10-2018

Measuring program comprehension: A large-scale field study with professionals

Xin XIA
Zhejiang University

Lingfeng BAO
Singapore Management University, lfbao@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Zhengchang XING
Australian National University

Ahmed E. HASSAN
Queen's University

See next page for additional authors

Published in IEEE Transactions on Software Engineering, 2017 July, Volume PP, Issue 99, Pages 1-26
http://doi.org/10.1109/TSE.2017.2734091

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 a considerable amount of time on program comprehension activities. 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, and investigate the program comprehension activities only within the IDEs. However, developers' program comprehension activities go well beyond their IDE interactions. In this paper, we extend our ActivitySpace framework to collect and analyze Human-Computer Interaction (HCI) data across many applications (not just the IDEs). We follow Minelli et al.'s approach to assign developers' activities into four categories: navigation, editing, comprehension, and other. We then measure the comprehension time by calculating the time that developers spend on program comprehension, e.g. inspecting console and breakpoints in IDE, or reading and understanding tutorials in web browsers. Using this approach, we can perform a more realistic investigation of program comprehension activities, through a field study of program comprehension in practice across a total of seven real projects, on 78 professional developers, and amounting to 3,148 working hours. Our study leverages interaction data that is collected across many applications by the developers. Our study finds that on average developers spend ∼58% of their time on program comprehension activities, and that 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 on the time that is spent on program comprehension, and we find senior developers spend significantly less percentages of time on program comprehension than junior developers. Our study also highlights the importance of several research directions needed to reduce program comprehension time, e.g., building automatic detection and improvement of low quality code and documentation, construction of software-engineering-specific search engines, designing better IDEs that help developers navigate code and browse information more efficiently, etc.

Index Terms—Program Comprehension, Field Study, Inference Model

1 Introduction

Program comprehension (aka., program understanding, or source code comprehension) is a process where developers actively acquire 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 [13], [15], [26], [36], [63]. Zelkowitz et al. claim that program comprehension takes more than half of the time spent on software maintenance [63]. A claim which is also confirmed by Fjeldstad and Hamlen [15], and Corbi [13]. Ko et al. find through controlled experiments on two debugging tasks and 10 participants, that understanding a program occupies around 35% of the total time [26]. Minelli et al. study the IDE interactions of 18 developers over 700 working hours, and find that on average developers spend 70% of their time performing program comprehension activities [36]. 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 investigats program comprehension activities within the IDE.

Current empirical studies that examine the role of program comprehension for software development have many shortcomings, most notable are: (1) several conclusions are based on anecdotal evidences [13], [15], [63], instead of empirical experiments on developers; (2) most prior studies are performed under controlled experiment with artificial setting, making difficult to generalize the results, e.g., [26]; (3) most prior studies involve a small number of participants (e.g., Ko et al.'s study has 10 participants [26], while Minelli et al.'s study has 18 participants [36]), and most of the participants are not professionals; (4) most prior studies only investigate program comprehension activities that occur within IDEs [26], [36]. 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.

- *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.*

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 activities *across many applications* that are used by developers instead of only using interaction data that is gathered from IDEs. Similar to past studies [13], [15], [26], [36], [63], our study tries to validate a well-known assumption (i.e., program comprehension takes much of the efficiency 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.

Relative to prior studies, our study is based on the analysis of a large number of developer activity data from their real working environment. Hence, we leverage the methodology of Minelli et al. [36] since their methodology can automatically infer developers' activities from developers' low-level interaction data. Different from several other previously-proposed models of program comprehension, e.g. [13], [58], our program comprehension model separates navigation from other activities since we believe that navigation is an important activity in software development and we want to highlight it separately. Ko et al. found that developers usually find the target information by navigating "information scents" [26], e.g., hyperlinks on a web page. Their study leads to a model of program comprehension grounded in the theories of information foraging. However, current IDEs do not support navigation well. For example, if a developer loses track of a relevant code fragment within Eclipse as she switches to other tasks, she is forced to find it again. Identifying navigation actions from developers' activities can give us deeper insight of developer's behavior. For example, we can get the procedural knowledge, which describes actions and manipulations that are carried out to complete programming tasks. We not only can know "what a developer codes", but also know "how a developer codes". However, separating navigation from other activities, e.g. coding and debugging, is very difficult, since developers interleave navigation and other activities. Furthermore, the process used in many prior studies has some limitations in a real working environment and requires extensive manual analysis. For example, Ko et al. use screen capturing techniques to record developers' working process in which they perform two debugging tasks and three enhancement tasks [26]. The collected screen-capturing videos are transcribed into different developer actions (e.g. reading code, or editing code). Ko et al.'s results are based on subjective interpretations of developers' behaviors, but it is unrealistic to analyze our collected data manually in our study because we collect developer's activity data for an extended period of time, i.e., two weeks in this paper. Hence, we extend the work of Minelli et al. [36] to investigate program comprehension activities. The question whether or not navigation is part of comprehension is a controversial one. Our methodology which reports both comprehension and navigation time allows readers to interpret the results in both ways, i.e., readers can simply sum up navigation and comprehension time if they consider navigation as part of comprehension.

Following by Minelli et al. [36], we categorize developers' activities into four categories: navigation, editing, comprehension, and other[1]. Navigation time refers to the time that developers spend in browsing through software [52], 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 that developers spend on editing source code. Comprehension time refers to the time that developers spend in program comprehension, including inspection activities such as inspecting console and breakpoint within the IDE, or reading through a piece of code (identified by e.g., detecting mouse drifting actions). We note that sometimes developers perform navigation activities to assist program comprehension activities, however, the navigation activities only involve some quick keyboard-/mouse activities, such as rolling the mouse, or clicking 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 the activities of 78 developers across 7 projects over 3,148 working hours in total. Moreover, we interviewed 10 of these developers. Our study finds that: (1) on average program comprehension takes up ∼58% of 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 .NET projects, (4) senior developers spend a significantly less percentage of time on program comprehension than junior developers, and (5) developers working on projects that are in the maintenance phase spend significantly higher percentage 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 during software development. Our study includes a total of 78 developers across 7 projects over 3,148 working hours. This study represents the largest field study on program comprehension to date. Different from prior studies, our study is conducted in a realistic setting.
2) We investigate the impact of programming language, developers' experience, and project phase on the time that is spent on program comprehension.

Paper organization. The remainder of this paper is organized as follows. Section 2 briefly reviews related work. Section 3 elaborates on the setup of our field study setup and our data collection process. 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. A number of prior studies measure program comprehension [13], [15], [26],

[1] For more details, please refer to Section 2.2.
[2] http://www.insigmaservice.com/
[3] http://www.hengtiansoft.com/en

[36], [63]. Zelkowitz et al. [63], Fjeldstad and Hamlen [15], and Corbi [13] 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 [26]. Minelli et al. study the IDE interactions of 18 participants over 700 working hours, and they find developers spend 70% of their time performing program comprehension activities [36]. 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 *78 professionals* working on *7 industrial projects* in a *realistic setting*. We collected a large amount of interaction data (a total of 3,148 working hours) by monitoring developer activities *across many applications* that they used during 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 do developers follow to comprehend programs, (2) what sources of information do developers use, (3) what information is missing, and (4) which tools do developers use and how do they use them [48]. Our field study is different and complements Roehm et al.'s study in several aspects: First, Roehm et al.'s study does not measure program comprehension time which is the focus of our study. Second, Roehm et al.'s study observes each participant for 45 minutes, while our study observes each participant over two weeks. Third, Roehm et al.'s study is more 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 participating developers. 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 do developers use during 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 Roehm et al.'s study.

In a later work, Maalej et al. further extended Roehm et al.'s study by surveying 1,477 respondents, and they analyzed the importance of certain types of knowledge for program comprehension, and the way that developers typically access and share knowledge [34]. Different from Maalej et al.'s study, our study did not involve an online survey. We complement their study by tracking user interaction data from 78 developers for two weeks, consisting a total of 3,148 working hours. Our study also highlights findings which were not investigated in Maalej et al.'s study.

Identifying Factors Affecting Program Comprehension. A number of prior studies investigate the impact of different factors on program comprehension. Siegmund et al. investigate the relationships between programming experience and program comprehension by performing short controlled experiments (i.e., 40 minutes experiments) using students as participants [51]. Teasley report that naming style impacts on program comprehension [57]. Latoza et al. identify working habit as a factor that impacts program comprehension [29]. In our study (i.e., RQ4), similar to Siegmund et al.'s work [51], we also investigate the impact of programming experience on program comprehension. However, different from their prior study, 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 were 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 are 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 a 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).

