The Journal of Systems and Software 132 (2017) 85–97

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Patterns of developers behaviour: A 1000-hour industrial study

Saulius Astromskis[a], Gabriele Bavota[b,*], Andrea Janes[a], Barbara Russo[a], Massimiliano Di Penta[c]

[a] *Free University of Bozen-Bolzano, Bolzano, Italy*
[b] *Università della Svizzera italiana (USI), Lugano, Switzerland*
[c] *University of Sannio, Benevento, Italy*

ARTICLE INFO

Article history:
Received 8 October 2015
Revised 8 June 2017
Accepted 27 June 2017
Available online 28 June 2017

Keywords:
Monitoring developers' activities
Case study

ABSTRACT

Monitoring developers' activity in the Integrated Development Environment (IDE) and, in general, in their working environment, can be useful to provide context to recommender systems, and, in perspective, to develop smarter IDEs. This paper reports results of a long (about 1000 h) observational study conducted in an industrial environment, in which we captured developers' interaction with the IDE, with various applications available in their workstation, and related them with activities performed on source code files. Specifically, the study involved six developers working on three software systems and investigated (i) how much time developers spent on various activities and how they shift from one activity to another (ii) how developers navigate through the software architecture during their task, and (iii) how the complexity and readability of source code may trigger further actions, such as requests for help or browsing/changing other files. Results of our study suggest that: (i) not surprisingly, developers spend most or their time ($\sim$ 61%) in development activities while the usage of online help is limited (2%) but intensive in specific development sessions; (ii) developers often execute the system under development after working on code, likely to verify the effect of applied changes on the system's behaviour; (iii) while working on files having a high complexity, developers tend to more frequently execute the system as well as to use more online help websites.

1. Introduction

While performing maintenance tasks, developers interact with their IDE and other applications following specific activity patterns driven by the modification they want to perform. For example, when they need to modify a source code file, they might want to look at other files to understand the context surrounding the change they have in mind or the impact of the change on those files. Another example is when developers navigate across different components or layers of the software architecture to update the business logic or the user interface. Other activities may make them use applications outside the development platform. For example, developers might need to browse internet pages to search or provide help in on-line forums (*e.g.*, Stack Overflow[1]).

In past and recent years, several authors have observed how developers perform comprehension or development activities (von Mayrhauser and Vans, 1994; Singer et al., 1997; Storey et al., 2000; Robillard et al., 2004; de Alwis and Murphy, 2006; Ko et al., 2006; Sillito et al., 2008; Ying and Robillard, 2011; Bavota et al., 2013; Soh et al., 2013; Negara et al., 2013; Fuchs et al., 2014). These studies allowed to learn precious insights about how developers behave during their task, what they need to know when performing them, what tool they use, and what their navigational patterns are. Such studies were performed *in vitro* (Fronza et al., 2011; Storey et al., 2000; Robillard et al., 2004; Ko et al., 2006; Bavota et al., 2013; Minelli et al., 2014c; Fuchs et al., 2014), by mining software repositories (Soh et al., 2013), by using reading-writing task monitoring from the IDE (Ying and Robillard, 2011), and observing, at a fine-grained level of detail, real-world development tasks (Janes et al., 2008; Johnson, 2001).

In general, most of the studies have been conducted over a limited interval of time (von Mayrhauser and Vans, 1994; de Alwis and Murphy, 2006)[2]; when longer projects were observed, the studies were mainly conducted from a qualitative standpoint by interviewing developers (Singer et al., 1997; Sillito et al., 2008), or by

* Corresponding author.
E-mail addresses: gabriele.bavota@usi.ch, gabriele.bavota@gmail.com (G. Bavota).
[1] http://stackoverflow.com.

[2] These studies lasted less than 30 h, compared to the over 1000 collected in our work.

http://dx.doi.org/10.1016/j.jss.2017.06.072
0164-1212/© 2017 Elsevier Inc. All rights reserved.

only collecting quantitative effort data (Astromskis et al., 2014), by focusing on specific aspects of the developers' interaction, *e.g.,* how IDE features are used (Murphy et al., 2006), tools are exploited in pair programming (Fronza et al., 2011), or specific activities such as refactorings are performed (Negara et al., 2013).

Paper contribution. Motivated by such literature, we conducted a study in which we observed how professional developers performed their activities during a time window of about 1000 h. The study has been run at a company developing real-time software for controlling mechanical devices used in the manufacturing and agriculture domains.[3] The study involved six professional developers working on three software projects. Using the PROM tool,[4] we were able to capture all interaction events that developers performed with their IDE and with any other application installed in their workstation. For example, we were able to gather information about which source code class a developer is working on right before she goes on a web site and reads about a discussion on how to solve the problem related to her code. Using the available information, we contributed to enhance knowledge on behaviour of professional developers during software maintenance and, in particular, we investigate (i) the amount of time they spend in doing different kinds of activities (*e.g.,* coding/browsing source code, using utility tools, communicating with teammates, searching for help on the Internet, *etc.*), and (ii) their transitions across the different kinds of activities and the software architecture when coding.

Relevance of the study. A deep understanding of developers' navigation/editing patterns in the source code and in the working environment can be of a paramount importance for various purposes. First, it can be useful to monitor in detail the development process in an organisation and more in particular in specific projects to understand where there can be room for improvement in the software documentation or in the set of resources a developer has available for performing certain tasks. Also, some specific navigation patterns can reveal issues in the software structure or readability, and therefore suggest re-documentation and refactoring activities. This holds for both navigation patterns across activities (*e.g.,* observing that when working on a specific code file developers tend to ask help to their teammates might indicate the need for refactoring/re-documenting the file) as well as across architectural layers (*e.g.,* some patterns might indicate the violation of the intended architecture). Last, but not least, understanding what triggers certain developers' activities can be useful to develop better IDEs and recommenders, able to guide developers in their code browsing and providing help/support considering the current context. Previous work has shown that contextual information is particularly useful to filter recommendations to developers (Fritz et al., 2007; Kersten and Murphy, 2006; Murphy et al., 2006), and that embedding online help in the IDE can improve productivity and code quality (Rahman et al., 2014; Holmes et al., 2005; Ponzanelli et al., 2014). A deeper knowledge of typical navigation patterns mined from past developers' activity can have great potential to further refine recommendations and better guide developers on what to do in certain contexts.

Main findings. While coding, developers tend frequently to communicate with other developers and/or using some kind of utilities supporting their activity (*e.g.,* tools for remotely controlling a server). While they rarely look for online help (this happens in 6% of the working sessions in our dataset), such activity is particularly intense in specific sessions, in which developers formulate multiple queries on search engines to find what they need and, once found the appropriate resource, start going back and forth between the code and the online help page in the browser. We also found that while working on highly complex/coupled files, developers tend to execute their code more frequently as compared to simpler files, likely to check whether the applied changes introduced errors. This might indicate less confidence of developers on working on highly coupled/complex files.

Paper structure. Section 2 provides the study definition and planning. Study results are reported and discussed in Section 3, while threats to its validity are discussed in Section 4. After a discussion of related work (Section 5), Section 6 concludes the paper and outlines directions for future work.

2. Study design

The *goal* of the study is to gather a deep understanding of developers' navigation patterns in the source code and in the working environment the *purpose* of investigating (i) the amount of time they dedicate to different kinds of activities (*e.g.,* development, looking for online help, *etc.*) and what the transitions between these activities are; (ii) how developers navigate across software architecture layers during their code change activities; and (iii) the influence of specific code characteristics (*i.e.,* size, readability, complexity, and coupling) on the likelihood of transitions between different activities (*e.g.,* is complex code more likely to trigger a "*looking for online help*" activity?).

2.1. Research questions

We aim at answering the following research questions:

- **RQ$_1$:***How much time do developers spend on different kinds of activities and how do they transit between them?* In our settings, we were able to collect six different types of developers' activities: (i) working on a source code file (*i.e.,* a code file is read and/or modified), (ii) executing the system under development, (iii) using utilities (*e.g.,* tools for remotely controlling a server, file compression tools, *etc.*), (iv) using external programs unrelated with code development (*e.g.,* interacting with a music player or playing video games), (v) communicating with other developers, and (vi) looking for online help (*e.g.,* accessing Stack Overflow). A better understanding of the time spent in such different activities, and how developers switch from one activity to the other, can help project managers to develop strategies aimed at optimising the maintenance process, *e.g.,* by developing tools and methods that provide appropriate support or identifying activities that can be avoided (Janes and Succi, 2014).
- **RQ$_2$:***How do developers navigate the system architecture during code change activities?* This research question analyses the navigational patterns among the architectural layers followed by developers while reading and/or modifying code files. All systems involved in our study are three-tier applications consisting of presentation, application logic, and data layer. Architectural layers may represent contextual boundaries for developers' activities. Activities in the presentation layer (*e.g.,* testing event triggers from the GUI) may be significantly different than the ones performed in the business logic layer (*e.g.,* testing algorithms) as they require different knowledge. In addition, changes in one layer can trigger changes in another (*e.g.,* changes to the GUI may require developers to browse the business logic or model). Stationariness (*i.e.,* focusing on a specific layer) or transition across such boundaries can characterise different types of development and maintenance processes (*e.g.,* changes to the GUI's design are expected to only affect the presentation layer, while the implementation of a new feature

[3] Due to confidentiality reasons, we cannot reveal the name of the company nor the specific applications.

