Quantifying Program Comprehension with Interaction Data

Roberto Minelli[1], Andrea Mocci[1], Michele Lanza[1] and Takashi Kobayashi[2]
1: REVEAL @ Faculty of Informatics – University of Lugano, Switzerland
2: Department of Computer Science – Tokyo Institute of Technology, Japan

***Abstract*—It is common knowledge that program comprehension takes up a substantial part of software development. This "urban legend" is based on work that dates back decades, which throws up the question whether the advances in software development tools, techniques, and methodologies that have emerged since then may invalidate or confirm the claim.**

We present an empirical investigation which goal is to confirm or reject the claim, based on interaction data which captures the user interface activities of developers. We use interaction data to empirically quantify the distribution of different developer activities during software development: In particular, we focus on estimating the role of program comprehension. In addition, we investigate if and how different developers and session types influence the duration of such activities. We analyze interaction data from two different contexts: One comes from the ECLIPSE IDE on Java source code development, while the other comes from the PHARO IDE on Smalltalk source code development. We found evidence that code navigation and editing occupies only a small fraction of the time of developers, while the vast majority of the time is spent on reading & understanding source code. In essence, the importance of program comprehension was significantly underestimated by previous research.

I. INTRODUCTION

Developers use IDEs (Integrated Development Environments) to read, understand, and write source code [7], [8]. Integrated development environments provide a number of facilities to support software development, such as source code browsers, refactoring engines, test runners *etc.* (*e.g.,* [10], [19]). While using an IDE, developers generate a large number of events, for example, browsing the source code of a method, editing the body of a method, or inspecting an object at runtime. We call the set of such events *interaction data*.

Among software engineering activities, program understanding has been estimated to be one of the most challenging tasks performed by developers [12]. According to Corbi, developers understand programs by(1) reading documentation, (2) reading source code, and (3) running the program itself [3]. According to some studies, understanding absorbs about half of the time of developers [3], [6], [24]. In fact researchers showed that developers spend more time reading than writing source code [22]. Navigation between code fragments is another essential activity for program comprehension [9], [14]. Ko *et al.* estimated that developers spend 35% of their time navigating the system at hand [10].

These facts have been taken for granted for quite some time, and some research fields, such as program comprehension and reverse engineering, base their reason to exist on such facts. With this work we want to investigate whether these facts can actually be confirmed.

We use interaction data to quantify the amount of time developers spend to navigate, write, and understand source code. Interaction is recorded silently while developers use an IDE to perform their activities.

To get a feeling for the type of data we are considering, refer to Fig. 1, which shows a development session at a glance.

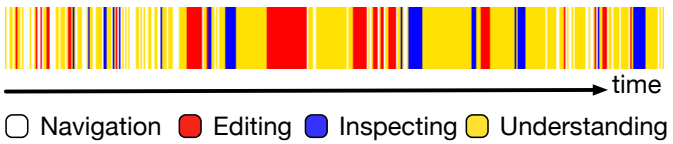

Fig. 1: Development Activities at a Glance.

The session lasted 30 minutes and 43 seconds. During that time a developer triggered a total of 455 interaction events:

- 403 *navigation* events (white). These are very quick "trigger" events, such as opening a browser, clicking on class or method names to see their contents, *etc.*
- *inspection* events (blue). These happen when the developer opened an inspector to examine the contents of an object.
- *editing* events (red). These represent the actual "writing" of the source code by editing new or existing code.
- The rest, depicted in yellow, is the time when the developer was seemingly "doing nothing", but in fact this is the time when the developer sits in front of the source code and looks at it. It thus represents the actual program understanding part.

We see that navigation events are mostly present at the beginning of the session, when the developer is getting his bearings in the system. Moreover, editing events are nearly always present after a longer understanding time interval. To collect such data we use two different means:

1) DFLOW, an extension to the PHARO Smalltalk IDE[1] to record all interactions. With DFLOW we collected 175 development sessions coming from 7 developers (both industrials and academics) performing their ordinary work. Development sessions have different types assigned by developers, *e.g.,* bug-fixing. DFLOW recorded more than 110,000 interaction events of different types (*e.g.,* navigation, editing)
2) PLOG, an ECLIPSE plugin. With PLOG we collected 15 sessions from 15 developers (*i.e.,* master students), totaling almost 4,000 interaction events.

[1]See http://www.pharo-project.org

We use both data sources in this paper to make the following contributions:

- A quantification of the development activities based on interaction data to estimate how much time developers spend to navigate, write, and understand source code;
- A comparison between our findings and the estimates available in the literature;
- A brief presentation of the tools with which we collect interaction data.

Structure of the Paper. In Section II we survey the related work. In Section III we describe the interaction data we have collected and briefly present the tools we developed in this context. In Section IV we describe the obtained datasets. In Section V we analyze and discuss our findings. In Section VI we draw our conclusions.

II. Related Work

In the past decades researchers have estimated that program understanding occupies a large part of the work time of developers. For example, Zelkowitz *et al.* estimated that program comprehension takes more than half the time spent on maintenance [24]. In turn, maintenance accounts for 55 to 95% of the total costs of a software system [4], thus the weight of program comprehension globally ranges between 30 to 50%. This estimation is corroborated by Fjeldstad and Hamlen, who claim that comprehension occupies half the time of developers [6]. Reverse engineering is the part of software maintenance that helps you understand the system prior to changing it [1]. While recovering the design of a software system, developers should understand what, how, and why a program does something. The cost of understanding software—that includes the time required to comprehend it and time lost in misunderstanding—is rarely seen as a direct cost, but is significant [1]. Also according to Erlikh, comprehending software is time-consuming and costly [4]. He estimated that about 90% of the budget goes to maintenance, leading to a substantial expense for understanding. More recently, Ko *et al.* claimed that understanding is also achieved through navigating source code fragments [10]. They estimated that developers spend about 35% of their time navigating the system at hand.

