spent on program comprehension activities. Thus, automatically generate and refine search queries based on the context in which a developer is working (e.g., by monitoring the state of his/her IDE) would help developers improve their performance of program comprehension. Some related research tools have been proposed in the literature to reformulate search queries for text retrieval in software engineering (e.g., [17]). For example, Haiduc et al proposed Refoqus which refines a user query based on the top-k (e.g., k=10) documents that retrieved by this query [17]. However, in practice, it would be possible that all top-k documents are irrelevant to the query, and for such cases, there is a need to investigate other ways to refine user queries. Thus, we still need more work to build a solution that can effectively help developers with online searching.

On average across the 7 projects, developers spend 57.62% of their time on program comprehension activities.

(RQ2) Which applications do developers use during program comprehension activities? How much time do they spend inside these applications during their program comprehension activities?

Results. In this RQ, we investigate program comprehension activities that are performed outside IDEs, the percentages of time developers spend inside various applications during these activities, and how developers switch between applications during program comprehension sessions. We calculate the lengths of time that developers spend on various applications during their program comprehension activities, and analyze the frequent sequences returned by ActivitySpace.

TABLE 7: The average percentage of time developers spent on program comprehension activities when they use IDEs, web browsers, and document editors.

Project	IDEs	Web Browsers	Document Editors
ALL	**19.95%**	**27.26%**	**10.38%**
A	36.76%	23.71%	2.91%
B	14.03%	31.26%	10.05%
C	14.04%	36.13%	8.68%
D	18.39%	34.23%	0.70%
E	16.08%	28.08%	10.45%
F	32.22%	24.13%	7.70%
G	8.58%	26.50%	16.72%

Table 7 presents the average percentages of time that developers spend when using IDEs, web browsers, and document editors to perform program comprehension activities for each of the 7 projects. **On average across the 7 projects, the percentages of the time that developers use IDEs, web browsers, and document editors to do program comprehension activities are 19.95%, 27.26%, and 10.38%, respectively.** Since we consider three groups (i.e., percentage of time developers spend when using IDEs, web browsers, and text editors during their program comprehension activities), and the distributions of percentage of time developers spend when using IDEs, web browsers, and text editors during their program comprehension activities are normally distributed as shown by the results of the Shapiro-Wilk test [38] (i.e., p-value is large than 0.05), we apply one-way analysis of variance (ANOVA) to determine whether there are any statistically significant differences between the means of these groups [45]. Table 8 presents the results for a one-way ANOVA test for the percentage of time that developers spend when performing program comprehension activities.using IDEs, web browsers, and document editors to/Since the F-value of the one-way ANOVA is 32.4, and the P-value is less than 0.001, we conclude that the difference between the different applications used to perform program comprehension activities is statistical significant.

Next, we also apply a pairwise t-test with Bonferroni Correlation [9] and Cohen's d [11][8] to determine whether the difference between different groups is statistically significant and the effect sizes are substantial. Table 9 presents Cohen's d and p-values for comparison of percentage of time that developers spend when using IDEs, web browsers, and document editors to perform program comprehension activities. We have the following observations:

1) Developers spend least time on program comprehension activities when using text editors, and the effect sizes are small and large when compared with the time that they spend using IDEs and web browsers, respectively.
2) Developers spend most time on program comprehension activities when using web browsers, and

[8]Cohen defines a D of between 0.01 to 0.20, between 0.20 and 0.50, between 0.50 and 0.80, above 0.80 as negligible, small, medium, and large effect size [11], respectively.

TABLE 8: One-way ANOVA test for percentage of time that developers spend when using different applications to perform program comprehension activities. DF = Degrees of Freedom. Sum Sq. = Sum of Square. Mean Sq. = Mean of Square.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Application	2	11,323	5,662	32.4	$3.9e^{-13}$***
Residuals	234	40,940	175	–	–

***p<0.001, **p<0.01, *p<0.05

TABLE 9: Cohen's D and P-values for comparison of percentage of time that developers spend when using different applications to perform program comprehension activities.

Application	IDE	Web Browser
Web Browser	**-0.49 (Small)*** **	–
Text Editor	**0.70 (Medium)** **	**1.55 (Large)*** **

***p<0.001, **p<0.01, *p<0.05

the effect sizes are large when compared with the time using IDEs and text editors.

Table 10 presents the top-5 frequent sequences and the percentage of program comprehension time for each sequence. We notice **developers frequently switch between IDEs and web browsers**. For example, the frequent sequence "IDE⇒Web Browser" and "Web Browser⇒IDE" correspond to 10.55% and 9.15% of the total effective working time of developers. Moreover, the frequency of switching between IDEs and document editors is much lesser. Among the top-5 frequent sequences, only "Web Browser⇒IDE⇒Document" captures the switching among web browsers, IDEs, and document editors, which corresponds to 3.35% of developers' total effective working time.

We also investigate what kinds of tasks lead to web browser use, see Table **??**. We use open card sorting [42] to group the tasks from the websites in our collected data. Our card sort process consists of two phases: In the preparation phase, we create one card for each web page. In the execution phase, cards are sorted into meaningful groups with a descriptive title. Our card sort was open, meaning we had no predefined groups; instead, we let the groups emerge and evolve during the sorting process. The first author and the other two graduate students of Zheiang University jointly sorted the card. Finally, we categorize six kinds tasks that lead to web browser use: *Communication, Project/Company Management*, *Debugging/Testing*, *Learning*, *Search for Solutions*, and others. We also count the number of web page that the developers open and calculate the percentage of web pages that belong to each task, see the last column in Table **??**. We find that the most frequent browser use is belong to *Search for solution*. Developers often need search online when they encounter some problems during software development. The search process is usually like this: First, a developer encounters a problem in IDE, e.g. an exception; then he/she switches to browser, opens the search engine and input a query; he/she open several web pages, e.g. a post in Stack Overflow, a technical blog, etc; Finally, he/she finds a solution and switch back to IDE to fix the problem. During this process, developers need to perform a lot of comprehension activities to understand the knowledge on these web pages. Anther import Another important reason that lead developers to use web browser is *Debugging/Testing*. There is at least one web application in all the studied projects and developers usually need to switch between IDE and browser frequently when they are debugging or testing the web application. These above two tasks might cause very frequent switchings between IDEs and browsers, which make developers perform programming comprehension more difficult. This is because the developers working context change fast and frequently during the switchings with across applications [4]. This suggests that effective techniques are required to track the information that flows implicitly during the context switching. Sometimes, developers use web browser to communicate with others, such as email, online forum. Developers also need to perform some *Project/Company Management* tasks through internal company websites. For instance, developers in Hengtian submit their daily and monthly task reports to manager in the task tracking system. Moreover, developers need learn programming skill and background knowledge related to project by online tutorial and company sharepoint.