[4] Developed at the Free University of Bozen-Bolzano (Astromskis et al., 2014).

likely results in traversing architectural layers) and navigational patterns can help construct models of development or maintenance based on such architectural boundaries.

- **RQ$_3$:***How does the internal quality of code components influence the likelihood of transitioning toward specific activities?* Size, readability, complexity, and coupling of a code component may impact the likelihood of transitioning toward a specific activity while working on a code component. For example, a developer working on a very complex code file might need to frequently search online help.

2.2. Context selection and data collection procedure

Our study has been conducted in 2013 in an industrial context, and in particular in a software company developing real-time software aimed at controlling mechanical devices used in the manufacturing and agriculture domains. We studied the activities of six developers working on the development and evolution of three C++ software systems (all related to the company's core business), having a size of 209, 157, and 88 classes, and 74, 34, and 12 KLOC, respectively. At the time of the study, these were the only three projects the involved developers' were working on.

The three projects subject of this study are embedded systems processing sensor data in real time to control industrial manufacturing processes. The requirement to process data in real time, led to the decision to use C++ for the development. The presentation layer of the embedded software consists of a simple user interface, which can be operated using a touch display. The sensors collect data about various properties of the monitored objects. The main programming task in this case is to develop the image and feature recognition components and on storing the identified properties into a file. The second part of the produced software is a Microsoft Windows application that can connect to the device running the embedded software component and visualises the collected data superimposing it onto the scanned object providing an augmented reality view. Using this visualisation, the operator is then able to define the next production steps.

The development approach of the company did not follow a specific method with steps or phases. Development tasks were distributed among the different developers by a project manager. The project manager of all three systems was the software developer with the longest experience in the company and the most product-specific knowledge. He was responsible for the architecture of the developed tools, the long-term planning of the development, and the communication with the management.

Each developer worked on different products alternating periods of development with external activities like customer relation or software deployment.

We employed the Prom tool[5] to monitor their interactions with the IDE[6] and with any other application (*e.g.*, web-browsers, word processors, mail clients, music players, *etc.*) installed in their workstation in a six-month time span.

Prom consists of various components that capture the interactions of a user (in our case a developer) with the programs she uses. Interactions are stored in logs as events. The tool includes *specific* and *general* components. Specific components capture the interactions with specific applications of interest at a high detail level. For example, Prom includes a specific component for Microsoft Visual Studio[7] able to log on which part of the code (the method, class, file, and project) a user works at a given point in time. Other specific components for the Microsoft Office suite[8] log the currently focused document (the document name and properties, file name) on which a user is working. General components are background processes of the operating system and log the currently focused application by reporting the process name and the title of the window opened within the application (if any).

Overall, for each event the tool allowed us to collect the following information:

- *The workstation id on which the interaction event was captured*. Each of the workstations on which we installed Prom was univocally identified by an alphanumeric id. Note that, while we are able to univocally identify each workstation, we do not have information on the specific developer who is using a workstation in a specific moment. This limitation is the result of privacy constraints set by the company when agreeing on hosting our study. Nonetheless, for the period we monitor the activities, one single computer was in fact shared among developers: Two developers alternate themselves in the use of the computer (*e.g.,* one developer working in the morning and one in the afternoon or in different days). This means that the two developers did not use the workstation at the same time. The other workstations were instead used exclusively by a single developer during the period we monitored the activities.
- *The time stamp of the event,* expressed with a precision of milliseconds.
- *The name of the software application on which the event has been captured*. As said before, this could be any software application installed on the monitored workstation.
- *The name of the "object" involved in the event*. The object (if any) opened with the application: it can be the path of a source code file if the application is an IDE, the path of a textual document if it is a word processor, the title of a visited webpage if it is a Web-browser, the email subject if it is a mail client *etc.*

Table 1
Examples of events captured by Prom.

Id	Timestamp	Application	Object
3a04	806733100	Visual Studio	C:/workspace/Main.cpp
3a04	807183800	Chrome	C++ Overloading - StackOverflow
3a04	821019090	Skype	Mario Rossi - Skype

Table 1 reports three examples of events as captured by Prom.

We also had access to the SVN repositories on which the three object systems were hosted. This gave us access to the source code components on which the developer worked.

2.3. Activities and working sessions

To address **RQ$_1$**, we started defining an *activity* as an event in the logs and the activity duration as the time difference between its timestamp and the timestamp of the subsequent event. Then, we removed all activities shorter than three seconds. By observing developers in action, we assumed that such short activities can be mainly due to non-relevant actions like scrolling across various windows. Then, we clustered logs into working sessions. We defined a *working session* as a sequence of events captured on the same workstation in which two subsequent events occur in less than λ minutes. As such, a new working session starts with an event that occurs more than λ minutes after the preceding event. With this heuristic, we aim at (i) associating a working session to a single task performed by a single developer and (ii) mitigate the risk of assigning a single task to multiple developers (*e.g.,* developers that use the same workstation during the day) or connecting multiple unrelated or non-development activities to a single task (*e.g.,* developers' activities interleaved by long meetings).

[5] Prom was developed by two of the authors (Astromskis et al., 2014).
[6] All involved developers used Microsoft Visual Studio as IDE.
[7] https://www.visualstudio.com/.
[8] https://products.office.com.

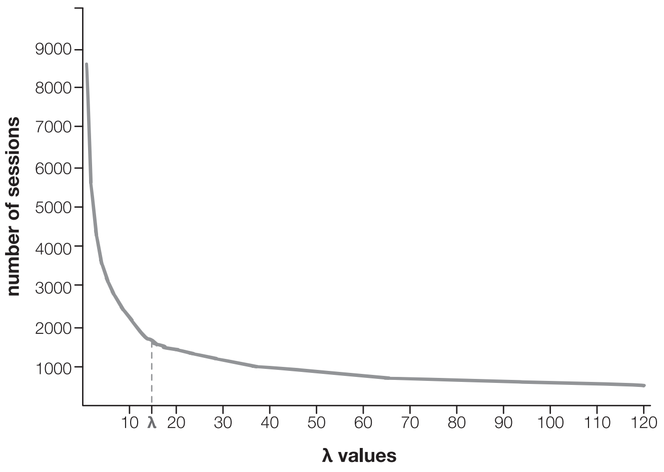

Fig. 1. Sensitivity analysis: changes in the number of sessions as the result of changes in the splitting threshold (*i.e.*, λ).

Defining the value for λ is not trivial, and almost any choice is likely to introduce imprecisions in the data. Also, to the best of our knowledge, there is not agreement in the literature on the time limit (λ in our case) separating working sessions. For this reason, we deeply looked into the data in order to identify a suitable λ value. In particular, we analysed the sensitivity with which the number of working sessions (dependent variable) changes in response to changes to λ (independent variable). We varied λ between 1 and 120 min at steps of 1 minute (thus experimenting with 120 different values). The plot depicting the obtained data is shown in the Fig. 1, with the number of working sessions on the *y*-axis and the λ values on the *x*-axis. The number of sessions strongly drops when increasing λ from 1 to 10 min, indicating a very low stability of the number of sessions in this interval. In particular, at each one unit threshold increase from 1 to 10, the number of sessions decreases, on average, of 22%. Selecting any of these thresholds would clearly strongly influence the results, given the very high instability in the number of sessions. For this reason, we selected as a suitable threshold λ_i for our dataset the lowest threshold ensuring a +/−5% stability with respect to its neighbours. In other words, the number of sessions obtained by applying λ_i as splitting threshold does not change of more than 5% with respect to λ_{i-1} and λ_{i+1}. This ensures stability in the data and a limited impact of the threshold's choice on the obtained results. In our case, the first λ_i value that satisfies this condition is 14 min. This is the value we used to split developers' activities into working sessions.

