Visualising Software Developers' Activity Logs to Facilitate Explorative Analysis

Alena Kovarova, Martin Konopka, Lukas Sekerak, Pavol Navrat

Slovak University of Technology, Ilkovicova 2, 84216 Bratislava, Slovakia
alena.kovarova@stuba.sk, martin_konopka@stuba.sk, xsekerakl1@stuba.sk, pavol.navrat@stuba.sk

Abstract: In this paper, we discuss whether data collected from monitoring software developers' logs can be considered big. We hypothesize that it falls within the category of Big Data. The main topic of our paper however, is how to facilitate analysis of such data. Due to the specificity of the monitored activity, the analysis is at least partially explorative in its nature. We hypothesize that visualisation can be a productive approach in such a case. We present several visualisation schemes (diagram types) and show those applied to explorative analysis of data gathered within one four year project that we have been participating in.

Keywords: Activity log; log stream; Programmer; Software development; Visualisation; Big data

1 Introduction

Let us consider a serious creative human activity, which is supposed to result in developing a very complex technical product. The human activity is inherently individual by definition and at the same time, due to the nature of the task, it can rarely be accomplished by a single person because of task complexity, delivery time requirements, etc. Thus, we envisage a complex process of multi-human activity that requires coordinated cooperation over a long period of time, with a collective very structured outcome.

There are many occasions during the development of a software product, when it is desirable to know what a particular developer was doing and how they performed, what they were doing with a particular part of the product being developed, or any other similar question. But perhaps, the nature of the technical product is such that it evolves over time or users of the product change their expectations. This translates to modifications of product requirements and specifications, and thus, onward to redeveloping the product, which may be a very complicated endeavour. Certain questions arise, such as "Who of the original developers should be assigned the task?" or "In which place should they attempt to make an amendment?"

To find answers to these and similar questions, we need to know more about the developers' activity. But do we know exactly what the required information is? On the side of data, assume that the development process takes place for several months and there are several, or even a few dozen developers involved. Let us consider that we are able to monitor developer's activity at the granularity of elementary steps taken once a few seconds if not more frequently. No matter how small the single record is, recording it every few seconds over a period of time, for example, several months for several developers will definitely surpass any reasonable volume of data storage available for the usual data processing tasks. More importantly, that would not be a reasonable modus of operation. The data are required to be processed in real-time so that we have the information when needed. Recently, it has become fashionable to describe such data as Big Data [5].

The outcome of software development is a software system, which is by itself a very complex product. The development process as a rule involves activities of several programmers, or more generally software developers, over a period of several months, sometimes years. Their work is essentially writing or modifying texts in a source programming language, and also writing a documentation, solving problems that occur during development, or communicating with each other. However, their actions can be logged at a very low level, way below the level of the programming language, not to speak of the level of developers' actions.

Currently, it is technically feasible to monitor not only when and which text (source code) they write, but also how they write it and which information and communication technology tool they employ. Monitoring generates a lot of data. In this paper, we discuss if data collected from monitoring software developer's logs can be considered big. We hypothesize that it falls within the category of Big Data. The main topic of our paper is, however, how to facilitate analysis of such data. Due to specificity of the monitored activity, the analysis is at least partially explorative in its nature. We hypothesize that visualisation can be a productive approach in such a case. We present several visualisation schemes (diagram types) and show those applied to an explorative analysis of data gathered within one four year project that we have been participating in.

The rest of the paper is structured as follows. In Section 2, we discuss whether data collected on software development can be considered Big Data. We identify so-called interaction data to be such data. In Section 3, we discuss what information could be extracted by visualising data and by what approaches this can be accomplished. We present our approach in Section 4. First, we describe what data is potentially the most interesting in the logs, and then we show our visualisation schemes. We evaluate the approach in Section 5. The paper is concluded in Section 6 where we also suggest some possibilities for future work.

2 Big Data in Software Engineering

In order to evaluate actual status and progress of a software project, it is required to monitor it on a suitable level of detail, from the lowest hardware interactions up to the highest level with performed development tasks [10]. Current research in empirical measurements and evaluation of software development focuses on monitoring developers on the level of interactions, which brings us into dealing with Big Data in software engineering. More precisely, interaction data fulfils the *4 Vs rule* of Big Data [24] and also occurs in real-time, thus the sensible approach is to represent this as a data stream that may be computationally analysed to reveal patterns [12], trends, and associations, especially related to human behaviour and interactions [21]. The motivation for gathering, processing and evaluating such streams of interaction data in software engineering is to get a detailed overview of the developers' work [14], to understand how they behave individually or in groups, and to avoid problems in development.

2.1 Interaction Data in Software Development

Traditional methods of evaluating progress on software projects are based on monitoring completion of development tasks by developers in task management systems. Developers are given tasks to complete, or they identify the tasks themselves, and then update the status of a task in the course of the work [14]. Team leaders are able to observe progress on a project, communicate the current status with developers, or reason upon it. However, this approach fails to identify causes of developers' mistakes, faults, or delays in completion of tasks. A more detailed approach is to monitor and evaluate a software project on the level of source code [14]. However, this still does not resolve the aforementioned problems.

Practical research in software engineering requires the monitoring of software developers in a greater detail [1, 10]. Evaluating software projects and developers through tasks and source code is limited to observing the results of the work only, not the processes that led to those results. Software developers interact with tools to fulfil a task assignment and are affected and surrounded by several different contexts. We summarize this information under the name *interaction data* [14]. The main sources of interaction data are tools and systems that developers work with during software development and maintenance, such as IDE – integrated development environments (Microsoft Visual Studio, Eclipse), revision control systems (Git, Microsoft Team Foundation Server, SVN), task management systems (Redmine, Atlassian Jira), software CASE tools (Sparx Enterprise Architect, IBM Rational Software Architect), operating systems and many other tools, e.g., a web browser, instant messaging, or e-mail client, note-taking software, etc.