TABLE 10: Top-5 frequent sequences and the percentage of program comprehension time for each sequence.

Frequent Sequence	Percentage
IDE⇒Web Browser	10.55%
Web Browser⇒IDE	9.15%
IDE⇒Web Browser⇒IDE	5.35%
Web Browser⇒IDE⇒Web Browser	4.65%
Web Browser⇒IDE⇒Document	3.35%

Interview Findings. From our interviews, all of the ten interviewees confirm that they frequently use a web browser to perform program comprehension activities. P6 stated: "*I will use web browser to search for something I cannot understand from the source code. For example, I just simply copy the piece of source code I do not understand into Bing*[9]*, and I will find something useful from the search results. It really helps me and I think the time to use web browser to do program comprehension takes half of my total time on program comprehension.*" From Table 7, we notice on average across the 7 projects, the percentages of the time developers use web browsers to do program comprehension activities is 27.26%, while the percentages of the time developers use IDEs and document editors to do program comprehension activities is 30.33%. P6's comments are consistent with our findings in Table 7.

Also, eight out of the ten interviewees complain that the frequent switching among web browsers, IDEs, and document editors may adversely impact productivity, since they *may forget what they really want to do after the switch, and they need to spend some time to recall something* (P1). P10 stated: "*although web browser and documents can help to do program comprehension, I still need to do the search process. Sometimes I cannot find the solutions that I want, so I keep on searching. Then after several tries, I may forget what I really want to do, and maybe go to visit some news in the web browser*". In practice, Mylyn[10] [19], an Eclipse plugin, can help reduce the side effect due to task switch, and improve productivity by reducing searching, scrolling, and navigation.

[9]In China, Google is blocked so developers use Bing more to search for things.

[10]http://www.tasktop.com/mylyn/resources

TABLE 11: The summary of web browser use

Task	Description	Website Example	Task Example	Perc.
Communication	Developers use some online tools in Web browser to communicate with others	Email Online Forum	*Developers write **emails** to ask something to colleagues.* *Developers discuss some interesting topics in **company forum**.*	6.5%
Project/Company Management	Currently, many project/company management systems (e.g., task tracking system, code quality management system) are web application	Intracompany Website	*Developers submit their monthly reports in **task tracking system**.*	14.2%
Debugging/Testing	If a developer works for a web application (e.g., J2EE), he usually need to visit the related web page when he is testing/debugging one certain function.	Project-related Website	*After developers receive a bug report, they open **the related web page** of the project to debug/test the related function*	24.3%
Learning	Developers learn some kinds of knowledge from online resources, such as technical tutorial, online company documentation.	Tutorial	*Developers learn project-related business knowledge through the documents on the **company sharepoint**.*	8.5%
Search for Solutions	During software development, developers often encounter lots of problems (e.g., runtime exception, configuration error) or are required to implement some kinds of code. They usually use search engines to navigate some websites and get some anwsers from the target websites.	Search engines Q&A websites API documentation Code hosting	*To solve some technical problems, developers usually use **Baidu/Bing** to search for solutions.* *Developers often visit **Stack Overflow** to find some code examples or solutions.* *Developers often visit the official API documentation (e.g., **Java API**) to know the usage of one certain API* *Developers find some popular repositories in **Github** to get a similar technical solution*	42.8%
Others	Some websites that are unrelated to developers' work	Entertainment	*When developers have a rest, they view **news** or visit **social network websites**.*	3.7%

Notice that in Table 7, time spent for program comprehension activities performed inside document editors is much lower than time spent inside IDEs and web browsers. We also check this observation with the interviewees, and seven of them agree that suitable documents are not always available or comprehensive enough. Thus, they prefer to use IDEs and web browsers more during their program comprehension activities (P1, P3, P5, P6, P7, P9, P10). P1 stated: "*due to the tight project schedule, most of the projects do not leave enough documentation. The help from the documentation is rather limited, reading the source code more or searching from the Internet can be more helpful*".

Implications. Based on the findings of RQ2, we have the following implications:

Integrating Multiple Applications into IDE. We notice that developers frequently switch between their IDEs and web browsers. Also, the percentage of time that a developer uses a web browser to perform program comprehension activities is $\sim$27%, which is more than the percentage of time that is spent on program comprehension activities performed inside IDEs or document editors. To reduce the time wasted due to switching among multiple applications, it will be worthwhile to integrate multiple relevant applications into IDEs, e.g., integrate web search functionalities into IDEs. Past studies (e.g., [34], [36]) also investigate how to integrate search engines or Stack Overflow into IDEs. Our findings support these existing research studies.

Search Engines. From RQ1, we found that query refinement might cause more time spent on program comprehension activities. From RQ2, in Table **??**, we found developers frequently search for solutions online. Thus, investigating what developers search and how they perform search activities could help understand how developers perform program comprehension activities better. In software engineering research, many previous studies (e.g., [1], [2], [24], [3], [25]) tried to develop domain-specific search engines (e.g., code search engines) to help developers to improve their search efficiency. However, it is still not clear whether domain-specific search engines can help developers improve their performance on program comprehension. Also, there are other open questions which are not answered: what do developers search online? Whether general search engines such as Google is enough to solve software engineering problems? Since these research questions are out of the scope of our paper, we plan to investigate them in the future.

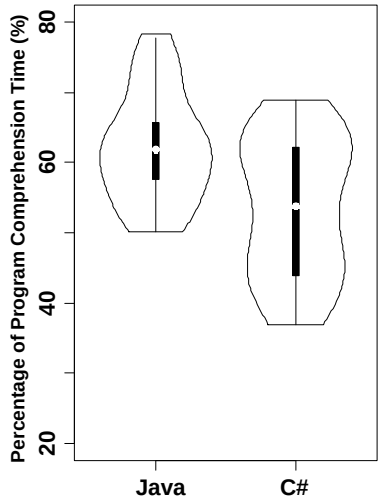

Fig. 6: Percentages of program comprehension time for different programming languages.

Aside from IDEs, developers use web browsers and document editors in their program comprehension activities. On average across the 5 projects, the percentages of time when developers use IDEs, web browsers, and document editors to do program comprehension activities are 19.95%, 27.26%, and 10.38%. Moreover, developers frequently switch between IDEs and web browsers, and the help gained from reading documents is limited.

(RQ3) Do different programming languages affect the percentage of time spent on program comprehension?

Results. In this research question, we investigate whether developers working on projects written in different programming languages spend different percentages of time on program comprehension. To address RQ3, we divide the 7 projects with two groups, i.e., Java and C#. The Java group consists of A, C, D, and F, and the C# group consists of B, E, and G.