2.3.1. Activity classification

We then classified the activities performed by the developers into the six types reported in Section 2.1. Worth noticing that such types have been defined (i) by observing the behaviour of developers in the company, (ii) being as much descriptive as possible while considering the limitations imposed by the events captured with the Prom tool, and (iii) by avoiding to speculate on activities, such as "testing", that have a clear definition in the literature and cannot be precisely captured with the events we have available.

1. *Code-related activities (C)*. As for similar automated tools, Prom is not able to report if a file opened in an application is modified or just read, without directly interrogating developers. As such, we do not discriminate between "writing" and "reading" code activities.
2. *Executing the system under development (X)*. An activity is classified as such if the developer runs the system she is developing. While the running of the system is likely due to testing purposes, we do not speculate by referring to such operations as "testing activities", since testing is a well-defined and formal process.
3. *Using utilities (U)*. This type includes all activities involving an application related to the development process (*e.g.*, tools for remotely controlling a server, file compression tools, image processing/manipulation tools, *etc.*). We manually identified them from the Prom's log. In our case, developers used rather frequently image processing tools while developing their software. This was due to the fact that many products that the company produces are based on image processing technologies, from general image acquisition to the recognition of objects and properties of objects, *e.g.*, the colour and properties a surface. The data that image processing components collect are used to control industrial manufacturing processes, *e.g.*, to decide which production step to take, depending on the determined quality of the surface of the recognised item. Developers often use image processing programs to manually verify the correct functioning of one of the tools under development.
4. *Using external programs (E)*. This type refers to all activities involving an application unrelated to development activities. We manually identified them from the Prom's log. For example, this category includes all browser-based activities not falling in the H category.
5. *Communicating with other developers (M)*. An activity is classified as such if it (i) involves a mail client (*e.g.*, Outlook), a messaging program (*e.g.*, Skype), or a browser window opened on a mail Web-client (*e.g.*, GMail), and (ii) reports the name of one of the developers or its email address (as mined from the SVN change log) in the string representing the object captured by Prom. For example, the last row of Table 1 shows a Skype chat with Mario Rossi. Such an event is classified in this category if Mario Rossi is the name of a developer. Otherwise, the activity is classified as U. We extracted all applications acting as a mail client, messaging program, and mail Web-client by analysing manually the complete set of 371 applications present in the Prom's log.
6. *Looking for online help (H)*. An activity is classified as such if it is performed in a browser and has as object a window title indicating: (i) the browsing of a Question & Answer website (*e.g.*, Stack Overflow), or (ii) a search-engine run with a query[9] clearly referring to the look for online help (*e.g.*, "how to use"). To manually tagged browsing activities as belonging to the H group. In particular, we started by considering all 14,766 browsing activities in our data. Then, we filtered out those lasting less than three seconds as well as those clearly do not related to online help (*e.g.*, those related to web addresses of online newspapers). We defined the set of browsing activities to exclude by manually looking into the set of visited websites. After this automatic filtering, we obtained 2098 browsing activities possibly related to searches for online help. One of the authors manually inspected these 2098 activities tagging the ones actually related to online help (*i.e.*, 327).

Each working session can be represented as a sequence of type labels with repetition. For example, a sequence CHCHC represents a set of five activities "coding - online help - coding - online help - coding" performed by a developer in a session. Each activity in a session has a its own duration.

Overall, we analysed 1038 working sessions for a total of 1008 h of developers' activity (*i.e.*, on average, each working session lasted for 59 min). These sessions include 86,835 activities distributed as reported in Table 2. Considering that (i) two of the six developers were part-time, (ii) the six-months monitoring period included

[9] The formulated query is reported as part of the window title.

Table 2
Descriptive statistics per activity type.

	Code-related	e**X**ecuting the system	co**M**municating with developers	online **H**elp	using **U**tilities	**E**xternal programs
Total number of activities	46,890	16,460	7531	327	14,868	759
Avg. activities duration (s)	51	20	57	27	30	33
Number of sessions with at least one instance of activity	1038	497	703	63	647	107
Percentage of sessions with at least one instance of activity	100%	49%	68%	6%	62%	10%
Avg. activities per session	45.10	15.87	7.25	0.30	14.26	0.73
Avg. activities per session (excluding sessions with zero instances)	45.10	33.12	10.71	5.19	22.98	7.09

Table 3
Descriptive statistics per session.

	Mean	Median	St. Dev.	Min	Max
Sessions per workstation	258	213	190	97	508
Sessions per system	424	272	431	89	910
Session length (# activities)	84	51	94	1	649
Session duration (min)	59	41	60	1	538

typical vacation periods in Italy (July and August), and (iii) the developers frequently travel for business reasons, the 1008 h of developers' work is a substantial amount of time.

2.4. Architecture and its layers

To answer **RQ$_2$**, we identified which class belonged to which architectural layer. Following the naming convention for namespaces in C++, (*e.g.,* a “GUI” namespace indicates the classes belonging to the presentation layer) and manually looking into every class, one of the authors assigned each class to one of the three layers: presentation (P), application logic (A), or data (D) layer. This information, combined with the developers activity monitored by using PROM, allows us to analyse how developers navigate the system architecture during code change activities.

Each working session can be represented as a sequence of layer labels with repetition. For example, a sequence PAPAD would represent the five file changes at “presentation - application logic - presentation - application logic - data” layer made by a developer during a working session.

2.5. Code quality

To collect data for **RQ$_3$**, we developed a tool that measures size, complexity, coupling, and readability of each source code file (only .cpp files) in the three object systems after each commit in which it has been involved (*i.e.*, we extracted the metrics' values for all files at each system's snapshot in the change history of the three systems).

We measured (i) file size as the total lines of code (LOC) composing it, including comments but excluding blank lines, (ii) file complexity as the sum of the McCabe's cyclomatic complexity (McCabe, 1976) of the methods it contains, (iii) file coupling as the sum of the Coupling Between Objects (Chidamber and Kemerer, 1994) of the classes it contains, and (iv) file readability with the metric proposed by Buse and Weimer (2010). This metric combines a set of low-level code features (*e.g.,* identifiers length, number of loops, *etc.*) and has been shown to be 80% effective in predicting developers' readability judgments. We used the authors' implementation of such a metric.[10] Given a code file, the readability metric takes values between 0 (lowest readability) and 1 (maximum readability).

[10] Available at http://tinyurl.com/kzw43n6.

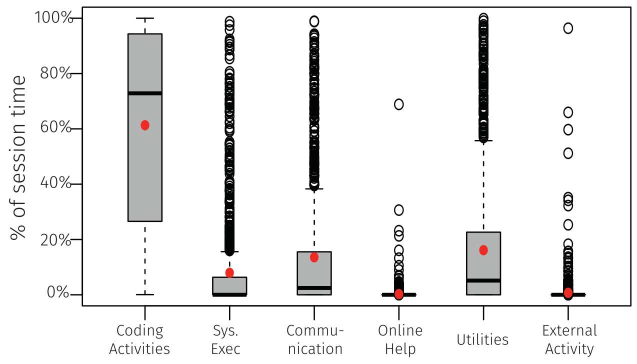

Fig. 2. RQ$_1$: Session time percentage spent for each activity over working sessions..

Table 4
RQ$_1$: Mean transition frequencies between activities.

From/To	C	X	M	H	U	E
C		**33%**	**32%**	1%	**30%**	4%
X	**58%**		21%	1%	17%	3%
M	**45%**	17%		0%	**34%**	4%
H	**70%**	12%	4%		14%	0%
U	**41%**	16%	**33%**	6%		4%
E	**41%**	15%	19%	0%	25%	

Table 5
RQ$_1$: Top activity patterns by percentage of working sessions.

Pattern	#Occ.	#Sessions	%Sessions
MC	2031	458	44%
CX	3364	431	42%
CM	1819	410	39%
UC	1883	407	39%
CXC	976	277	27%
XCX	869	259	25%
CMC	458	186	18%
MX	721	168	16%
XM	700	161	16%
MCM	221	91	9%
CMU	177	80	8%
MCU	167	76	7%

2.6. Method of analysis

The sample for the analysis consists of the set of working sessions of the three object systems. Working sessions can be represented as sequences of activity types or sequence of layer types as described in Sections 2.3 and 2.4. For each file changed in a working session, we extracted quality metrics described in Section 2.5.