Researchers proposed a number of approaches and tools to track the way developers work inside the IDE. They also leveraged this data for different purposes. Singer *et al.* implemented NAVTRACKS, a tool that records and leverages navigation histories of developers to better support browsing through software [20]. Kersten *et al.* developed MYLAR, a tool that monitors the programmer to identify the most important program entities [9]. Murphy *et al.* later used this data to answer the question *"How Are Java Software Developers Using the* ECLIPSE *IDE?"* [13]. They showed that developers use most of the ECLIPSE perspectives while developing and that they often use keyboard shortcuts to perform activities. Yoon and Myers developed FLUORITE, a tool that logs low-level events in the ECLIPSE IDE [23]. They claimed that FLUORITE can be used to evaluate existing tools through the analysis of coding behavior. Robbes and Lanza proposed SPYWARE, a tool that records semantic changes in real time [16]. They later devised ad-hoc metrics and used their data to understand and characterize development sessions [15].

Researchers also tried to automatically identify tasks and activities in development sessions. Coman and Sillitti collected low-level events and presented a technique to split sessions into task-related sub-sessions [2]. Interaction data was also used for change prediction. Kobayashi *et al.* developed PLOG, a tool to capture interaction histories inside the ECLIPSE IDE. They used the recorded data to devise a prediction model for change propagation based on interaction histories [11]. Similar work was performed by Robbes *et al.* [17].

What struck us while reviewing the related work is the dichotomy between claims regarding the importance of program comprehension which are often not backed up by empirical evidence, and a large of body of work that uses fine-grained interaction to perform other types of research. With the present paper we try to close this gap by using fine-grained interaction data to validate claims pertaining to the role of program comprehension in the context of software development.

III. Interaction Data and Tools

We first describe interaction data and its properties. We then briefly present the tools we developed to record that data. Since the data we collect comes from two different development contexts, *i.e.,* Smalltalk and Java, we also discuss how the interaction data differs depending on the language and the tool to collect it.

We classify interaction data according to the following simple taxonomy:

- *Navigation events*, used to browse (but not modify) code entities, like opening a browser to list the methods of a class or a file to depict its contents;
- *Inspection events* (Smalltalk-only), that happen when developers inspect the state of run-time objects;
- *Editing events*, that modify source code, like adding a new class or modifying the code of a method.

Each profiled event has the following properties:

- A **creation time**, the timestamp of the event;
- A set of program **entities involved** in the event, such as classes and methods.

A sequence of interaction events make up a development session, which consists of the following elements:

- A **title**, describing of the intention of the developer;
- An **author name**, that is, the name of the developer;
- A **type** (Smalltalk-only), describing its intended purpose:
  - **General purpose** (*default*): The developer performs various activities.
  - **Refactoring**: The developer mainly performs refactoring activities.
  - **Enhancement**: The developer mainly performs perfective maintenance activities [21].
  - **Bug-fixing**: The developer mainly performs bug-fixing activities.
- A **start time** and **end time**;
- A number of **sub-sessions**, with the following data:
  - A **start time** and an **end time**;
  - A collection of **interaction events**;
  - **Windows information** (Smalltalk-only).

Title, author name, and type are submitted by the developer before the session starts. A session might last for hours or days, but the developer will probably not program uninterruptedly. Sub-sessions indicate pauses during development. In DFLOW, when a developer stops programming for any reason (*e.g.,* a conference call) she can explicitly pause (and later resume) the recording. In addition, when we post-process the interaction histories of DFLOW and PLOG, we automatically detect (and remove) **idle times** longer than 10 minutes and create implicit sub-sessions without idle periods.

As we use two different tools that track two different IDEs which support a different development philosophy, the information above cannot be mapped to the interaction data collected by the tools. To make an example: The PHARO Smalltalk IDE is a multi-window environment, and it is normal for users to spawn several windows during development. Moreover, it is an IDE based on program entities, and not files, which entails that a developer is looking always at methods in isolation. As opposed to that, ECLIPSE is an IDE based on tabs and files, which requires that developers open files which are presented in the editor and it is therefore normal that a developer has several methods in front of his eyes in the same window. This in essence means that in the ECLIPSE case it is not possible for us to unambiguously understand which entity is being looked at. However, we are not interested into what exactly the developer is doing, but more when she is doing it.

A. DFlow and Smalltalk Interaction Histories

DFLOW is the tool we developed to record interaction histories in Smalltalk. DFLOW collects, in a non-intrusive way, 33 different events. Table I lists these events with an identifier and a short description.

The initial character of each identifier represents the event type, *i.e.,* **N**avigation, **I**nspection, and **E**diting. DFLOW also records **window information** to observe how a developer interacts with the windows of the target IDE. DFLOW profiles how and when the user opens, resizes, moves, activates, and closes a window. Developers use a simple UI of DFLOW to start, pause, stop, and resume the recording of sessions.

Table II and Table III summarize the data collected with DFLOW, grouped per type and per developer respectively. We have 175 sessions by 7 developers, totaling more than 110,000 events. The developers are all from the PHARO open-source community, with a background in both industry and academia, located in 4 different sites (INRIA Lille, France; University of Bern, Switzerland; University of Santiago, Chile; University of Lugano, Switzerland). The vast majority of the sessions are Enhancement and General, where general sessions are usually the longest, with an average length of more than 1.5 hours. We also note that bug-fixing sessions are generally short, which can be explained by the fact that often they were dedicated and guided sessions to fix particular known bugs in existing code. Across all session types, navigation events are one order of magnitude more than the number of editing events. General purpose sessions are, on average, those that contain more sub-sessions and have the higher number of navigation and edit events (778.75 and 76.29 on average). Bug-fixing sessions have the highest number of inspect (44.63 on average). This is not surprising since inspects are often triggered while debugging.

TABLE I: List of Interaction Events.

ID	Description
N_1	Opening a Finder UI
N_2	Selecting a package in the system browser
N_3	Selecting a method in the system browser
N_4	Selecting a class in the system browser
N_5	Opening a system browser on a method
N_6	Opening a system browser on a class
N_7	Selecting a method in the Finder UI
N_8	Starting a search in the Finder UI
I_1	Inspecting an object
I_2	Browsing a compiled method
I_3	Do-it on a piece of code (*e.g.,* workspace)
I_4	Print-it on a piece of code (*e.g.,* workspace)
I_5	Stepping into in a debugger
I_6	Run to selection in a debugger
I_7	Exiting from an active debugger
I_8	Proceeding in a debugger
I_9	Browsing full stack in a debugger
I_{10}	Stepping over in a debugger
I_{11}	Entering a full debugger
I_{12}	Browsing the hierarchy of a class
I_{13}	Browsing all implementors of a method
I_{14}	Browsing all senders of a method
I_{15}	Closing the current Smalltalk image
I_{16}	Saving the current Smalltalk image as...
I_{17}	Browsing the version control system
I_{18}	Browsing the stack trace in the debugger
I_{19}	Browse versions of a method
E_1	Creating a new class
E_2	Adding/removing instance variables from a class
E_3	Removing a method from a class
E_4	Adding a method in a class
E_5	Remove a class from the system
E_6	Automatically creating accessors for a class

It appears that, while fixing bugs, developers navigate less than in other sessions (264.96 navigations on average). Our hypothesis on this is that these sessions are highly focused since the developer already knows the subset of program entities involved in a given bug-fixing task. One third of times developers do not know what they are going to do, *i.e.,* they select "general purpose" as session type. Those sessions have a very high concentration of navigation and edit events (respectively 778.75 and 76.29) and a higher number of windows with respect to other types (115.85 where the average over all the sessions is 89.52). This justifies their general, broad purpose: Developers do a little bit of everything.