As pointed out by the authors in [14], interaction data encompasses mainly the following four types of data:

- Interactions – observable actions of a developer within tools, e.g., navigation in source code, code editing, mouse movements, committing changes to a source code repository, etc.
- Artefacts – entities that a developer interacts with, e.g., source code documents, documentation web pages, physical artefacts or even people.
- Tools – software applications and systems that a developer works with.
- Contexts – decisions, reasons and other circumstances of developer's work, e.g., what, when, how, and why affected them during their work.

As can be seen, interaction data cover almost everything that developers may come into interaction with, and also how they interact in that situation. Although developers' *interactions* are monitored at the lowest possible level of detail, they can still be used to track development tasks (as artefacts) in task management systems/tools, as well as source code (artefacts) and changes (interactions) within revision control systems/tools. Recording this amount of data about software developers allows us to attempt to identify their expertise or familiarity with code [13, 14], to annotate source code with important information [19], or even search for unknown or hidden connections between source code documents [11]. Several other approaches based on utilizing interaction data nowadays arise using the proposed systems [14], although always with predefined assumptions.

2.2 Monitoring Software Developer's Activity

Many approaches and systems have been proposed for monitoring interaction data in software development [1, 6, 10, 18]. However, many operate at different levels of detail. We can observe developers' activity at the following levels [21] (starting with the lowest level):

- Hardware interactions, e.g., mouse moves, key presses, touch gestures.
- Widget interactions – a developer interacts with areas (widgets) of a tool.
    - Single widget interactions, e.g., scrolling in code editing window in an IDE, copying code fragments, selecting a text.
    - Multi widget interactions, e.g., searching for references of a source code entity and using them in a code editing window.
- Activities – enclosed parts of developer's work on a task, e.g., studying a code, adding a new code, debugging, documenting completed work.
- Tasks – described with a goal which a developer is about to accomplish, e.g., fix a bug no. 324, added service endpoint ABC.

Based on the employed level of interactions, we encounter problems with the collecting, processing and storing of interaction data [14]. For example, Mylyn [10] aggregates interaction events in an IDE into five pre-selected types of activities, the IDE++ project [6] records even the key presses and mouse events. However, although authors of IDE++ record almost every event in Eclipse IDE, they do not attempt to store them due to a high volume of data. PerConIK [1] records various interaction events in an IDE. At first, authors in PerConIK attempted to persistently store compressed keyboard button presses and mouse events, but later left them out for the same reasons, despite it being the finest grained data. Such data could help us determine user activity duration or some individual characteristics of either the user or the domain. IDE++ alternatively records all available data and redirects it to other connected tools that examine these events.

Monitoring a developer on the hardware and widget levels is possible thanks to available application programming interfaces in operating systems and development tools. However, identification of activities is not possible with tools because of a vague notion of what an activity in fact is. The authors in [19] attempted to automatically identify activities using Hidden Markov Models, because this technique matched attributes of difference between activity and interaction. Activity is composed of interactions, but only those of a certain intention for a developer, which they had to undertake during their work on a task. From the automation viewpoint, we are not able to unambiguously distinguish between types of activities, e.g., adding a new functionality or refactoring. Because a developer’s interactions occur in real-time, we may attempt to incrementally identify developer’s activities in real-time as well [12]. The motivation behind identifying activities is to better describe development tasks or untangle them when they appear to be overlapping [12, 20].

Whether we monitor developers’ interactions, activities or tasks, the motivation remains the same: to understand how developers approach work on a software project individually and/or cooperatively, how they progress, and how they identify issues with respect to finishing their work in the desired quality and on time. Interaction data occurs in real-time and may be used for identifying patterns, trends or associations in developers’ interactions and activities.

2.3 Software Developer Interaction Data Can Become Big

Interaction events occur very fast during a developer’s work, from the finest level of granularity with recording of every keystroke, up to recording changes in source code contents after every widget interaction, e.g., navigation or scrolling in a document. Using the 4 Vs characteristics of Big Data, we may look upon interaction data as Big Data as well [24]:

- Volume – recording interactions in tools, e.g., every change in a source code document after navigation, mouse button presses and moves.

- Velocity – multiple interactions may occur within a second, or changes in mouse coordinates may be recorded with high frequency (1000 positions per second), multiplied by large development teams.
- Variety – monitoring various keystrokes, mouse, or events in an IDE, e.g., from opening a source code document, through to adding a new line of code, to identifying changes in abstract syntax trees of source code.
- Veracity – monitoring of IDE events cannot be predicted – a developer may study code while not interacting with tools or a computer in any way, thus unexpectedly not generating any data.

Our work is part of the research project PerConIK [1] where we monitor software developers at a medium size software company. Besides, we monitor students of Masters study programmes in Software Engineering and Information Systems who develop their semester projects. For the monitored interaction events within the infrastructure of PerConIK, see section 3.1. Also, note that we do not monitor mouse movements and key strokes that IDE++ does (but without storing them) [6], only interactions in tools because of the high volume which has to be stored if doing so. As an example of a possible data flow, consider the following statistics even for 10 developers monitored within the PerConIK project during 38 workdays in February and March 2015:

- Average velocity if any interaction event occurred in a time frame:
    - 3.682 per second and 18.893 per minute,
- Average velocity if over 10 interactions occurred in that time frame:
    - 71.351 per second and 42.849 per minute.

Although data of 10 developers may not seem really big, we use them as a representative sample from our dataset because of different work habits and experiences that students have, and because the setup of the PerConIK infrastructure has evolved over time. During this period, there were exactly 27,994 interactive minutes (i.e. minutes with at least 1 recorded interaction; if counting only interactive seconds, there were exactly 29,628 of such seconds). Pointedly, monitoring more developers for a longer time would increase these numbers in orders of magnitude.