To answer **RQ$_1$**, we first analysed the percentage of time spent per activity type in a working session (Fig. 2). Then, we measured the proportion of different activity transitions (Table 4). Additionally, we identified the most frequent activity transition patterns in working sessions (Table 5). This was done by matching regular ex-

Table 6
RQ$_2$: Mean transition frequencies between layers in working sessions.

From/To	P	A	D
P	49%	32%	19%
A	16%	56%	29%
D	12%	42%	46%

Table 7
RQ$_2$: Top architectural patterns by frequency ordered by percentage of sessions (all systems).

Pattern	#Occ.	#Sessions	%Sessions
DA	2784	419	40%
AD	2765	414	40%
ADA	1155	287	28%
DAD	913	247	24%
AP	1305	223	21%
PA	1287	217	21%
DP	616	122	12%
APA	396	117	11%
PAP	375	109	11%
DPD	190	60	6%
(DA)+	123	55	5%

Table 8
RQ$_3$: Spearman correlation between quality metrics and likelihood of transiting toward a specific activity.

Metric	X	M	H
Size	0.07	0.11	0.11
Readability	−0.13	0.01	−0.05
Complexity	**0.27**	0.11	**0.29**
Coupling	**0.21**	**0.14**	0.03

pressions of length varying from two to five[11] onto a file representing activities performed by a developer during a working session (*e.g.,* a sequence CHCHC represents a set of five activities performed by a developer in a session) and determining for each pattern whether it was iterated (*e.g.,* in the sequence CHCHC the pattern CH is iterated multiple times).

We used the same approach to address **RQ$_2$** and extract the navigation patterns followed by developers across the architectural layers in working sessions. In this case, we computed the percentage of navigations between two layers in working sessions (Table 6), and the most frequent navigation patterns in working sessions of length from two to five (Table 7).

Finally, to address **RQ$_3$** we computed the Spearman correlation (Cohen, 1988) between size, readability, coupling, and complexity of a file *F* and the percentage of times that developers transited toward a specific activity while working on *F* (Table 8). It is worth noting that, since *F* can change over time, also the values for its size, readability, coupling, and complexity can change. Having available the value for the four metrics measured after each commit in which *F* has been involved, we considered the median value of each metric as the variable to correlate with the percentage of times that developers transited toward a specific activity while working on *F*. We preferred the median over the mean since the latter is strongly influenced by outliers. Cohen (1988) provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed that there is no correlation when $0 \leq \rho < 0.1$, small correlation when $0.1 \leq \rho < 0.3$, medium correlation when $0.3 \leq \rho < 0.5$, and strong correlation when $0.5 \leq \rho \leq 1$. Similar intervals also apply for negative correlations.

[11] Note that we did not match regular expressions longer than five since, as will be clearer during the results' discussion, long transition patterns are unfrequent.

3. Results discussion

In this section, we discuss the study results and summarize the findings for each research question.

3.1. RQ$_1$: How much time do developers spend on different kinds of activities and how do they transit between them?

Table 2 shows descriptive statistics per activity type and in particular: (i) the total number of activities per type, (ii) the average activities duration (in seconds), (iii) the number of sessions with at least one instance of each activity type, (iv) the percentage of working sessions containing at least one instance of each activity type, (v) the average number of activity type per session, and (vi) the average number of activity type per session when excluding sessions do not having that specific type. In addition, Table 3 reports descriptive statistics related to the characteristics of the considered sessions, *i.e.,* their distribution per workstation and per system as well as their length/duration in terms of number of activities (of all types) and minutes.

Table 2 shows that 100% of working sessions include coding activities (C), which includes both modifying a code file as well as just opening it in the IDE. Activities related to communicating with other developers (M) and using some kind of utilities (U) are highly spread, being present in 68% and 62% of the monitored sessions, respectively. This means that during most of the working session, developers communicate among them and use some kind of utility application. Also, almost half of the sessions include the execution of the system under development (X, 49%).

The limited proportion of online help (H) activities (6% of the working sessions) is balanced with the frequency of such activities in each session (on average, five H activities per session when considering only sessions with at least one instance of H). Altogether, this indicates that, while a low percentage of implementation activities trigger online help, developers generally need multiple iterations (*e.g.,* multiple Google search/Stack Overflow page readings) to locate the solution they need. For example, in one of the sessions we analysed, the developer was looking for online help on how to set the colours of a combobox in C++. This required a Google query (*i.e.,* “set combobox backcolour c++”) and the reading of two different Stack Overflow discussions (*i.e.,* “c++ - Change the item background colour of a combobox in MFC?”, and “c++ - Edit control border and WM_CTLCOLOREDIT?”) multiple times (18) going back and forth from the source code. Manually inspecting our data, we had the perception that the online help was mainly sought when developers had to use a library (in the example related to UI) for which the online help was expected to be available, and the developer had limited experience with that.

The box-plots in Fig. 2 show the percentage of time per working session spent by developers across the six activity types[12] (filled dots indicate mean values). A large proportion of the overall time is dedicated to source code files (C, mean= 61%). Percentage sessions' time across the other activities is distributed, on average, as follows: 8% to execute the system under development (X), 16% to use utilities (U), 2% to look for online help (H), 14% to communicate with other developers (M), and 1% to perform external activities (E).

Table 4 reports the mean percentage of transitions between different types of activities (the most frequent transition(s) from each activity type is highlighted in **bold**). From a C activity, there is almost equal chance of moving toward a X, M, or U activity, while it is very unlikely to reach a H or an E activity. Interesting is especially to see as in 32% of cases developers execute the system

[12] Note that the boxplots refer to all sessions, not just to the ones in which a specific activity type occurred.

under development (X) right after modifying or simply reading code (C). This might indicate that (i) developer verify the impact of their changes on the system's behaviour, and/or (ii) developers use the running system as a support for program comprehension while reading the source code.

The most likely transition (70%) is from browsing online help (H) to working on source code (C). This may indicate that developers have often found online hints that soon want to incorporate into their code. Communication with other developers (M) and use of utilities (U) are also likely to occur in sequence (34% and 33%). This can happen for various reasons. For example, both M and U have a high probability to appear in a working session (Table 2), which increases the probability they occur in sequence. Another reason might be related to the definition of U activities. In our settings, U activities also contain communication exchanges with external people (*i.e.,* communication with non-developers). So if developers tend to dedicate specific periods of the day to communicate with people, the likelihood they communicate with developers and external people in the same working session and even in sequence can be high.

Table 5 shows the number of occurrences of the most frequent patterns we observed (column #Occ), and the number (#Sessions) and percentage (%Sessions) of sessions containing them ordered by such percentage. Results in Table 5 indicate that, developers often transit from code-related to communication activities forward, back, and forward again (see patterns CM, MC, CMC, MCM). Although transitions between execution the system and working on its code (CX, CXC, XCX) are less generally present in working sessions, the number of such transitions is the highest (*e.g.,* we found 3364 instances of the CX pattern). On average, the sessions containing CX have 7 repetition of this pattern, indicating that developers repeatedly work on the code, execute the system, and eventually work on the code again.

3.2. *RQ$_2$: How do developers navigate the system architecture during code change activities?*

Table 6 reports the transition frequencies between architectural layers performed by developers during code change activities while Table 7 shows the most frequent architectural transition patterns we found. The notation (P)+ means that the pattern P was present multiple times in sequence with no interleaving of other activities (*e.g.,* PPP).

When we classified working sessions by the layer in which the first code-related activity occurs, we observed that in 61% of the cases the code-related activity started in A, 25% in D, and 14% in the P layer. Thus, it is unlikely that developers start their code-related activities by navigating the system's architecture from the top layer (*i.e.,* P), while they rather start from within the application logic layer.

The diagonal values in Table 6 additionally show that code-related activities occurring in one layer are likely to trigger other code-related activities in the same layer. This is particularly evident for the application logic layer (56%). Such layer is also a frequent target of code-related activities triggered by any other layer (column A in the table).

To analyze the prevalence of the patterns across the three different systems, we used Kendall's τ correlation after having ranked the architectural patterns in decreasing order of frequency. The pattern rankings highly correlate, and the results indicate a $\tau = 0.86$ (which can be considered a strong correlation) between System1 and System2, a $\tau = 0.72$ (moderate to strong) between System1 and System3, and $\tau = 0.70$ between System2 and System3 (moderate to strong). These values suggest homogeneity of the pattern prevalence in the three systems.