Note 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 used programming languages, 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 who work on these 7 projects. We send emails to these 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 refers to the time when a developer stays in front of the computer, doing things which are related to the project. We exclude idle periods during which a developer usually performs personal activities (e.g., eating lunch/dinner) or attends 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 four participants who 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 an industrial conference. Also, we notice that one participant worked for more than 90 hours (i.e., 96 hours) during two weeks, and the participant informed us that since he is new to the project team, and he worked many hours per day to become familiar with the project. We removed the data collected from these five participants to reduce the noise, so in total we analyze data from 78 participants.

Table 1 presents the statistics of the seven studied projects[4]. 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, developer inputs[5], and the two companies' definition[6], in this study, we consider these three projects as large-size projects. Also, projects B, C, and F contain 1M to 3M LOCs, and 12 - 45 developers, we label them as medium-sized projects. Moreover, project D only has 0.3M LOCs, and 10 developers, and we label it as a small-sized project. 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 projects use C# (B, E, and G) as their main programming language. We also asked developers to categorize each project into either maintenance or development phase depending on whether the corresponding software product has been released or not. Four projects are in the development phase (C, D, E, and G), and three projects are in the maintenance phases (A, B, and F).

TABLE 1: Statistics of the studied projects. Start. = Start Date, # D. = No. of Developers, # S. = No. of participants in our study, Pro. = Programming Language.

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

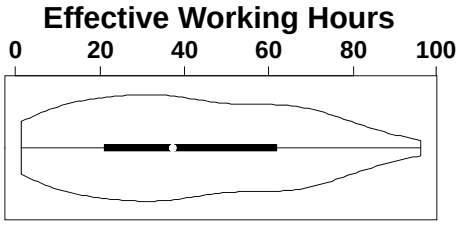

Fig. 1: A violin plot of the distribution of effective working hours.

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 our data collection and analysis process: First, we use the *ActivitySpace* framework to collect time-ordered events while a developer is interacting with applications. Then we divide a sequence of time-ordered events into working sessions by identifying idle periods and we divide each working session into sprees by the reaction time. Next, we classify these sprees by the information that is provided by the collected events. Finally, we compute the time for each of the different activities.

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

[5]From our interview, several developers mentioned that they consider projects that have more than 50 developers, 10 - 50 developers, and less than 10 developers as large, medium, and small projects respectively.

[6]The two companies define the size of a project according to the fee a client company will pay in the contract, and they simply ranked the top 25% projects with highest fee as large projects, followed by the next 50% as medium size, and the lowest 25% as small projects.

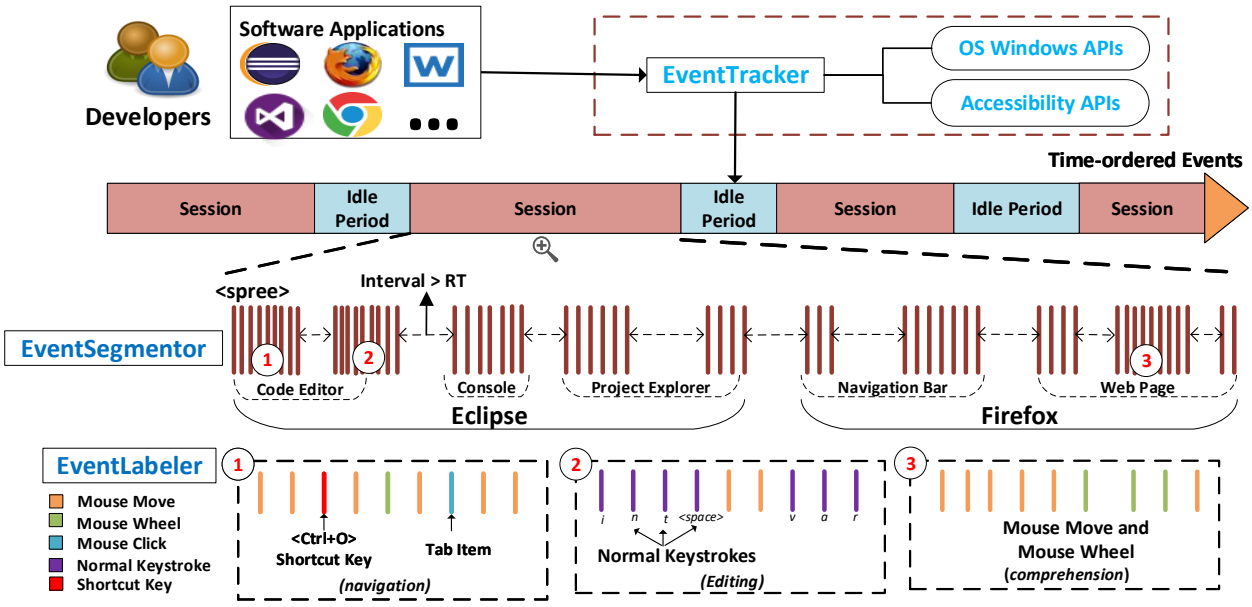

Fig. 2: Our data collection and analysis process.

3.2.1 Tracking Events

As a developer interacts with an application, *ActivitySpace* generates time-ordered events (see Figure 3 for an example). Each event has a time stamp down to milliseconds precision. Each event is composed of an event type, basic window information that is collected using OS window APIs, and the focused UI information that the application exposes to the operating system through accessibility APIs (available for mouse click event only). *ActivitySpace* monitors three types of mouse events (namely mouse move, mouse wheel, and mouse click) and two types of keyboard events (namely 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 a 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 a 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 a 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. 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 collected UI information 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 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[7]. Furthermore, if developers use command line tools to execute one task, they usually switch to other applications and do not need to wait for the results from the command line. So, the spent tme on the command line tools is very small. *ActivitySpace* generates a placeholder event of "unknown" event type when a mouse or keyboard event occurs in other applications. We analyse the proportion of time that developers spend on such "unknown" applications, and it is typically less than 2%.

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 one hour. We acknowledge that there often short idle periods (five

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

minutes in this study) when developers could have a short break or chat with other colleagues. We remove these short idle periods when calculating the comprehension time. 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 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, or planning), 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 or 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 [42]. According to this theory, developers cannot perform different activities (i.e., programming comprehension, navigation, or editing) at 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 navigation action (a scroll or a menu click to view a call hierarchy). 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 depending on human factors (e.g., personality, or age) and the task at the hand. Different settings of RT might generate different results, but Minelli et al. [36] reported that the different RT values did not affect their findings. So, in this study, we set RT at one second, following their RT setting. We also discuss the effect of different RT values in Appendix.

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

If *EventLabeler* cannot determine the category of a 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 a mouse click event, *EventLabeler* classifies the event as a *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 that developers perform, for example, if the *UI Type* is *tree item* or *scroll bar*, 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 commonly used shortcut keys in our collected data, as summarized in the *Navigation Shortcut Key* and *Comprehension Shortcut Key* part of Figure 4. *EventLabeler* labels normal keystroke events as *editing*.

If all the events in a 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 a 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 a *navigation* event, otherwise as a *comprehension* event.

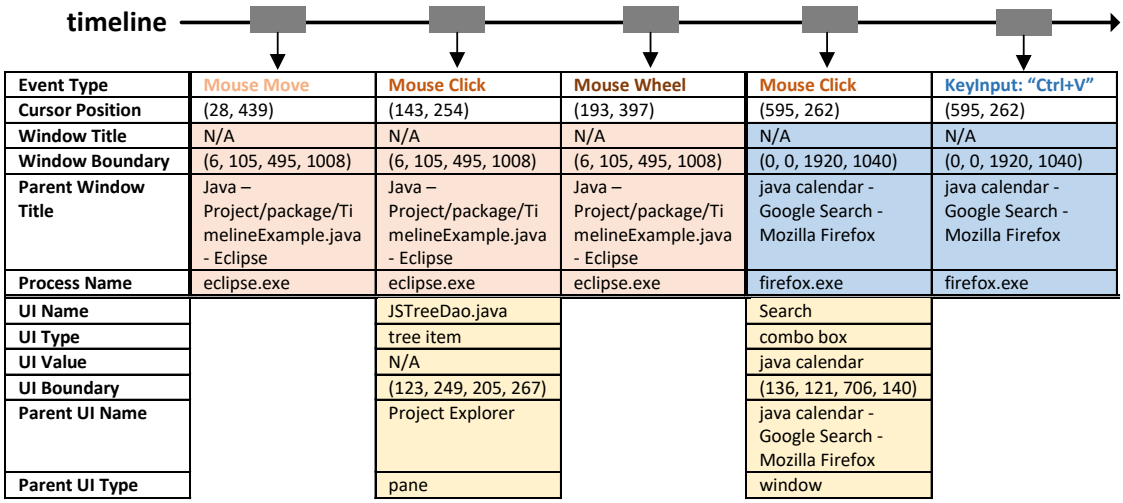

Fig. 3: An example of low level events.