One-way ANOVA Analysis. Figure 6 presents the percentages of program comprehension time for Java and C# projects. We notice that on average, developers working in the Java

TABLE 12: One-way ANOVA test for percentage of time that developers spend on program comprehension when working on Java and C# projects.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Language	1	1,492	1,492	18.7	$4.6e^{-5}$***
Residuals	77	6,158	80	–	–

***p<0.001, **p<0.01, *p<0.05

TABLE 13: Two-way ANOVA test for the interaction effects of the programming language of projects and the applications used for program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Language	1	497	497.4	3.3	0.0488*
Application	2	11,323	5661.6	38.0	$5.2e^{-15}$***
Lang:Appl	2	6,066	3,033	20.4	$7.1e^{-9}$***
Residuals	231	34,377	148.8	–	–

***p<0.001, **p<0.01, *p<0.05

and C# projects spend 63.22% and 53.54% of their time on program comprehension activities. Similar to RQ2, we apply one-way analysis of variance (ANOVA) to determine whether there are any statistically significant differences between the means of the two groups [45]. Table 12 presents the results for one-way ANOVA test for percentage of time that developers spend on program comprehension when working on Java and C# projects. Since F value of one-way ANOVA is 18.7, and the P-value is less than 0.001, we conclude that there is statistical significance for the time developers spent on program comprehension in Java and C# projects.

Next, we also apply pairwise T-test with Bonferroni Correlation and Cohen's d to test whether the difference between these two groups (Java and C#) are statistically significant and the effect sizes are substantial. The P-value is less than 0.001, and Cohen's D is 0.97, which corresponds to large effect size. Thus, we conclude that developers working on Java projects spend more time on program comprehension than these working on C# projects.

Two-way ANOVA Analysis. In RQ2, we investigate the percentages of time that developers spend when using IDEs, web browsers, and document editors to perform program comprehension activities. Here, we would like to investigate the interaction effects of the programming language of projects and the applications used for program comprehension. For example, we would like to investigate that whether developers in C# projects spend more time on comprehension in the web browsers than these in Java projects. To do so, we apply two-way ANOVA [9] to test the statistical significant. Two-way ANOVA extended one-way ANOVA by examining the influence of two different categorical independent variables (in our case, programming languages, and applications) on one continuous dependent variable (in our case, percentage of time on programming comprehension). Table 13 presents the results of two-way ANOVA test for the interaction effects of the programming language of projects and the applications used for program comprehension. We find that programming language of projects, applications used for program comprehension, and the interactions of these two factors all have the statistically significant impact on the percentage of time spent on program comprehension.

Next, we also apply pairwise T-test with Bonferroni Correlation and Cohens d to test whether the difference between these two factors (i.e., programming languages, and applications) are statistically significant and the effect sizes are substantial. Table 14 presents the Cohen's D and P-values for the interactions of programming languages of projects and applications used for program comprehension, we have the following conclusions:

1) Developers in C# projects spend more time on program comprehension inside the web browsers than these in C# projects using IDEs or text editors, respectively, and the effect size are large. However, there is negligible effect size and non statistical significance when comparing the time on program comprehension by using IDEs and text editors in C# projects.
2) Developers in Java projects spend less time on program comprehension inside the text editors than these in Java projects using IDEs or web browsers, respectively, and the effect size are large. However, there is negligible effect size and non statistical significance when comparing the time on program comprehension by using IDEs and web browsers in Java projects.
3) Developers in Java projects spend more time on program comprehension inside the IDEs than these in C# projects using IDEs. However, there is non statistical significance when comparing the time on program comprehension by using web browsers or text editors in C# and Java projects.

From Table 14, we also find that the main difference of time spent on program comprehension in Java and C# projects are due to difference of time spent on program comprehension inside IDEs, and we find that on average developers in Java projects spend a higher percentage of their time performing program comprehension activities inside IDEs than their counterparts that work on C# projects (28.72% vs. 11.82%).

Interview Findings. We also interview developers to better understand why Java projects need more program comprehension time. One possible reason is that Java projects often extensively use third party libraries. P5 stated: "*Different from C# projects, Java projects often use a number of third party open source libraries. These libraries lead quite often to an increased need for additional program comprehension effort, since we need to understand what is in these libraries*". To further analyze whether the number of third party libraries will affect the time spent on program comprehension, we also count the number of third party libraries used in these 7 projects by analyzing their build files (e.g., build.xml in Ant, pom.xml in Maven, or MSBuild in C#). Table 15 presents the number of third party libraries to the percentage of time spent on program comprehension. We notice Java projects will use much more third party libraries than C# projects. We use the Spearman correlation coefficient [52] to measure how much correlated two variables are C in our case, number of libraries used in the seven projects, and program comprehension time. The Spearman correlation

TABLE 14: Cohen's D and P-values for the interactions of programming languages of projects and applications used for program comprehension.

Lang.(Appl.)	C# (IDE)	C# (Web)	C# (Text)	Java (IDE)	Java (Web)	Java (Text)
C# (IDE)	–	**1.20 (Large)*** **	0.16 (Negligible)	**1.14 (Large)*** **	**1.23 (Large)*** **	-0.46 (Small)
C# (Web)	**-1.20 (Large)*** **	–	**-1.21 (Large)*** **	0.05(Negligible)	-0.12(Negligible)	**-1.86 (Large)*** **
C# (Text)	-0.16 (Negligible)	**1.21 (Large)*** **	–	**1.13 (Large)*** **	**1.29 (Large)*** **	-0.77 (Medium)
Java (IDE)	**-1.14 (Large)*** **	-0.05(Negligible)	**-1.13 (Large)*** **	–	-0.16 (Negligible)	**-1.70 (Large)*** **
Java (Web)	**-1.23 (Large)*** **	0.12(Negligible)	**-1.29 (Large)*** **	0.16 (Negligible)	–	**-2.12 (Large)*** **
Java (Text)	0.46 (Small)	**1.86 (Large)*** **	0.77 (Medium)	**1.70 (Large)*** **	**2.12 (Large)*** **	–

***p<0.001, **p<0.01, *p<0.05

TABLE 16: Spearman's rho and P-value for the number of libraries and program comprehension time. Statistically significance is in bold.

Factors	Spearman's rho	p-value
Overall Compre. Time	**0.88**	**0.008**
IDE Compre. Time	**0.81**	**0.027**
Web Browser Compre. Time	-0.31	0.504
Text Editor Compre. Time	-0.74	0.058

TABLE 17: Spearman's rho and P-value for the number of months and program comprehension time. Statistically significance is in bold.