To understand how these transitions occurred in the studied systems, we looked into the events involving the different presentation layers. In particular, we manually analysed both the events recorded by PROM as well as the commits performed by developers in the versioning system to identify the type of activity (*e.g.,* implementation of new feature) developers were carrying out when transitioning across architectural layers. For example, let us assume that we observe developers working on files `A.cpp` and `B.cpp` from System1 in a working session recorded by PROM on January 7th 2014. `A.cpp` and `B.cpp` do not exist in the System1's versioning system until January 8th 2014, when they are added in the context of a commit mentioning in the commit message "*Implemented the new sorting feature*". In such a scenario, we infer that the session recorded by PROM concerns coding activities related to the implementation of a new feature. This analysis has been manually performed to discuss some qualitative examples helping in explaining the transitions we observed. For example, we observed that transitions $P \rightarrow P$ (*i.e.,* subsequent activities performed to different code files in the presentation layer) mainly occur when: (i) a new feature is added to the system, and all the source code files implementing its GUI are created one after the other; (ii) the look-and-feel of the GUI is changed, requiring an update to multiple files in the P layer; and (iii) logically coupled GUIs (*e.g.,* those belonging to a wizard) are updated, to reflect changes applied to the A and/or D layer. The implementation of new features is also the most frequent reason why developers perform writing activities in A and D after changes in P. For instance, during the implementation of a new feature (that we will call *Sort* for confidentiality reasons) the developer performed 17 $P \rightarrow A$ transitions between the class implementing the *Sort* GUI and the class implementing its business logic. Also, the implementation of the *Sort* feature required 11 $P \rightarrow D$ transitions, aimed at the modelling of the information gathered through the GUI forms in the corresponding data objects.

Table 7 shows that very high frequency of transitions between the application logic and the data layer (DA and AD). These transitions are also quite recurring in the sessions containing them (on average 13 transitions per working session). The pattern DA+ further illustrates that developers iteratively move back and forward between the two layers.

Such behaviour can be explained by the peculiar type of software systems (*i.e.,* real-time software controlling mechanical devices) developed by the company. Namely, the data objects of these systems mostly represent real parts of the mechanical devices controlled by algorithms in the application logic layer (*e.g.,* a square board with $n \times n$ wells containing material manipulated by the device). Thus, changes to code components in D (*e.g.,* the value of n is modified) are very likely to result in changes needed in the business logic manipulating such objects. Differently, the classes in the P layer are only linked to D by the data objects modelling the information collected in the GUI's forms that, however, are very few (*i.e.,* no much user interaction is required by this type of real time software).

3.3. *RQ$_3$: How does the internal quality of code components influence the likelihood of transitioning toward specific activities?*

Table 8 reports the Spearman correlation between quality metrics and the likelihood of transiting (percentage of transitions) toward specific activity types. The statistically significant results (*i.e.,* p-value < 0.05), on which we will focus our discussion, are reported in **bold** face. We excluded activities C, E, and U since no significant results were found for any of the investigated metrics.

We first notice that there are no statistically significant correlations for the size and readability metrics. For file complexity, we found two moderate correlations. Namely, when code files are complex, there is a moderate likelihood that developers work on them and, right after, execute their code (0.27) or seek for online help (0.29). Inspecting individual files, we found a very complex file (complexity = 9474) that triggered an execution activity in 76% of the cases. In other words, almost after every activity performed on this file, developers checked the system correctness.

In terms of coupling, the more a file is coupled the more developers execute their systems (0.21) or communicate with other developers (0.14). While this correlations are quite low, this might indicate less confidence of developers on working on high coupled files, *e.g.,* because changes could cause unwanted ripple effects.

4. Threats to validity

Threats to *construct validity* concern the relation between the theory and the observation. In our study, this threat can mainly be due to errors in the construction of the working sessions, the definition of development activities, and the association of code classes to architecture layers of the application. When collecting data to build working sessions, we discarded cases of short-time events (less than three seconds) that are unlikely to be an indication of development activities, but for example of moving the focus among different windows. This might have meant loosing some quick, but valid, activity. We also allowed up to 14 min of elapsing time between two activities in a working session in order to take into account idle time related to non-computer-interaction activities associated with development (*e.g.,* long code readings). Thus, time for non-development activities, like coffee breaks, could be accounted for development. Working sessions are also associated with workstations' and not with developers' ID. Thus, we might not have been able to isolate the single developer's activity in case of concurrent use of more machines by one developer or in case of more developers using the same machine. Nonetheless, according to feedback provided by company's employees, we can assert that a workstation is being used almost always by a single developer.

PROM does only capture the interactions of the developer with the "active window" (*i.e.,* the window having the active focus), even when multiple windows are shown on the screen. This clearly limits our monitoring data. Indeed, if a developer is debugging code in the IDE (active window) while discussing with a colleague on Skype on how to fix a bug (non-active window), we only observe her interaction with the IDE.

Finally, heuristics used for the classifications of developers' activities have been defined by manually analysing the PROM's logs. Although the information included in each event we collected is rich, we cannot exclude the misclassification of some activity types (*e.g.,* a Google query looking for online Help was not recognised as a help (H) activity by our manual inspection, and thus classified as an External activity). Also, the association of code classes to the three-tier architectural layers has been done manually, and it might have implied some misclassification.

Threats to *internal validity* concern any confounding factor that could influence our results. In principle, internal policies of the company, like access to specific resources, or the awareness of employees about the data collection might have had some effects on our results. For instance developers might have not navigated specific resources on the Web knowing that the navigation was being monitored. However, the over 1000 h of monitoring make us confident on the reliability of the observed developers' behaviour.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. Although most of the analyses performed in this paper mainly have an observational nature, we used, where appropriate (**RQ$_3$** in particular), statistical procedures to support our claims.

Threats to *external validity* concern the generalisation of our findings. This study has been conducted with employees of one company, and for this reason the obtained results may not generalise to other companies, which, for example, might have used different policies in accessing applications external to the IDE. Also, as already stressed in our results discussion, some of our findings are possibly dictated by the specific type of software systems developed by the company.

5. Related work

This section describes related work mainly concerning (i) studies investigating the developers' behaviour, and (ii) existing data collection (monitoring) tools.

5.1. Studying developers' behaviour

Roughly speaking, studies on developers' behaviour are divided into two kinds: the ones that collect data by visual observations, eye tracking, survey, and interviews and the ones that install tools in the work station of developers to collect data automatically from the software environment. While our work belongs to the latter category, the majority of the studies performed in the literature pertain to the former.

von Mayrhauser and Vans (1994) published one of the first works on the analysis of developer's behaviour. They visually observed the behaviour of five professional developers in two-hours maintenance sessions. They found that programmers use to switch between different applications as well as different sources of documentation while they try to understand their source code.

Singer et al. (1997) studied the daily activities of developers. They performed three separate analyses on applications' usage by studying single individuals, teams, and the whole company. Interestingly, they found that search tools (*e.g.,* variations of the grep command) are the most frequently used tools by developers. With this study they provided new guidelines for tool designers based on users' cognitive processes and mental models with the emphasis on usability.

Robillard et al. (2004) performed an exploratory study to analyse the factors that contribute to effective program investigation behaviour. They observed that effective developers exploit a methodical approach during program investigation, trying to understand the high-level structure of the system before designing and then implementing the change. In contrast, non-effective developers employ an opportunistic approach, skimming the code components during code understanding (Robillard et al., 2004). In this paper, we found that developers move among different abstraction layers (e.g., between application and data layer) in a more opportunistic way motivated by the type of product developed at the company.

Sillito et al. (2008) performed two qualitative studies aimed at understanding what pieces of information programmers seek when they need to modify code components, how they collect such information and how well today's programming tools help in that process. As output of these studies they proposed a catalog of 44 types of questions programmers ask while coding. These questions have been categorised into four categories expressing the type of information needed to answer the question (Sillito et al., 2008).

Eye tracking systems have been used to investigate the comprehension of UML diagrams (Guéhéneuc, 2006; Yusuf et al., 2007), the effect of the layout on the comprehensibility of software documentation (Sharif and Maletic, 2010), and the effect of design patterns on comprehension (Jeanmart et al., 2009). These studies have the power to collect data at low-level granularity (*e.g.,* identify the specific part of the code observed by the developer in a given moment), but cannot automatically detect interactions among different events generated in applications running on the work station or tracking the developers' activities in parallel.

Our study differs from these studies because, while we can not capture the level of granularity ensured by eye-tracking technologies, we can afford to capture complex interactions with the IDE and with a wide set of software applications (*e.g.,* browsers) developers use everyday. Also, none of the previous studies has been run for a period of time comparable to the study presented in this paper.