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 a *navigation* event.

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 our study) and shorter than a threshold (5 minutes in our study). Based on our observation and interview, time intervals longer than 5 minutes usually represent the time period during which developers have short breaks or chat with their colleagues. We do not consider these time intervals as idle periods because a developer is still in a working mode on the computer, unlike a long meeting or a 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 the visited web pages (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 manually determined and fine recorded 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*. Note that most of the example keywords in Table 2 are translated from Chinese. Our filtering results in more accurate statistics of developers' work habits.

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*

To understand program comprehension across different applications, *ActivitySpace* identifies all application switching by identifying the difference between the process names of two consecutive events. It can find all application switching, e.g., IDE ⇒ Web Browser, IDE ⇒ Web Browser ⇒ IDE, of length 2 to 4, and compute the duration of sequences that have these application switching. 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.

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 split the collected data into many sprees using our proposed approach and link these sprees to the corresponding timestamp of the screen-capturing video. Then we confirm the categories of these sprees with the two developers. For the two developers, there are 2,840 and 1,643 sprees in total, respectively. Among these sprees, 1,051 and 343 sprees are categorized as comprehension by our ActivitySpace tool. For these ones, we ask the developers to tell us what they

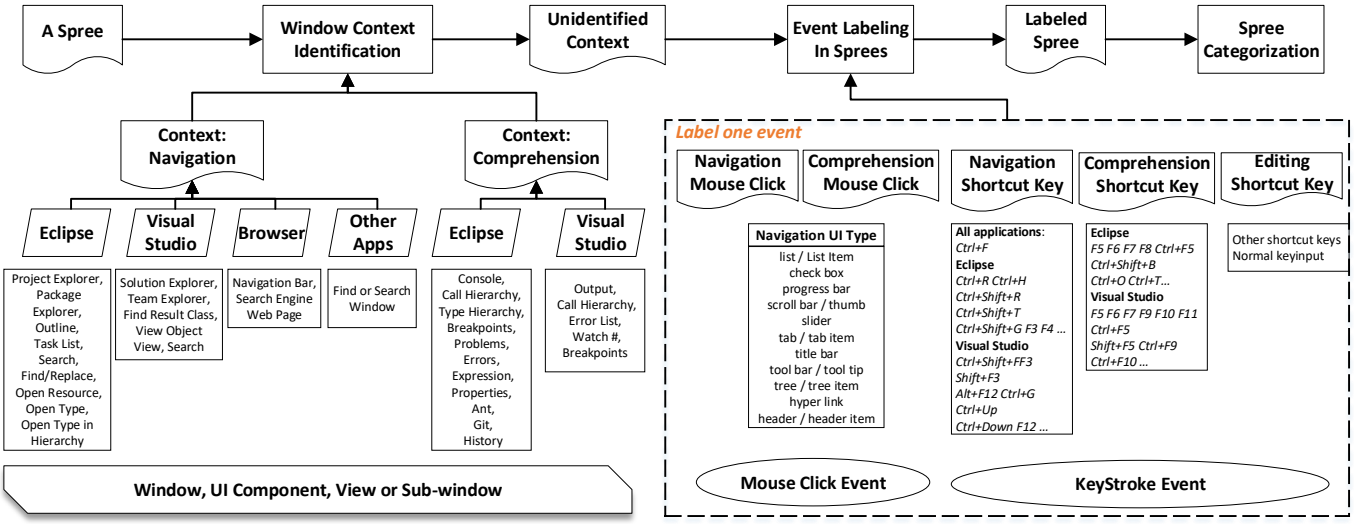

Fig. 4: The Process of Spree Categorization of *EventLabeler*

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%

did in the sprees, and categorize what they did into one of the four activities (i.e., navigation, editing, comprehension, and others). Although more than a thousand sprees need to be analyzed by the two developers, a large number of continuous sprees belong to one activity. Thus, the developers only need to recognize the boundaries of the activities, and do not need to analyze the sprees one by one.

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

3.3 Interview

In addition to analyzing the collected data, we interviewed 10 out of the 78 participants, to confirm and interpret our findings. We performed the interviews at the end of the monitoring process. We sent emails to all of the 78 participants to inquire about their availability, and 10 participants indicated their availability (seven worked on Java projects, and eight worked on C# projects). Table 4 presents the working experience, programming languages, and project teams of the 10 interviewees. The participants have varying numbers of years of professional experience.

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

TABLE 4: Statistics of the 10 interviewees. Exp. = Experience, Lang = Language.

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

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

After the interviews, we used a transcription service named LuyinBao[8] provided by iFlyTek in China to transcribe audio into text. We then read the text, and performed open card sorting [53] to group the statements from the 10 interviewees into different categories. To do so, we first removed 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. Then they grouped the statements with similar key phrases

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

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 are Chinese, we used Chinese as the main language to discuss with them. In the paper, we translated all Chinese communications into English.

3.4 Research Questions

We would like to investigate the following five 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 can take up as much as half of a developer's time [13], [15], [26], [36], [63]. However, some conclusions are based on anecdotal evidence [13], [15], [63], instead of being derived from empirical studies. Some studies are performed under controlled experiment instead of real project settings [26]. Furthermore, some studies only involve a small number of participants [26], [36], and most of them are not professionals. To address the limitations of prior works, in this work, we revisit the same question by monitoring the time spent on program comprehension activities of the 78 developers working on 7 real world projects of 0.3-10 millions lines of code over a period totalling of 3,148 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 [26], [36]. However, to understand a piece of source code, a developer may not only navigate and search for related source code inside the IDE, but also search online resources, such as Stack Overflow. Investigating program comprehension activities across multiple applications helps us better understand how developers perform program comprehension in practice.

Specially, in RQ2, we investigate the time that developers spent when using IDEs, web browsers, and text editors to perform program comprehension activities, keep in mind that these three applications represent different ways for performing program comprehension activities: in IDEs, developers mainly comprehend the source code; in web browsers, developers mainly comprehend the searched content (e.g., bug fixing solutions, feature implementation suggestions, or tool installation guides) returned by search engines; in text editors (e.g., MS Word, MS Excel, and Notepad++), developers mainly comprehend technical/project documents (e.g., project requirement or design documents). Note that some developers might use text editors to write/edit source code. For such cases, since the text editor now serves as an IDE, we count the time that they spent on text editors as program comprehension time inside IDE. We use file extension to identify whether participants edit/write/comprehend source code or not, i.e., if a participant opened or edited a file in a text editor with an extension such as "java", "cs", "c", "cpp", "h", "html", "htm", "js", or "xml", we count the time spent on such files as program comprehension time inside IDE. Moreover, three of our investigated projects (B, E, and F) need to develop web portals, and developers sometimes debug the web pages in web browsers. For such cases, since the web browser now serves as an IDE, we count the time that they spent on web browser as program comprehension time inside IDE, i.e., if a developer opened a URL such as "localhost:8080" or a specific IP address like "10.171.10.99", we still count the time spent on such web pages as program comprehension time inside IDE.

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

A number of factors (e.g., programming languages, developer experience, and project phase) would affect the time that is spent on program comprehension, and investigating the impact of programming languages on the percentage of time that is spent on program comprehension could help developers understand their program comprehension activities better. Our findings can help developers consider an additional factor when deciding which programming language to use, and help inform language and IDE designers on areas for improvement. Here, we consider the effect of two programming languages, i.e., Java and C#, on the time that is spent on program comprehension. According to a Stack Overflow survey 2017, Java and C# are two most popular programming languages[9].

(RQ4) Do the senior developers spend less 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 lead to varying time spent on program comprehension activities. In this research question, we investigate whether senior developers spend less time than junior developers (e.g., novice or less experienced developers) on program comprehension. The answer of this RQ can help identify the target beneficiary (e.g., senior or junior developers) for automated tools to improve the efficiency of program comprehension.

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

Different project phases, such as the development and maintenance phase, may affect the time spent on program comprehension activities. In this research question, we investigate whether projects at different phases require different amounts of program comprehension effort. Similar to RQ4, the answer of this RQ can provide inputs to tool builders in designing automated tools to improve the efficiency of program comprehension by 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.

