I Know What You Did Last Summer
An Investigation of How Developers Spend Their Time

Roberto Minelli, Andrea Mocci and Michele Lanza

REVEAL @ Faculty of Informatics — University of Lugano, Switzerland

Abstract—Developing software is a complex mental activity, requiring extensive technical knowledge and abstraction capabilities. The tangible part of development is the use of tools to read, inspect, edit, and manipulate source code, usually through an IDE (integrated development environment). Common claims about software development include that program comprehension takes up half of the time of a developer, or that certain UI (user interface) paradigms of IDEs offer insufficient support to developers. Such claims are often based on anecdotal evidence, throwing up the question of whether they can be corroborated on more solid grounds.

We present an in-depth analysis of how developers spend their time, based on a fine-grained IDE interaction dataset consisting of ca. 740 development sessions by 18 developers, amounting to 200 hours of development time and 5 million of IDE events. We propose an inference model of development activities to precisely measure the time spent in editing, navigating and searching for artifacts, interacting with the UI of the IDE, and performing corollary activities, such as inspection and debugging. We report several interesting findings which in part confirm and reinforce some common claims, but also disconfirm other beliefs about software development.

I. INTRODUCTION

Software development is a complex activity that requires both technical knowledge and extensive abstraction capabilities [1]. Even if the outcome of software development is code, the development process is far from being just code writing [2]. In fact, software systems are so large and complex [3] that developers need to build links between the source code and their *mental model* (*e.g.,* [4], [5], [6]).

Constructing the mental model is the base for program comprehension, and has a key role in the development process: Researchers estimated that it absorbs half of the time of a developer [7], [8], [9]. Such a claim, however, is mainly based on anecdotal evidence, throwing up the question of whether it can be corroborated or confuted on more solid grounds.

While being a fundamental part of software development, it is unclear how program comprehension is supported by IDEs (Integrated Development Environments): IDEs offer various tools and facilities to support the development process, like (1) Code Editors, (2) Code Browsers, and (3) Debuggers [10], [11]. None of these components is dedicated to program comprehension. Instead, comprehension emerges from the complex interleaving of such activities. Moreover, researchers discovered that some UI paradigms (*e.g.,* windows- or tabs-based IDEs) may negatively influence development, hindering comprehension and generally developer productivity [12]. While this claim is intuitively convincing, there is no quantitative evidence on how much time is spent on simply fiddling with the UI of an IDE.

While interacting with IDEs, developers generate a large amount of data [13], [14]. These interactions happen at different levels of abstraction: they can be conceptual events, like adding a method to a class, or low level events like pressing keys or moving the mouse to navigate between entities. While researchers believe that this data is valuable [14], [15], most of these interactions are not captured.

Our aim is to record interaction data and measure the time effectively devoted to different activities. With this data, we provide insights on the distribution of development activities and we quantitatively answer questions like:

1) What is the effective time spent in program comprehension/understanding? What about the other activities like editing, inspecting and navigating through code entities?
2) How much time do developers spend in fiddling with the UI of an IDE?
3) What is the impact of the fragmentation of the development flow?

We present an in-depth analysis on how developers spend their time. We collected fine-grained interaction data using DFLOW [16], our non-intrusive profiler for the PHARO IDE[1]. Our dataset consists of 740 development sessions of 18 developers. DFLOW collected about 200 hours of development time, amounting to more than 5 million of IDE events. We propose an inference model of development activities to precisely measure the time spent in editing, navigating and searching for artifacts, interacting with the UI of the IDE, and performing activities such as inspection and debugging. Among our findings, we discovered that program comprehension absorbs more time than generally assumed, and that fiddling with the UI of IDEs can substantially hinder the productivity of developers. We make the following contributions:

- An in-depth analysis of how developers spend their time;
- An inference model of development activities to precisely measure the time of different activities;
- A comparison between our findings and the common claims and beliefs available in the literature;
- A brief presentation of DFLOW, the tool with which we collect interaction data.

Structure of the Paper: Section II presents our dataset and the target IDE. In Section III we explain how we make sense of the data and present the results, and in Section IV we illustrate our findings. Section V discusses the related work while Section VI concludes our work.
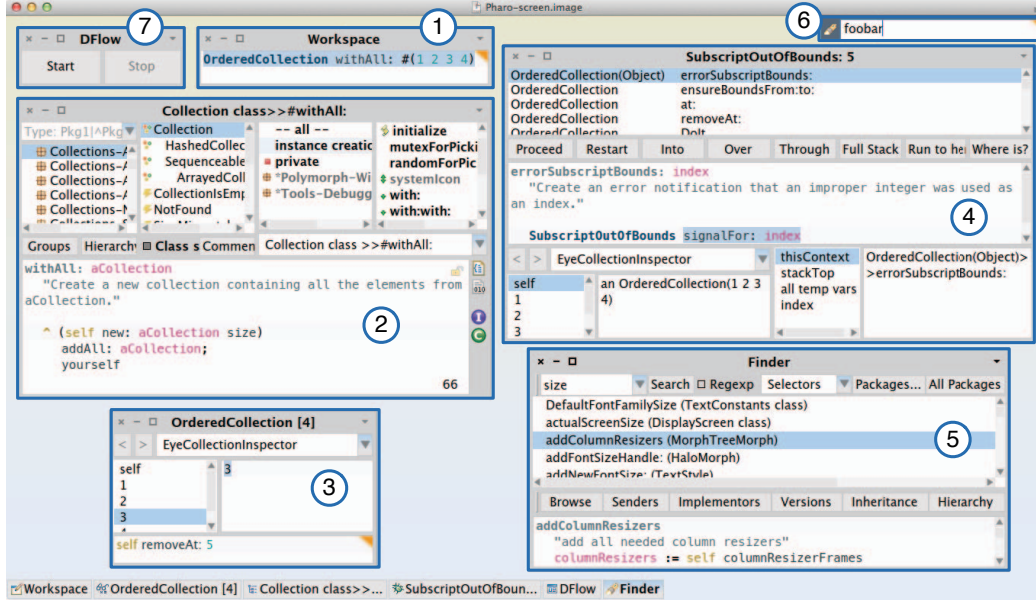
[1]See http://pharo.org

Fig. 1. The PHARO IDE with: (1) workspace, (2) code browser, (3) inspector, (4) debugger, (5) Finder UI, (6) "spotlight" search, and (7) UI of DFLOW

This paper consists of many assumptions that could be subject of a critical stance towards them. For this reason, many sections contain an explicit *advocatus diaboli* paragraph, that explicitly illustrates possible objections to the assumptions and generally threats to validity.

II. INTERACTION DATA

This section introduces the PHARO IDE (II-A), our interaction profiler DFLOW (II-B) and details the dataset (II-C).