Studies based on the automated collection of data by relying on tools installed on the developers' work stations generally focus on IDE/utilities usage during maintenance activities. In this category, it falls research that aims at understanding what tool and source of information developers need in their maintenance activities. In general, data collection in such papers focusses on the sole IDE.

Ko et al. (2006) performed a study in which 31 developers had to implement changes for 70 min to an already existing system, which they have not worked on previously. Authors captured the developers' activities by recording the screen of the workstation, and analysed how did the subjects search for the relevant code, and how the changes were implemented. They observed that developers often start their coding activity by searching (either manually or with a tool) for the code components they need to change. Then, they start modifying the identified code component, by navigating back and forth towards its dependencies. Our study involves a wider context (*i.e.,* we focus on a wider set of development activities), it is performed in an industrial context, and lasted a much longer time.

Soh et al. (2013) analysed how the developers effort is distributed during the maintenance activities, and whether the complexity of the code artefact influences the amount of effort needed to implement or fix it. Moreover, the authors analysed how the effort is distributed during the exploration phase, and find out that on average 62% of files opened during this phase are not significantly related to the final implementation of the task. This issue can be addressed by analysing the behaviour of software developers with the final aim of defining approaches which would be able to reduce the information overload (*e.g.,* number of artefacts to be analysed) of developers by filtering and ranking the information presented by the development environment (Fritz et al., 2007; Kersten and Murphy, 2006; Murphy et al., 2006). The findings of our paper can complement the body of knowledge in this field. Indeed, the usage patterns identified in our study can be used to complement such approaches providing a more effective support during program comprehension.

Bavota et al. (2013) performed a controlled experiment to investigate how developers navigate different sources of documentation (*i.e.,* javadoc, sequence diagrams, and use cases) available in the IDE while performing program comprehension and change impact analysis activities. The study was conducted with computer science students, and found that although participants spend a conspicuous proportion of the available time by focusing on source code, they browse back and forth between source code and either static (class) or dynamic (sequence) diagrams. Less frequently, participants—especially more experienced ones—follow an "integrated" approach by using different kinds of artefacts (Bavota et al., 2013).

Another study monitoring the developers' behaviour was presented by Singh et al. (2014), who monitored developers activities to analyse the technical debt payments of developers. They combine code maintainability and comprehension effort data to create a framework supporting real-time prioritisation of technical debt removal efforts. The proposed framework has been evaluated with ABB developers, showing its applicability. Data about developers' behaviour have been also used to measure the costs per activity or per code artefacts (Astromskis et al., 2014).

de Alwis and Murphy (2006) analysed how programmers experience disorientation when using Eclipse, identifying three factors that may lead to disorientation: the absence of connecting navigation context during program exploration, thrashing between displays to view necessary pieces of code, and the pursuit of sometimes unrelated subtasks.

Storey et al. (2000) performed a study aimed at analysing whether program understanding tools enhance or change the way that programmers understand programs. Based on the results achieved the authors suggested that tools should support multiple strategies (top-down and bottom-up, for example) and should aim to reduce cognitive overhead during program exploration. The results of our work can be used to produce smarter IDEs, limiting programmers disorientation.

Several works have been carried out by relying on the Eclipse IDE interaction data collected by the Mylyn tool (Murphy et al., 2006). Ying and Robillard (2011) studied data from over 4000 programming sessions of open source projects using the Mylyn task monitoring mechanism. Their study aimed at investigating different editing styles when working on source code. The achieved results showed that the developers' editing style changes on the basis of the specific task they need to perform. For example, when fixing bugs developers tend to edit source code earlier during the programming session with respect to what they do while enhancing existing features. In this paper, we found that although developers might change their style for specific tasks, in general they often move from looking for on-line help to coding related activities and, within a working session, from code related activities to communication with other developers or code execution back and forward.

Murphy et al. (2006) used Mylyn to investigate how developers use the Eclipse IDE. As the result of this analysis they reported: (i) the most used views (1st package explorer, 2nd the console, 3rd the search), (ii) the most used perspectives (1st Java, 2nd Debug, 3rd Team synchronisation), (iii) the most used commands in the IDE (1st delete, 2nd save, 3rd past), and (iv) the most used refactoring operations (1st rename, 2nd move, 3rd extract).

Parnin and Rugaber (2011) used similar data to investigate how developers resume their activities after interruptions. They found that quite rarely (in 10% of the monitored sessions) developers resume coding in less than one minute. Indeed, developers mostly start their coding sessions by focusing on activities helping them to rebuild their task context.

Murphy-Hill et al. (2011) analyzed eight different datasets trying to understand how developers perform refactoring. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment, data from the Eclipse Usage Collector aggregating activities of 13,000 developers for almost one year, and information extracted from versioning systems. Some of the several interesting findings they found were (i) almost 41% of development activities contain at least one refactoring session, (ii) programmers rarely (almost 10% of times) configure refactoring tools, (iii) commit messages do not help in predicting refactoring, since rarely developers explicitly report their refactoring activities in them, (iv) developers often perform *floss refactoring*, namely they interleave refactoring with other programming activities, and (v) most of the refactoring operations (close to 90%) are manually performed by developers without the help of any tool.

Negara et al. (2013) collected and analysed IDE interaction data to detect manual refactoring activities, and investigate them in comparison with automatic refactorings. They found that developers perform refactoring 11% more manually than automatically and that refactoring operations have different "popularity" in manual and automated refactoring.

Fuchs et al. (2014) performed a study where they monitored students' interaction with the IDE and browsing data to assess their behaviour during a programming task. They provide a categorisation of the types queries performed by students on search engines while programming (*e.g.,* troubleshooting, basic programming, *etc.*). Also, they show the most visited websites, with the first in the ranking being Stack Overflow.

While we share with such works the analysis of code browsing and writing activities, we also observe other kinds of activities developers perform within the IDE (*e.g.,* executing the program) and outside it. Also, we relate the activities being performed with the architectural layers being navigated and with the code readability and complexity.

In a related research thread, (Mark et al., 2005) observed 24 information workers showing that they frequently switch between tasks and 57% of their activities are interrupted. Table 3 reports similar findings: we found that a work session lasts 59 and involves 84 activities on average.

Finally, studies have explicitly focused the attention on the identification of IDE's weaknesses. DeLine et al. (2005) identified several usability issues of conventional development environments when a developer has to update a software system, including maintaining the number and layout of open text documents and relying heavily on textual search for navigation. As previously said, our study has a different focus, and expands to developers' activities performed outside the IDE.

5.2. Data collection tools

The research area of developer behaviour analysis has been constantly growing, therefore new tools and methods to collect and analyse developer behaviour data emerged. Kersten and Murphy (2005) created the Mylar tool, which monitor the usage of Eclipse IDE[13] to analyse which are its most used features. Mylar evolved into Mylyn, which has been used in many studies analysing the developer interactions with the Eclipse IDE (Ying and Robillard, 2011; Murphy et al., 2006; Murphy-Hill et al., 2009; Negara et al., 2013; Fuchs et al., 2014).

Robbes and Lanza (2007) presented the SpyWare tool, able to capture the set of events during a development session. SpyWare has implementations for the Eclipse and Squeak[14] development environments.

Minelli et al. (2014a, 2014b, 2014c) presented the DFlow tool to collect and visualise data about developers activities within the Pharo IDE.[15]

Johnson et al. (2003) developed the Hackystat platform, to collect software product and process data. The platform is composed of sensors collecting data from various sources. Authors also developed the Zorro tool (Johnson and Kou, 2007), which uses the data collected by Hackystat to automatically identify whether Test Driven Development (TDD) practices are applied within the development work.

Snipes et al. (2013, 2014) present the Blaze tool to automatically infer best practices in developer behaviour, and use this information to gameify the development process, by rewarding the developers with points and badges. The tool works in Visual Studio IDE, and uses other tools (*e.g.,* Mylyn, Hackystat) to collect the data.

There also exist commercial tools, such as Codealike[16] or WakaTime[17] which monitor developer behaviour within IDE's.

In our work, we chose to use PROM, a monitoring tool developed at the Free University of Bozen-Bolzano (Sillitti et al., 2003; Remencius et al., 2009; Coman et al., 2009; Sillitti et al., 2011). Like Hackystat it uses sensors to collect data from various development environments (Eclipse, Visual Studio, Netbeans, *etc.*), operating systems (OS X, Windows, Linux), source code versioning systems (Svn, Git), and productivity software (MS Office, Open Office). In Table 9 we compare PROM with the other available data collection tools. The comparison shows as most of the tools which collect the data about developer behaviour from the IDE, do not integrate other data sources, like source code management (SCM) systems, browsing history, e-mail systems, productivity software. For instance, no one of the available tools is able to capture browsing events from every web browser. Also, no one monitors events generated inside productivity suites (*e.g.,* Microsoft Office). These features, together with our high knowledge of the PROM tool, driven our choice of the tool to use in our study.