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

Project	Comprehension	Navigation	Editing	Others
Average	**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%

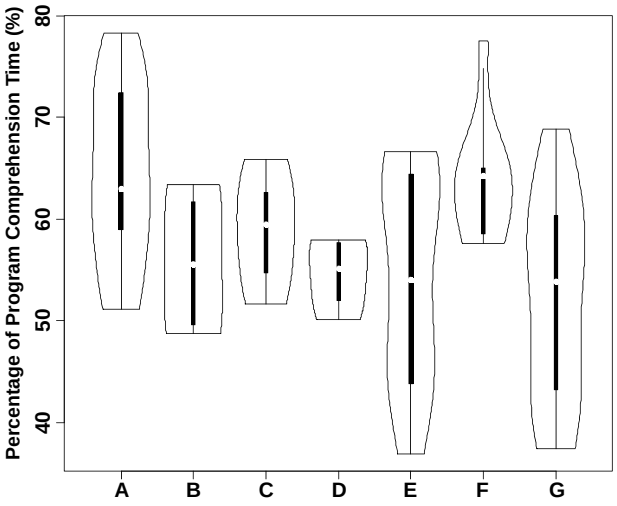

Fig. 5: A violin plot of the percentage of program comprehension time.

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

4.1.1 Results

Table 5 presents the average percentage of time that developers spent on comprehension, navigation, editing, and others for the 5 projects. **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 observe that developers in different projects spend varying time on program comprehension activities, which vary from 51.80% (G) - 64.05% (F). Our finding is consistent with previous studies [13], [15], [26], [36], [63].

To further investigate why developers spent so much time on program comprehension, we performed the following two steps: (1) we invited 10 interviewees to speak freely on other root causes of long program comprehension time, and we came out with an initial set of causes; (2) we label the 200 sessions based on these root causes, and we create new root causes if the existing ones are not sufficient. Based on the interviewee's input, we identify nine root causes as shown in Table 6, and we randomly choose from our collected data 200 sessions where developers spent more than 20 minutes on program comprehension. By using the snapshots and the events that are provided by our ActivitySpace tool, the first two authors can trace back and replay 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 tried to categorize these long sessions into the nine root causes that we derived, and we categorize the sessions which do not belong to any root causes as "others". We observe that some sessions can be assigned to multiple root causes, and thus we assign multiple causes to these sessions as needed. We use Fleiss Kappa [16] to measure the agreement between the two labelers. The overall Kappa value between the two labelers on the 200 sessions is 0.78, indicating a substantial agreement between the labelers. For the sessions for which both labelers cannot reach an agreement, we invited a Ph.D student (not one of the co-authors) who has 5 years of professional experience to make the judgement. Finally, we reached agreement on all of the 200 sessions. We find that all the 200 sessions can be put into eight out of the nine root causes that we identified – no session belongs to "unfamiliarity with business logic" category. Table 6 presents the nine root causes. Since a session can be mapped to multiple root causes, the sum of the number of sessions for the nine root causes is more than 200.

The following paragraphs describe the details of the nine root causes and the 200 sessions:

1. No comments or insufficient comments. A large amount of the code that we inspected has no comment or insufficient comments among the 200 sessions. For example, in Java projects, many comments that say "TODO Auto-generated method stub" (the default comments when automatically generating a class/method in Eclipse), or "To be added". Moreover, in 30 sessions, developers finally added comments to the class/method after they spent a long time comprehending 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 causes difficulties in program comprehension. Previous studies highlighted the importance of comments in the process of software maintenance [22], [38], [54], [55], [61]. Also, sometimes comments are not updated along with the code, which in turn causes difficulties for 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 reported that the turnover rate in IT companies varies from 20% - 35% [17], [23], [47], [60].

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 that we analyzed, we observe that one method called "readHistory"

[9]https://stackoverflow.com/insights/survey/2017

TABLE 6: Nine root causes that were identified by us.

Root Cause	Source	Application	# Sessions
No comments or insufficient comments	Interview/Session	IDE	92 (46%)
Meaningless classes/methods/variables names	Interview/Session		75 (38%)
Large number of LOC in a class/method	Interview/Session		63 (32%)
Unconsistent coding styles	Interview/Session		42 (21%)
Navigating inheritance hierarchies		38 (19%)	
Query refinement	Interview/Session	Web Browser	83 (42%)
Query Refinement, and browsing a number of search results/links	Interview/Session		42 (21%)
Lack of documents, and ambiguous/incomplete docu- ment content	Interview/Session	Text Editor	79 (40%)
Searching for the relavent documents	Interview/Session		12 (6%)
Unfamiliarity with business logic	Interview	NA	NA

needs to open 5 files, and the code simply names the needed 5 " BufferedReader" instances as "br1" to "br5". When developers comprehended this method, we noticed that they frequently traced back to the definition statement of "br1" to "br5" whenever they saw operations on these five methods.

In our interview, nine out of the ten interviewees agree that meaningless classes or methods or variables lead to program comprehension difficulties, since it increases the difficulty of understanding the semantic meanings of classes/methods/variables, as P8 stated "*Some developers name a variable casually, such as int a, double b, which makes the program hard to understand and maintain*". De Lucia et al. highlighted that name of a class/method/variable is a crucial element for program readability [14]. Lawrie et al. found that the quality of class/method/variable names affects the efficiency of program comprehension, and they recommended the use of full word identifiers [30]. Thus, in practice, we recommend developers to pay attention to the name of classes/methods/variables, and to try to use meaningful words to describe the meaning of each class/method/variable.

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 note that large classes or methods cause difficulty in understanding since the code logic is often complex. For example, in the 200 sessions that we analyzed, one class named "StockMarketOperation", which provides stock buying, selling, buying on margin, and short selling functionalities, has more than 2,000 LOCs. A developer spent 30 minutes to comprehend this class when he was trying to locate a bug. A large number of LOC in a class/method is often a sign cause of an anti-patterns named god class/method, where one class/method controls too many processes in a software system [18]. One common practice to resolve a large number of LOC in a class/method is to divide the implemented functionalities in the class/method across several focused sub-classes/methods [18], [46].

4. Inconsistent coding styles. Due to the evolution of a software system and lack of strict style guidelines, the coding styles of a project, a class, and even a method can be different. Among the 200 sessions that we analyzed, 21% of the sessions needed long program comprehension time 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 as local variables. A developer needs to trace back multiple times to understand the meaning of these similarly named variables.

In our interviewee, nine out of the ten interviewees agree that inconsistent coding styles (e.g., camelCase or under_score) [8], [50] 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 prior program comprehension studies also argue whether camelCase is superior to under_score in practice [8], [50]. For example, Binkley et al. performed an eye tracking study on 135 programmers and non-programmers to better understand the impact of identifier style on code readability, and they found that camelCase is superior to under_score [8]. Later, Sharif and Maletic performed a replication study of 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 [50]. 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 a consistent coding style and naming convention.

5. Navigating inheritance hierarchies. 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 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 the factory design pattern to wrap the implementation of detailed classes, to locate the buggy method in one of the inherited classes, a developer needed to comprehend the method in the abstract class, and to navigate and comprehend each of the inherited methods in the inherited classes, and finally located the buggy method. To reduce the effort due to navigating inheritance hierarchy, Lanza and Ducasse propose a lightweight view named polymetric views which is based on the combination of software visualization and software metrics [28].

In our interview, seven out of the ten interviewees mention that high-level abstractions in source code might increase navigation time. 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.

6. Query Refinement, and browsing a number of search results/links. When developers perform online queries to comprehend an exception/error/bug or an API, they might need to refine their queries multiple times to find desired results. For example, in our collected data, a developer needed to comprehend an exception on a database connection, and since he has limited experience on database connections, he performed this Google query[10] "how to connect a database using Java". After reading and comprehending several of the top results, he found that none of them was relevant. But he noticed a new word "JDBC", and refined the query as "Java Database Connection JDBC", however again after reading and comprehending the source code of several of the top results, none of them were considered relevant. Finally, he refined the query as "Java Database Connection JDBC pool", and found a relevant answer on Stack Overflow.

In our interview, only three out of ten interviewees agreed that query refinement is one of the root causes for 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 searching online*". Haiduc et al. highlighted the importance of query refinement in the performance of text retrieval in software engineering [21]. They proposed Refoqus which refines a user query based on the top-k (e.g., k=10) documents that are retrieved by an initial query. Nie et al. expanded a query based on crowd knowledge to improve the performance of code search [40]. The effectiveness and efficiency of search can be improved if a search engine could help refine queries intelligently.

7. Lack of documents, and ambiguous/incomplete document content. From our study, we observe that the contents of some documents are either ambiguous or incomplete, which causes developers to spend a considerable amount of time to comprehend these documents. For example, in one requirement documents, the description of the rules governing how a fund should be transferred are too short and not clear. A developer spent more than two hours to comprehend this requirement.