A. The Pharo IDE

The target IDE for our study is the PHARO IDE. Figure 1 shows a screenshot of the IDE during a development session. The PHARO IDE is a window-based environment, as opposed to a tab-based environment like the Eclipse IDE[2]. In a window-based paradigm users interact with different, possibly overlapping, windows to perform development activities. A window is "active" when it has the focus and is the target of interactions. The commonly used windows are:

- The **Code Browser** lets the user navigate through source code entities (upper part) and edit source code (lower part). The example in Figure 1.2 displays the source code of the method #withAll: of Collection class.
- The **Workspace** lets the user write and run snippets of code. This window is mainly used to test small snippets of code on-the-fly. The example in Figure 1.1 initializes an OrderedCollection from the elements of an array.
- The **Inspector** allows to browse instances of objects. The example in Figure 1.3 shows the object resulting from evaluating the code in Figure 1.1. The lower part of the Inspector can be used to execute code on the current

object. In this example self is bound to the object that is currently being inspected.
- The **Debugger** is the user interface designed for code debugging. Figure 1.4 shows the debugger that results from the execution of the code in the inspector of Figure 1.3, *i.e.,* the attempt to remove the 5th element in an array of size 4. The upper part lets the user browse the call stack, the middle part lets the user read and modify the source code of the current entity selected in the call stack, and the lower part is similar to an inspector.
- The **Finder UI** (see Figure 1.5) and a **"Spotlight-like" UI** (see Figure 1.6) are the UIs to search for code artifacts.

B. DFlow: An Interaction Profiler

To collect interaction data inside the PHARO IDE we built DFLOW [16], a non-intrusive interaction profiler for the PHARO IDE. It offers a minimalistic user interface (see Figure 1.7) that lets developers start and stop the recording of a session. When a developer stops the recording, DFLOW asks for additional (optional) information, such as a brief description of the session and the type, before sending the data to our server. The session type can be *gereral purpose*, *refactoring*, *enhancement*, or *bug-fixing*. DFLOW collects interaction data of different types and records a timestamp down to millisecond precision, for each event. Our approach records two different categories of events: *Meta events* and *Low-level events*.

Meta events represent actions, triggered from the developer inside the IDE, conceptually related to development. Examples include browsing the source code of a method or adding a new method to a class. These events contain a pointer to the program entity involved in the action (*e.g.,* class or method).

[2]See https://www.eclipse.org

Meta events can be classified as:

- *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*, that happen when developers inspect the state of run-time objects (*e.g.,* with the debugger);
- *Editing events*, that modify source code, like adding a new class or modifying the code of a method.

Low Level Events are basic UI events that deal with mouse, keyboard, or UI manipulation. In particular, DFLOW records:

- *Window events* are all the events that deal with the different windows of the PHARO IDE, like opening, closing, moving or resizing a window;
- *Mouse events* include all movements, scrolls, and clicks with the mouse inside the UI of the IDE. Each event knows the position of the cursor; in particular, movement events are specialized when the mouse moves outside the main PHARO window to other areas of the screen (*mouse-out* event) or back inside (*mouse-in* event);
- *Keyboard events* represent all the keystrokes in the session. Each of this event knows the exact keystroke (or combination of keystrokes with modifiers like *shift*) that has been typed.

Table I details all the different events collected by DFLOW.

TABLE I
EVENTS RECORDED BY DFLOW

Meta Events: Navigation, Inspection, Editing	
NE_1	Opening a Finder UI
$NE_{2,3,4}$	Selecting a package, method, or class in the code browser
$NE_{5,6}$	Opening a system browser on a method or a class
NE_7	Selecting a method in the Finder UI
NE_8	Starting a search in the Finder UI
IE_1	Inspecting an object
IE_2	Browsing a compiled method
$IE_{3,4}$	Do-it/Print-it on a piece of code (*e.g.,* workspace)
$IE_{5,6,7}$	Stepping into/Stepping Over/Proceeding in a debugger
IE_8	Run to selection in a debugger
$IE_{9,10}$	Entering/exiting from an active debugger
$IE_{11,12}$	Browsing full stack/stack trace in a debugger
$IE_{13,14,15}$	Browsing hierarchy, implementors or senders of a class
IE_{16}	Browsing the version control system
IE_{17}	Browse versions of a method
$EE_{1,2}$	Creating/removing a class
$EE_{3,4}$	Adding/removing instance variables from a class
$EE_{5,6}$	Adding/removing a method from a class
EE_7	Automatically creating accessors for a class
User Interface Events	
$WE_{1,2}$	Opening/closing a window
WE_3	Activating a window, *i.e.,* window in focus
$WE_{4,5,6,7}$	Resizing/moving/minimize/maximize a window
Mouse & Keyboard Events	
$ME_{1,2}$	Mouse button up/down
$ME_{3,4}$	Scroll wheel up/down
ME_5	Mouse move
$ME_{6,7}$	Mouse-out/in
KE_1	Keystroke pressed

Previous versions of DFLOW collected only meta events and window events [16], [17]; with such events only it is not possible to provide precise estimates of high-level development activities like code editing and program understanding [18].

Our goal is to provide precise measures of such activities, thus we enriched DFLOW with the recording of low-level UI events like mouse and keyboard events.

Advocatus Diaboli. In this study we considered the PHARO IDE and the Smalltalk programming language. One may argue that it is unconventional and not industry standard. However, in a previous preliminary study we analyzed both Java and Smalltalk interaction histories and our results were not showing significant differences in terms of the role of understanding between the two different environments [18]. Nonetheless, we plan to implement a fine-grained recorder like DFLOW on at least another IDE to give us confidence about the generalizability of our results.

C. The Dataset

Table III summarizes our dataset. The first two columns report anonymized identifiers of the developer with their total number of sessions collected with DFLOW. We call "session" a sequence of IDE interactions without *"idle periods"* longer than, say, 5 minutes. DFLOW detects when the developer is away from the keyboard and splits the interactions into multiple sessions (discarding the idle period). For each developer we collected: 1) the recording time, 2) the number of low-level events, 3) the number of meta events, and 4) the number of windows used during development sessions. Each row in the table reports values for a single developer. The leftmost part of the table reports the total values, while the rightmost part the average values (per session). In the last row (*i.e.,* All) total values accumulate the values for all developers while average values are computed using a weighted arithmetic mean across all developers weighted on the number of sessions.

To invite participants we sent a call on the PHARO-DEV mailing list[3]. Eighteen developers, both professionals and academics, answered the call and helped us in the collection of their interactions. Participants were not assigned specific tasks. Instead they have been working on their own personal projects. They all share a common code base (*i.e.,* the source code of PHARO) but we have no information on the size of their own private projects. The dataset features 2 master students, 9 PhD students, and 7 professionals. We distinguish 3 levels of expertise, *i.e.,* how many years they have been programming in PHARO. D9 is the only developer that can be considered a *novice*. The others are quite familiar with the PHARO IDE, with an expertise between 1 and 5 years (6 developers) or longer than 5 years (11 developers). Table II summarizes demographics information.