6. Conclusion and lessons learned

This paper reported results of an exploratory study aimed at recording how six developers—working in three industrial projects related to the development and evolution of real-time software—interacted with their working environment (IDE and other applications in the workstation) for a period of over 1000 h. More specifically, we analysed (i) the proportion of time spent in various kinds of activities and the likelihood of transition between different activities, (ii) how developers navigated across the software architecture during a development activity, and (iii) how the size, complexity, coupling, and readability of source code might have triggered certain activities such as browsing other files or searching for help. The achieved results provide several valuable findings for the research community:

Lesson 1. *The identification of useful sources of (informal) documentation when looking for online help is far from trivial, and should be better supported, possibly with integration in the IDE*. Our results showed that the usage of online help is fairly limited (2%), but it is pretty intense during specific sessions, in which developers worked on parts of the systems—such as the GUI—for which they needed (and expected) to find online help available. In these working sessions, developers formulate multiple queries to find what they need and, once found the appropriate resource, they tend to go back and forth between the code and the online help page in the browser. This continuous context switching could negatively affect their productivity. Recommender systems integrated with the IDE and able to automatically identify (informal) sources of documentation useful for a task at hand (see *e.g.,* Ponzanelli et al., 2014) could really make the difference in such a context.

Lesson 2. *Monitoring developers' activities both inside and outside the IDE can provide interesting insight on code components creating issues to software developers*. For example, we observed a very complex file that triggered the execution of the code in 76% of cases after the developers work on it (*i.e.,* likely to check the correct behaviour of the system after the applied changes). This information, combined with standard quality metrics, can be exploited to develop a new family of quality checkers, possibly identifying symptoms of poor design and implementation choices (similarly to what done by code smells detectors (Palomba et al., 2015)), as well as customised practices of development modelled on developers' behaviour. Indeed, while in the specific example discussed above, the complexity metric might be enough to identify the quality issue, some classes might represent maintainability issues even if they do not exhibit worrying quality metrics profiles (Palomba et al., 2015). We plan to investigate in the future the usage of developers' interaction data to identify poor design and implementation choices.

Lesson 3. *Characteristics of the code under development can be exploited to generate contextual recommendation*. Our results showed that changes to files having a high complexity have a moderate correlation with the likelihood of executing the system or searching for help. This result can be potentially useful for the prioritisation of contextual recommendations in the IDE (Ponzanelli

[13] www.eclipse.com.
[14] www.squeak.org.
[15] www.pharo.org.
[16] www.codealike.com.
[17] www.wakatime.com.

Table 9
Comparison of developer behaviour data collection tools .

	PROM (Sillitti et al., 2003; Remencius et al., 2009; Coman et al., 2009; Sillitti et al., 2011)	**Hackystat** (Johnson et al., 2003)	**Mylyn** (Ying and Robillard, 2011; Murphy et al., 2006; Murphy-Hill et al., 2011; Negara et al., 2013; Fuchs et al., 2014)	SpyWare (Robbes and Lanza, 2007)	DFlow (Minelli et al., 2014a; 2014b; 2014c)	Codealike	WakaTime
Type	Academic	Academic	Free, professional	Academic	Academic	Commercial	Commercial
OSS	No	Yes	Yes	No	Yes	No	Some
IDE	Eclipse, Netbeans, Intellij, Visual Studio	Emacs, Eclipse, JBuilder, Visual Studio	Eclipse	Eclipse, Squeak	Pharo	Visual Studio	Visual studio, Sublime, Brackets, Eclipse, Vim, Xcode, Emacs, Intellij
Browsers	Any	No	No	No	No	Chrome	No
SCM	Svn, Git	CVS	No	No	No	No	Github, Bitbucket, Slack
Build systems	No	Ant	No	No	No	No	No
Testing systems	None	JUnit	No	No	No	No	No
Issue trackers	Built in, Bugzilla, Trac, IBM Jazz	Jira	Bugzilla, Trac, Github, Jira	No	No	No	No
Mailing systems	MS Outlook	No	No	No	No	No	No
Productivity systems	MS Office, Open office	No	No	No	No	No	No
Plugins	Yes	Yes	No	No	No	No	Yes

et al., 2014), making such recommendations more "talkative" when changes in complex code occur. Also, this result confirms the usefulness of approaches that based on the characteristics (*e.g.*, complexity) of a change indicate whether it is likely to induce a fix (Kim et al., 2008), and therefore whether analysis or testing activities are highly desirable after the change.

These lessons learned represent the main input for our future research agenda on the topic, mainly focused on designing and developing recommenders integrated in the IDEs, such as those described above.

References

Astromskis, S., Janes, A., Sillitti, A., Succi, G., 2014. An approach to non-invasive cost accounting. In: Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Verona, Italy.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S., 2013. An empirical investigation on documentation usage patterns in maintenance tasks. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22–28, 2013, pp. 210–219.

Buse, R.P., Weimer, W.R., 2010. Learning a metric for code readability. IEEE Trans. Software Eng. 36 (4), 546–558.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Software Eng. (TSE) 20 (6), 476–493.

Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences, second ed. Lawrence Earlbaum Associates.

Coman, I., Sillitti, A., Succi, G., 2009. A case-study on using an automated in-process software engineering measurement and analysis system in an industrial environment. In: IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009, pp. 89–99. doi:10.1109/ICSE.2009.5070511.

de Alwis, B., Murphy, G.C., 2006. Using visual momentum to explain disorientation in the Eclipse IDE. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE Computer Society, Brighton, UK, pp. 51–54.

DeLine, R., Khella, A., Czerwinski, M., Robertson, G.G., 2005. Towards understanding programs through wear-based filtering. In: Proceedings of the ACM 2005 Symposium on Software Visualization. ACM, St. Louis, Missouri, USA, pp. 183–192.

Fritz, T., Murphy, G.C., Hill, E., 2007. Does a programmer's activity indicate knowledge of code? In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Dubrovnik, Croatia, pp. 341–350.

Fronza, I., Sillitti, A., Succi, G., Vlasenko, J., 2011. Analysing the usage of tools in pair programming sessions. In: Agile Processes in Software Engineering and Extreme Programming - 12th International Conference, XP 2011, Madrid, Spain, May 10–13, 2011. Proceedings, pp. 1–11.

Fuchs, M., Heckner, M., Raab, F., Wolff, C., 2014. Monitoring students' mobile app coding behavior data analysis based on IDE and browser interaction logs. In: 2014 IEEE Global Engineering Education Conference (EDUCON), pp. 892–899. doi:10.1109/EDUCON.2014.6826202.

Guéhéneuc, Y.-G., 2006. TAUPE: towards understanding program comprehension. In: Proceedings of the 2006 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006), October 16–19, 2006, Toronto, Ontario, Canada. IBM, pp. 1–13.

Holmes, R., Walker, R.J., Murphy, G.C., 2005. Strathcona example recommendation tool. In: Proceedings of ESEC/FSE 2005, pp. 237–240.

Janes, A., Sillitti, A., Succi, G., 2008. Non-invasive software process data collection for expert identification. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE). Knowledge Systems Institute, San Francisco, CA, USA.

Janes, A., Succi, G., 2014. Lean Software Development in Action. Springer.

Jeanmart, S., Guéhéneuc, Y.-G., Sahraoui, H.A., Habra, N., 2009. Impact of the visitor pattern on program comprehension and maintenance. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida, USA, pp. 69–78.

Johnson, P., 2001. You can't even ask them to push a button: toward ubiquitous, developer-centric, empirical software engineering. In: Proceedings of the NSF Workshop for New Visions for Software Design and Productivity: Research and Applications, Nashville, TN, USA.

Johnson, P., Kou, H., 2007. Automated recognition of test-driven development with Zorro. In: Agile Conference (AGILE), 2007, pp. 15–25. doi:10.1109/AGILE.2007.16.

Johnson, P., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S., Doane, W., 2003. Beyond the personal software process: metrics collection and analysis for the differently disciplined. In: 25th International Conference on Software Engineering, 2003. Proceedings, pp. 641–646. doi:10.1109/ICSE.2003.1201249.

Kersten, M., Murphy, G.C., 2005. Mylar: a degree-of-interest model for ides. In: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA, March 14–18, 2005, pp. 159–168.