In our interview, nine out of the ten interviewees agreed that the 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 that it is infeasible to create complete and consistent requirements documents, which might cause long-term problems for agile teams [41]. And the Agile manifesto [7] also pointed out: Working software [is valued] over comprehensive documentation. Unfortunately, a limited focus on documentation in agile development increases the program comprehension cost. P5 stated: "*Agile can increase the productivity of a developer, however, it will increase the program comprehension time when new developers join the project team since there are limited documents to which they can refer.*"

In practice, developers prefer to write code more than documents[11], thus the lack of documentation is problematic in every development process, which causes difficulties in program comprehension.

8. Searching for the relevant documents. In project C, we noticed that they have different types of documents, e.g., requirement documents, design documents, API usage documents, and test case documents. And each type of document has 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 interviews, only one interviewee (P4) mentioned that too much documents hinders program comprehension activities.

Besides the 8 root causes, during the interviews, we also uncovered one additional root cause for long program comprehension times, i.e., unfamiliarity with business logic.

9. Unfamiliarity with Business Logic. Five out of the ten interviewees mention that unfamiliarity with business logic also hinders program comprehension activities. P1 stated: *"unfamiliarity with the business logic is very common for developers who just joined a 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 problems that a newcomer faces, and it can be relieved when the newcomer stays longer in the project team, or he/she gains more experience on software development.

4.1.2 Implications

In RQ1, we find that developers spend 58% of their time on program comprehension, which validates the well-known assumption (i.e., program comprehension takes much of developer's time) that drives the line of work on supporting program comprehension [13], [15], [26], [36], [63]. Our results also show that the efforts of previous studies on

[10]Although Google is blocked in China, developers use VPN to access Google.

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

program comprehension are necessary, and we still in need for more advanced program comprehension tools. Here, we list some potential tools based on our analysis of our collected data, and interviews:

Code and Documentation Quality Control. From our study and interviews, we found no comments 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 no comments or insufficient comments, and lack of documentation are the two root causes which lead to more time being spent on program comprehension activities. In software engineering community, many studies proposed the automated generation of comments [38], [54], [55], [61], and documents [25], [35]. Our findings support these prior research studies, it would be interesting to deploy these tools in practice to improve the efficiency of program comprehension.

Automated Generation and Refinement of Search Queries. From our study, we observe that 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 being spent on program comprehension activities. Thus, automatically generating and refining 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 during program comprehension activities. Some related research tools have been proposed in the literature to reformulate search queries for text retrieval in software engineering. For example, Haiduc et al proposed Refoqus which refines a user query based on the top-k (e.g., k=10) documents that are retrieved by an initial query [21]. However, in practice, it is possible that all top-k documents are irrelevant to the posed 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.

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

Project	IDEs	Web Browsers	Document Editors
Average	**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%

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

4.2.1 Results

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

Table 7 presents the average percentages of time that developers spent 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 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 [49] (i.e., p-value is larger than 0.05), we apply a one-way analysis of variance (ANOVA) test to determine whether there are any statistically significant differences between the means of these groups [56]. Table 8 presents the results for a one-way ANOVA test for the percentage of time that developers spent when performing program comprehension activities using IDEs, web browsers, and document editors. 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 statistically significant.

Next, we also apply a pairwise t-test with a Bonferroni correction [9] and we measure Cohen's d [12][12] 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

[12]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 [12], respectively.

TABLE 8: One-way ANOVA test for percentage of time that developers spent 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 spent 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

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 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 observe that **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 11). There are 20,678 web pages in our collected data. So, we randomly sample 3,000 web pages from the collected data and perform an open card sort to group the tasks. 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 that we had no predefined groups; instead, we let the groups emerge and evolve during the sorting process. The first author and another two graduate students of Zhejiang University (who are not co-authors of this paper) jointly sorted the card. Finally, we categorize six kinds of 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 pages that the developers open and calculate the percentage of web pages that belong to each task – see the last column of Table 11. While reading their emails, developers need to comprehend email contents (e.g., bug description and solution proposal) to complete their work. Thus, the comprehension time in a web browser also includes email time. We group email and online forum under the same task (i.e., communication) since both of them can be used for communication.

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%

We find that the most use of browser belongs to Searching for Solution category. Developers often need to search online when they encounter some problems during software development. The search process is usually as follows: First, a developer encounters a problem while working in the IDE, e.g. an exception; then he/she switches to the browser, opens the search engine and inputs a query; he/she visits several web pages, e.g., a post on Stack Overflow, a technical blog, etc; Finally, he/she finds a solution and switches back to the IDE to fix the problem. During this process, developers need to perform many comprehension activities to comprehend the knowledge on these web pages.

Another important reason that leads developers to use a web browser is Debugging/Testing. There is at least one web application in all the studied projects and developers usually need to switch frequently between the IDE and the browser when they are debugging or testing the web application.

The aforementioned two tasks might cause very frequent switchings between the IDE and the browser, which increases program comprehension cost. The increased cost is due to the developers working context changing fast and frequently during the switchings across applications [4]. This suggests that effective techniques are required to track the information that flows implicitly during the context switching.

In addition to the two above mentioned tasks, developers also need to learn programming skills and background knowledge related to their project by reading online tutorials and accessing the company's sharepoint site.

4.2.2 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 a web browser to search for something that I cannot understand from the source code. For example, I just simply copy the piece of source code that I do not understand into Bing*[13]*, 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 seven projects, the percentages of time that developers use web browsers to perform program comprehension activities is 27.26%, while the percentages

[13]In China, Google is blocked so developers use Bing more frequently instead.

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 read **emails** to comprehend reported bugs.* *Developers discuss some interesting topics in **company forums**.*	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 a **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 code skills through online turotials.*	8.5%
Searching for Solutions	During software development, developers often encounter many obstacle (e.g., runtime exceptions, or configuration errors) or are required to implement some code. They usually use search engines to get relevant anwsers.	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	Websites that are unrelated to developers' work	Entertainment	*When developers have a rest, they view **news** or visit **social network websites**.*	3.7%

of time that developers use IDEs and document editors to perform program comprehension activities is 30.33%. P6's comments are consistent with our findings listed in Table 7.

Also, eight out of the ten interviewees complain that the frequent switching among web browsers, IDEs, and document editors adversely impacts their 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 read some news in the web browser*".

Notice that in Table 7, the time spent for program comprehension activities that are 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 frequently 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*".

4.2.3 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 total percentage of time spent on program comprehension activities that are performed within IDEs and document editors. To reduce the time wasted due to the switching among multiple applications, it will be interesting to integrate multiple relevant applications into IDEs, e.g., integrate web search functions into IDEs. In practice, Mylyn[14] [24], can help reduce the side effect due to task switching, and improve productivity by reducing searching, scrolling, and navigation. Past studies (e.g., [43], [45]) also investigate how to integrate search engines or Stack Overflow into IDEs. Our findings support these prior studies.

Search Engines. From RQ1, we found that query refinement might cause more time being spent on program comprehension activities. From RQ2, in Table 11, we found that developers frequently search for solutions online. Thus, investigating what developers search and how they perform search activities could help us better understand how developers perform program comprehension activities. In software engineering research, many prior studies (e.g., [1]–[3], [31], [32]) 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? Are general search engines such as Google good enough to solve software engineering problems? Future studies are needed to further investigate these questions.

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 that 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.

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

4.3.1 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 seven projects into two groups, i.e., Java and C#. The Java group consists of projects A, C, D, and F, and the C# group consists of projects B, E, and G.

One-way ANOVA Analysis. Figure 6 presents the percentages of program comprehension time for Java and C# projects.

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

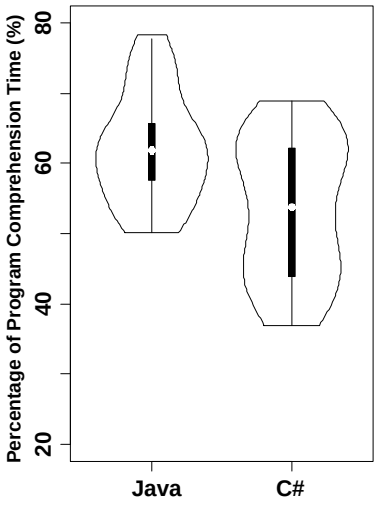

Fig. 6: A violin plot of the percentages of program comprehension time for different programming languages.