TABLE II
DEMOGRAPHICS OF DEVELOPERS

Developer	Role	Expertise (years)
D9	Master Student	< 1
D14	Master Student	1–5
D1, 2, 10, 15, 16	PhD Student	1–5
D3, 12, 13, 18	PhD Student	> 5
D4, 5, 6, 7, 8, 11, 17	Professional	> 5

[3]See http://pharo.org/community

TABLE III

Dev.	#S	Total Rec. Time	Total Low Level ME	KE	WE	All	Total #Meta	Total #Win	Average Rec. Time	Average Low Level ME	KE	WE	All	Average #Meta	Average #Win
D1	407	89h 21m 46s	1,436,332	104,622	16,402	1,557,356	80,030	3,966	00h 13m 10s	3,529.07	257.06	40.30	3,826.43	196.63	9.74
D2	136	52h 09m 52s	1,945,028	143,852	33,801	2,122,681	58,468	5,677	00h 23m 01s	14,301.68	1,057.74	248.54	15,607.95	429.91	41.74
D3	76	28h 50m 44s	596,928	66,717	9,376	673,021	35,168	2,080	00h 22m 46s	7,854.32	877.86	123.37	8,855.54	462.74	27.37
D4	32	06h 17m 34s	129,492	6,441	1,426	137,359	6,653	539	00h 11m 48s	4,046.63	201.28	44.56	4,292.47	207.91	16.84
D5	19	01h 31m 45s	21,575	1,709	344	23,628	1,087	90	00h 04m 50s	1,135.53	89.95	18.11	1,243.58	57.21	4.74
D6	14	04h 20m 26s	62,857	8,628	449	71,934	4,037	132	00h 18m 36s	4,489.79	616.29	32.07	5,138.14	288.36	9.43
D7	11	04h 13m 54s	82,294	9,670	1,573	93,537	3,201	453	00h 23m 05s	7,481.27	879.09	143.00	8,503.36	291.00	41.18
D8	9	01h 14m 47s	19,550	328	103	19,981	441	34	00h 08m 19s	2,172.22	36.44	11.44	2,220.11	49.00	3.78
D9	9	03h 19m 54s	26,970	3,194	732	30,896	2,120	231	00h 22m 13s	2,996.67	354.89	81.33	3,432.89	235.56	25.67
D10	8	01h 03m 33s	14,797	1,232	252	16,281	1,471	67	00h 07m 57s	1,849.63	154.00	31.50	2,035.13	183.88	8.38
D11	5	01h 12m 38s	33,775	2,521	283	36,579	3,510	80	00h 14m 32s	6,755.00	504.20	56.60	7,315.80	702.00	16.00
D12	5	01h 07m 48s	31,186	2,554	321	34,061	1,381	89	00h 13m 34s	6,237.20	510.80	64.20	6,812.20	276.20	17.80
D13	2	00h 05m 56s	3,332	273	54	3,659	12	10	00h 02m 58s	1,666.00	136.50	27.00	1,829.50	6.00	5.00
D14	1	01h 52m 38s	10,420	551	920	11,891	5,033	182	01h 52m 38s	10,420.00	551.00	920.00	11,891.00	5,033.00	182.00
D15	1	00h 01m 58s	714	21	11	746	32	5	00h 01m 58s	714.00	21.00	11.00	746.00	32.00	5.00
D16	1	00h 15m 57s	4,741	565	60	5,366	305	20	00h 15m 57s	4,741.00	565.00	60.00	5,366.00	305.00	20.00
D17	1	00h 04m 55s	1,347	49	33	1,429	3,423	8	00h 04m 55s	1,347.00	49.00	33.00	1,429.00	3,423.00	8.00
D18	1	00h 07m 46s	5,197	38	82	5,317	292	28	00h 07m 46s	5,197.00	38.00	82.00	5,317.00	292.00	28.00
All	738	197h 13m 54s	4,426,535	352,965	66,222	4,845,722	206,664	13,691	00h 16m 02s	5,998.01	478.27	89.73	6,566.02	280.03	18.55

Our dataset features 738 sessions amounting to roughly 200 hours of *actual* development time, *i.e.,* in the table, the total (and average) recording time column do not include the timespans in which the developers were idle (*i.e.,* DFLOW recorded no interactions with the IDE for more than 5 minutes). The dataset includes more than 5 million of events (*i.e.,* both low level and meta). Sessions, on average, last for 16 minutes and count ca. 7,000 events. DFLOW recorded events for more than 13,000 windows, an average of 18.55 per session.

The total number of low-level mouse and keyboard events (*i.e.,* mouse and keyboard) is significantly (and not surprisingly) larger with respect to meta events, which begs the question whether these low-level events are related to meta-events. For example, sequences of mouse events can be related to specific entity inspections or navigation, but also with simple UI fiddling or adjustment.

There are substantial differences between different developers. The first 8 developers' average session time varies from 4 minutes and 50 seconds to almost 23 minutes and 5 seconds. This pinpoints the differences in their programming flow: Since recording time is free of idle time, this value is the "pure" time the developers spent in doing actual work. A developer with a short session time is a developer whose development flow is highly fragmented. Among the first eight developers, D2 and D7 are the developers with the less fragmented flow: Their sessions last, on average, more than 23 minutes, a duration which is in line with, for example, time management methods such as the *"Pomodoro Technique"* of extreme programming developed by Cirillo [19].

On the other hand, developers like D5 and D8 have a fragmented flow: They work on average for around 4m 50s and 8m 19s respectively before having an interruption of at least 5 minutes. This corroborates the findings of LaToza *et al.* who established that developers are frequently interrupted, and that recovering from the interruptions can be difficult [20].

Observing the distribution of low-level and meta events per developer we can speculate on how developers use the IDE. For example, on average D2 triggers more low-level events with respect to other developers (on average 15,607.95, more than twice the overall average). An interpretation for this is that she is constantly fiddling with the UI of the IDE to better accomodate her needs. Researchers already pointed out possible problems in dealing with the UIs of IDEs. For example, Rötlisberger *et al.* called *"window plague"* the problem developers might have while dealing with multiple windows or tabs [12]. Developer D8, instead, seems to be at ease with the UI of the IDE, since her number of low-level events is well below average and, in particular, she has the lowest average number of window events per session.

The number of meta events can be a rough indicator of productivity: They represent actions like creating/removing a class/method, or exploring code artifacts or inspecting objects. In terms of meta events, D2, D3, and D7 seem to be the most productive developers, while D5 and D8 are the less productive ones. This correlates with the fact that the development flow of D5 and D8 is more fragmented than the one of D2 and D3.