Factors	Spearman's rho	p-value
Overall Compre. Time	0.09	0.85
IDE Compre. Time	0.04	0.94
Web Browser Compre. Time	-0.04	0.94
Text Editor Compre. Time	0.16	0.73

coefficient ranges from -1 to 1, where -1 and 1 correspond to perfect negative and positive relationships respectively, and 0 means that the variables are independent of each other. Table 16 presents Spearman's rho and P-value for the number of libraries and program comprehension time. We notice that there are high positive and statistically significant correlations between the number of libraries, and the overall program comprehension time and the time spent on program comprehension inside IDEs. Thus, the increase number of libraries can increase the number of time spent on program comprehension, especially the time spent on program comprehension inside IDEs.

Besides, one interviewee mentioned that the difference of time spent on program comprehension for Java and C# projects might be that Java projects have been existed for longer time than C# projects, since Java has been popular used for around 25 years, while C# appeared around 15 years. Since all of the seven projects are still active until April, 2017, we count the number of the months passed from the start date of the seven projects to April, 2017, as shown in Table 15. We also use the Spearman correlation coefficient to measure how much correlated two variables are C in our case, number of months passed for the seven projects, and program comprehension time. Table 17 presents Spearman's rho and P-value for the number of months and program comprehension time. We notice there are no statistically significant correlations between the number of months, and the overall program comprehension time, and the time spent on program comprehension inside IDEs, Web browses, or text editors. Thus, the number of time a project existed has limited effect to time spent on program comprehension.

TABLE 18: Two-way ANCOVA test for the interaction effects of the programming language of projects and the number of used libraries for program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Language	1	1492.3	1492.3	19.4	$3.5e^{-5}$***
Lib	1	377.1	377.1	4.9	0.03*
Lang:Lib	1	0.5	0.5	0.007	0.935
Residuals	75	5779.9	77.06	–	–

***p<0.001, **p<0.01, *p<0.05

Here, we also investigate the interaction effects of the programming language of projects and the number of libraries used in these projects to the percentage of time spent on program comprehension. Since we have a continuous independent variable (i.e., the number of libraries), and a categorical independent variable (i.e., the programming language of projects), we use a two-way ANCOVA (Analysis of covariance) [9] test to check whether the interaction effects of programming languagee and the number of used libraries have the statistically significant impact on the time spent on program comprehension. Table 18 presents the two-way ANCOVA test for the interaction effects of the programming language of projects and the number of used libraries for program comprehension. We find that programming languages of projects, and the number of used libraries have statistically significant impact on the percentage of time that is spent on program comprehension. However, the the interactions of these two factors does not have statistically significant impact on the percentage of time that is spent on program comprehension.

Another reason is that many developers find that Visual Studio (IDE for C# projects) provides a better support for program comprehension activities than Eclipse (IDE for Java projects). Among the ten interviewees, six of them have experience on both Java and C#, and all of them agreed that the difference between the IDEs play a major role in the difference in program comprehension time. They agreed that Visual Studio provides better search and navigation functions than Eclipse.

Implications. Based on the findings of RQ3, we have the following implications:

Library Usage. Different from C# which is owned by Microsoft, Java is one kinds of open source programming languages. One advantage of Java is that there are many third-party libraries, and textbooks on software engineering [35], [15] often encouraged developers to reuse the existing code instead of writing new code, in order to reduce development time. From our study, we found that using more third-party libraries would increase the time on program comprehen-

TABLE 15: Number of third party libraries and number of months to the percentage of time spent on program comprehension.

Project	Language	#No. Libs	#No. Months	% Compre.	% IDE	% Web	% Text
A	Java	22	68	63.37%	36.76%	23.71%	2.91%
C	Java	18	35	58.86%	14.04%	36.13%	8.68%
D	Java	14	18	53.32%	18.39%	34.23%	0.70%
F	Java	25	14	64.05%	32.22%	24.13%	7.70%
B	C#	4	58	55.80%	14.03%	31.26%	10.05%
E	C#	6	50	56.15%	16.08%	28.08%	10.45%
G	C#	2	22	51.80%	8.58%	26.50%	16.72%

sion. Thus, it would be interesting to investigate whether the decreased time on development is equal, larger, or smaller than the increased time on program comprehension. Moreover, considering there are a large number of third party libraries, and some are of high quality, while others are of low quality. Thus, recommending suitable libraries for software development would be useful. A previous study by Thung et al. recommended third party libraries for new software projects [47]. Our findings support their study.

Better Design of IDE. In RQ1, we found that navigating multiple times in IDEs would also lead to more time spent on program comprehension. And in RQ3, we found that the main difference of time spent on program comprehension in Java and C# projects are due to difference of time spent on program comprehension inside IDEs. In our interview, five out of ten interviewees mentioned IDEs like Eclipse do not provide sufficient support for developers to fully understand and navigate through relationships (e.g., containment, inheritance, invocations, etc.) between code elements spread in multiple source code files. Some tools are proposed to improve IDE according to developers' typical behavior [22], [10]. Ko and Myers proposed a debugging tool Whyline, which allows programmers to ask "Why did" and "Why didn't" questions about their program's output [22]. Bragdon et al. proposed Code Bubbles to help developers define and use working sets, where working sets refer to the group of functions, documentation, notes, and other information that a programmer needs to accomplish a particular programming task (e.g., feature implementation or bug fixing) [10]. Moreover, as shown from our interview findings, Eclipse community might also draw lessons from some interesting design ideas and functionalities from Visual Studio. In the future, we plan to performance another study on the difference between Eclipse and Visual Studio, and how the difference affect the performance on program comprehension.

Developers in the Java projects spend more percentages of their time on program comprehension than developers in the C# projects.

(RQ4) Do senior developers spend less percentages of their time on program comprehension?

Results. To address RQ4, we first divide the working experience of developers into 3 groups according to the number of years of professional experience, i.e., low experience (less than 3 years of professional experience), medium (3 to 5 years of professional experience), and high (more than 5 years of professional experience).

One-way ANOVA Analysis. Figure 7 presents the percentages of program comprehension time for developers with different professional experience. We notice that on average, developers of low, medium, and high experience spend 66.37%, 55.97%, and 44.43% of their time on program comprehension activities. Similar to previous RQs, we apply one-way analysis of variance (ANOVA) to determine whether there are any statistically significant differences between the means of the three groups. Table 19 presents the results for one-way ANOVA test for percentage of time that developers with difference professional experience spend on program comprehension. Since F value of one-way ANOVA is 79.4, and the P-value is less than 0.001, we conclude that there is statistical significance for the time developers with different professional experience spent on program comprehension. Table 20 presents Cohen's D and p-values for comparison of percentage of time that developers with low, medium, and high professional experience spend to perform program comprehension activities. We have the following conclusions:

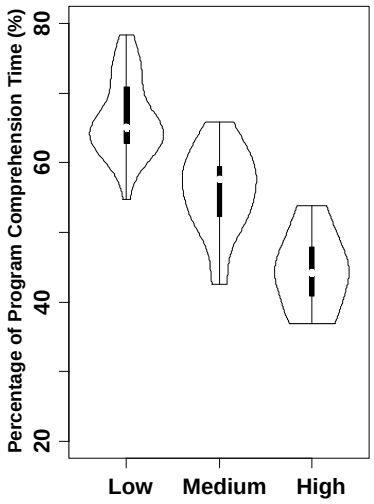

Fig. 7: Percentages of program comprehension time for developers with different professional experience.