Table III gives insights on how different developers behave. All developers, but SD4, have a number of navigation events that are one order of magnitude more than the number of editing. It remains to be investigated the behavior of SD4 that on average performs one edit event every two navigations. Developer SD1 has a very high number of windows per session (262.00). She is developing a visualization engine for Smalltalk. PHARO is a window-based environment and her tool generates visualization inside windows. This explains why her use of windows is significantly higher than others. This developer, in general, is an outlier: She has significantly higher number of navigations and edits with respect to other developer and further investigation on her behavior is required. Developers SD3 and SD5 are the two subjects that used

TABLE II: Smalltalk Sessions Data per Type.

Session Type	Sessions			Events						Windows	
				Navigation		Inspect		Edit			
	#	Avg. Subsessions	Avg. Duration (hh:mm:ss)	#	Avg.	#	Avg.	#	Avg.	#	Avg.
Bug Fixing	27	2.11	44:38	7,154	264.96	1,205	44.63	1,537	56.93	1,739	64.41
Enhancement	86	2.26	57:08	40,973	476.43	3,631	42.22	3,992	46.42	7,331	85.24
General	55	4.18	1:33:38	42,831	778.75	1,767	32.13	4,196	76.29	6,372	115.85
Refactoring	7	1.71	48:46	3,294	470.57	22	3.14	331	47.29	224	32.00
All	**175**	**2.38**	**1:06:21**	**94,252**	**538.58**	**6,625**	**37.86**	**10,056**	**57.46**	**15,666**	**89.52**

TABLE III: Smalltalk Sessions Data per Developer.

Developer	Sessions			Events						Windows	
				Navigation		Inspect		Edit			
	#	Avg. Subsessions	Avg. Duration (hh:mm:ss)	#	Avg.	#	Avg.	#	Avg.	#	Avg.
SD1	12	6.08	03:01:24	21,617	1,801.42	183	15.25	2,458	204.83	3,144	262.00
SD2	3	1.00	16:27	393	131.00	157	52.33	24	8.00	71	23.67
SD3	65	1.49	52:32	20,468	314.89	2,157	33.18	2,091	32.17	3,183	48.97
SD4	6	1.83	48:13	2,183	363.83	353	58.83	1,196	199.33	608	101.33
SD5	70	2.84	56:26	35,495	507.07	2,952	42.17	3,289	46.99	7,336	104.80
SD6	7	4.29	01:25:18	6,862	980.29	337	48.14	472	67.43	555	79.29
SD7	12	6.67	01:34:25	7,234	602.83	486	40.50	526	43.83	769	64.08
All	**175**	**2.82**	**01:06:21**	**94,252**	**538.58**	**6,625**	**37.86**	**10,056**	**57.46**	**15,666**	**89.52**

DFLOW the most. The former is pretty new to the Smalltalk programming language. She navigates and edits less than the average of other developers. SD5, an experienced Smalltalk developer, instead is mostly in line with average values. It remains to be investigated how the expertise of developers impact on their behavior inside the IDE. Developers SD1 and SD6 are the two subjects that navigate the most (1,801.42 and 980.29 events on average). Interestingly, while SD1 uses a very high number of windows, SD6 despite navigating a lot more than other developers, uses few windows. Developers SD1 and SD7 have the highest numbers of sub-sessions (6.08 and 6.67, where the average is 2.82), symptoms of highly interrupted development sessions.

B. Plog and Java Interaction Histories

In addition to Smalltalk interaction histories collected with DFLOW, we also analyzed 15 Java development sessions. These sessions were captured using the PLOG tool developed by Kobayashi *et al.* [11].

PLOG captures interactions at two granularities: file and method level. For this study we only used histories at file level which are comparable to the interaction histories recorded with DFLOW (a class in Smalltalk is conceptually similar to a file in Java, which often contain one class). Interaction histories captured by PLOG have the following meta-information: (1) an **author name**; and (2) a list of **events**, with the following information: a **timestamp** when the event was recorded and a **type**. PLOG records two types of events:

- N_R: a navigation without any editing;
- N_W: a navigation with an editing event.

Navigation events happen when the developer moves between tabs and opens new tabs. N_R events are comparable to navigation events in DFLOW, while N_W events contain implicit editing events. Inspection events are not captured in PLOG, as ECLIPSE does not offer a live programming environment, as opposed to PHARO.

Table IV summarizes the data collected with PLOG.

TABLE IV: Java Sessions Data per Developer.

Developer	Sub-sessions	Duration (hh:mm:ss)	Events	
			Navigation	Edit
JD1	5	1:15:39	48	17
JD2	7	2:02:51	121	13
JD3	11	3:49:00	208	56
JD4	5	1:28:38	139	48
JD5	14	4:42:31	248	82
JD6	8	2:48:00	204	57
JD7	18	12:38:03	803	295
JD8	5	1:28:24	111	11
JD9	6	2:18:31	121	11
JD10	19	3:36:59	231	61
JD11	7	1:48:24	109	20
JD12	11	2:01:47	121	21
JD13	1	1:28:37	70	28
JD14	16	5:16:42	454	27
JD15	11	2:35:52	132	29
ALL	**144**	**49:19:58**	**3,120**	**776**

There are 15 sessions by 15 different developers. Naïvely, PLOG has no concept of sub-session, but we pre-processed the data to automatically identify sub-sessions according to periods of idle (minimum idle set to 10 minutes) in the interaction histories. This generates 144 sub-sessions without idle.