Advocatus Diaboli. The dataset is large enough to draw statistical conclusions. However, it has flaws related to the distribution of recorded sessions among the developers: More than half of the sessions come from the same developer and some developers provided us with only few minutes of interaction data. Since the last fact may influence conclusions about developer diversity, we will not consider such individuals when we reason about single developers. We included their values from completeness, but we grayed them out from all the tables in this paper. Another argument can be formulated about the missing purpose (*e.g.,* debugging, refactoring, *ex novo* implementation) of sessions. Further investigation is needed to understand how the purpose of a session and the code base and project size influence studies like the one we propose.
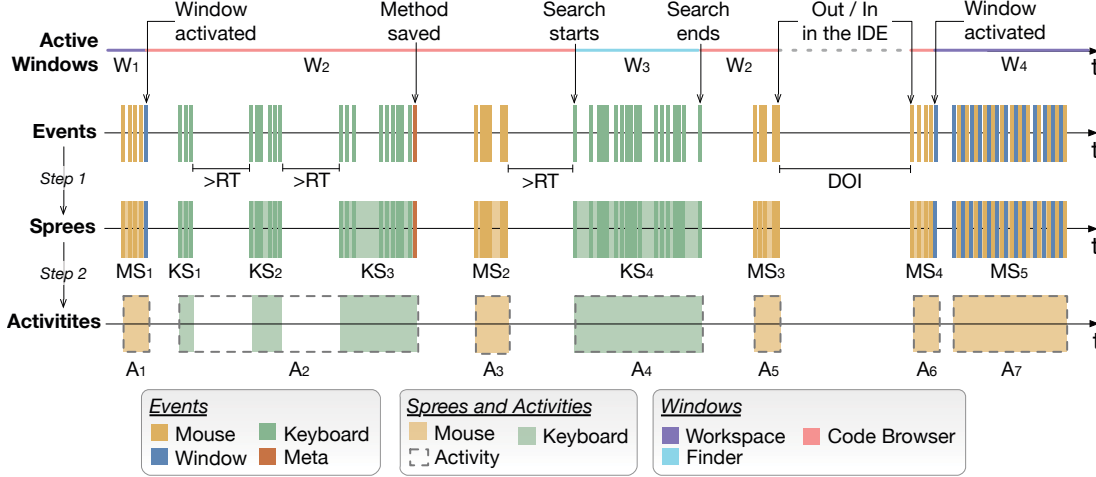
Fig. 2. Sprees and Activities from Fine-Grained Interaction Histories

III. MEASURING DEVELOPMENT ACTIVITIES

A. Events, Sprees, and Activities

Interaction histories recorded with DFLOW are composed of sequences of meta-events and low-level events. Each event has a timestamp associated to it. Events have virtually no duration, *e.g.,* a keystroke happens at a certain point in time, and we record that moment with millisecond precision, but we cannot precisely measure its duration, *i.e.,* we do not have information about for how long the user pressed the key. This is however not a problem, as the point is to group these low-level events into sequences of higher level events for which it is easy to measure the precise duration.

Our model uses the concept of *reaction time* (RT) to aggregate events. While typing a piece of code, for example, a developer performs a sequence of keystrokes. These keys are separated by small pauses, in the range of milliseconds, due to the physical actions required involved, *i.e.,* pressing keys on a keyboard. In this time, the developer is focused on the writing activity per se. When the sequence of keystrokes terminates, the developers pause, reflecting on the just written piece of code, and planning the next steps. The *reaction time* is the time that elapses between the end of a physical action sequence (typing, moving the mouse, *etc.*) and the beginning of concrete mental processes like reflecting, planning, *etc.* which represent the basic moments of program understanding.

We denote the reaction time with RT and assign the duration of 1 second to it. This duration, known also as "Psychological Refractory Period" [21] varies among humans, also depending on the task at hand, between 0.15 and 1.5 seconds. This might appear as a threat to validity, but as iterating through all possible values in that range did not affect our findings, we settled on the 1 second compromise, which is a conservative choice.

Our inference model uses the reaction time to group low-level events into higher-level abstractions: *mouse (or keyboard) sprees* and *development activities*.

Mouse/Keyboard Sprees. A spree is a sequence of mouse/keyboard events where each subsequent pair of events satisfy the following temporal and conceptual constraints:

- The time difference between their timestamps is smaller than the reaction time RT.
- They are performed on the same *window* of the IDE.
- Between them there is no *trigger event*, *i.e.,* a *meta event* which conceptually breaks a spree. Examples include adding or editing a method, navigating in the code browser (see Figure 1.2), or inspecting an object.
- At most one of its events should conceptually initiate a new spree or terminate the current one, *e.g.,* the keyboard shortcut `<Shift-cr>` that triggers a search action in the PHARO IDE should initiate a spree while the mouse moving outside the IDE terminates the current spree.

Development Activities. An activity is a sequence of mouse/keyboard sprees satisfying a number of constraints. We identify three kinds of activities:

- *Search Activities* are all the activities where the user performs a search inside the IDE (*e.g.,* on the Finder UI, Figure 1.5).
- *Inspection Activities*: examining an object by means of an inspector (see Figure 1.3).
- *Browser Activities* are all remaining activities after removing both search and inspection activities. They happen on specialized windows of PHARO, like the code browser, such as editing and navigation.

In Practice: Figure 2 exemplifies our two-step process to construct development activities from raw interaction histories. The timeline on top shows a sample recorded interaction history, *i.e.,* a sequence of low-level and meta events.

Step 1: *From Events to Sprees:* The first step towards the construction of activities is aggregating the events into mouse and keyboard sprees. At the beginning of the sample interaction history shown in Figure 2 there is a sequence of mouse events. We construct a new mouse spree (MS_1) by

adding these events until one of the interrupting conditions is met. In this case, the reaction time is not elapsed (the difference between the last mouse event and the following event is smaller than RT), but there is a window event that activates a new window. MS_1 is complete. The next event in the sequence is a keystroke. We start composing a new keyboard spree (KS_1). After adding some keystrokes to it, the reaction time elapses (the difference between the timestamp of the next event and the last event in the current spree is greater than RT), thus we finalize KS_1. The same situation happens for both KS_2 and KS_3. In the case of KS_3, however, there is also a meta event of type EE_5 (see Table I), *i.e.,* the action a developer performs to either add or edit the method of a class. We call this a **trigger** event that we associate to the current spree, *i.e.,* KS_3. MS_2, the next mouse spree, is interrupted due to the expiration of the reaction time. KS_4 is a keyboard spree that starts when the user invokes the action that triggers the search in the PHARO IDE. Its stopping condition is the end of the search. The next mouse spree, MS_3 is interrupted because the mouse moves outside the PHARO IDE window. The time between the end of the spree is marked as DOI (Duration Outside IDE). The next event, a mouse event in this case, originates the next mouse spree, MS_4, interrupted due to the change of the window in focus. The last mouse spree, instead, is a dense sequence of mouse events with interleaving window events (not window activations, as they would have triggered the end of the spree). The timeline in the middle shows the results of this step: From dozens of low-level events we generated 5 mouse sprees and 4 keyboard sprees.