1) Developers of low professional experience spend more time on program comprehension activities compared with these with medium and high professional experience, and the effect sizes are large.
2) Developers of medium professional experience spend more time on program comprehension activities than these of high professional experience, and the effect size is large.

Two-way ANOVA Analysis. Here, we would like to investigate the interaction effects of professional experience and the applications used for program comprehension, and we

TABLE 19: One-way ANOVA test for percentage of time that developers with different professional experience spend on program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Language	2	5174	2587.1	79.4	$<2.2e^{-16}$***
Residuals	76	2476	32.6	–	–

***p<0.001, **p<0.01, *p<0.05

TABLE 20: Cohen's D and P-values for comparison of percentage of time that developers with low, medium, and high professional experience spend to perform program comprehension activities.

Exp	Low	Medium
Medium	**1.80 (Large)*** **	–
High	**3.99 (Large)*** **	**1.98 (Large)*** **

***p<0.001, **p<0.01, *p<0.05

apply two-way ANOVA to test the statistical significant. Table 21 presents the results of two-way ANOVA test for the interaction effects of professional experience and the applications used for program comprehension. We find that professional experience, applications used for program comprehension, and the interactions of these two factors all have the statistically significant impact on the percentage of time spent on program comprehension.

Next, we also apply pairwise T-test with Bonferroni Correlation and Cohens d to test whether the difference between these two factors (i.e., professional experience, and applications) are statistically significant and the effect sizes are substantial. Table 22 presents the Cohen's D and P-values for the interactions of professional experience and applications used for program comprehension, we have the following conclusions:

1) Developers of low and medium experience spend less time on program comprehension inside text editors than these inside IDEs or web browsers, and the effect sizes are large.
2) Different from developers of low and medium experience, developers of high experience spend less time on program comprehension inside IDEs than these inside text editors or web browsers, and the effect sizes are large.
3) Developers of low and medium experience spend more time on program comprehension inside IDEs than these of high experience, and the effect size is large. However, there is non-statistical significance between developers of low experience and medium experience on time spent on program comprehension inside IDEs.
4) There is non-statistical significance among developers of low, medium, and high experience on time spent on program comprehension inside web browsers or text editors, although the effect sizes are small or medium.

Interview Findings. All of the ten interviewees agree that the more senior a developer is the more likely he/she spends less time on program comprehension. Senior developers accumulate enough software development experience, and some of them have done a number of similar projects before. In an IT company, to better allocate human resources,

TABLE 21: Two-way ANOVA test for the interaction effects of professional experience and the applications used for program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Experience	2	1,725	862.4	5.5	0.005**
Application	2	11,323	5661.6	36.2	$2.3e^{-14}$***
Exp:Appl	4	3,523	880.7	5.6	0.0002***
Residuals	228	35,693	156.5	–	–

***p<0.001, **p<0.01, *p<0.05

typically developers are required to do projects in the same domain. For example, P6 has done 5 projects which are all related to financial systems. The accumulated experience can help to reduce the time spent on program comprehension activities. P1 who is a senior developer stated: "*I have worked more than 7 years, and done more than 20 projects. Currently, given a requirement document, I can even know how the source code will be written since most of these projects are similar. However, if I come to a new project which I have never done before, such as a Matlab project, I will still spend a lot of time on program comprehension*".

Implications. Based on the findings of RQ4, we have the following implications:

Program Comprehension Behavior Learning. We manually checked and compared behaviors of senior and junior developers during program comprehension activities, and we noted some interesting observations. For example, when switching between an IDE and a web browser, some senior developers will first copied some code from the web browser to the IDE, and then compared the differences between the copied code and the original code in the IDE. In this way, they can reduce the time to switch between IDE and web browser multiple times. However, for some junior developers, they just simply switched between IDE and web browser multiple times, which required more time. Thus, it would be interesting to develop a tool which can automatically monitor developers' behaviors when they perform program comprehension activities, and recommend good behaviors to developers to help them reduce program comprehension time. The good behaviors can possibly be learnt automatically by mining behavior patterns from activities of senior developers.

Senior developers spend less time on program comprehension activities than novices/less experienced developers.

(RQ5) Do different project phases affect the percentage of time spend on program comprehension?

Results. To address RQ5, we divide the 7 projects into two groups, i.e., development phase and maintenance phase. The development phase group contains C, D, E, and G, and the maintenance phase group contains A, B, and F.

One-way ANOVA Analysis. Figure 8 presents the percentage of time spent on program comprehension activities for projects in the development and maintenance phases. We notice that on average, developers of projects in the development phase and those in the maintenance phase spend 53.54% and 63.22% of their time on program comprehension activities. Table 23 presents the results for one-way ANOVA test for percentage of time that developers in projects of maintenance and development phases spend on program

TABLE 22: Cohen's D and P-values for the interactions of professional experience and applications used for program comprehension.

Exp(Appl)	Low(IDE)	Low(Web)	Low(Text)	Med(IDE)	Med(Web)	Med(Text)	High(IDE)	High(Web)	High(Text)
Low(IDE)	–	0.18 (Neg)	**-1.00(Lar)*** **	-0.36(Sma)	0.15(Neg)	**-1.37(Lar)*** **	**-1.13(Lar)*** **	-0.31(Sma)	-0.63(Med)
Low(Web)	-0.18 (Neg)	–	**-1.67(Lar)*** **	-0.68(Med)	-0.05(Neg)	**-2.39(Lar)*** **	**-1.95(Lar)*** **	-0.69(Med)	**-1.19(Lar)***
Low(Text)	**1.00(Lar)*** **	**1.67(Lar)*** **	–	0.73(Med)	**1.56(Lar)*** **	-0.45(Sma)	-0.33(Sma)	0.91(Lar)	-0.47(Sma)
Med(IDE)	0.36(Sma)	0.68(Med)	-0.73(Med)	–	-0.06(Neg)	**-0.33(Sma)** **	**-0.95(Lar)***	0.06(Neg)	-0.33(Sma)
Med(Web)	-0.15(Neg)	0.05(Neg)	**-1.56(Lar)*** **	0.06(Neg)	–	**-2.21(Lar)*** **	**-1.82(Lar)*** **	-0.62(Med)	**-1.09(Lar)***
Med(Text)	**1.37(Lar)*** **	**2.39(Lar)*** **	0.45(Sma)	**0.33(Sma)** **	**2.21(Lar)*** **	–	-0.06(Neg)	**1.58(Lar)***	1.1(Lar)
High(IDE)	**1.13(Lar)*** **	**1.95(Lar)*** **	0.33(Sma)	**0.95(Lar)***	**1.82(Lar)*** **	0.06(Neg)	–	1.18(Lar)	-0.80(Med)
High(Web)	0.31(Sma)	0.69(Med)	-0.91(Lar)	-0.06(Neg)	0.62(Med)	**-1.58(Lar)***	-1.18(Lar)	–	-0.45(Sma)
High(Text)	0.63(Med)	**1.19(Lar)***	-0.47(Sma)	0.33(Sma)	**1.09(Lar)***	-1.1(Lar)	-0.80(Med)	0.45(Sma)	–