Because of complying with the 4 Vs characteristics, interaction data and developers' activity are often subject to visualisation. Omoronyia et al. provide a good overview of 12 tools [18] visualising developers' activity. One example is the visualisation tool Team Tracks that displays the current activity of a developer, or which files are currently edited and who altered them. Their plugin to an IDE monitors which files were frequently visited and assumes that these files can be problematic. However, none of the mentioned tools [18] dealt with the visualisation of software development from a perspective similar to ours. This paper indicates the importance of capturing data for tracking developers' activity and, of course, the visualisation of this data.

3 Visualisation of Software Developers' Activity

In order to record huge amounts of data, it is usually required to choose a form of representation that facilitates eliciting important information. The information may be in the form of patterns, trends or associations found in raw data. There exists a variety of methods for acquiring such information based on statistics, artificial intelligence, machine learning, or formal concept analysis. In most of the cases, we need two things to choose the right method – 1. to have the data, and 2. to know what we are looking for in the data (either exact or abstract). For example, we can track certain quantities of events, or find certain patterns in them, and so on. But there are moments when we do not know the data and/or we do not know what we are looking for – we just want to find something new. Existing methods often fail if they do not have a goal set beforehand. In such a case, an explorative analysis (visual data mining) can be very useful. In general, it is the first step on the following path (see Figure 1):

- *Step 1*: In the case that data contains unknown information, it is visualised using different graphs or visualisation approaches. One can find, by inspecting them, something interesting and set it as an assumption.
- *Step 2*: In the case that there has been formulated an assumption (either resulting from the first step or simply from knowing a domain), this can be verified either by an experiment or by broader systematic data analysis. This verification can either support or refute the assumption and thus can be reformulated or accepted for the next step (e.g. as a new part of user or domain model).
- *Step 3*: In the case that it is already known what information is contained in the old data (either resulting from the first two steps or simply from knowing the data domain), there is a chance that it will be contained in the new data. To find it there, one can reason upon the live (streaming) data, and log only the found (derived) information – metadata.

In our work, we deal only with the first step of this (meta) information retrieval. Therefore, to maximize the success of a search, the visualisation has to follow the *type by task* taxonomy (TTT) of information visualisations proposed by Shneiderman [23]. It has its roots in the Visual Information Seeking Mantra: "Overview first, zoom and filter, then details-on-demand," and contains the following seven tasks:

- Overview: Gain an overview of the entire collection.
- Zoom: Zoom in on items of interest.
- Filter: Filter out uninteresting items.
- Details-on-demand: Select an item or group and get details when needed.

- Relate: View relationships among items.
- History: Keep a history of actions to support undo, replay, and progressive refinement.
- Extract: Allow extraction of sub-collections and of the query parameters.

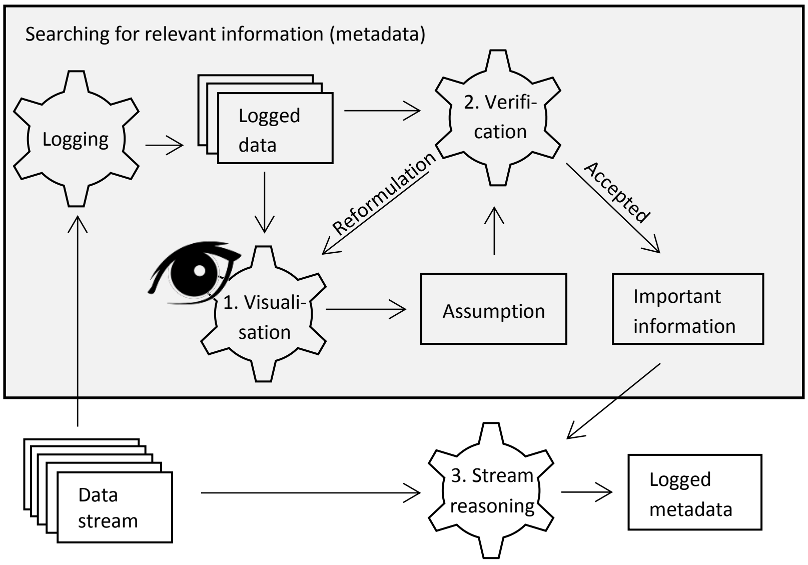

Figure 1
Exploratory analysis of logged data involving assumption formation based on visualisation

On the other side, the goal of a visualisation must be clear. Maletic, et al. [15] proposed that each software visualisation system supporting large-scale software development and maintenance has to have the following five dimensions. These dimensions reflect the why, who, what, where and, how questions to be addressed for the developed software visualisation:

- Tasks – *why* is the visualisation needed?
- Audience – *who* will use the visualisation?
- Target – *what* is the data source to represent?
- Representation – *how* to represent it?
- Medium – *where* to represent the visualisation?

In our case we answer these questions: Tasks – find new information in the data that could possibly improve software development, Audience – software development analysts, researchers, possibly project managers, Target – developers' interactions, Representation – graphs, Medium – computer monitor.

There are various approaches to visualisation in the domain of software development. Most of them visualise code structure [9] and/or source code metrics, and/or tasks (Gantt chart), e.g., Source Miner Evolution [17], YARN [8], Forest Metaphor [4], ClonEvol [7], GEVOL [3], EPOSee [2]. Some visualisations are even animated in time. DFlow [16] visualises developer's interactions during the development process, it is oriented only to navigating, writing, and understanding the source code and thus miss the wider context of developers' work. More complex tools, which are more similar to our solution, can be found in the time management domain, e.g., ManicTime[1] application.