Step 2: *From Sprees to Activities:* The second step is to aggregate the sprees into high-level development activities. From the refined interaction history with sprees (*i.e.,* the middle timeline in Figure 2) our approach extracts, in sequence, search, inspection, and browsing activities. A spree can be part of a single activity, thus when we assign sprees to activities we *mark* them as already used. A search activity can be either performed on a Finder UI or triggered by a keyboard shortcut to start/confirm/abort a "spotlight-like" search (*i.e.,* <Shift-cr> to start the search, <cr> to confirm it, or either a mouse click or the <esc> keystroke to abort it). In this case, there is a search activity composed of the single key spree KS_4, triggered by the spree containing the shortcut <Shift-cr>. Inspect activities are performed on an inspector or triggered by inspection meta events (see Table I). In the sample interaction history there are neither inspection meta events nor inspector windows, thus there are no inspection activities. All the remaining sprees are aggregated into "browser activities". Starting from the beginning of the interaction history, MS_1 is the first activity. The activity is interrupted because the next spree is on a different window due to the window activation at the end of MS_1. The next three keyboard sprees happen on the same window, and thus they get grouped into a single activity. The following activity is composed by the single mouse spree MS_2, because the next spree, KS_4, is marked as part of another activity, A_4. MS_3, the next mouse spree, creates an activity because there

is an interruption, *i.e.,* out of the IDE. The second to last activity is only composed of the spree MS_4, because then there is a window focus change. Finally, the last remaining spree, MS_5, concludes the interaction history and makes up the last activity. The bottom timeline in Figure 2, shows the final result: From 9 sprees we end up with 1 search activity and 6 browser activities.

Our dataset is thus reduced from 5 million of recorded events to 174,366 sprees and to 31,609 development activities.

B. Decomposing Software Development

We decompose software development into the following distinct and disjunct categories: *understanding*, *navigation*, *editing*, and *UI interactions*. Moreover, we also track the time spent outside of the IDE.

Understanding (U) is the time spent in program comprehension, that aggregates three main components:

1) The *Basic Understanding (BU)* is the sum of all the basic moments of program understanding. It is represented by all time intervals between sprees which are longer than the *reaction time*. Basic understanding can be performed inside development activities (*i.e.,* intra-activities) and across subsequent activities (*i.e.,* inter-activities).

   - BU_{intra} is the *Basic Intra-Activity Understanding Time* that is the sum of all the time intervals, longer than RT, between the sprees composing an activity.
   - BU_{inter} is the *Basic Inter-Activity Understanding Time* that is the sum of all the time intervals, longer than RT, between subsequent activities.

2) *Inspection (I)* is the time a developer spends in inspection activities (mostly using inspector windows), computed as the sum of the duration of all the sprees that have as trigger an inspection meta event (see Table I).

3) *Mouse Drifting (MD)* is the time the user "drifts" with the mouse without clicking. It is computed as the sum of the duration of the mouse sprees that are only composed or mouse movements, and no clicks. We also recorded the screen casts of several of the sessions collected by DFLOW and discovered that a large part of this time is absorbed by what we call *mouse-supported reading*, *i.e.,* when a developer uses the mouse as a "pointer" to support the reading of source code (*e.g.,* MS_2 in Figure 2).

Navigation (N) is the time spent in browsing through software [22]. This time includes both navigation using code browsers or package explorers and searching for particular program entities or pieces of code.

1) *Browsing (B)* is the time the developer spend while navigating between program entities. It is computed as the sum of the duration of the sprees that have as trigger a navigation meta event (see Table I).

2) *Searching (S)* is the time a user spends in searching particular program entities such as methods or classes. This can be achieved using UIs such as the Finder UI (see Figure 1.5) or dedicated keyboard shortcuts, *e.g.,* <Shift-cr> in the PHARO IDE triggers a search

dialog, see Figure 1.6. This time is the sum of the duration of the sprees happening inside user interfaces that support search activities. We remove from this time both the *user interactions*, *mouse drifting*, and *editing* time that might happen inside search UIs.

Editing (E) is the time that the developer spend editing source code. This is computed by summing up the duration of all the sprees that have as trigger an editing meta event (see Table I). For browsing activities, this definition is refined depending on the window where the activity is performed. In a code browser, for example, all the keystroke sprees that have no trigger for navigation contribute to editing time. Examples are KS_1, KS_2, and KS_3 in Figure 2.

User Interface Interaction (UI) is the time explicitly devoted in fiddling with the UI. This includes, for example, moving or resizing windows to better organize the IDE. It is computed as the sum of the duration of the mouse sprees that have interleaving window resize and move events in their timespan. An example is MS_5 in Figure 2.

Time Spent Outside the IDE (OI) is the time that the developer spend outside the PHARO IDE window. It is computed by summing up all the timespans that elapse between all activities that terminate with the PHARO IDE losing focus (*e.g.,* mouse goes outside the main IDE window) and the beginning of the next activity in the interaction history. It is denoted as DOI (*i.e.,* Duration Outside the IDE) in Figure 2.

Advocatus Diaboli. We inferred activities starting from low-level events like keyboard and mouse sprees, and meta-events from the IDE like saving a method or inspecting a field of an object. Recording low-level events minimizes the possibility that we discard relevant events and do not capture exact duration of activities. However, since we may not monitor every possible meta event of the IDE (*e.g.,* special ad-hoc plugins and widgets) we may potentially interpret some activity in the wrong way. To cope with this threat, we made sure that all developers used the standard widgets of PHARO for which our model correctly classifies events and sprees in the correct class of activities. As future work, we plan to cross-validate our automated activity extraction with concrete observations (*e.g.,* think-aloud) to understand to what extent the extracted activities match the actual activities.

The same applies with basic understanding. In principle, the fact that small periods of idleness (inter- and intra-activities) are mapped to program understanding is an explicit assumption that we made, but indeed they could be mini interruptions unrelated to development, like the programmer checking his phone. However, the reverse critic could be done to some of the moments spent outside the IDE. They could be timespans spent in checking documentation or other development artifacts supporting program understanding, that are completely absent from our model. We still need cross-validation to ensure that our interpretation is correct, but we believe that the issues above compensate themselves and do not invalidate our measurements involving program understanding.

IV. HOW DEVELOPERS SPEND THEIR TIME

Figure 3 summarizes the average distribution of activities of the developers and their sessions in our dataset.
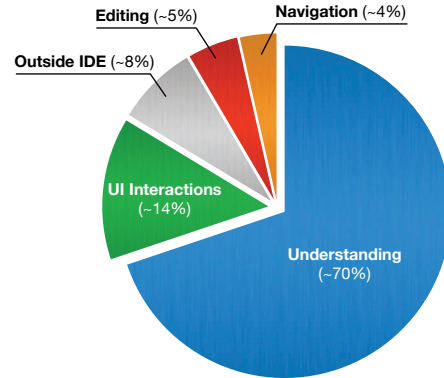

Fig. 3. How do developers spend their time?

Program understanding is as expected the dominant activity, but as we see our analysis attributes to it even more importance than what the common knowledge suggests, reaching a value of roughly 70%. This is a strong point in favor of the research done in the field of program comprehension and reverse engineering. A rather worrisome finding is the time spent in UI interactions: roughly 17% of the time is spent in fiddling with the user interface of the IDE. The relatively small amount of time spent in editing and navigation (roughly 5% for both of them) is not surprising. In the case of editing it corroborates previous research, which established that when it comes to actual writing of source code the so-called "productivity" of developers is rather low [13]. This is yet another argument against measuring productivity with metrics like lines of code. In the case of navigation it emphasizes the fact that programming is not only writing, but rather a complex mental activity where a software system is perceived and navigated like a *graph* composed of nodes (*i.e.,* program entities) and edges (*i.e.,* relationships and dependencies between them), and not like a flat collection of textual artifacts. Last, the time spent outside of the IDE, *during* a session, corroborates the findings of LaToza *et al.* [20]: Developers are often exposed to micro-interruptions of voluntary nature (*e.g.,* emails, instant messages, social networks notifications). Our dataset establishes that roughly 10% of the development time is spent on such interruptions. Table IV presents the detailed results aggregated per developer. The remainder of this section discusses in details the dynamics of program understanding, the time spent outside the IDE, and the impact of UI, editing, and navigation.