TABLE 12: One-way ANOVA test for percentage of time that developers spent 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

We notice that on average, developers working in the Java and C# projects spend 63.22% and 53.54% of their time on program comprehension activities. Similar to RQ2, since the distribution of percentage of time developers that spent program comprehension is normally distributed as shown by the results of the Shapiro-Wilk test [49] (i.e., p-value is larger than 0.05), we apply a one-way analysis of variance (ANOVA) to determine whether there are any statistically significant differences between the means of the two groups [56]. Table 12 presents the results for a one-way ANOVA test for percentage of time that developers spend on program comprehension when working on Java and C# projects. Since the F- value of the one-way ANOVA is 18.7, and the P-value is less than 0.001, we conclude that there is statistical significance difference for the time that developers spend on program comprehension in Java versus C# projects.

Next, we also measure Cohen's d to test whether the effect size between these two groups (Java and C#) is substantial. The 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 those working on C# projects (at least for our studied projects).

Two-way ANOVA Analysis. In RQ2, we investigate the percentages of time that developers spent on program comprehension activities when using IDEs, web browsers, and document editors. Here, we investigate the interaction effects of the programming language of projects and the applications (i.e., IDE, Web browser, or text editor) that are used for program comprehension. For example, we would like to investigate whether developers in C# projects spend more time on comprehension in web browsers than those developers in Java projects. Since the distributions of percentage of time that developers spent 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 (i.e., p-value is larger than 0.05), we apply a two-way ANOVA test [9]. A two-way ANOVA test extends a one-way ANOVA by examining the influence of two different categorical independent variables (in our case, programming languages, and used applications) on one continuous dependent variable (in our case, the percentage of time that is spent on program comprehension). Table 14 presents the results of our two-way ANOVA test for the interaction effects of the programming language of projects and the used applications for program comprehension. We find that the programming language of a project, the used applications for program comprehension, and the interactions of these two factors all have statistically significant impact on the percentage of time spent on program comprehension.

Next, we also apply a pairwise t-test with a Bonferroni correction and a Cohen's d to test whether the difference between these two factors (i.e., programming languages, and used applications) are statistically significant and that the effect sizes are substantial. Table 15 presents the Cohen's d and p-values for the interactions of programming languages of projects and used applications for program comprehension, we have the following observations:

1) Developers in C# projects spend more time on program comprehension inside web browsers than developers in C# projects using IDEs or text editors, respectively, and the effect sizes are large. However, there is a negligible effect size and non-statistical significant difference when comparing the time spent on program comprehension by using IDEs and text editors in C# projects.
2) Developers in Java projects spend less time on program comprehension inside text editors than developers in Java projects using IDEs or web browsers, respectively, and the effect sizes are large. However, there is a negligible effect size and non-statistically significant difference when comparing the time spent 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 developers in C# projects using IDEs. However, there is no statistically significant difference when comparing the time on program comprehension using web browsers or text editors in C# and Java projects.

To visualize the results, we plot the interaction effect between programming languages of projects and used applications for program comprehension as shown in Table 13. ↑ denotes the staticstically significant with a large positive effect size, while ↓ denotes statistically significant with a large negative effect size. From Table 15, we also find that the main difference between the time spent on program comprehension in Java and C# projects is due to difference in the time spent on program comprehension inside IDEs, and we find that on average developers in Java projects

TABLE 13: Interaction effect of programming languages of a project and used applications for program comprehension. ↑ denotes the staticstically significant with a large positive effect size, while ↓ denotes statistically significant with a large negative effect size.

(a) Interaction effect of programming languages (i.e., Java and C#)

Interaction Effect	Java	C#
IDE		
Web Browser		↑
Text Editor	↓	

(b) Interaction effect of applications (i.e., IDE, web browser, and text editor).

Interaction Effect	Java	C#
IDE	↑	
Web Browser		
Text Editor		

TABLE 14: 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

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%).

Furthermore, we analyze if the difference in program comprehension time is correlated to the age of a project. We first count the number of the months that passed from the start date of the seven projects to the month when we performed our study (i.e., June 2016); these are shown in Table 16. We then use the Spearman correlation coefficient to measure the strength of correlation between the two variables C- in our case, number of months that passed for each of the seven projects, and program comprehension time. The Spearman correlation 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 17 presents Spearman's rho and P-value for the number of months and program comprehension time. The correlations between the number of months, and the overall program comprehension time, the time spent on program comprehension inside IDEs, web browser, and text editors are all small. Thus, the age of a project has a limited effect on the spent time spent on program comprehension. Note that these correlations may not be statistically significant, due to the small size of the investigated data.

4.3.2 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 make extensive use of 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

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

TABLE 18: 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

whether the number of third party libraries affects the time spent on program comprehension, we count the number of third party libraries that are used in these seven projects by analyzing their build files (e.g., build.xml in Ant, pom.xml in Maven, or MSBuild in C#). Table 16 presents the number of third party libraries to the percentage of time spent on program comprehension. We observe that Java projects use a larger number of third party libraries than C# projects. We use the Spearman correlation coefficient [62] to measure the correlation between the two variables – in our case, number of used libraries in the seven projects, and the program comprehension time. Table 18 presents Spearman's rho and p-value for the number of libraries and program comprehension time. Correlations between the number of libraries, and the overall program comprehension time and the time spent on program comprehension inside IDEs are high and statistically significant. Thus, an increase in the number of libraries is associated with an increase in the amount of time spent on program comprehension, especially the time spent on program comprehension inside IDEs. Note that although we get medium to large correlations between the number of libraries and time spent on program comprehension inside web browser and text editors, the correlations may are not statistically be significant, due to the small size of the investigated data.

We also investigate the interaction effects of the programming language of a project and the number of used libraries in these projects to the percentage of time spent on program comprehension. We have a continuous independent variable (i.e., the number of libraries) and a categorical independent variable (i.e., the programming languages of a project). Hence , while a two-way ANOVA only works when the independent variables are of categorical type, we use a two-way ANCOVA test [9] since a two-way ANOVA only works when the independent variables are categorical. to check whether the interaction effect of programming language and the number of used libraries has a statistically significant impact on the time spent on program comprehension. Table 19 presents the two-way ANCOVA test for the interaction effects of the programming language of a project and the number of used libraries for program comprehension. We find that the programming languages of a project, and the number of used libraries have statistically

TABLE 15: Cohen's d and p-values for the interactions of the programming languages of a project and used applications 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: 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%

TABLE 19: Two-way ANCOVA test for the interaction effects of the programming language of a project and the number of used libraries for on the time spent on 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

significant impact on the percentage of time that is spent on program comprehension. However, the interaction of these two factors does not have a statistically significant impact on the percentage of time that is spent on program comprehension. Thus, the programming languages of a project, and the number of used libraries impact the percentage of time spent on program comprehension independently.

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

4.3.3 Implications

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

Library Usage. One advantage of Java is that there are many third-party libraries, and SE introductory book (e.g., [19], [44]) often encourage developers to reuse existing code instead of writing new code, in order to reduce development time. From our study, we find that using more third-party libraries increases the time spent on program comprehension. 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 that 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.

Better Design of IDE. In RQ1, we observed that navigating inheritance hierarchies leads to more time spent on program comprehension. And in RQ3, we observed that the main difference of spent time on program comprehension in Java versus C# projects is due to difference of time spent on program comprehension inside IDEs. In our interviews, 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, and invocations) between code elements that are spread across multiple source code files. Prior research proposed several tools to improve IDEs according to developers' typical behavior [10], [27]. 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 [27]. Bragdon et al. proposed Code Bubbles to help developers define and use working sets, where a working set refers to the group of functions, documentation, notes, and other information that a programmer needs for accomplishing a particular programming task (e.g., feature implementation or bug fixing) [10]. Moreover, from our interviews we observe that the Eclipse community might also draw lessons from some interesting design ideas and functionalities from Visual Studio. Future studies are needed to better under the key differences between Eclipse and Visual Studio, and the impact of these differences on program comprehension activities.

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

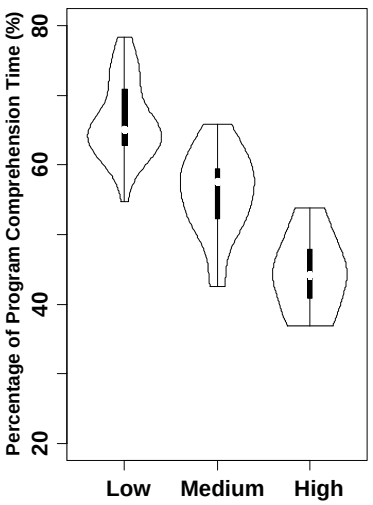

Fig. 7: A violin plot of the percentages of program comprehension time for developers with different professional experience.

4.4 (RQ4) Do senior developers spend less time on program comprehension?