3.1 Data Available on Software Developers' Activity

We monitor interaction events in tools that a software developer interacts with during their work. Although we may record even elementary events such as key presses or mouse movements that make our data big, we empirically selected only some of the events provided by tools (Microsoft Visual Studio, Eclipse, Git, Bash shell, and Mozilla Firefox). An interaction event is reported by a tool noting that a developer performed a meaningful operation. For each developer we also log which processes and applications were running in the operating system.

With a web browser we record *navigation* to URL addresses (through a link/URL bar/bookmark/other), *actions with tabs* (switch to/open/close), saving a *document* (and its name), or even creating a *bookmark* (with its name).

In an IDE we record interactions with source code *documents* (add, open, close, switch to, remove, save, rename), *projects* and *solutions* (add, open, close, switch to, remove, rename, refresh). We also record source code content interactions, specifically *code fragments manipulation* (copy, paste, cut, paste from a webpage) and *searching in source code* (searched expression, used search options, number of searched documents and results). We are also interested in a work to Git (commits) or Microsoft Team Foundation Server (check-ins). Because developers also use bash shell during their work, thus we record executed *bash commands*.

Recording of interaction events in tools is realized by custom plugins that communicate with local client application, called UACA, running on a developer's workstation. This application sends recorded data in chunks to a centralized repository using REST web services. With this approach we are able to monitor high quantities of interesting and very detailed data about software development, but still use it for evaluation, visualisation, or further processing. Further details on the PerConIK infrastructure can be found in [1].

Countless numbers of charts can be devised from the listed data and source code of monitored software projects. In this paper, we cannot discuss all the devised

[1] http://www.manictime.com/

data or graphs, and in any case, we did not employ all of them. However, to explain our visualisation approach in a simple and comprehensible manner, while still showing the potential of such visualisations, we include only charts that contain two activities – those of a developer interacting with a web browser and with an IDE. These are not further fragmented. We also did not include activities of bash commands because of their insufficient number. Answers to these questions can be found in such graphs: When and which developer was the most active? Is it always at the same time? Do all of the developers use a web browser and IDE by the same share? Is there a pattern or dependency at work in a web browser or an IDE? Which web sites were visited by developers the most? Is there a relation between the amount of time spent in a web browser during a work in an IDE, to the number of source code changes by a developer?

The graphs may also contain various metrics, e.g., for source codes: time spent by a developer typing them, studying them (reading without changing them), number of functions edited, how many times and how long a certain file was opened, the number of time the file was changed, etc. For web browsing: the ratio between the number of web pages related to work and private purposes, the ratio between the time spent on work vs. private web pages, the absolute number of web pages or absolute time spent on work/private web pages, correlation of these numbers with other quantities, the number of copied elements, etc.

Other potential graphs using mentioned metrics may answer the following questions: Can we categorize developers according to their read/edit/copy/paste behaviour? Can we evaluate their productivity? Is there any correlation between different types of activities that help us to indicate developers' experience, their strengths and weaknesses? Is there any harmful behaviour occurring in specific situations, e.g., introducing a bug by the end of a project? Is there any visible trend or pattern between the number of source code changes and other situations?

Answers to some of these questions can be found only if data is from a long enough period of time, which could take many years.

3.2 Graphs Devised from Software Developer's Activity

In order to identify any information from interaction data, it is necessary to combine events into higher level activities [12, 20] and then to visualise them. It is not easy to determine which actions form an activity. One way to do so is to apply a time threshold, as we describe in more detail in [22]. The visualisation graphs themselves should be transparent, readable and adaptable. We propose three kinds of graphs:

1) Timeline graphs, which depict selected type of activities – e.g., when and how long a developer worked in a certain application (see ManicTime graph in Figure 2); here it cannot easily be seen how many of them there are or how much time is spent in total. This type of graph can be found also in [16]

2) Scatter graphs – the dependency of selected activities (see Figure 6)
3) Column graphs – the cumulative numbers of the selected types of activities:
    a. One column is for one time step – e.g., it depicts number of web and IDE activities done within an hour; columns are ordered by time and have the same width (see Figure 3 and 4).
    b. One column is for one type of activity – e.g., it depicts the number of visits on stackoverflow.com domain; columns are ordered by height and have the same width.
    c. One column is for one block of activity – e.g., it depicts the number of changed lines in source code; columns are ordered by time and each block has a different width equal to the duration of a depicted block (see Figure 5)

Our implementation of the visualisation allows the user an interactive modifying time axis scale for each graph with a time axis. It is possible to choose time units to be used for data accumulation in the graph. By clicking on a column in a graph, a new graph with a finer time scale will open.

4 Evaluation

In order to be able to evaluate our proposed approach, we devised a prototype tool IVDA (Interactive Visualisation of Developers' Actions). It is implemented as a service that provides visualisations of logged data [22]. The visualisations are shown directly on the web browser, while computation takes place on the server.

There are tools that are able to monitor a software developer, but not to visualise their activity. There are also tools, on the other hand, that visualise software development but do not monitor the software developer so closely. Since our tool IVDA is no exception to this, some experiments are by design not fully supported.

For comparison with existing systems, we compared our tool with several other similar tools that were compared in [18]. Moreover, we also include the tool ManicTime in the comparison.

The tool is not from the domain of software development, but from the more general domain of time management. It monitors computer usage in any work. It is oriented toward applications or environments used, and documents (context) that the user is working within the given application (see Figure 2). The tool monitors how long activities lasted. Recorded data can be seen on the time axis. They serve the user as an overview of their work with the computer. The tool also offers cumulative results, albeit only in numerical form.

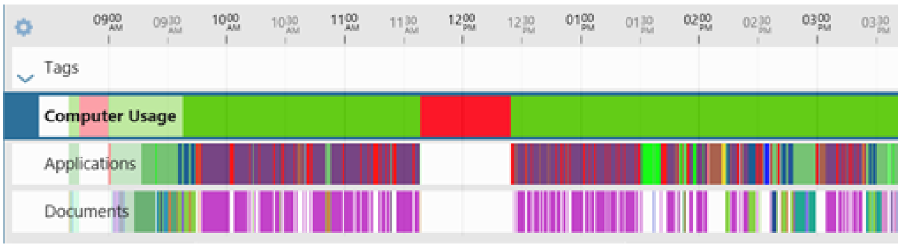

Figure 2
Timeline in ManicTime interface

See Table 1 for results of the comparison with our IVDA tool with existing tools mentioned in [18] that monitor developers' work, along with ManicTime. The comparison is done by classification based on workspace awareness elements and does not take into account visualisation ability. The number of individual elements across all 14 tools represents the need of developers to know that information. As we can see, IVDA offers the five most wanted elements. ManicTime offers three of them, although this tool is not specific to the domain of software development.

Table 1
Omoronyia, et al. [18] classification of visualisation tools with added IVDA and ManicTime

Tool	Identity	Location	Activity level	Actions	Intentions	Changes	Objects	Extents	Abilities	Influence	Expectations
TagSEA	x				x	x	x				
Jazz	x				x	x	x	x		x	
Expertise browser	x	x				x			x	x	
Sysiphus	x	x			x	x					
Hipikat	x										
Palantír	x		x			x	x	x			
FASTDash	x	x		x		x	x				
Team tracks			x							x	x
CASS	x				x	x	x				
Augur	x				x	x	x				
Ariadne	x							x			
Mylyn		x	x	x		x	x				x
ManicTime	**x**	**x**	**x**								
IVDA	**x**	**x**	**x**			**x**	**x**				
14 tools	12	6	6	2	5	10	8	3	1	3	2

To proceed with the evaluation of our approach, we performed a user test aimed at its visualisation quality. The tools from Table 1 do not provide visualisation as IVDA or ManicTime. Therefore we needed to choose other tools. As we already mentioned, there are tools, which visualise code structure and/or source code metrics, and/or tasks [2, 3, 4, 7, 8, 9, 16, 17], but they do not visualise programmer's activities with their context. This led us to find a tool for visualisation of activities, but from another domain, e.g., ManicTime, and to compare it with IVDA. The test included 10 test cases (TC) of differing difficulty (see Table 2). We conducted this test with 7 participants (4 males and 3 females) ages 19-35, with a university degree in an engineering field. None of them had previous experience with visualisations or tools that monitor time spent with a computer, and they use a web browser for 2 to 8 hours a day.

Table 2
10 test cases (TC1-TC10) conducted for evaluation of the IVDA tool

TC1	Inspect activity of any developer over the previous week.
TC2	Determine by inspection, for any developer, what activities prevailed during the previous day.
TC3	Find out which processes were run on the user's computer on a given day.
TC4	During which part of the last 3 days was the developer most productive?
TC5	Which files and environments did the developer work with yesterday?
TC6	Which particular file has been the most frequently modified one by a developer during the whole previous year?
TC7	Did the developer write source code over the last week more frequently in the mornings or in the evenings?
TC8	For how long did the developer use the browser after 11 o'clock?
TC9	Try to identify some habit in the developer's behaviour within the last month.
TC10	There are a developer's activities that do not relate to the actual development. If they indeed occur, find out when they started, how long they lasted and what is their nature. Choose a single August day in 2014.

Table 3 shows the success rates of tasks completion (within the allotted time limit) – the test participants completed most of the required tasks on time for both of the compared tools. Moreover, we noticed that the test participants frequently had problems with initial tasks, which took them longer than expected. This may be caused by a lack of intuitiveness of the tool in the initial phase. Keeping the main goal of this research in mind, i.e., to facilitate explorative analysis, the TC9 task was the most critical one. Our IVDA tool has higher potential to analyse monitored and visualised data, and to seek new information from it. Moreover, IVDA offers information, which other visualising tools do not. One of the interesting feedbacks from one participant was that she became curious after experiments and started to explore the data herself to answer her own questions.

In Figure 3, the graphs reveal information about the logged developer "Puma". It appears "Puma" mostly writes code during working days. There are a few days when "Puma" works really hard (maybe refactoring some code since a lot of lines of the code were changed). But there are also days (and even weeks), when "Puma" uses only the web browser. The gap in August suggests that Puma was on vacation.

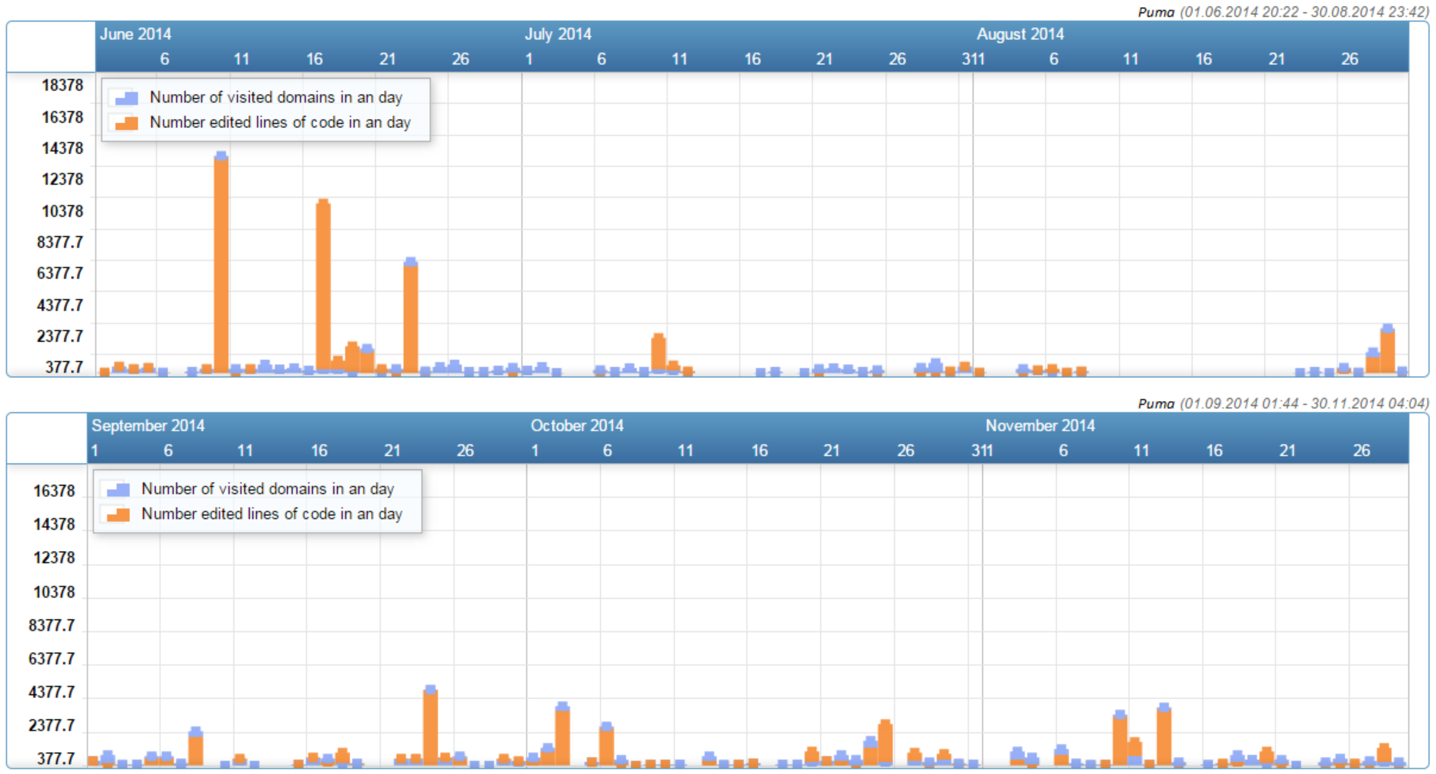

Figure 3
Overview of work by a developer "Puma" within a timeframe of 6 months –
columns in graphs depict cumulated number of visited domains and edited lines of code per day

Table 3

Success rate of tasks TC1-TC10 completion in IVDA and ManicTime (%)

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10
IVDA	85	100	85	100	57	100	100	85	100	100
ManicTime	100	100	100	57	100	85	85	85	57	71

Graphs in Figure 4 show that different developers work differently during the same time period. “Jaguar” edited hundreds lines of code during selected days. “Puma” worked a lot with a browser – probably searching for something. In both graphs it is clearly visible that both of them are more active during the day and resting during the night. No regularities like lunch breaks, meetings, etc. can be found in this part of a year, but in our dataset there are other time periods, where such events are visible.

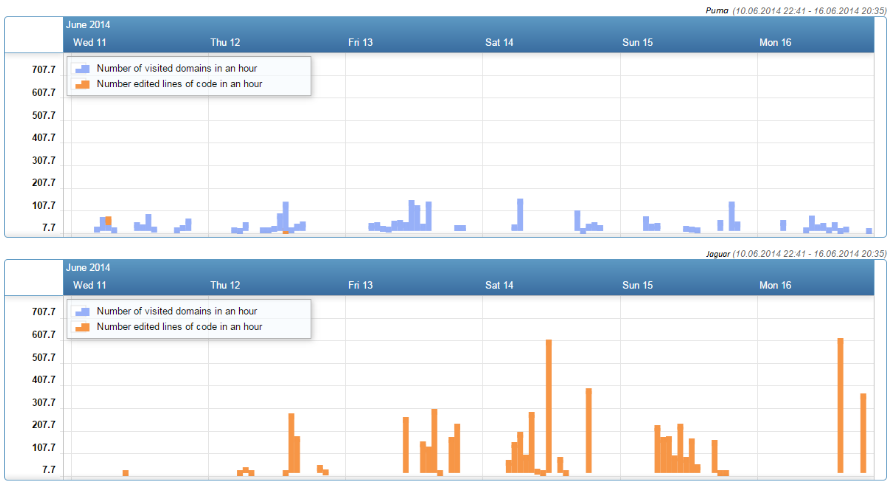

Figure 4

Comparison of two developers “Puma” and “Jaguar” over one week –the number of visited domains together with the number of edited lines, both cumulated per hour

Graphs in Figure 5 depict columns of different widths since the activities are cumulated per continuous activity (either continuous in a web browser or continuous in IDE). This type of visual representation helps the explorer quickly find the most active periods in a developer’s working day – the highest and the widest columns. As the example of the developer “Puma” shows, developers' days can vary widely – there is a different time of day for the highest number of edited lines of code, different amounts of visited domains and different ratios of these two activities. Although these three days look so different, they have also something in common – the Pareto principle also plays a role here: it looks like the most active periods cover 20% of the developer’s working day and during these peaks, 80% of their activities are completed.

In another type of graph one can try to find an answer to the following question: "Does the number of edited lines of the code depend on the time the developer spent using a web browser when writing it?" As we can see in Figure 6, there is no dependency. However, we can postulate that when this developer spends more than 20 minutes using a browser, they are probably not writing a code.

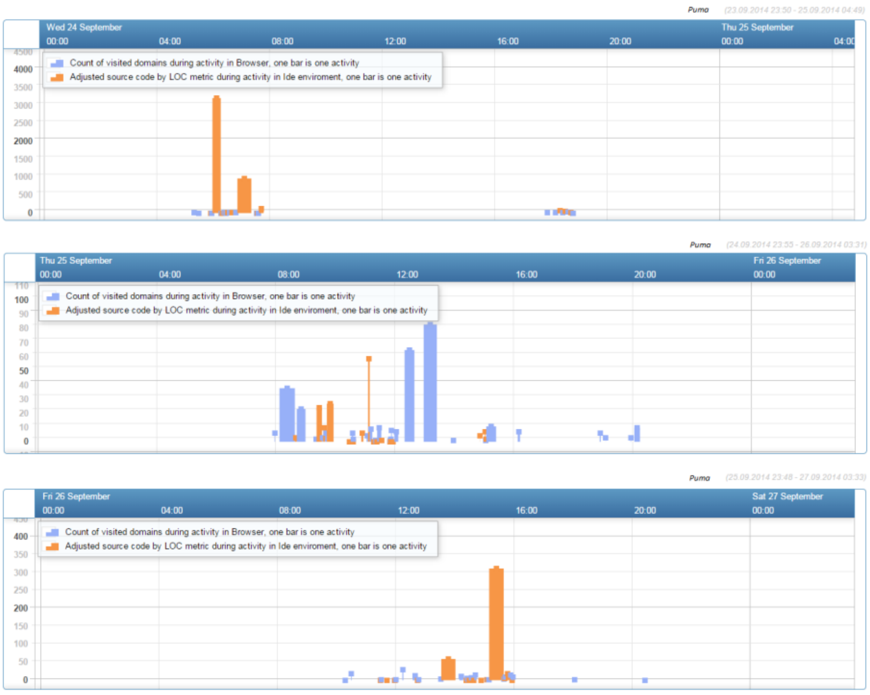

Figure 5
Number of visited domains together with number of edited lines by developer "Puma", both cumulated per continuous activity, within three subsequent days

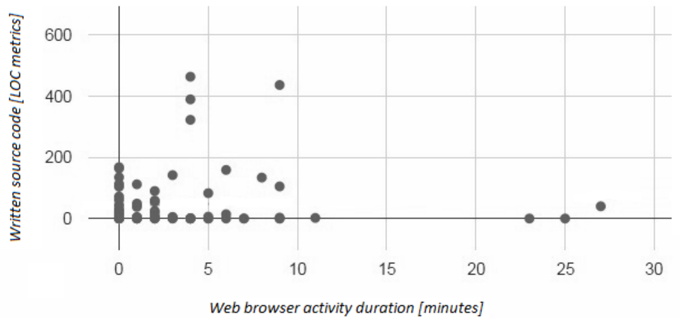

Figure 6
Scatter graph, where every point represents the duration of the web browser activity which was immediately followed by a depicted amount of source code changes in IDE

Conclusions

The work presented in this paper contributes to the methods of better understanding a software developer’s activities. Understanding them is vitally important during software evolution. In development, a project manager can better reassign tasks and allocate resources. In maintenance, any revealed code dependencies suggest places for a possible remedy. Activities emerge as high level outcomes of exploratory analysis of low level data, logging a developer’s rather elementary actions. These elementary actions are such that many of them occur within small periods of time. As a result, logging them also requires some kind of pre-processing, since without reduction, storing them would be too prohibitive. This is essentially one of the characteristics of Big Data.

Software development is a creative process, which together with maintenance are the two most extensive elements of a software project. A software developer, as an executor committed to bringing specifications and ideas into usable product, is often difficult to monitor, evaluate and reason upon identified information, in order to support the particular software project meeting the deadlines on time and in acceptable quality. As we pointed out, this motivates current research in software engineering to employ new approaches for monitoring software development directly on the level of interaction events performed by developers in tools that they use during their work.

Our contribution is to transform low level logged data into higher level information on activities and then to attempt various schemes of visualisation in order to facilitate better understanding of the data. We employ interactive visualisation in a customizable manner to find answers to various interested questions of team leaders or software developers. We understand visualisation as a tool to open the way to explorative analysis of a software developer’s activity. We described explorative analysis as a three step process, depicting the importance of a proper visualisation tool. We devised a number of simple graph schemes. They visualise the partially processed data and allow a human analyst to make assumptions by generalising what they see in visualising graphs. We applied the devised graph schemes to data gathered within the project PerConIK that we have been participating in over the last four years. We devised and implemented the interactive visualisation tool IVDA. User-defined graphs generated by the tool show information about activities, developers' actions and their metrics, which a team leader may use to identify possible causes of problems, delays in delivering results, anomalies and trends in activities of developers in a team. Furthermore, this tool may help any scientist to endorse or refute assumptions about a software developer’s activity. By inspecting the visualised data, the analyst is able to gain much useful information on the software development of a particular project.

Since data that can be received by logging is way below the level any analysis of software development should be performed, one challenge for future work is to further automate identification of developers' activities from interactive events.

Such identification, especially when accomplished in real-time during a developer's work, may support not only a team leader or scientist in answering hypotheses, but also developers themselves before committing their results to revision control systems or when describing completed tasks in task management systems. Developers often work on several development tasks, more or less arbitrarily, that may tangle them into composite change (commit) in the end. Such tangled changes in an RCS are difficult to review, describe, or merge with other changes. Existing approaches attempt to identify and untangle such changes by analysing a static snapshot of commit history. However, we argue that the approach of doing so in real-time is required, to prevent a software developer from tangling their changes even before committing them to an RCS.

Acknowledgement

This work was supported by the Research and Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF. This work was also partially supported by the Scientific Grant Agency of The Slovak Republic, Grant No. VG 1/0752/14.

References

[1] Bieliková, M., Polášek, I., Barla, M., Kuric, E., Rástočný, K., Tvarožek, J., Lacko, P.: Platform Independent Software Development Monitoring: Design of an Architecture. In: Proc. of 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014) Springer-Verlag, 2014, pp. 126-137

[2] Burch, M., Diehl, S., Weissgerber, P.: EPOSee – A Tool For Visualizing Software Evolution. In: Proc. of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005) IEEE, 2005, pp. 1-2

[3] Collberg, C., Kobourov, S., Nagra, J., Pitts, J., Wampler, K.: A System for Graph-based Visualization of the Evolution of Software. In: Proc. of the 2003 ACM Symposium on Software Visualization (SoftVis '03) ACM, 2003, pp. 77-86

[4] Erra, U., Scanniello, G., Capece, N.: Visualizing the Evolution of Software Systems Using the Forest Metaphor. In: Proc. of the 16th International Conference on Information Visualisation (IV 2012) IEEE, 2012, pp. 87-92

[5] Garzo, A., Benczur, A. A., Sidlo, C. I., et al.: Real-time Streaming Mobility Analytics. In: Proc. of the 2013 IEEE International Conference on Big Data, IEEE, 2013, pp. 697-702

[6] Gu, Z., Schleck, D., Barr, E. T., Su, Z.: Capturing and Exploring IDE Interactions. In: Proc. of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014) ACM, 2014, pp. 83-94

[7] Hanjalic, A.: ClonEvol: Visualizing Software Evolution with Code Clones. In: Proc. of the 1st IEEE Conference on Software Visualization (VISSOFT 2013) IEEE, 2013, pp. 1-4

[8] Hindle, A., Ming Jiang, Z., Koleilat, W., Godfrey, M. W., Holt, R. C.: YARN: Animating Software Evolution. In: Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007) IEEE, 2007, pp. 129-136

[9] Kapec, P.: Knowledge-based Software Representation, Querying and Visualization. In: Information Sciences and Technologies. Bulletin of the ACM Slovakia, Vol. 3, No. 2, Slovak University of Technology Press, Bratislava, Slovakia, 2011, pp. 1-11

[10] Kersten, M., Murphy, G.C.: Using Task Context to Improve Programmer Productivity. In: Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14) ACM, 2006, pp. 1-11

[11] Konôpka, M., Bieliková, M.: Software Developer Activity as a Source for Identifying Hidden Source Code Dependencies. In: Proc. of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2015) Springer-Verlag, LNCS 8939, 2015, pp. 449-462

[12] Konôpka, M., Návrat, P.: Untangling Development Tasks with Software Developer's Activity. To appear in: Proc. of the 2nd International Workshop on Context for Software Development (CSD 2015) in companion with the 37th International Conf. on Software Engineering (ICSE 2015) IEEE Press, Florence, Italy, 2015, 2 p.

[13] Kuric, E., Bieliková, M.: Estimation of Student's Programming Expertise. In: Proc. of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14) ACM, 2014, Article No. 35

[14] Maalej, W., Fritz, T., Robbes, R.: Collecting and Processing Interaction Data for Recommendation Systems, in Recommendation Systems in Software Engineering, Robillard, M. P., Maalej, W., Walker, R.J., Zimmermann, T. (Eds.) Springer Berlin Heidelberg, 2014, pp. 173-197

[15] Maletic, J. I., Marcus, A., Collard, M. L.: A Task Oriented View of Software Visualization. In: Proc. of the 1st Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002) IEEE, 2012, p. 32

[16] Minelli, R., Mocci, A., Lanza, M.: Visualizing Developer Interactions. In: Second IEEE Working Conference on Software Visualization (VISSOFT 2014) IEEE, 2014, pp. 147-156

[17] Novais, R., Lima, C., de F. Carneiro, G., Paulo, R. M. S., Medonca, M.: An

Interactive Differential and Temporal Approach to Visually Analyze Software Evolution. In: Proc. of 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011) IEEE, 2011, pp. 1-4

[18] Omoronyia, I., Ferguson, J., Roper, M., Wood, M.: Using Developer Activity Data to Enhance Awareness during Collaborative Software Development. In: Computer Supported Cooperative Work (CSCW) Vol. 18, Issue 5-6, Springer Netherlands, 2009, pp. 509-558

[19] Rástočný, K., Bieliková, M.: Enriching Source Code by Empirical Metadata. In: Proc. of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14) ACM, 2014, Article No. 67

[20] Roehm, T., Maalej, W.: Automatically Detecting Developer Activities and Problems in Software Development Work. In: Proc. of 33rd International Conference on Software Engineering (ICSE 2012) IEEE, 2012, pp. 1261-1264

[21] Roehm, T., Gurbanova, N., Bruegge, B., Joubert, C.: Monitoring User Interactions for Supporting Failure Reproduction. In: Proc. of the 21st IEEE International Conference on Program Comprehension (ICPC 2013) IEEE, 2013, pp. 73-82

[22] Sekerák, L.: Interactive Visualization of Developer's Actions. In: Proc. of the 11th Student Research Conf. on Informatics and Information Technologies (IIT.SRC 2015) Slovak University of Technology Press, Bratislava, Slovakia, 2015, pp. 281-286

[23] Shneiderman, B.: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: Proc. of the 1996 IEEE Symposium on Visual Languages, IEEE, 1996, pp. 336-343

[24] Zhang, J., Huang, M. L.: 5Ws Model for Big Data Analysis and Visualization. In: Proc. of 16th IEEE International Conference on Computation Science and Engineering (CSE 2013) IEEE, 2013, pp. 1021-1028