Fig. 2: Visualizing Java Development Activities.

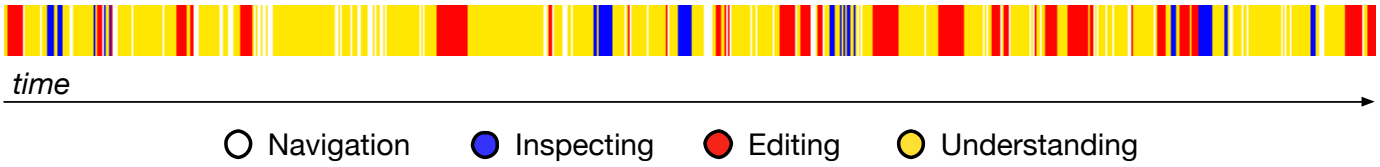

Fig. 3: Visualizing Smalltalk Development Activities.

Developers were given different tasks, namely to extend an existing system to fulfill a series of change requests without any prior knowledge of the system (*i.e.,* enhancement sessions). Each developer has a personal way of approaching the task, as some of them took several hours (up to 12) to implement the requested changes, while others took little bit more than one hour. We can infer that about 20% of all navigation events led to an editing activity.

IV. Quantifying Activities

Our goal is to estimate how much time developers spend to navigate, write, and understand source code using interaction data. Interaction data, as we described in Section III, is the information we leverage to quantify development activities. Fig. 2 and 3 depict two development sessions captured with our recording tools. Fig. 2 shows a Java session recorded with PLOG. That session lasted 1 hours 48 minutes and 24 seconds and counts 109 navigation activities (white), 20 edits (red), and a large amount of understanding (yellow). The developer (JD11 in Table IV) spends 65.1% of her time understanding the system whilst performing perfective maintenance. The first half of the session is essentially composed of understanding driven by the navigation of the system at hand. After that, the developer acquired the necessary knowledge to perform the changes. The second part of the session encloses all the 20 editing activities interleaved with navigation events and understanding time frames. Editing activities have different durations. The first one lasts for a quite long time, probably due to the fact that the developer is not confident with her understanding of the system. Finally, she performs a series of editing activities and increases her confidence on the system.

Fig. 3 shows a Smalltalk development session recorded with DFLOW. The session lasted for 1 hour 14 minutes and 8 seconds and counts 491 navigations, 25 inspections, and 34 editing activities. The first difference with Fig. 2 is the presence of inspection activities (blue). In Smalltalk an inspection happens when a developer observes some property of an instance of an object, *i.e.,* the value of its fields. Inspections are, most of the times, triggered while debugging when, upon a crash, the developer wants to know more about the reasons of the failure. Fig. 3 depicts a bug-fixing session of the developer SD5 (in Table III). The developer spent 60.91% of her time in understanding tasks, 15.28% on editing activities, and 10.93% navigating between code fragments. In the session, there are some peculiarities. For example, inspections have often an edit preceding them. Our hypothesis is that the developer first changes the code, then executes and debugs it. This is possible in the PHARO Smalltalk IDE as it is a live programming environment, *i.e.,* even when the system raises an exception it is still "alive" and can be modified on the fly. Most important editing activities (*i.e.,* the ones with the longest durations) are often preceded by significant understanding time frames. This is a symptom of the fact that developers want to gather a substantial understanding of the system prior to changing it. As in the Java session, editing activities are concentrated in the second part of the session while the first part is mainly comprehension.

A. Estimation Model for DFlow

Fig. 4 depicts a fragment of a raw interaction history recorded with DFLOW, that is, a sequence of events with their timestamp. As a base to estimate the amount of program understanding during development, we need first to estimate the amount of time spent for other development activities, starting from the recorded events.

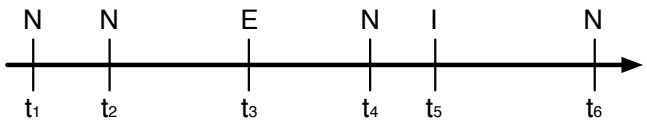

Fig. 4: A Sample DFLOW Interaction History

Considering DFLOW interaction histories, we have three types of recorded events: navigation, inspection, and editing, for which we estimate the duration of the corresponding activities as follows.

Navigation Activities. Navigation events are clicks in the user interface of the IDE. To perform the "click" a user spend a relatively small amount of time. A navigation implies an additional time required to move to a target area. This is known as Fitts's Law and computed as a function of the distance and the size of the target [5]. In this work we approximate this time

to a fixed average duration (ΔN). Fig. 5 shows the updated interaction history after estimating navigation activities.

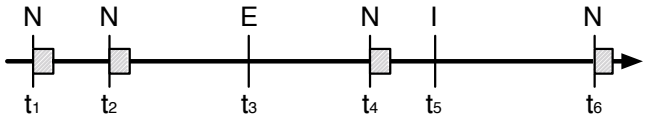

Fig. 5: History with Navigation Activities

Editing Activities. For an editing activity E_i, DFLOW records an event when the user is done with the editing: We denote this time as $end(E_i)$. We assume that the duration of the edit (ΔE_i) is a fraction (P_E) of the time interval between the end time of the previous activity $end(prev(E_i))$ and the end time of E_i.

$$\Delta E_i = P_E \times (end(E_i) - end(prev(E_i))) \quad (1)$$

Fig. 6 shows the updated interaction history assigning a duration to editing events.

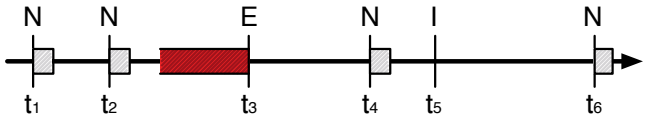

Fig. 6: History with Navigation & Editing

Inspection Activities. For an inspection activity I_i, DFLOW records an event at time $start(I_i)$, that is, when the user starts the activity. We assume that the duration of the inspection (ΔI_i) is a fraction (P_I) of the time interval between $start(I_i)$ and the start time of the following event $start(next(I_i))$.

$$\Delta I_i = P_I \times (start(next(I_i)) - start(I_i)) \quad (2)$$

Fig. 6 shows the updated interaction history assigning a duration to inspection events.