***p<0.001, **p<0.01, *p<0.05

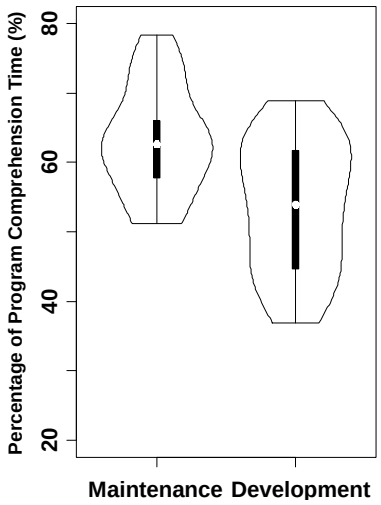

Fig. 8: Percentages of program comprehension time for projects in different phases.

TABLE 23: One-way ANOVA test for percentage of time that developers in projects of maintenance and development phases spend on program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Phase	1	1,802	1,802	23.7	$5.8e^{-6}$***
Residuals	77	5,847	75.9	–	–

***p<0.001, **p<0.01, *p<0.05

comprehension. Since F value of one-way ANOVA is 23.7, and the P-value is less than 0.001, we conclude that there is statistical significance for the time developers spent on program comprehension in projects of maintenance and development phase.

Next, we also apply pairwise T-test with Bonferroni Correlation and Cohen's d to test whether the difference between these two groups (maintenance and development) are statistically significant and the effect sizes are substantial. The P-value is less than 0.001, and Cohen's D is 1.11, which corresponds to large effect size. Thus, we conclude that developers working on maintenance projects spend more time on program comprehension than these working on development projects.

Two-way ANOVA Analysis. Here, we would like to investigate the interaction effects of the project phases and the applications used for program comprehension, and we apply two-way ANOVA to test the statistical significant. Table 24 presents the results of two-way ANOVA test for

TABLE 24: Two-way ANOVA test for the interaction effects of the project phases and the applications used for program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Phase	1	601	601	4.4	0.0372*
Application	2	11,323	5661.6	41.3	$4.5e^{-16}$***
Phase:Appl	2	8,701	4354.8	31.8	$6.3e^{-13}$***
Residuals	231	31,630	136.9	–	–

***p<0.001, **p<0.01, *p<0.05

the interaction effects of project phases and the applications used for program comprehension. We find that project phases, applications used for program comprehension, and the interactions of these two factors all have the statistically significant impact on the percentage of time spent on program comprehension.

Next, we also apply pairwise T-test with Bonferroni Correlation and Cohens d to test whether the difference between these two factors (i.e., project phases and applications) are statistically significant and the effect sizes are substantial. Table 25 presents the Cohen's D and P-values for the interactions of project phases and applications used for program comprehension, we have the following conclusions:

1) Developers in development projects spend more time on program comprehension inside the web browsers than these in development projects using IDEs or text editors, respectively, and the effect size are large. However, there is small effect size and non statistical significance when comparing the time on program comprehension by using IDEs and text editors in development projects.
2) Developers in maintenance projects spend less time on program comprehension inside the text editors than these in maintenance projects using IDEs or web browsers, respectively, and the effect size are large. However, there is small effect size and non statistical significance when comparing the time on program comprehension by using IDEs and web browsers in maintenance projects.
3) Developers in maintenance projects spend more time on program comprehension inside the IDEs than these in development projects using IDEs (28.72% vs. 11.46%). And Developers in maintenance projects spend less time on program comprehension inside the text editors than these in development projects using text editors (5.71% vs. 13.72%). However, there is small effect size and non

TABLE 25: Cohen's D and P-values for the interactions of project phases and applications used for program comprehension.

Lang (Appl)	Dev (IDE)	Dev(Web)	Dev(Text)	Main (IDE)	Main (Web)	Main (Text)
Dev(IDE)	–	**1.28 (Large)*** **	0.20 (Small)	**1.48 (Large)*** **	**1.21 (Large)*** **	-0.53 (Medium)
Dev(Web)	**-1.28 (Large)*** **	–	**-1.27 (Large)*** **	0.24(Small)	-0.21(Small)	**-1.98 (Large)*** **
Dev(Text)	-0.20 (Small)	**1.27 (Large)*** **	–	**1.52 (Large)*** **	**1.25 (Large)*** **	**-0.94 (Large)***
Main (IDE)	**-1.48 (Large)*** **	-0.24(Small)	**-1.52 (Large)*** **	–	-0.47 (Small)	**-2.20 (Large)*** **
Main (Web)	**-1.21 (Large)*** **	0.21(Small)	**-1.25 (Large)*** **	0.47 (Small)	–	**-2.24 (Large)*** **
Main (Text)	0.53 (Medium)	**1.98 (Large)*** **	**0.94 (Large)***	**2.20 (Large)*** **	**2.24 (Large)*** **	–

statistical significance when comparing the time on program comprehension by using web browsers in maintenance and development projects.

Interview Findings. There are several reasons for the difference in the percentages of program comprehension time for developers of projects in development phase and maintenance phase. First, in the development phase, the project team is relatively stable, but in the maintenance phase, some developers will leave and some new developers will join the project team. P6 stated: "*The high turnover rate for project in the maintenance phase causes the long program comprehension time. Sometimes, even 50% of the developers will leave my team. The newcomers need to spend more time to understand the source code*".

Second, in the development phase, developers focus on understanding requirements; while in the maintenance phase, developers focus on understanding the source code. P3 stated: "*In the development phase, we spend more time on understanding the requirements but less time on the source code. Understanding requirements is high level, while understanding code is low level, which will take much more time*".

Third, the lines of code (LOCs) of projects in the development phase are much less than the LOCs in the maintenance phase. Thus, the workload to understand the source code in the development phase is much less than that in the maintenance phase. P9 stated: "*the search space for projects in the development phase and maintenance phase is different. The larger number of LOCs for projects in the maintenance phase translates to the need to put more effort to searching for relevant source code, and hence lead to more time on program comprehension activities.*".