A. Program Understanding

Components of Understanding. The attentive reader has probably noted that Figure 3 does not include some of the components described in the previous section, such as inspection and mouse drifting – even if their contribution is relatively low, it is not negligible. They have not been elided, instead, we grouped them as components of program understanding.

TABLE IV
TIME COMPONENTS AGGREGATED PER DEVELOPER

| | Understanding | | | | | Navigation | | | Edit (%) | UI (%) | OI (%) |
| | Base | | Other | | | | | | | | |
Dev.	BU_{intra} (%)	BU_{inter} (%)	I (%)	MD (%)	Tot. (%)	B (%)	S (%)	Tot. (%)			
D1	35.07%	36.37%	0.25%	2.96%	74.64%	2.37%	0.38%	2.75%	3.07%	9.01%	10.53%
D2	37.41%	6.65%	3.23%	5.51%	52.79%	4.81%	1.19%	6.00%	9.76%	28.51%	2.94%
D3	47.68%	22.22%	0.87%	3.76%	74.54%	4.47%	0.26%	4.73%	5.44%	12.21%	3.08%
D4	38.06%	27.86%	0.53%	3.28%	69.74%	3.21%	0.14%	3.35%	3.75%	14.13%	9.03%
D5	22.90%	45.67%	0.07%	1.89%	70.53%	1.20%	0.00%	1.20%	2.74%	10.90%	14.63%
D6	52.85%	23.40%	0.11%	2.25%	78.61%	3.41%	0.05%	3.46%	9.18%	8.76%	0.00%
D7	56.77%	10.70%	0.07%	2.06%	69.59%	1.82%	0.00%	1.82%	10.67%	17.57%	0.35%
D8	45.66%	24.84%	0.00%	3.60%	74.09%	2.26%	0.00%	2.26%	1.29%	11.34%	11.03%
D9	58.68%	17.93%	0.73%	0.93%	78.26%	1.00%	0.09%	1.09%	6.08%	13.45%	1.12%
D10	36.94%	28.34%	0.57%	1.96%	67.81%	5.41%	0.00%	5.41%	4.10%	22.41%	0.28%
D11	39.11%	7.77%	0.00%	4.27%	51.14%	5.90%	0.00%	5.90%	6.66%	11.59%	24.70%
D12	28.58%	8.25%	0.00%	4.24%	41.07%	2.51%	0.00%	2.51%	10.49%	31.73%	14.20%
D13	52.97%	15.36%	0.30%	4.67%	73.29%	1.47%	0.00%	1.47%	4.07%	21.16%	0.00%
D14	7.04%	86.76%	0.07%	0.21%	94.08%	0.56%	0.00%	0.56%	1.24%	4.13%	0.00%
D15	54.14%	22.58%	0.00%	1.54%	78.26%	1.73%	0.00%	1.73%	2.66%	17.34%	0.00%
D16	64.80%	1.09%	2.93%	3.47%	72.28%	1.39%	0.16%	1.54%	10.01%	15.91%	0.26%
D17	73.74%	3.48%	0.00%	0.85%	78.07%	4.46%	0.00%	4.46%	1.97%	15.50%	0.00%
D18	29.15%	6.71%	0.00%	4.21%	40.06%	4.16%	0.00%	4.16%	1.32%	33.45%	21.01%
Average	**37.82%**	**27.70%**	**0.87%**	**3.46%**	**69.85%**	**3.09%**	**0.47%**	**3.56%**	**4.90%**	**13.81%**	**7.88%**

Inspection is an activity, performed on objects at runtime, to check their status, and ultimately to understand the dynamic aspects (*i.e.,* the behavior) of the code. It is essential in any process involving running code, like debugging. In live environments like PHARO, inspection can be used to inspect any runtime object created by running any piece of code, *i.e.,* it directly supports the understanding of run-time behavior.

Mouse drifting is another component of program understanding that corresponds to mouse movements without any apparent consequent action. One of the typical examples of mouse drifting is to support the reading of a piece of code: Developers support the reading activity by slowly moving the mouse pointer as a guide to read and understand the code.

There is significant variability of program understanding among developers. In our dataset, it ranges from 41% for D12 to 94% for D14; however, in both cases, we do not have many recorded sessions, so they are probably simply outliers for specific tasks that require respectively a minimal or a maximal amount of understanding. In case of D12, most of the remaining time is actually spent on fiddling with the UI (around 32%) and being outside the IDE (ca. 14%), which suggests she is not concentrated on the task at hand.

Inspection also varies between developers: D2, in particular, spends around 3% of his time in inspecting objects at runtime, while on average the inspection is below 1%. Similar higher time spent on inspection can be seen on D16. Higher variety is present on the usage of mouse drifting. D2 and D12 spend much of their time fiddling with the mouse.

Our data also provides insights on other mechanics of program understanding, and in particular on how understanding is distributed among activities. On average, basic inter-activity understanding amounts to 10 percent points more than intra-activities understanding. It is evident that base understanding is prevalent inside activities, that is, inside conceptually related

sequences of keyboard or mouse sprees. In other words, the process of program understanding is not really an activity per se, but it is interleaved with other activities like editing. Again, there is significant variability between developers. The process of base understanding for D1, for example, is almost equally divided between intra- and inter-activity understanding. For D5 and D14, there is significantly more inter-activity understanding, which probably means that the activities of these developers are contiguous, and less affected by interruptions.

B. Time Spent Outside the IDE

Switching the context between the IDE and other applications (*i.e.,* reading e-mails) impacts the focus, flow, and productivity of a developer [20], [23]. A developer who spends time outside the IDE, once back in the IDE, is likely to need time to "recover": Her sessions are likely to exhibit more time spent in program understanding. Another factor that may impact the duration of understanding time is the number of such breaks: A session may end up in a "fragmented" state where the flow is frequently interrupted by context switches that lead to spending time outside the IDE, and it might have an impact on the time spent in program understanding. The number of context switches might also have an impact on the time spent in fiddling with the UI of the IDE. After a context switch, it is likely that a developer needs to re-arrange her environment to "recover" the context inside the IDE. This is what we call UI time.