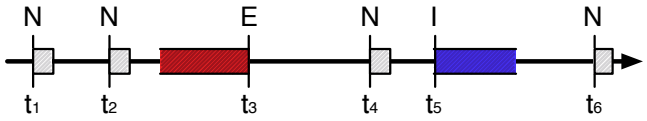

Fig. 7: History with Navigation, Editing, and Inspecting Activities

Understanding Activities. Understanding activities are all the *gaps* between the other three types of development activities. We consider that everything that is not navigation, inspection, and editing is program understanding. For this reason, we first assigned duration to the other types of events, and we identify every remaining gap in the interaction history as understanding activities. Fig. 8 shows the final interaction history, with all the activities.

Interaction between Inspection and Editing. Our estimation model uses time intervals between events to estimate

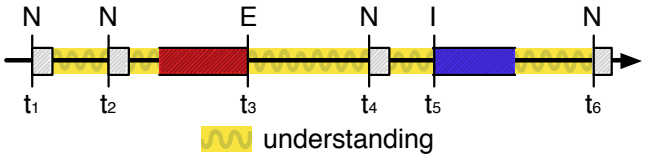

Fig. 8: A DFLOW History with all Activities

the duration of the corresponding activities. There are 3 types of events, thus nine possible combinations which lead to the same number of possible intervals: NN, NI, NE, EN, EI, EE, IN, II, and IE. The last case – that is, an inspection event followed by an editing event – is relatively more complex, because the duration of the inspection activity depends on the duration of the editing activity, which in turn depends on the duration of the inspection. Fig. 9 illustrates the situation.

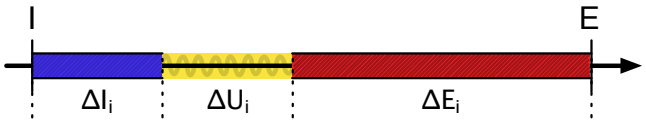

Fig. 9: The Case of Editing after Inspection

According to our estimation model, the duration of the inspection activity ΔI_i and the duration of the editing activity ΔE_i are as follows:

$$\Delta I_i = P_I \times (start(next(I_i)) - start(I_i)) \quad (3)$$

$$\Delta E_i = P_E \times (end(E_i) - end(prev(E_i))) \quad (4)$$

To solve Equation (3) we should know the start time of $next(I_i)$, that is E_i. We cannot know this before solving Equation (4).

In turn, to determine ΔE_i we need the end time of $prev(E_i)$, that is I_i, and we cannot know this a priori. To simplify, we can rewrite Equations (3) and (4) as follows:

$$\begin{cases} \Delta I_i = P_I \times (\Delta - \Delta E_i) \\ \Delta E_i = P_E \times (\Delta - \Delta I_i) \end{cases} \quad (5)$$

where $\Delta = \Delta I_i + \Delta U_i + \Delta E_i$. ΔI_i depends on ΔE_i, and vice-versa. We want to find two percentages, P_I^{IE} and P_E^{IE}, such that ΔE_i and ΔI_i can be computed from the whole interval Δ:

$$\begin{cases} \Delta I_i = P_I^{IE} \times \Delta \\ \Delta E_i = P_E^{IE} \times \Delta \end{cases}$$

By definition, these two fractions are:

$$\begin{cases} P_I^{IE} = \frac{\Delta I_i}{\Delta} \\ P_E^{IE} = \frac{\Delta E_i}{\Delta} \end{cases}$$

By dividing Equations (5) by Δ we obtain:

$$\begin{cases} P_I^{IE} = P_I \times (1 - P_E^{IE}) \\ P_E^{IE} = P_E \times (1 - P_I^{IE}) \end{cases} \quad (6)$$

Solving Equations (6) we obtain:

$$\begin{cases} P_I^{IE} = \frac{P_I - P_I \times P_E}{1 - P_I \times P_E} \\ P_E^{IE} = \frac{P_E - P_I \times P_E}{1 - P_I \times P_E} \end{cases}$$

To summarize, the three time intervals depicted in Fig. 9, can be computed in function of the whole interval Δ as follows:

$$\begin{cases} \Delta I_i = P_I^{IE} \times \Delta \\ \Delta E_i = P_E^{IE} \times \Delta \\ \Delta U_i = \Delta - \Delta I_i - \Delta E_i \end{cases}$$

Reflections. We do not have "perfect" interaction data. For example, during an editing activity we do not know exactly what the developer is doing, only that she is editing. Our future work in this context is to track interactions at finest level possible, *i.e.,* the keystroke level for editing operations, and mouse events (including scrolling, *etc.*) for navigation and inspection activities.

B. Estimation Model for Plog

PLOG interaction histories are composed of just two types of recorded events: N_R and N_W. The former represents pure navigation events, while the latter represents a navigation event that follows an editing activity. Fig. 10 shows a PLOG interaction history.

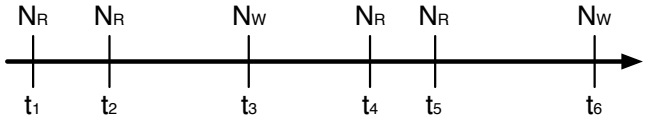

Fig. 10: A Sample PLOG Interaction History

Since PLOG interaction histories lack inspection events, and editing events are implicit, to estimate the duration of developer activities we use a slightly different model with respect to the case of DFLOW histories.

Navigation Activities. Events of type N_R denote pure navigation. As in the DFLOW case, we approximate this time to a fixed duration (ΔN). Fig. 11 shows the updated interaction history with explicit pure navigation activities of fixed duration.

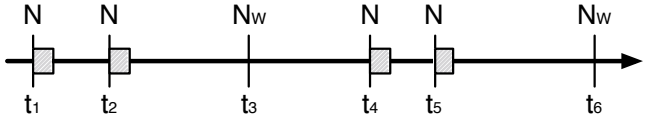

Fig. 11: History with Navigation Activities

Editing Activities. Editing activities are implicit: PLOG records an event of type N_W at time $t(N_W)$ when the user performs a navigation after editing. The editing happened in an unknown moment in the time interval between the end of the previous event, denoted as $end(prev(N_W))$ and $t(N_W)$. To make the editing activity E_i explicit, we place it in the middle of the interval and with a duration (ΔE_i) that is half the duration of that interval. The N_W event is then converted to a navigation activity with a fixed duration ΔN.

$$\Delta E_i = P_E \times (t(N_W) - end(prev(E_i))) \quad (7)$$

Fig. 12 shows the updated interaction history assigning a duration to both navigation and editing activities.