4.4.1 Results

In RQ4, we examine whether there exists major differences for participants with different professional experience, thus we create buckets of participants according to their professional experience. Specifically, we divide the developers into 3 groups according to their number of years of professional experience. We define those with low, high, and medium experience as the 25% with the least experience in years, 25% with the most experience in years, and the rest, respectively. This grouping of participants follows prior work, e.g., Carver et al. [11] and Lo et al. [33]. By using this approach, participants who have less than 3 years of professional experience, 3 to 5 years of professional experience, and more than 5 years of professional experience are categorized into the low, medium, and high experience groups.

One-way ANOVA Analysis. Figure 7 presents the percentages of program comprehension time for developers with different professional experience. 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 a one-way analysis of variance (ANOVA) to determine whether there are any statistically significant differences between the means of the three groups. Table 20 presents the results for a one-way ANOVA test for the percentage of time that developers with different professional experience spend on program comprehension. Since the F value of the one-way ANOVA is 79.4, and the p-value is less than 0.001, we conclude that there is a statistical significant difference for the time that developers with different professional experience spend on program comprehension. Table 21 presents Cohen's d and p-values for comparison of percentage of time that developers with low, medium, and high professional experience spend on program comprehension activities. We have the following observations:

1) Developers of low professional experience spend more time on program comprehension activities compared to developers 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 compared to developers with high professional experience, and the effect size is large.

TABLE 20: One-way ANOVA test for the percentage of time that developers with different professional experience spent on program comprehension.

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

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

TABLE 21: Cohen's d and p-values for comparison of percentage of time that developers with low, medium, and high professional experience spent 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

Two-way ANOVA Analysis. Here, we want to investigate the interaction effects of professional experience and the used applications (i.e., IDE, Web browser, and text editor) for program comprehension, and we apply a two-way ANOVA test to check whether the interaction effect of professional experience and the used applications has a statistically significant impact on the time spent on program comprehension. Table 22 presents the results of a two-way ANOVA test for the interaction effects of professional experience and the used applications for program comprehension. We find that professional experience, used applications for program comprehension, and the interactions of these two factors all have statistically significant impact on the percentage of time that is spent on program comprehension. Here, the interaction of these two factors has a statistically significant impact on the percentage of time that is spent on program comprehension, meaning that the simultaneous influence of the two factors (in our case, professional experience and the used applications) on the dependent variable (in our case, percentage of time that is spent on program comprehension) is not additive.

Next, we also apply a pairwise t-test with a Bonferroni correction and we measure Cohen's d to test whether the difference between these two factors (i.e., professional experience, and used applications) are statistically significant and whether the effect sizes are substantial. Table 23 presents the Cohen's d and p-values for the interactions of professional experience and used applications for program comprehension, we have the following observations:

1) Developers of low and medium experience spend less time on program comprehension inside text editors than 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

TABLE 22: Two-way ANOVA test for the interaction effects of professional experience and the used applications 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

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 developers with high experience, and the effect size is large. However, there is no statistically significant difference between developers of low experience and medium experience on the time spent on program comprehension inside IDEs.
4) There is no statistically significant difference among developers of low, medium, and high experience on the time spent on program comprehension inside web browsers or text editors, although the effect sizes are small or medium.

4.4.2 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, 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 helps 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*".

4.4.3 Implications

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

Program Comprehension Behavior Learning. We manually checked and compared the behavior 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 copy some code from the web browser to the IDE, then compare 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 best practices to developers to help them reduce program comprehension time. The best practices can possibly be learnt automatically by mining the activities of senior developers.

As an example, in a project team, we can analyze how senior developers navigate source code to acquire a good program understanding to perform various maintenance tasks (e.g., implementing newly requested features, fixing newly reported bugs, etc.). Based on senior developers' navigation patterns, we can build new behavior-driven change impact analysis and bug localization techniques. Given a particular source code file to change, we can recommend what other source code files and documentation to inspect to get the needed information to perform the change. Given a new feature request, we can highlight the files that developers need to inspect to get a better understanding of relevant parts of the code base. Given a new bug report, we can highlight the files that developers need to inspect to get a better understanding as to what went wrong and how to fix it. As another example, we can recommend sites that senior developers frequently visited to get information that is needed during program comprehension phase, or to learn new technology and tips to improve their skills.

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

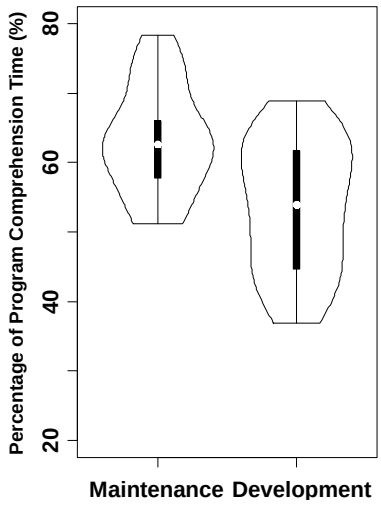

Fig. 8: A violin plot of the percentages of program comprehension time for projects in different phases.

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

4.5.1 Results

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

One-way ANOVA Analysis. Figure 8 presents the percentage of time that is spent on program comprehension activities for projects in the development and maintenance phases. We notice that on average, developers of projects in the new

TABLE 23: Cohen's d and p-values for the interactions of professional experience and used applications 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

TABLE 24: One-way ANOVA test for the percentage of time that developers in projects of maintenance and new development phases spent 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

TABLE 25: Two-way ANOVA test for the interaction effects of the project phases and the used applications 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

development phase and those in the maintenance phase spend 53.54% and 63.22% of their time on program comprehension activities. Table 24 presents the results for a one-way ANOVA test for the percentage of time that developers of projects in maintenance and new development phases spend on program comprehension. Since the F value of the one-way ANOVA is 23.7, and the p-value is less than 0.001, we conclude that there is statistically significant difference for the time that developers spent on program comprehension in projects in the maintenance versus new development phase.

Next, we also apply a pairwise t-test with a Bonferroni correction and we measure 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 developers working on new development projects.

Two-way ANOVA Analysis. Here, we would like to investigate the interaction effect of the project phase and the used applications for program comprehension, and we apply a two-way ANOVA to test the statistical significant. Table 25 presents the results of a two-way ANOVA test for the interaction effect of project phase and the used applications for program comprehension. We find that the project phase, used applications for program comprehension, and the interaction of these two factors all has a statistically significant impact on the percentage of time that is spent on program comprehension.

Next, we also apply a pairwise t-test with a Bonferroni correction and measure Cohen's d to test whether the difference between these two factors (i.e., project phases and used applications) are statistically significant and the effect sizes are substantial. Table 26 presents Cohen's d and p-values for the interaction of project phases and used applications for program comprehension, we have the following observations:

1) Developers in development projects spend more time on program comprehension inside the web browsers than developers in new development projects using IDEs or text editors, respectively, and the effect size is large. However, there is a small effect size and a non-statistically significant difference 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 those in maintenance projects using IDEs or web browsers, respectively, and the effect size is large. However, there is a small effect size and a 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 developers in new 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 developers in new development projects using text editors (5.71% vs. 13.72%). However, there is a small effect size and a non-statistically significant difference when comparing the time on program comprehension using web browsers in maintenance versus new development projects.

4.5.2 Interview Findings

There are several reasons for the difference in the percentages of program comprehension time for developers of projects in new development phase and maintenance phase. First, in the new development phase, the project team is relatively stable, but in the maintenance phase, some developers will leave and some new developers will join the

TABLE 26: Cohen's d and p-values for the interaction of project phase and used applications 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)*** **	–

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 new development phase, developers are more likely to focus on understanding requirements; while in the maintenance phase, developers are more likely to focus on understanding the source code. P3 stated: "*In the new 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 new development phase are much less than the LOCs in the maintenance phase. Thus, the workload to understand the source code in the new development phase is much less than that in the maintenance phase. P9 stated: "*the search space for projects in the new 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 on searching for relevant source code, and hence lead to more time on program comprehension activities.*".

4.5.3 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, namely the large size of the source code and relevant documentation. So, an 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 new development and maintenance phases, such a tool will make developers understand source code more effectively.

Developer Turnover Management. We also found that the high developer turnover rate in the maintenance phase is associated with developer spending more time on program comprehension. Many researchers have studied developer turnover. For example, Mockus finds that developers leaving a project had a negative impact on quality but that newcomers had no effect on it due to the loss of knowledge and experience [37]. On the contrary, Foucault et al. find that newcomers have an impact on quality while project leavers do not have such an effect [17]. Understanding and preventing developer turnover can help a 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 program comprehension.