Implications. Based on the findings of RQ5, we have the following implications:

Code Search. In RQ5, we found one important reason that developers in maintenance projects spend more time on program comprehension is the increasement of source code and relevant documentation. So, a effective code search tool can help developers find the target information quickly. Furthermore, if such code search tool can link the source code to other materials during software development and maintenance, this will make developers understand source code more effectively.

Developer Turnover Management. We also found the high developer turnover rate in maintenance phase make developer spend more time on programming comprehension. Many researchers have studied developer turnover. For example, Mockus finds that only leavers have relationship with software quality since the loss of knowledge and experience [29]. On the contrary, Foucault et al. find that newcomers have a relationship with quality and leavers do not have such relationship [14]. Understanding developer turnover can help the company retain talented developers and reduce the loss due to developers departure. The talented developers who remain in the project can spend less time on programming comprehension. This is one very important factor that makes the project success.

Developers of projects in the maintenance phase on average spend a higher percentage of their time on program comprehension activities than developers of projects in the development phase. A statistical test shows that the difference is significant.

5 Discussion

5.1 Different Settings of Reaction Time (RT)

In this study, we set reaction time (*RT*) value to 1 when computing programming comprehension time. This might be a threat to validity. The range of *RT* value is usually from 0.5 to 1.5 seconds, which depends on different human factors (e.g. personality, age, etc.) and the task on the hand [49]. Hence, we also try different *RT* values ($[0.5, 0.8, 1, 1.2, 1.5]$) to investigate the effects on our findings. Table 26 shows the average percentage of time developers spend on comprehension, navigation, editing, and others in different *RT* values. We find that the larger the *RT* value is, the less the percentage of *comprehension* time is. On the other hand, the percentage of *navigation* time become larger as the *RT* value increases. This result makes sense because all intervals that is larger than *RT* among developers' interactions are computed as *comprehension* in our study. However, we think these results in different *RT* values do not affect our findings. In all results, the comprehension activities take more than half of developers' working time, which is consistent with previous studies [12], [13], [21], [28], [53]. Furthermore, the results of all individual developers in our study is consistent with the average results in Table 26. So, the different *RT* values do not affect our findings about the effect of program language, developer experience, and project phase on program comprehension. Moreover, as we shown in Section 3.2.6, when we set RT to be 1 second, our approach shows similar results as manual annotations. Thus, in this paper, we set RT as 1 second.

5.2 Cross-Company Analysis

Our study collect data from two companies Hengtian and IGS. Projects A and G are fom IGS, and projects B to F are from Hengtian. Here, we would like to investigate whether the developers in these two companies spend similar time on program comprehension. The answer of this question will affect the generalizability of our study, e.g., if we find that developers in different companies spend different

TABLE 26: The average percentage of time developers spend on comprehension (Compre.), navigation, editing, and others in different *RT* values.

RT	Compre.	Navigation	Editing	Others
0.5	61.05%	17.01%	5.70%	16.24%
0.8	59.15%	21.38%	5.50%	13.97%
1.0	58.87%	24.83%	6.36%	9.94%
1.2	56.78%	25.85%	4.95%	12.42%
1.5	53.03%	31.28%	4.45%	11.23%

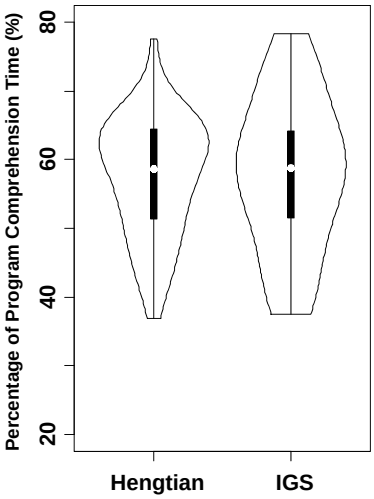

Fig. 9: Percentages of program comprehension time for developers in Hengtian and IGS.

time on program comprehension, company will become a dominant factor to affect time on program comprehension.

One-way ANOVA Analysis. Figure 9 presents the percentage of time spent on program comprehension activities for developers in Hengtian and IGS. We notice that on average, developers in Hengtian and IGS spend 57.49% and 57.68% of their time on program comprehension activities. Table 27 presents the results for one-way ANOVA test for percentage of time that developers in projects of maintenance and development phases spend on program comprehension. Since P-value is larger than 0.05, we conclude that there is non-statistical significance for the time developers in the two companies spent on program comprehension.

Two-way ANOVA Analysis. We would like to investigate the interaction effects of the companies and the applications used for program comprehension, and we apply two-way ANOVA to test the statistical significant. Table 28 presents the results of two-way ANOVA test for the interaction effects of companies and the applications used for program comprehension. We find that the interaction of these two factors has the non-statistically significant impact on the percentage of time spent on program comprehension.

TABLE 27: One-way ANOVA test for percentage of time that developers Hengtian and IGS spend on program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Company	1	0.7	0.7	0.007	0.9338
Residuals	77	7649	99.3	–	–

***p<0.001, **p<0.01, *p<0.05

TABLE 28: Two-way ANOVA test for the interaction effects of companies and the applications used for program comprehension.

Factor	DF	Sum Sq.	Mean Sq.	F Value	P Value
Company	1	0	0.2	0.0001	0.9711
Appl	2	11,323	5661.6	37.4	$4.0e^{-16}$***
Comp:Appl	2	547	273	1.6	0.2117
Residuals	231	40,393	174.9	–	–

***p<0.001, **p<0.01, *p<0.05

From the above analysis, we conclude that developers in these two companies spend similar time on program comprehension, thus our results can be generabilized to other companies.

5.3 Feedback from Participants

After completing our paper, we have also sent the results section to the interviewees, and asked them to validate the findings. All of them agree that our observations and writing are consistent with their raised intents during the interviews. Some comments we collected are:

- I really like the (nine) root causes concluded by the authors. I will ask my team members to write comments in source code, to reduce the difficulty in program comprehension.
- It is interesting to note (that) developers spend most of the time on program comprehension inside Web browser. Although I know I use web search frequently, I never notice that I even spent more time inside Web browsers than inside IDEs. Yes, I agree context switch will cause an increase of time spent on program comprehension.
- Java and C# are two most popular programming languages, and I have experience on both of the two programming languages. From my experience, when I develop Java projects, I spent more time on code understanding, since we would use a large amount of external code. I also agree that third-party library usage might be the cause of the difference of the time spent on program comprehension between Java and C#.
- With the development experience increased, the time spent on program comprehension will be decreased. It is a useful finding, since we can encourage developers work hard when they are young, so when they become senior, they can have more time to relax.
- As an outsourcing company, there are many projects in the maintenance phases. Sometimes, the boss think maintaining a project is much easier than developing a project, and thus we should deliver a maintenance project on time. However, we (developers) do not agree with that. The finding of the paper provides us the evidence, and we will use it to argue with our boss next time.