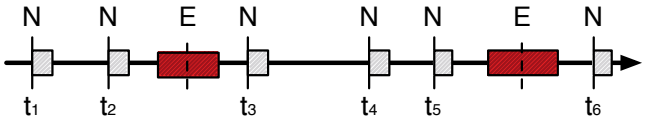

Fig. 12: History with Navigation and Editing Activities

Understanding Activities. We assign every remaining gap in the interaction history to understanding activities. Fig. 13 shows the final shape of the PLOG interaction history.

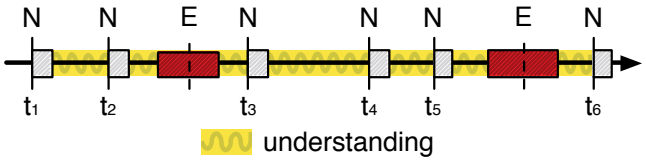

Fig. 13: A PLOG History with all Activities

Reflections. Again, this interaction data is not perfect. Since we do not have keystroke-level events, we must approximate the editing activities. Is it possible that if a user open a new file and spends one minute in that file, 59 seconds could be spent on doing nothing. We believe our approximation is however reasonable, as it does not assign the editing activity an exaggerated weight. Future work in this context is the recording of keystroke-level events, as done for example in ECLIPSEEYE [18].

C. Discussion: ΔN, P_E, and P_I

The estimation model we propose to quantify development activities has three degrees of freedom: i) ΔN, that represents the conventional average duration of navigation events; ii) P_E, that models the average percentage of editing activities between an edit event and the preceeding event; and iii) P_I, which similarly represents the percentage of inspection activities in Smalltalk interaction histories. We intend *navigation* as the "mechanics of navigation", *i.e.,* the clicks in the user interface. Thus, we conventionally assume that each navigation event lasts, on average, 0.5 seconds. We fix ΔN to 0.5s.

Quantifying the right amount of P_E and P_I is out of the scope of this paper, since it would require more fine-grained events to be collected (*e.g.,* keystroke events). Instead, we

discuss how the results of our quantification model change by varying these parameters, obtaining possible lower and upper bounds to the amount of time spent in program understanding. Table V shows how the amount of program understanding changes by varying P_E and P_I in the case of Smalltalk development sessions collected by DFLOW.

TABLE V: Amount of Understanding in Smalltalk Sessions Varying the Estimates of P_E and P_I

P_I \ P_E	0.10	0.25	0.50	0.75	0.90
0.10	**88.44%**	83.74%	75.87%	67.94%	63.16%
0.25	86.17%	81.63%	73.97%	66.20%	61.46%
0.50	82.34%	78.02%	70.71%	63.24%	58.62%
0.75	78.46%	74.28%	67.28%	60.16%	55.73%
0.90	76.10%	71.97%	65.09%	58.16%	**53.93%**

The most pessimistic estimate is shown in the bottom right cell of the table, and corresponds to $P_E = P_I = 90\%$. In this case understanding amounts to 54% of time, which is slightly above the upper-bound of previous estimates [3], [6], [24]. The most optimistic estimate is on the top-left cell of the table, corresponding to $P_E = P_I = 10\%$: In this case, understanding consumes around 88% of the time of developers, which is significantly above previous estimates. Obviously, more accurate estimates lay between these two quite unrealistic extremes, and they are shown in the other cells of the table. It appears very likely that actual time spent by developers in program understanding has been underestimated by previous research by at least 10-20%.

Table VI shows similar results in the case of Java development sessions recorded by PLOG. In this case, since the tool did not record inspection events, the only parameter to vary is P_E. Pessimistic and optimistic estimates for program understanding are similar to the case of DFLOW, suggesting similar considerations about the time spent in program understanding.

TABLE VI: Amount of Understanding in Java Sessions Varying the Estimate of P_E

P_E	0.10	0.25	0.50	0.75	0.90
	94.33%	87.13%	75.12%	63.11%	**55.91%**

V. RESULTS

Tables VII, VIII, and IX summarize the distribution of development activities for all the recorded sessions in the case of DFLOW and PLOG. For each development activity, we calculate the average of the percentage of time spent by developers in each activity.

As we pointed out in the previous section, program understanding is a dominant activity, as it accounts, on average, from 54 to 94% of the total development time in each session. The values are similar despite the profound difference in the way developers produce Java and Smalltalk code. This suggests that the role of program understanding has been underestimated by previous research. Decades ago researchers claimed that understanding a program absorbs about 50% of the time of developers [3], [6], [24], while we find that the percentage is likely to be substantially higher.

Fig. 14 depicts the distribution of the relative importance of the activities using boxplots. Although there can be substantial differences between individual sessions and developers, what emerges is a clear pre-dominance of program understanding activities.

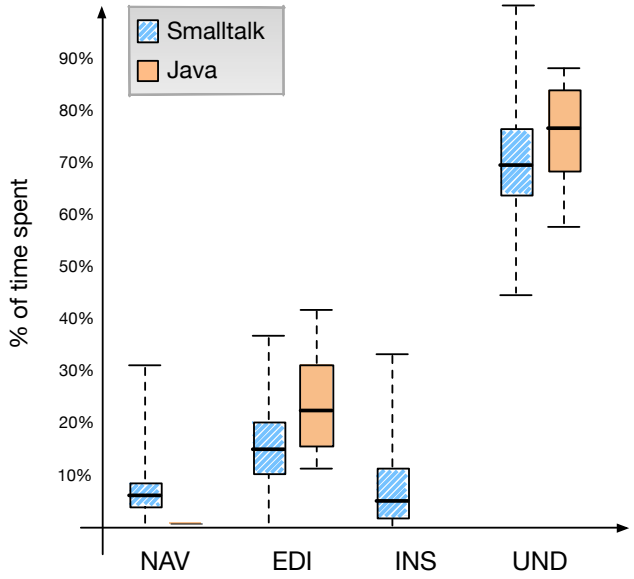

Fig. 14: Development Activities for All Sessions.
Note: inspect (INS) applies only to Smalltalk sessions.