To investigate these conjectures, we analyze the correlation between the *time* spent outside the IDE (*i.e.,* DOI) and the duration of understanding for each session. Another correlation involves the *number* of times the developer goes outside the IDE (*i.e.,* OI Events) and the duration of understanding. The last analysis involves the correlation between the number of OI Events and the time spent in pure UI interactions. We use the Pearson Correlation Coefficient (PCC) to determine the

TABLE V

CORRELATION BETWEEN NUMBER OF IDE EVENTS (OI) AND TIME SPENT OUTSIDE THE IDE (DOI) WITH COMPREHENSION TIME (UNDERSTANDING)

Dev.	Sess.	OI Events	Avg. OI Events	PCC OI Events vs. Understanding	p-value	PCC DOI vs. Understanding	p-value
D1	407	2,101	5.16	0.72	2.20×10^{-16}	0.66	2.20×10^{-16}
D2	136	989	7.27	0.76	2.20×10^{-16}	0.40	1.83×10^{-6}
D3	76	154	2.03	0.47	2.27×10^{-5}	0.00	9.53×10^{-1}
D4	32	91	2.84	0.91	7.07×10^{-13}	0.80	2.89×10^{-8}
D5	19	36	1.89	0.82	2.09×10^{-5}	0.61	6.03×10^{-3}
D7	11	73	6.64	0.74	9.82×10^{-3}	0.64	3.55×10^{-2}

linear correlations using the R[4] tool. Our analyses involve the 704 sessions that have time spent outside the IDE.

Time Spent Outside the IDE vs. Understanding Time. The PCC is 0.46 and is thus a weak linear correlation; using the corresponding statistical test [24], we reject the hypothesis that values are not correlated at 95% confidence level with the lowest possible p-value returned by R (2.20×10^{-16}).

Number of OI Events vs. Duration of Understanding. The PCC is 0.63, and the statistical test at confidence level of 95% is in favor of rejecting the null hypothesis of non-correlation with a p-value similar to the previous test. Even if *correlation is not causation*, these findings are consistent with the hypothesis that the number of time intervals spent outside the IDE increases the understanding time.

Number of OI Events vs. Duration of UI Time. The PCC is 0.65, and the statistical test at confidence level of 95% is in favor of rejecting the null hypothesis of non-correlation, with the same p-value. These results support the fact that the more context switches happen in a session, the more a developer fiddles with the UI of the IDE to recover her focus.

Table V shows the first two correlation analyses discussed above for each developer with at least 10 sessions with at least one time interval spent outside the IDE in a session (*i.e.,* D6 is not in the table because she has zero OI Events). At first sight, there is evidence of diverse developer behavior in terms of the number of time intervals spent outside of the IDE per session, which varies from a minimum of 2.37 to a maximum of 8.50 intervals per session. Results for correlation are also diverse: All p-values are very low (*i.e.,* below 4×10^{-2}) and suggest rejection of the hypothesis of non correlation. The exception is D3, for whom the duration of the time spent outside the IDE is not correlated with the duration of program understanding. However, there is a mild but significant correlation with the number of intervals spent outside the IDE. This likely means that it does not matter how much time she spent outside the IDE in total, but just the number of times her sessions are fragmented. D2 shows a similar behavior: she seems more affected by the number of times she exits the IDE rather than time spent outside. D4 is also interesting: Her sessions are not very fragmented; however, she is the developer mostly affected by the time spent outside the IDE, with strong correlation with both duration and number of intervals spent outside the IDE.

C. The Impact of the UI, Navigation, and Editing

Our data shows that on average around 14% of the time of developers is spent on rearranging the UI of the IDE, that is, resizing or dragging windows. Different experience may explain variability when aggregating data per developer. D14, for example, rearranges windows only for 4% of the time, while D2 and D12 spend around 30% of their time for this task. This might indicate that they often end up in chaotic environments [12] that need to be reordered or restructured.

Our data shows that pure navigation between entities occupies around 3.6% of the time of developers, and that browsing occupies most of the time spent in navigation. Only 7 developers used searching. Among the people who use search, the time spent on these activities low. Presence of editing activities and editing time is also quite variable: it ranges from 1.24% for D14 to 10.67% for D7. Moreover, D7 and D16 correlate their high time spent in editing with very short time spent outside the IDE, which is probably a sign of highly focused development sessions.

D. Wrapping Up

Our analysis indicates that the role of program understanding has not been overrated, but on the contrary is has been significantly underestimated. We believe that our analysis corroborates the importance of research in approaches and tools that deal with program comprehension and reverse engineering. This analysis also provides evidence that fragmentation with time spent outside the IDE influences the time spent on comprehension. To draw a simplistic conclusion: Developers should not be interrupted during programming activities. Last, the non-trivial amount of time spent on fiddling with the UI of the IDE highlights the need for research on novel user interface and interaction paradigms for the IDE, a still underexplored field with some notable exceptions, *e.g.,* [25] [26] [27] [28].

Advocatus Diaboli. The positive correlation is consistent with the fact that the time outside the IDE influences the total program understanding time, but obviously the dynamics of development are complex, and other factors also influence the duration of program understanding. For example, developer experience on the task at hand may strongly decrease the impact of session fragmentation on program understanding. Moreover, the extent and impact of fragmentation depends on the specific activity performed outside the IDE – it is likely that a chat on an unrelated matter or browsing a social network's feed may impact more than reading a related STACK OVERFLOW post.

---

[4]See http://www.r-project.org

V. RELATED WORK

The related work can be classified in approaches which (1) study the role of program understanding, and (2) use interaction data as the empirical basis for their research.

Role of Program Understanding. Zelkowitz *et al.* estimated that program comprehension takes more than half the time spent on maintenance [9]. In turn, maintenance accounts for 55 to 95% of the total costs of a software system [29], thus the weight of program comprehension was estimated to be between 30 to 50%, which is also in the range of the claim of Fjeldstad and Hamlen, who asserted that comprehension occupies half of the time of developers [8]. Considering the processes used by developers to understand source code, Ko *et al.* claimed that understanding is also achieved by navigating source code fragments [15], and that such process occupies around 35% of the total development time.

Interaction Data. Researchers proposed a number of approaches and tools to track the way developers work inside the IDE and leveraged this data for different purposes. Fritz *et al.* used data from psycho-physiological sensors to assess the nominal difficulty of code comprehension tasks [30]. They combined data coming from an eye-tracker, an electrodermal activity sensor, and an electroencephalograph to predict task difficulty. Compared to these approaches, recording IDE events with DFLOW is less invasive from the point of view of the developer. FLUORITE is a tool proposed by Yoon and Myers that records low-level events in the Eclipse IDE [31]. 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 [32]. Singer *et al.* proposed NAVTRACKS, a methodology backed up with a tool that records navigation histories of developers and leverages them to support browsing through software [22]. Kersten *et al.* proposed MYLAR, an approach that monitors the programmer activities to identify the program entities which are more important [13]. Murphy *et al.* used the data collected by MYLAR to study how Java developers use the Eclipse IDE [14], finding that developers use most of the Eclipse perspectives while developing, and that keyboard shortcuts are a frequently used alternative to faster reach some IDE features. Some approaches also tried to automatically reconstruct activities in development sessions. Coman and Sillitti collected low-level events and presented a technique to split sessions into task-related sub-sessions [33]. Researchers also studied the impact of work fragmentation [34], [35], [23]. Sanchez *et al.*, for example, observed how the productivity of developers decreases with the increase of the number and the duration of interruptions in the developer workflow [23]. These finding are consistent with the results of our correlation study described in Section IV-B. Interaction data has also been used for change prediction. Kobayashi *et al.* presented PLOG, a tool to capture interaction histories inside the Eclipse IDE. The recorded data was used to devise a prediction model for change propagation based on interaction histories [36]. Robbes *et al.* also performed similar work [37].