5.4 Limitations

Threats to Construct Validity. One of the threats to construct validity relates to the ability of *ActivitySpace* to accurately infer program comprehension activities. There could

be activities that are wrongly labelled. Still, we have done many steps to minimize errors, e.g., detecting and removing idle time, ignoring accesses to websites which are irrelevant to software development, etc.

Another threat relates to wrong conclusions that we draw about participant's perceptions from their comments. To minimize this threat, we have recorded our interviews and listened to it several times. Also, the first two authors work together to ensure that the results are accurate. After completing our paper, we have also sent it to the interviewees, and asked them to validate the findings. All of them agree that our findings are consistent with their interviews.

Threats on External Validity. The number of participants that we monitor and interview is limited. In total, we monitor 79 developers for a total of 3,244 working hours and interview 10 of them. All these developers come from 2 companies. Although these numbers may limit the generalizability of our study, the number of developers we interview are on par with other interview-based studies [31], [16], and the number of developers we monitor are more than other studies that also monitor developers, e.g., [21], [28]. Furthermore, many of our participants have worked in many other companies before, and have experience with developing projects in various programming languages and sizes. Besides, our study is setup in two companies, and our cross-company analysis shows that the time spent on program comprehension on these two companies are non-statistically significant, however, it is still not clear whether our conclusions are still held if we analyze the time spent on program comprehension for more developers from more companies. In the future, we plan to reduce these threats further by monitoring and interviewing an even larger number of developers across more companies over a longer period of time.

.

6 Conclusion and Future Work

In this paper, we present a large-scale field study on how program comprehension is performed in practice. We record the activities of 79 developers working on 7 real industrial projects spanning a period totaling of 3,244 working hours. We analyze this recorded data, and we find that on average, developers spend up to $\sim 58\%$ of their time on program comprehension, and they frequently use web browsers and document editors to perform program comprehension activities. Our findings are validated through relatively extensive empirical data long-held assumptions about program comprehension, including that senior developers spend less time on program comprehension, more time on program comprehension is required in the maintenance phase, and that program comprehension activities occupy a non-trivial amount of a developer's day.

In the future, we plan to send out a survey to study practitioners' perception on program comprehension to better understand the conclusion of our study, and design better program comprehension tools to help developers improve their productivity, e.g., we plan to integrate online resources into IDEs. We also plan to mine historical expert/senior developers' program comprehension behaviors to recommend good behaviors to novice/junior developers. Moreover, we plan to conduct an even larger field study with more projects and developers to further reduce the threats to validity and more precisely quantify the effect of different factors on program comprehension.

References

[1] Krugle. http://opensearch.krugle.org/projects/, March 2014.

[2] Koders. http://www.koders.com, March 2016.

[3] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Proceedings of the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[4] L. Bao, J. Li, Z. Xing, X. Wang, X. Xia, and B. Zhou. Extracting and analyzing time-series hci data from screen-captured task videos. *Empirical Software Engineering*, pages 1–41, 2016.

[5] L. Bao, Z. Xing, X. Wang, and B. Zhou. Tracking and analyzing cross-cutting activities in developers' daily work. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[6] L. Bao, D. Ye, Z. Xing, and X. Xia. Activityspace: A remembrance framework to support interapplication information needs. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (Tool Track)*, 2015.

[7] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[8] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To camelcase or under_score. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 158–167. IEEE, 2009.

[9] S. Boslaugh. *Statistics in a nutshell*. " O'Reilly Media, Inc.", 2012.

[10] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464. ACM, 2010.

[11] J. Cohen. Statistical power analysis for the behavioral sciences lawrence earlbaum associates. *Hillsdale, NJ*, pages 20–26, 1988.

[12] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[13] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. *Proceedings Guide*, 48, 1983.

[14] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 829–841. ACM, 2015.

[15] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[16] M. Greiler, A. van Deursen, and M. Storey. Test confessions: a study of testing practices for plug-in systems. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 244–254. IEEE, 2012.

[17] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 842–851. IEEE, 2013.

[18] J. J. Jiang and G. Klein. Supervisor support and career anchor impact on the career satisfaction of the entry-level information systems professional. *Journal of management information systems*, 16(3):219–240, 1999.

[19] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.

[20] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *ACM Transactions on Information Systems (TOIS)*, 31(1):1, 2013.

[21] A. J. Ko, B. Myers, M. J. Coblenz, H. H. Aung, et al. 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.

[22] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.

[23] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[24] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: using test-cases to search and reuse source code. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering (ASE)*, pages 525–526. ACM, 2007.

[25] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[26] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31, 2014.

[27] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290. ACM, 2014.

[28] R. Minelli, A. Mocci, and M. Lanza. I know what you did last summer- an investigation of how developers spend their time. In *Proc. of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*, pages 25–35. IEEE, 2015.

[29] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 67–77. IEEE Computer Society, 2009.

[30] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 23–32. IEEE, 2013.

[31] E. R. Murphy-Hill, T. Zimmermann, and N. Nagappan. Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development? In *Software Engineering (ICSE), 2014 36th International Conference on*, pages 1–11, 2014.

[32] F. Paetsch, A. Eberlein, and F. Maurer. Requirements engineering and agile software development. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 308–313. IEEE, 2003.

[33] H. Pashler. Dual-task interference in simple tasks: data and theory. *Psychological Bulletin*, 116(2):220–44, 1994.

[34] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[35] R. S. Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[36] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 194–203. IEEE, 2014.

[37] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering*, pages 255–265. IEEE Press, 2012.

[38] S. Shaphiro and M. Wilk. An analysis of variance test for normality. *Biometrika*, 52(3):591–611, 1965.

[39] B. Sharif and J. I. Maletic. An eye tracking study on camelcase and under_score identifier styles. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 196–205. IEEE, 2010.

[40] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[41] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 173–175. IEEE, 2005.

[42] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[43] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[44] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 101–110. ACM, 2011.

[45] B. G. Tabachnick, L. S. Fidell, and S. J. Osterlind. Using multivariate statistics. 2001.

[46] B. E. Teasley. The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies*, 40(5):757–770, 1994.

[47] F. Thung, L. David, and J. Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE 2013): Proceedings: Koblenz, Germany, 14-17 October 2013*, pages 182–191, 2013.

[48] A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[49] G. M. Weinberg. The psychology of computer programming. 1998.

[50] A. Whitaker. What causes it workers to leave. *Management Review*, 88(9):8, 1999.

[51] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567. IEEE, 2013.

[52] J. H. Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.

[53] M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon. *Principles of software engineering and design*. Prentice-Hall Englewood Cliffs, 1979.