Role of session types. While in interaction histories collected by DFLOW and PLOG program understanding is dominant in both cases, there is difference ranging from 2% to 6% depending on the estimates of parameters, as shown in Table VII. On all the Smalltalk sessions, program understanding ranges from 54 to 88% while on Java sessions these bounds range from 56 to 94%. A possible interpretation of this difference can be found in the fact that PLOG sessions were all enhancement sessions, while the sessions we collected with DFLOW are of different type and include, for example, bug-fixing and refactoring sessions.

Table VII shows the different distribution of the duration of development activities per session type in the case of Smalltalk sessions collected by DFLOW.

The role of understanding is still dominant in all session types: However, there is some significant difference in the distribution of editing and navigation activities. Refactoring sessions, for example, have a high understanding component and make minimal use of inspection activities (0.2-1.7%). While program understanding does not appear to be significantly different between session types, there are some differences on the distribution of other activities. Enhancement and Bug Fixing sessions spend significant time on inspection (around 10%), while Enhancement and Refactoring sessions spend more time on editing.

Developer Diversity. Table VIII lists the relative importance of the activities for each Smalltalk developer. We can

TABLE VII: Development Activities per Session Type.
Min: $P_E = 0.9$ and $P_I = 0.9$, Med: $P_E = 0.5$ and $P_I = 0.5$, and Max: $P_E = 0.1$ and $P_I = 0.1$

Type	N (%)	E (%)			I (%)			U (%)		
		Min	Med	Max	Min	Med	Max	Min	Med	Max
DFLOW: Smalltalk										
Enhancement	7.8%	30.6%	17.4%	3.6%	13.4%	7.8%	1.7%	48.2%	67.0%	86.9%
Refactoring	9.1%	30.4%	16.9%	3.4%	1.7%	0.9%	0.2%	58.9%	73.1%	87.3%
General	6.2%	21.3%	12.1%	2.5%	9.5%	5.5%	1.2%	63.0%	76.2%	90.1%
Bug Fixing	4.6%	25.5%	14.5%	3.0%	17.4%	10.0%	2.1%	52.6%	70.9%	90.3%
All	6.8%	26.9%	15.3%	3.2%	12.3%	7.2%	1.6%	54.0%	70.7%	88.4%
PLOG: Java										
All	0.9%	43.2%	24.0%	4.8%	–	–	–	55.9%	75.1%	94.3%

TABLE VIII: Smalltalk Development Activities per Developer.
Min: $P_E = 0.9$ and $P_I = 0.9$, Med: $P_E = 0.5$ and $P_I = 0.5$, and Max: $P_E = 0.1$ and $P_I = 0.1$

Developer	N (%)	E (%)			I (%)			U (%)		
		Min	Med	Max	Min	Med	Max	Min	Med	Max
SD1	5.1%	30.9%	17.6%	3.6%	13.7%	8.0%	1.7%	50.3%	69.4%	89.5%
SD2	8.9%	21.6%	12.1%	2.4%	20.3%	11.3%	2.3%	49.2%	67.7%	86.4%
SD3	8.0%	30.2%	17.0%	3.5%	5.6%	3.3%	0.7%	56.2%	71.8%	87.9%
SD4	8.5%	26.5%	15.1%	3.1%	12.8%	7.4%	1.6%	52.2%	69.0%	86.8%
SD5	4.6%	16.8%	9.7%	2.1%	9.3%	5.6%	1.3%	69.4%	80.1%	92.1%
SD6	8.4%	10.5%	6.0%	1.2%	10.5%	6.0%	1.2%	70.6%	79.6%	89.1%
SD7	5.7%	22.3%	12.6%	2.6%	10.5%	6.1%	1.3%	61.5%	75.6%	90.4%

deduce diverse "profiles": SD3 has a tendency towards more navigation and editing, and less understanding. In essence he could be characterized as more "aggressive" towards the code base. Similarly, SD2 spends almost the same amount of time on navigation, but distributes almost equally the remaining time between editing and inspection, being thus more "cautious" and she probably frequently verifies the implemented changes. SD5 and SD6 are even more cautious, denoted by their high understanding values. In essence they reflect more on the code before they change it. SD3 however is making very little use of inspecting, which is a preferred activity of skilled developers. SD1 and SD3 have the highest editing and the among the lowest understanding values. In essence, they seem to be at ease with the code base and confidently change it without the need to rely on extensive navigation.

Table IX lists the relative importance of the activities for each Java developer. Again we can infer some developer "profiles". For example JD8 took a short time to implement the changes, but has high understanding and low editing (refer to Table IV for durations). JD2 and JD14 have a similar behavior but they took more time to implement the task. We can say that JD8 is someone who thinks deeply about what to do, and then does it quickly and firm. JD13 is at the opposite end of the spectrum: The high amount of editing time, with relatively high navigation time, denotes a developer who meanders heavily in the code base until she slowly implements the changes.

Reflections. The importance of program understanding has been underestimated. This finding is all the more relevant as it holds across diverse IDEs and programming languages. We believe this to be an important insight that corroborates the importance of research in approaches and tools that deal with program comprehension and reverse engineering.

TABLE IX: Java Development Activities per Developer.
Min: $P_E = 0.9$ and $P_I = 0.9$, Med: $P_E = 0.5$ and $P_I = 0.5$, and Max: $P_E = 0.1$ and $P_I = 0.1$

Dev.	N (%)	E (%)			U (%)		
		Min	Med	Max	Min	Med	Max
JD1	0.5%	60.2%	33.4%	6.68%	39.3%	66.0%	92.78%
JD2	0.8%	20.3%	11.3%	2.25%	78.8%	87.9%	96.90%
JD3	0.8%	52.5%	29.1%	5.83%	46.8%	70.1%	93.41%
JD4	1.3%	32.0%	17.8%	3.56%	66.6%	80.9%	95.12%
JD5	0.7%	68.7%	38.2%	7.63%	30.6%	61.1%	91.64%
JD6	1.0%	40.7%	22.6%	4.52%	58.3%	76.4%	94.46%
JD7	0.9%	62.1%	34.5%	6.90%	37.0%	64.6%	92.22%
JD8	1.1%	24.9%	13.8%	2.76%	74.0%	85.1%	96.19%
JD9	0.7%	28.3%	15.7%	3.14%	71.0%	83.5%	96.13%
JD10	0.9%	34.1%	18.9%	3.79%	65.0%	80.2%	95.32%
JD11	0.9%	27.6%	15.4%	3.07%	71.5%	83.8%	96.05%
JD12	0.8%	49.1%	27.3%	5.46%	50.0%	71.9%	93.71%
JD13	0.7%	75.2%	41.8%	8.35%	24.2%	57.6%	90.98%
JD14	1.2%	22.4%	12.4%	2.48%	76.4%	86.4%	96.32%
JD15	0.7%	50.1%	27.9%	5.57%	49.1%	71.4%	93.72%

A. Threats to Validity

Construct Validity. We record interaction data in terms of events in the IDE with their timestamp. We devised a model to estimate the duration of activities: A threat to validity for our results is the accuracy of this model, that may not precisely capture, for example, the moment when editing activities start. To get a more accurate model to estimate activity durations, one should record more fine-grained interaction data and corresponding events. However, in Section IV-C we described how even varying the degrees of freedoms the essence of this paper remains true.