Summing up. We found a dichotomy between claims regarding the importance of program comprehension and the fact that they are often not backed up by quantitative evidence. With this paper we 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.

VI. CONCLUSIONS

We presented an in-depth analysis of how developers spend their time, based on a corpus of fine-grained IDE interaction data. We proposed a model that aggregates low-level events into sprees and those into activities. In the end, 31,609 development activities originated from the 5 million events recorded with DFLOW. We measured the time spent by developers in 5 distinct and disjunct categories: *understanding*, *navigation*, *editing*, *UI interactions*, and *time spent outside of the IDE.*

Our results reinforce common claims about the role of program understanding: On average, developers spend 70% of their time performing program comprehension. In addition, developers spend 14% of their time in fiddling with the UI of the IDE, which calls for novel and more efficient user interfaces. The time spent for editing and navigating source code is respectively 5% and 4%. The large part of development is occupied by mental processes (*i.e.,* understanding) and, in the remaining time, a developer has to deal with inefficient user interfaces to read, write, and browse source code. We believe that future IDEs should tackle these problems to enable developers to focus on the tangible part of development: writing source code. We also observed that the number of context-switches (*i.e.,* times the IDE loses focus in a development session) and their duration, is linearly correlated with both the understanding time and the time spent in fiddling with the UI. This corroborates results obtained previously by researchers like LaToza *et al.* [20]. Finally, the time spent outside of the IDE (ca. 8%), the frequency of such interruptions, and their subsequent negative impact on understanding, points out that developers are exposed probably too often to distractions.

Overall, we believe this paper makes a number of contributions to the state of the art: First, with respect to the field of program comprehension, it confirms what has long been an accepted, but never validated, ground truth: program comprehension is the activity with which developers spend the vast majority of their time. The motionless staring at the screen is thus legitimized. Second, it points out that IDEs are far from perfect when it comes to the way their user interfaces are built. We believe this calls for research in novel approaches and metaphors, which so far still represent a niche research area. Third, it confirms that like many other modern workers, software developers are exposed to frequent interruptions with negative consequences. To end on a light note: When developers stare at their screens without any movement: *Don't worry, they're ok, leave them alone.*

REFERENCES

[1] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Manage. Sci.*, vol. 44, no. 4, pp. 433–450, Apr. 1998.

[2] G. Weinberg, *The Psychology of Computer Programming*, silver anniversary ed. Dorset House, 1998.

[3] M. Lehman, "Laws of software evolution revisited," in *Proceedings of EWSPT 1996 (5th European Workshop on Software Process Technology)*. Springer, 1996, pp. 108–124.

[4] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of CASCON 1997 (Conference of the Centre for Advanced Studies on Collaborative Research)*, 1997, pp. 21–36.

[5] M. Robillard, W. Coelho, and G. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE TSE 2004 (Transactions on Software Engineering)*, vol. 30, no. 12, pp. 889–903, 2004.

[6] T. Frey, M. Gelhausen, and G. Saake, "Categorization of concerns: A categorical program comprehension model," in *Proceedings of PLATEAU 2011 (3rd Workshop on Evaluation and Usability of Programming Languages and Tools)*. ACM, 2011, pp. 73–82.

[7] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[8] R. K. Fjeldstad and W. T. Hamlen, "Application Program Maintenance Study: Report to Our Respondents," in *Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, Eds. IEEE, 1982, pp. 13–30.

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

[10] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *Proceedings of IWPSE 2005 (8th International Workshop on Principles on Software Evolution)*. IEEE, 2005, pp. 113–122.

[11] O. Greevy, T. Gîrba, and S. Ducasse, "How developers develop features," in *Proceedings of CSMR 2007 (11th European Conference on Software Maintenance and Reengineering)*. IEEE, 2007, pp. 265–274.

[12] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in IDEs," in *Proceedings of WCRE 2009 (16th Working Conference on Reverse Engineering)*, 2009, pp. 237–246.

[13] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of AOSD 2005 (4th International Conference on Aspect-Oriented Software Development)*. IEEE, 2005, pp. 159–168.

[14] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[15] 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)*, vol. 32, no. 12, pp. 971–987, 2006.

[16] R. Minelli and M. Lanza, "Visualizing the workflow of developers," in *Proceedings of VISSOFT 2013 (1st IEEE Working Conference on Software Visualization)*, 2013, pp. 1–4.

[17] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Proceedings of VISSOFT 2014 (2nd IEEE Working Conference on Software Visualization)*, 2014, pp. 147–156.

[18] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *Proceedings of QSIC 2014 (14th International Conference on Quality Software)*, 2014, pp. 276–285.

[19] F. Cirillo, *The Pomodoro Technique*. FC Garage, 2013.

[20] T. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*. ACM/IEEE, 2006, pp. 492–501.

[21] S. Pinker, *How the Mind Works*. W. W. Norton, 1997.

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

[23] H. Sanchez, R. Robbes, and V. M. Gonzalez, "An empirical study of work fragmentation in software evolution tasks," in *Proceedings of SANER 2015 (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*. IEEE, 2015, pp. 251–260.

[24] M. Triola, *Elementary Statistics*, 10th ed. Addison-Wesley, 2006.

[25] A. Bragdon, S. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. LaViola, "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*. ACM/IEEE, 2010, pp. 455–464.

[26] R. DeLine and K. Rowan, "Code canvas: Zooming towards better development environments," in *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering) – ERA*. ACM/IEEE, 2010.

[27] F. Olivero, "Object-focused environments revisited," Ph.D. dissertation, University of Lugano, Switzerland, Apr. 2013.

[28] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes, "Enabling program comprehension through a visual object-focused development environment," in *Proceedings of VL/HCC '11 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, 2011, pp. 127–134.

[29] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, May 2000.

[30] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*. ACM/IEEE, 2014.

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

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

[33] I. Coman and A. Sillitti, "Automated identification of tasks in development sessions," in *Proceedings of ICPC 2008 (16th International Conference on Program Comprehension)*. IEEE, 2008, pp. 212–217.

[34] L. Zou and M. Godfrey, "An industrial case study of program artifacts viewed during maintenance tasks," in *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, 2006, pp. 71–82.

[35] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of ICSE 2007 (29nd International Conference on Software Engineering)*. IEEE Computer Society, 2007.

[36] T. Kobayashi, N. Kato, and K. Agusa, "Interaction histories mining for software change guide," in *Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, 2012, pp. 73–77.

[37] 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)*. IEEE CS Press, 2010, pp. 161 – 170.