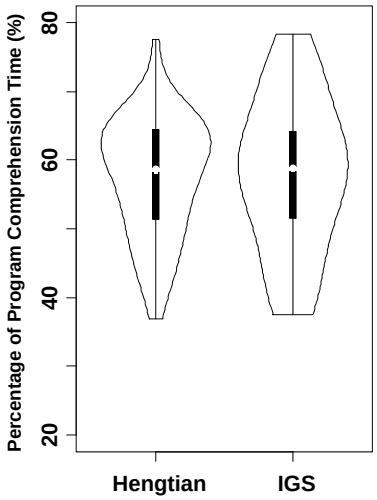

Fig. 9: A violin plot of the percentages of program comprehension time for developers in Hengtian and IGS.

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 new development phase.

5 Discussion

5.1 Cross-Company Analysis

Our study collects data from two companies Hengtian and IGS. Projects A and G are from IGS, and projects B to F are from Hengtian. Here, we would like to investigate whether developers across both companies spend similar time on program comprehension. The answer to this question will affect the generalizability of our study, e.g., if we find that developers in different companies spend different amount of time on program comprehension, then future studies should consider this aspect in the design of their experiments.

One-way ANOVA Analysis. Figure 9 presents the percentage of time spent on program comprehension activities for developers in Hengtian and IGS. 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 a one-way ANOVA test for the percentage of time that developers in maintenance and development phases spend on program comprehension. Since the p–value is larger than 0.05, we conclude that there is no statistically significant difference in the time, that developers across both companies, spent on program comprehension.

Two-way ANOVA Analysis. We now investigate the interaction effect of the company and the applications that are used

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 effect of company and the used applications 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

for program comprehension. Hence we apply a two-way ANOVA to test the statistical significance. Table 28 presents the results of a two-way ANOVA test for the interaction effect of company and the applications that are used for program comprehension. We find that the interaction of these two factors has a non-statistically significant impact on the percentage of time spent on program comprehension.

From the above analysis, we conclude that developers in these two companies spend similar time on program comprehension, thus our results are more likely to generalize to other companies. Nevertheless, future studies are needed to further examine the generality of our findings.

5.2 Feedback from Participants

After completing our paper, we sent the results section along with the abstract and introduction to the ten interviewees, and asked them for feedback about our findings, i.e., we asked them whether they agree, disagree or have no comments (neutral) on the results of each RQ. Figure 10 presents the survey results for the ten interviewees. In general, most of the respondents agreed with the results of the five RQs. For RQ1 and RQ2, all of the ten respondents agreed with our findings. For RQ3, two of the respondents (P9 and P10) choose the option "neutral", since they only worked in C# projects, and they never joined any Java project. For RQ4, two of the respondents disagreed with our findings; one commented that "*the results in RQ4 are surprising. In my experience, senior developers need to spend more time on program comprehension, since they have to do more advanced things (e.g., code reviews, and architecture design), which require them to understand source code and the implementation of systems more*" (P1). For RQ5, we have two respondents who disagreed with our findings; one commented that "*I think developers spend similar time on program comprehension activities in different projects phases, since in any phase developers need to read code, search online, and reuse third-party libraries. Maybe I am wrong, but I have to admit that it is an interesting*" (P6). Some of the received comments which support our results are as follows:

- 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 of program comprehension.

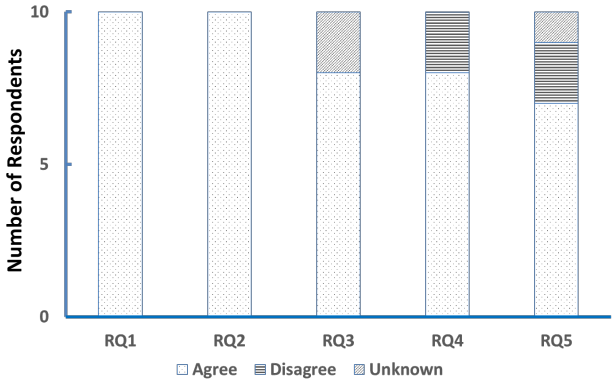

Fig. 10: Survey results.

- 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 increases the time spent on program comprehension.
- Java and C# are the 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 spend 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#.
- As an outsourcing company, there are many projects in the maintenance phases. Sometimes, the boss thinks maintaining a project is much easier than developing a project from scatch, 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.

5.3 Limitations

Threats to Construct Validity. One of the threats to construct validity relates to the ability of our *ActivitySpace* tool 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, and ignoring accesses to websites which are irrelevant to software development. We also perform a preliminary study on two developers to verify the correctness of our data collection tool on inferring programming comprehension activities. Since the two developers need to analyze many sprees, it is possible that they make some mistakes. Still, we believe that the developers are correct for most of the sprees.

Another threat relates to wrong conclusions that we draw about participant's perceptions from their comments. To minimize this threat, we recorded our interviews and listened to them several times. Also, the first two authors worked together to ensure that the results are accurate. After

completing our paper, we also sent it to the interviewees, and asked them for feedbacks on our 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 78 developers for a total of 3,148 working hours and interview 10 of them. All these developers come from 2 companies. Due to the limited availability of the participants, we only interviewed ten participants, which might impact the generalizability of our results. Although these numbers may limit the generalizability of our study, the number of developers that we interviewed is on par with other interview-based studies [20], [39], and the number of developers that we monitored are more than other studies that also monitor developers, e.g., [26], [36]. Furthermore, many of our participants have worked in many other companies before, and have experience with developing projects of various programming languages and sizes.

We only study two companies and both are from China. Time spent on program comprehension may be different if we investigate projects from other companies, especially those outside China. Additionally, developers participating in our study may use different search engines, e.g., Bing or Google (via VPN). The usage of different search engines may affect the time spent on program comprehension. Moreover, the company strategy on open source projects usage might affect the time spent on program comprehension, since developers are likely to do more web searching for projects that make heavy use of open source components. Future studies are needed to further examine these points.

Moreover, in this paper, we only consider two programming languages, i.e., Java and C#. The time spent on programming languages might be vary when we consider more programming languages. 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, and investigating projects written in a more diverse set of programming languages.

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 78 developers working on 7 real industrial projects spanning a period totaling of 3,148 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. Through relatively extensive empirical data, our work revisits long-held assumptions about program comprehension, including that senior developers spend less time on program comprehension, that 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. We encourage future work to use our findings to construct in-depth surveys that can be distributed to a much wider audience so we can get a much wider understanding.

A replication package for this paper can be downloaded from: https://goo.gl/DHyfJa

TABLE 29: The average percentage of time that developers spend on comprehension (Compre.), navigation, editing, and others for 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%

Appendix

1. Different Settings of Reaction Time (RT)

In this study, by default, we set the reaction time (RT) value to 1 second when computing program 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 hand [59]. Hence, we also try different RT values (i.e., in $\{0.5,0.8,1,1.2,1.5\}$) to investigate the effect of choosing different RT values on our findings.

Table 29 shows the average percentage of time that developers spend on comprehension, navigation, editing, and others when using 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 becomes larger as the RT value increases. This result makes sense because all intervals that are larger than RT among developers' interactions are computed as comprehension in our study.

However, these variations in the results when using different RT values do not affect our findings. In all results, comprehension activities take more than half of developers' working time, which is consistent with prior studies [13], [15], [26], [36], [63]. Furthermore, the results of all individual developers in our study is consistent with the average results in Table 29. So, the different RT values do not affect our findings about the effect of programming language, developer experience, and project phase on program comprehension. Moreover, as we show in Section 3.2.6, that when we set RT to be 1 second, our approach shows similar results as manual annotations. Thus, in this paper, we set RT to 1 second.

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. C. Carver, O. Dieste, N. A. Kraft, D. Lo, and T. Zimmermann. How practitioners perceive the relevance of esem research. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 56. ACM, 2016.

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

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

[14] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 193–202. IEEE, 2012.

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

[16] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[17] 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.

[18] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

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

[20] 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.

[21] 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.

[22] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304, 2012.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] M. Lanza and S. Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[29] 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.

[30] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Whats in a name? a study of identifiers. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 3–12. IEEE, 2006.

[31] 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.

[32] 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.

[33] D. Lo, N. Nagappan, and T. Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 415–425. ACM, 2015.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, 2016.

[41] 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.

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

[43] 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.

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

[45] 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.

[46] A. J. Riel. *Object-oriented design heuristics*, volume 335. Addison-Wesley Reading, 1996.

[47] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus. Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1006–1016. ACM, 2016.

[48] 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.

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

[50] 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.

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

[52] 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.

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

[54] 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.

[55] 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.

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

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

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

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

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

[61] 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.

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

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