In our study we only considered the part of software development carried on inside an IDE. Since program comprehension can be carried out throughout the whole software development lifecycle, our findings are a lower bound of the total time devoted to software understanding.

Statistical Conclusion. We considered a total of 190 sessions with around 120,000 interactions, which we repute to be substantial enough to deduce some conclusions, but we did not measure the statistical confidence of our results. We consider this to be part of our future work, once the data collection is precise to the finest level possible (*i.e.,* keystroke-level and accurate mouse tracking).

External Validity. The weight of different development activities may significantly vary with different programming languages and IDE user interfaces. To ameliorate this possible threat, we considered two significantly different programming languages and IDEs, obtaining similar estimates that give us confidence about the generalizability of our results.

A similar argument can be formulated about the developer diversity, which may influence the amount of time required from program comprehension. In our study, we considered 15 different Java developers in the case of interaction data recorded on the ECLIPSE IDE, and 7 different developers with different background and experience in the case of Smalltalk and the PHARO IDE. Further investigation is needed to understand how developers' expertise influences the way they interact with the UI of the IDE.

VI. CONCLUSIONS AND FUTURE WORK

Program understanding is considered one of the most time consuming activities of software development. This claim is taken as an *ipse dixit*, a dogmatic statement to be accepted as it is. We believe that the advances on software engineering practice may have mutated the role of program comprehension, and that this topic needs investigation, also to motivate the importance of areas of research, such as program comprehension, reverse engineering, and mining software repositories.

We presented a study whose aim was to get empirical evidence about the role of program comprehension. We computed an estimate of the duration of developer activities leveraging interaction data, that captures the interaction events of the developer with the user interface of the IDE. Our study considered two significantly different development contexts, one coming from the ECLIPSE IDE and the Java programming language, and another one considering Smalltalk code development in the PHARO IDE. We collected interaction data from 22 developers, 15 working in Java with ECLIPSE and 7 working in Smalltalk with PHARO, totaling hundreds of hours of recorded activities. Our findings strongly suggest that the role of program comprehension has been significantly underestimated by previous research.

Acknowledgments. We acknowledge the support of the Swiss National Science foundation for the project No. 146734 "HI-SEA". We thank Max Leske, Lorenzo Baracchi, and all the developers that helped us gathering the data.

REFERENCES

[1] E. Chikofsky and I. Cross, J.H. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[2] I. Coman and A. Sillitti. Automated identification of tasks in development sessions. In *Proceedings of ICPC 2008 (16^{th} International Conference on Program Comprehension)*, pages 212–217. IEEE, 2008.

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

[4] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.

[5] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.

[6] R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In G. Parikh and N. Zvegintzov, editors, *Tutorial on Software Maintenance*, pages 13–30. IEEE, 1982.

[7] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of IWPSE 2005 (8^{th} International Workshop on Principles on Software Evolution)*, pages 113–122. IEEE, 2005.

[8] O. Greevy, T. Gîrba, and S. Ducasse. How developers develop features. In *Proceedings of CSMR 2007 (11^{th} European Conference on Software Maintenance and Reengineering)*, pages 265–274. IEEE, 2007.

[9] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of AOSD 2005 (4^{th} International Conference on Aspect-Oriented Software Development)*, pages 159–168. IEEE, 2005.

[10] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TSE 2006 (Transactions on Software Engineering)*, 32(12):971–987, 2006.

[11] T. Kobayashi, N. Kato, and K. Agusa. Interaction histories mining for software change guide. In *Proceedings of RSSE 2012 (3^{rd} International Workshop on Recommendation Systems for Software Engineering)*, pages 73–77, 2012.

[12] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of ICSE 2006 (28^{th} International Conference on Software Engineering)*, pages 492–501. ACM, 2006.

[13] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[14] D. Piorkowski, S. Fleming, C. Scaffidi, L. John, C. Bogart, B. John, M. Burnett, and R. Bellamy. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Proceedings of VL/HCC 2011 (Symposium on Visual Languages and Human-Centric Computing)*, pages 109–116. IEEE, 2011.

[15] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension)*, pages 155–164. IEEE CS Press, 2007.

[16] R. Robbes and M. Lanza. Spyware: A change-aware development toolset. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*, pages 847–850. ACM Press, 2008.

[17] R. Robbes, D. Pollet, and M. Lanza. Replaying ide interactions to evaluate and improve change prediction approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 161 – 170. IEEE CS Press, 2010.

[18] Y. Sharon. Eclipseye - spying on eclipse. Bachelor's thesis, University of Lugano, June 2007.

[19] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE TSE 2008 (Transactions on Software Engineering)*, 34(4):434–451, 2008.

[20] J. Singer, R. Elves, and M. Storey. Navtracks: supporting navigation in software maintenance. In *Proceedings of ICSM 2005 (21^{st} International Conference on Software Maintenance)*, pages 325–334. IEEE, 2005.

[21] E. B. Swanson. The dimensions of maintenance. In *Proceedings of ICSE 1976 (2^{nd} International Conference on Software Engineering)*. IEEE, 1976.

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

[23] Y. Yoon and B. A. Myers. Capturing and analyzing low-level events from the code editor. In *Proceedings of PLATEAU 2011 (3^{rd} Workshop on Evaluation and Usability of Programming Languages and Tools)*, pages 25–30. ACM, 2011.

[24] M. Zelkowitz, A. Shaw, and J. Gannon. *Principles of software engineering and design*. Prentice Hall, 1979.