Kersten, M., Murphy, G.C., 2006. Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Oregon, USA, pp. 1–11.

Kim, S., Jr., E.J.W., Zhang, Y., 2008. Classifying software changes: clean or buggy? IEEE Trans. Software Eng. 34 (2), 181–196.

Ko, A., Myers, B., Coblenz, M., Aung, H., 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Software Eng. 32 (12), 971–987. doi:10.1109/TSE.2006.116.

Mark, G., González, V.M., Harris, J., 2005. No task left behind? Examining the nature of fragmented work. In: Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2–7, 2005, pp. 321–330.

McCabe, T., 1976. A complexity measure. IEEE Trans. Software Eng. SE-2 (4), 308–320.

Minelli, R., Baracchi, L., Mocci, A., Lanza, M., 2014. Visual Storytelling of development sessions. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 416–420. doi:10.1109/ICSME.2014.65.

Minelli, R., Mocci, A., Lanza, M., Baracchi, L., 2014. Visualizing developer interactions. In: 2014 Second IEEE Working Conference on Software Visualization (VISSOFT), pp. 147–156. doi:10.1109/VISSOFT.2014.31.

Minelli, R., Mocci, A., Lanza, M., Kobayashi, T., 2014. Quantifying program comprehension with interaction data. In: 2014 14th International Conference on Quality Software (QSIC), pp. 276–285. doi:10.1109/QSIC.2014.11.

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

Murphy-Hill, E., Parnin, C., Black, A.P., 2011. How we refactor, and how we know it. Trans. Software Eng. 38 (1), 5–18.

Murphy-Hill, E.R., Parnin, C., Black, A.P., 2009. How we refactor, and how we know it. In: 31st International Conference on Software Engineering, ICSE 2009, May 16–24, 2009, Vancouver, Canada, Proceedings, pp. 287–297.

Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D., 2013. A comparative study of manual and automated refactorings. In: Castagna, G. (Ed.), ECOOP 2013 Object-Oriented Programming. In: Number 7920 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 552–576.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., De Lucia, A., 2015. Mining version histories for detecting code smells. IEEE Trans. Software Eng. 41 (5), 462–489. doi:10.1109/TSE.2014.2372760.

Parnin, C., Rugaber, S., 2011. Resumption strategies for interrupted programming tasks. Software Quality J. 19 (1), 5–34.

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M., 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In: 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31, - June 1, 2014, Hyderabad, India, pp. 102–111.

Rahman, M., Yeasmin, S., Roy, C., 2014. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In: Proceedings of CSMR/WCRE 2014, p. Toappear.

Remencius, T., Sillitti, A., Succi, G., 2009. Using metric visualization and sharing tool to drive agile-related practices. In: Abrahamsson, P., Marchesi, M., Maurer, F. (Eds.), Agile Processes in Software Engineering and Extreme Programming. In: Number 31 in Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, pp. 255–256.

Robbes, R., Lanza, M., 2007. Characterizing and understanding development sessions. In: 15th IEEE International Conference on Program Comprehension, 2007. ICPC '07, pp. 155–166. doi:10.1109/ICPC.2007.12.

Robillard, M.P., Coelho, W., Murphy, G.C., 2004. How effective developers investigate source code: an exploratory study. IEEE Trans. Software Eng. 30 (12), 889–903.

Sharif, B., Maletic, J.I., 2010. An eye tracking study on the effects of layout in understanding the role of design patterns. In: Proceedings of the 26th IEEE International Conference on Software Maintenance. IEEE Computer Society, Timisoara, Romania, pp. 1–10.

Sillito, J., Murphy, G.C., Volder, K.D., 2008. Asking and answering questions during a programming change task. IEEE Trans. Software Eng. 34 (4), 434–451.

Sillitti, A., Janes, A., Succi, G., Vernazza, T., 2003. Collecting, integrating and analyzing software metrics and personal software process data. In: Euromicro Conference, 2003. Proceedings. 29th, pp. 336–342. doi:10.1109/EURMIC.2003.1231611.

Sillitti, A., Succi, G., Vlasenko, J., 2011. Toward a better understanding of tool usage (NIER track). In: Proceedings of the 33rd International Conference on Software Engineering. ACM, New York, NY, USA, pp. 832–835. doi:10.1145/1985793.1985917.

Singer, J., Lethbridge, T.C., Vinson, N.G., Anquetil, N., 1997. An examination of software engineering work practices. In: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research. IBM, Toronto, Ontario, Canada, p. 21.

Singh, V., Snipes, W., Kraft, N., 2014. A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension. In: 2014 Sixth International Workshop on Managing Technical Debt (MTD), pp. 27–30. doi:10.1109/MTD.2014.16.

Snipes, W., Augustine, V., Nair, A.R., Murphy-Hill, E., 2013. Towards recognizing and rewarding efficient developer work patterns. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 1277–1280.

Snipes, W., Nair, A.R., Murphy-Hill, E., 2014. Experiences gamifying developer adoption of practices and tools. In: Companion Proceedings of the 36th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 105–114. doi:10.1145/2591062.2591171.

Soh, Z., Khomh, F., Gueheneuc, Y.-G., Antoniol, G., 2013. Towards understanding how developers spend their effort during maintenance activities. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 152–161. doi:10.1109/WCRE.2013.6671290.

Storey, M.-A.D., Wong, K., Müller, H.A., 2000. How do program understanding tools affect how programmers understand programs? Sci. Comput. Program 36 (2–3), 183–207.

von Mayrhauser, A., Vans, A.M., 1994. Comprehension processes during large scale maintenance. In: Proceedings of the 16th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 39–48.

Ying, A.T.T., Robillard, M.P., 2011. The influence of the task on programmer behaviour. In: The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22–24, 2011. IEEE, pp. 31–40.

Yusuf, S., Kagdi, H.H., Maletic, J.I., 2007. Assessing the comprehension of UML class diagrams via eye tracking. In: Proceedings of the 15th International Conference on Program Comprehension. IEEE Computer Society, Banff, Alberta, Canada, pp. 113–122.

Saulius Astromskis Saulius Astromskis currently is a software engineer at Vertical-Life. He received his PhD degree from University of Bolzano in 2015. His main research interests are empirical software engineering, software processes, agile and lean software development, process mining, software measurement, and data visualisation.

Gabriele Bavota Gabriele Bavota is an Assistant Professor at the Università della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is author of over 80 papers appeared in international journals, conferences and workshops. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, ICSME, MSR, ICPC, SANER, SCAM, and others.

Andrea Janes Andrea Janes is a researcher at the Free University of Bolzano-Bozen. His research interests include Lean software development, value-based software engineering, and empirical software engineering. Dr. Janes holds a doctorate degree in informatics from the University of Klagenfurt, Austria.

Barbara Russo Barbara Russo is associate professor at the Faculty of Computer Science of the Free University of Bozen-Bolzano. She got her PhD in Mathematics from the University of Trento, Italy. She was visiting researcher at the Max-Planck Institut fr Mathematik in Bonn, DE and Marie Curie fellow at the University of Liverpool, UK. She has more than sixty publications both in mathematics and computer science in international journals including Bulletin of the London Mathematic Society and Journal of Empirical Software Engineering and Journal of Software Systems. She has co-authored the book Adopting Open Source Software, A Practical Guide, MIT Press, 2011. She participated to several national and international projects (research projects in EU FP5, FP6, and FIRB programmes and educational projects in TEMPUS and Erasmus Mundus programmes). Barbara Russo was awarded by the Italian Association for Enterprises for the best example of Public-Private collaboration with the project Bachelor studies for working students and Reviewer of the Yearå for the International Journal of Strategic Information System, Elsvier in 2015. Her research interest is in the field of empirical software engineering and software maintenance. Her technical competences are in data mining and modelling for software reliability.

Massimiliano Di Penta Massimiliano Di Penta is associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is author of over 190 papers appeared in international journals, conferences and workshops. He serves and has served in the organizing and program committees of over 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, GECCO, MSR WCRE, and others. He has been general co-chair of various events, including SCAM 2010, SSBSE 2010, and WCRE 2008. Also, he has been program chair of events such as the ICSM 2012, ICPC 2013, MSR 2012, MSR 2013, WCRE 2006, WCRE 2007, SSBSE 2009, and other workshops. He is currently member of the steering committee of ICSME, MSR, SSBSE, and PROMISE. Previously, he has been steering committee member of other conferences, including ICPC, SCAM, and WCRE. He is in the editorial board of IEEE Transactions on Software Engineering, the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley.