Mining and Extraction of Personal Software Process measures through IDE Interaction logs

Alireza Joonbakhsh
CSE and IT Department, Shiraz University
Shiraz, Iran
ac_ali2582@yahoo.com

Ashkan Sami
CSE and IT Department, Shiraz University
Shiraz, Iran
sami@shirazu.ac.ir

ABSTRACT

The Personal Software Process (PSP) is an effective software process improvement method that heavily relies on manual collection of software development data. This paper describes a semi-automated method that reduces the burden of PSP data collection by extracting the required time and size of PSP measurements from IDE interaction logs. The tool mines enriched event data streams so can be easily generalized to other developing environment also. In addition, the proposed method is adaptable to phase definition changes and creates activity visualizations and summarizations that are helpful for software project management. Tools and processed data used for this paper are available on GitHub at: https://github.com/unknowngithubuser1/data.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; *IDE*; Mining Software Repositories, Enriched Event Streams; • **Software and its engineering → Software development process management**; PSP

KEYWORDS

IDE, Personal Software Process

ACM Reference format:

Alireza Joonbakhsh and Ashkan Sami. 2018. Mining and Extraction of Personal Software Process measures through IDE Interaction logs. In *Proceedings of MSR '18: 15th International Conference on Mining Software Repositories (MSR '18). ACM, New York, NY, USA, 4 pages.* https://doi.org/10.1145/3196398.3196462

1 INTRODUCTION

The Personal Software Process (PSP) [1] is a framework that follows specific planning and quality principles and uses measured processes that help software engineers enhance their development performance by tracking their personal plans and actual development of code. PSP relies heavily on collection of vast amounts of data about development time, product size and details about defects, all of which are hard to collect.

Manual data recording produces overheads and causes many distractions due to context switching [2]. Thus, numerous tools and software have been developed to facilitate recording programmers' planning and performance stats with minimal context switches and manual efforts.

PSP tools provide various set of features to collect logs like development time, program/code-size measures and defect summaries. However, the functionality of these tools concerning time logging happen mostly manual or semi-auto. In addition, changes to processes and preferred metrics can present difficulties since their data collection capabilities are based on a specific standard or purpose.

This paper aims to incorporate enriched event data streams [3] (which resemble IDE interactions by the user) into PSP principles in order to mine some of the summaries required by PSP processes without interfering with the user through GUI. The results and the method employed in this study show how time logs for coding and debugging phases of PSP process can be extracted by classifying and mining IDE interaction events. Furthermore, analysis of software codes obtained from Simplified Syntax Trees [4] (SST) and Edit Events provide fully automated and detailed LOC measurements for each coding class grouped by development session and the programmer.

The rest of paper is as follows: Section 2 reviews related works based on two categories of tools and presents their weaknesses and strengths. Section 3 discusses the main differences between the proposed method and previous works. It also explains techniques and mining process thoroughly. Section 4 presents results and discussion and Section 5 concludes the paper.

2 LITERATURE REVIEW

As stated in the introduction, different tools and software like Process Dashboard [5], PSPA [6] and Hackystat [7] to name a few exist that assist developers in appliance of PSP. These tools differ from each other in their accessibility, collected observations and environments. Based on the required amount of direct influence of user and their level of manual input, two categories of these tools are reviewed in this section.

MSR '18, May 28--29 2018, Gothenburg, Sweden

https://doi.org/10.1145/3196398.3196462

2.1 Manual Data Collection for PSP

PSP Studio [8] aids in PSP data measurement, and stores them in a historical database. It comprises a wizard which allows the user to plan their process before actual development. However this tool needs direct and manual input.

PSP Palm [9] is another PSP tool that is intended for Palm OS. The start and stop of the recording in this tool is based on stopwatch timers that entail user manipulation.

2.2 Automatic Data Collection for PSP

Hackystat [7] automates the collection of various metric data by introducing "sensors" that are installed on developers' machine. Some of these metrics include size, time effort and defect. Although sensors make the data recording unobtrusive, the variety that exists among metrics and development tools imply the need for development of these sensors and their registration on client machines.

PSPA [6] uses "Knowledge Management Portal" as its metaphor and gathers data from its agents which function in IDE as plugins.

The proposed method advocates storing and mining of IDE interaction logs as a generic tool for extracting unrestricted and flexible facts about software processes as well as PSP data. While previous PSP automation tools in practice emphasize building rigid agents and sensors for data collection, the proposed method is not highly dependent on the environment and the performed analysis.

3 CURRENT WORK

3.1 Input Dataset

The March 1, 2017 Release of KaVE dataset belonging to MSR 2018 Challenge [10] consisting of over 11 Million enriched IDE event data streams was used for this study.

3.2 Mining Approach

3.2.1 Inputs. The inputs to the mining workflow are interaction logs of each developer containing information such as the time at which the event was triggered, its duration and other context information that designate the exact type of the action by the developer (i.e.: Command, Window, Debugger, etc…).

3.2.2 Method. The process applied on the input only involves execution of SQL queries in order to achieve the results in a timely manner.

3.2.3 Output. The output is a list of time intervals that specify the exact times the developer started or ended a specific development phase like Coding or Debugging. These intervals can contain up to *P* minutes of break between activities. *P* is an input parameter that defines the maximum gap allowed between event logs for them to be linked. Similar studies such as [12] consider five minutes of inactivity as a break. But in this study in order to disallow packing of coding phases that contain short bursts of debugging inside them, this threshold was decreased and a maximum gap of 180 seconds were used.

3.3 Data Preparation

In order to prepare the dataset for time based processes, a C# application (tool) was developed that iterates over all user zip files and all JSON files and converts the dataset to a relational one.

As an optional step, this phase also filtered out co-triggered event logs (with the same timestamp and details) to limit the size of the resulting dataset. It also shortened some of the names of commands by removing the GUID part of them to keep them readable. For example, command name *"{X-Y-Z-T}:Text.Paste"* is replaced with *"Text.Paste"*.

The result is a raw SQLServer database containing both generic and context information of events with enough details saved for the next phase of mining process. This database along with the tools and queries used to prepare and process the data are available for replication. Fig. 1 Shows the relation between Tables for Events, Sessions, Files (User Zip Files) and Users on a diagram. Each event belongs to a session and each session also belongs to File (User zip). File table also has a one to many relationship with User, indicating that each file should represent activities of one user.

For each type of event, a dedicated Table exists that contains its context information, but to be able to process these information faster with minimal number of joins between tables, a column named "Details" is added to the generic Events table that enables for an instant look over context data.

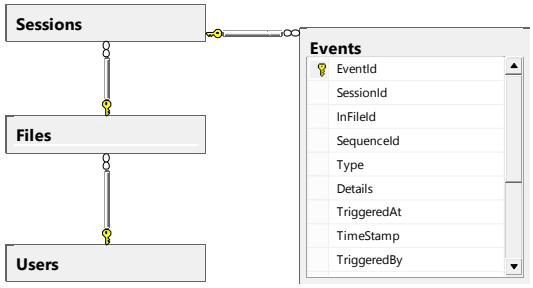

Figure 1: Database diagram illustrating table relationships.

3.4 Mining time logs

3.4.1 Total activity time. This measure takes the entire amount of time that developer had any interaction with IDE into consideration. These interactions may include text editing, navigating between documents, testing, debugging the project and so on…

Initially, in order to extract the total time that each developer spent working, intersecting logs that happened not more than *P* minutes apart (by taking their timestamp and duration into account), where grouped into one single continuous interval.

Fig. 2 shows a graphical representation of different inputs and the output of this task (having *P*=3 minutes) in relation to each other, displaying a time window from 12:00 to 12:08. Each rectangle represents a log that presumably starts and ends some

time in this window. Arrows denote the resulting interval from packing (collapsing) logs together. In "Example 1", events with ids 1, 2 and 3 were joined together as they are not more than three minutes apart. But event 4 does not match this criteria with the first group, so it belongs to another interval. Some events in the data set might have overlaps with each other. An instance of this situation is shown in "Example 2".

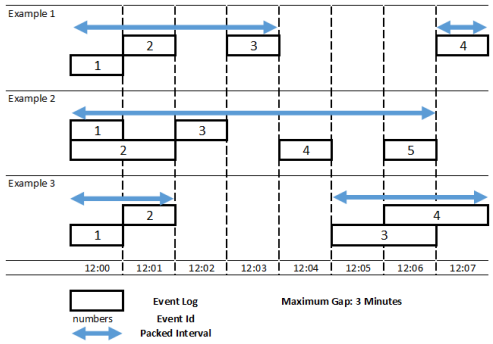

Figure 2: Examples of Interval packing with consideration of gaps up to 3 minutes.

SQL queries were developed to calculate the measures presented. These queries primarily used SQL aggregation functions and window functions like ROW_NUMBER(). To grasp a clearer picture over the step-by-step execution, see Table 1.

Table 1: Step-by-Step procedure to pack intervals using SQL queries

Step No.	Explanation
1	Select event id, session id, start time and ending time for each event related a user. The ending time for an event is calculated by adding its duration to its starting time. To allow for gaps, the ending time should be extended by value of *P*.
2	Separate starting times and ending times into two rows, start times and ending times having t=+1 and t=-1 respectively.
3	Order all the rows first by their time, then descending by t (start=+1, end=-1). This places the logs that are contained inside other logs between them, making the longest log stand for the union of them. Since each event was extended by *P* seconds, events that are not more than *P* seconds apart, overlap.
4	Mark each unique interval with a group number and calculate the time difference between its start and end time. Also subtract *P* seconds from each of their time spans.

3.4.2 Coding time. Extracting the exact amount of time a developer spent in coding phase, requires first selecting logs associated with textual commands, completion events and edit events, and subsequently collapsing them using the mentioned interval packing method. Specifically, constraints below classify a log as a coding activity log:

- Event is a command event and its name matches either of the rules: *"%Edit.%"* or *"%TextControl.%"*.
- Event is an Edit Event.

3.4.3 Debugging time. The stated technique is also applied to extract a rough estimate for debugging times for "Debugger Event" logs. However results may vary based on the value chosen for *P* and the definition of debugging events. Picking considerably higher *P*'s will merge more distant events as an interval and cause the Debugging times to ramp up.

3.4.4 Activity Visualization. The output of methods discussed in this section can directly be used to create performance charts and timelines. Fig. 3 and Fig. 4 illustrate visual presentations of a developers activities on a timeline both separated by activity type and the name of document/class they have been working on. Each bar in the center of timeline shows an activity which is classified as Debugging, Coding or Total activity. For example on Fig. 4, the user worked on "MainForm.cs" for 73 minutes, and continued working on "ApplicationSettings.cs" after a 24 minute break.

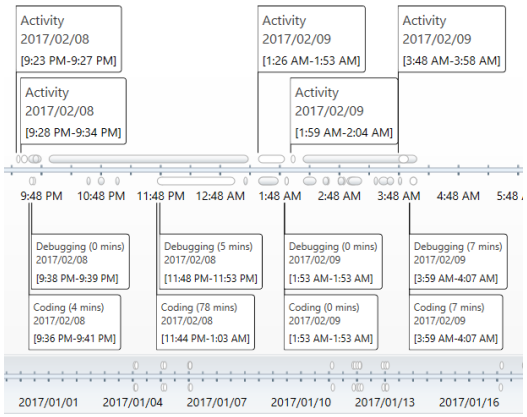

Figure 3: Visualization of a developers' Activity on a timeline.

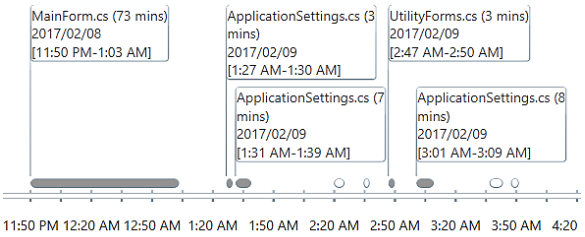

Figure 4: Timeline showing activity time spent on each document.

3.5 Mining size measures

The measurements for program size that are required in PSP include: Base, Modified, Added, Deleted, Reused, Total New & Changed, Total LOC, Total New Reused, Total Object LOC.

Most of these measures can be gathered by applying a Diff algorithm over different versions of a document. Yet some like "Reused" require more context information about the codes that are taken from a reuse library.

The dataset used in this study stores codes in form of SST. While the number of nodes in a tree can be used as measurements of product size, it is also possible to reconstruct the program codes of the product and calculate size measures with the preferred and precise definition of LOC.

The devised method employed "SSTPrintingVisitor" (which visits nodes in a tree recursively and prints codes into an output) together with "DiffPlex" [11] package (an open-source C# library that generates textual diffs) in order to compare the Codes of SSTs in textual format and extract total size measures (LOC Added, Modified and Deleted) for each class as shown in Table 3.

4 RESULTS

This method of extracting time logs allows the developers to have a more flexible and iterative software process and be able to switch back and forth between phases frequently.

Table 2 contains sample outputs that summarize the time (in minutes) spent on each PSP phase in five days of development from four different developers in the dataset. These developers were randomly chosen from users that had at least five days of activity. The selection also contains developers with different education levels: Master (259 and 256), Autodidact (249 and 247), Bachelor (206).

Results of a study conducted in 2013 showed that developers spend as much time debugging as they did producing (coding) [13] (Specifically 50-50%). Applying the presented method over all of the 81 users in the dataset shows that on average each user spends 45% of their total activity time coding and 20% of it debugging, while the rest consists of other types of IDE Events like Navigation, Document and simple Activity Events.

Table 2: Output of Mining Process for four developers separated by each PSP Phase

User Id	Coding	Debugging	Total Activity (m)
259[1]	194	184	646
249[2]	487	113	1163
206[3]	128	106	823
247[4]	234	420	971

[1] Profile Id: 9377aee4-055c-423e-a841-6cb8512a4f60

[2] Profile Id: f4386b50-076a-445c-9256-59158ed351fa

[3] Profile Id: 78c2056b-bc39-456d-a152-1a1b31b75486

[4] Profile Id: 153b500c-eb39-45cb-8a5f-8ba9aca0f5d0

Table 3: Summary of size measures grouped by developer and Class Name

User Id	Class Name	A	M	D	Coding Time (h)
256[5]	AuthController	19	12	0	0.9
	AccountController	113	107	88	4.5
249	Node	2371	1753	2270	13.4
	Test.Program	94	76	85	7.4

On the other hand, some inaccuracies regarding coding time may exist. These inaccuracies can be caused by errors in how duration times for "Edit Events" were collected in the dataset, but they can be tweaked by taking the time intervals of each document (as in Fig. 4) into account.

5 CONCLUSIONS

In summary, this paper proposed a method for mining IDE interaction logs that allow for a non-intrusive generic data collection and lets developers have a more flexible and iterative software process. Results showed how this method is adaptable to changes and provides timeline visualizations and summarizations that are helpful for managing software projects. Tools and processed data used for this paper are freely available on GitHub at [14].

REFERENCES

[1] W. Humphrey. 2005. PSPSM: A Self-Improvement Process for Software Engineers, Addison-Wesley Professional.

[2] P. Ferguson, W. S. Humphrey, S. Khajenoori, S. Macke and A. Matvya. 1997. Introducing the Personal software process: Three industry cases. IEEE Computer, Vol. 30, No 5, (May 1997), pp. 24-31

[3] Proksch, S. et al. 2017. Enriching in-IDE process information with fine-grained source code history. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (Feb. 2017).

[4] Proksch, S. et al. 2016. A dataset of simplified syntax trees for C#. Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16 (2016).

[5] Tuma Solutions LLC. Process Dashboard. Retrieved from: http://www.processdash.com.

[6] Sison, R. 2005. Personal software process (PSP) assistant. 12th Asia-Pacific Software Engineering Conference (APSEC'05) (2005).

[7] Hackystat Development Team. Hackystat. Retrieved from: http://code.google.com/p/hackystat/.

[8] 97 Design Studio Team. PSP Studio. Retrieved From: http://www.cs.etsu.edu/psp/

[9] Leon Ioannides. PSP Palm. Retrieved From: http://innovexpo.itee.uq.edu.au/2003/exhibits/s360903/

[10] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2017. Website of the MSR Challenge Proposal. http://www.kave.cc/msr-mining-challenge. (2017).

[11] DiffPlex - Netstandard 1.0 C# library to generate textual diffs. Retrieved From: https://github.com/mmanela/diffplex/

[12] Minelli, R. et al. 2016. Taming the IDE with fine-grained interaction data. 2016 IEEE 24th International Conference on Program Comprehension (ICPC) (May 2016).

[13] Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T. 2013. Reversible debugging software. Cambridge Judge Business School; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf.

[14] https://github.com/unknowngithubuser1/data

[5] Profile Id: SarojEkka