Mining Micro-practices from Operational Data

Minghui Zhou[1], Audris Mockus[2]
[1]School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies, MoE
Beijing 100871, China
[2]Department of Electrical Engineering and Computer Science, University of Tennessee
1520 Middle Drive Knoxville, TN 37996-2250
Avaya Labs Research, 233 Mt Airy Rd, Basking Ridge, NJ 07920, USA
zhmh@pku.edu.cn, audris@utk.edu

ABSTRACT

Micro-practices are actual (and usually undocumented or incorrectly documented) activity patterns used by individuals or projects to accomplish basic software development tasks, such as writing code, testing, triaging bugs, or mentoring newcomers. The operational data in software repositories presents the tantalizing possibility to discover such fine-scale behaviors and use them to understand and improve software development. We propose a large-scale evidence-based approach to accomplish this by first creating a mirror of the projects in the open source universe. The next step would involve the inductive generalization from in-depth studies of specific projects from one side and the categorization of micro-practices in the entire universe from the other side.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*process metrics*

General Terms

Human Factors,Measurement

Keywords

micro-practice, fine-scale activity pattern, operational data

1. INTRODUCTION

Best practices in software engineering have always been the goal of practitioners and researchers striving to improve software productivity and quality. Rich literature on the subject, e.g., [6, 19, 3, 11], considers system-level design, schedules, and defined processes as mechanisms to assure the projects' success. However, it is not always clear to what extent and how these practices affect individual behavior. Also, methodologies and processes are generally used as templates, but in practice they are implemented, customized, adapted, and evolved to suit a particular context.

In contrast, micro-practices are actual fine-scale patterns of activity used by developers or projects to accomplish basic software development tasks, such as finding and fixing bugs, testing, or mentoring newcomers. They are usually undocumented (or incorrectly documented [25]) and are learned through practice of doing the tasks. In other words, they represent tacit knowledge [18] implicitly defined in the way individuals act and projects operate.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2666611

We argue for the need to discover and compare micro-practices as means to better understand and improve software development. The study of micro-practices is likely to provide the basis for valuable insights into when, how, and how well methodologies and processes are used and help choose micro-practices that are appropriate for developers' particular goals and contexts.

Operational support tools such as issue tracking system (ITS), version control system (VCS), mail system, forum, etc., have been extensively adopted by software projects, and the operational data produced by these tools is increasingly being used to observe how people develop software and how they interact with each other to accomplish the tasks [17, 10, 24, 7, 21, 13]. In other words, software engineering community has accumulated enormous experiences of software micro-practices that are latent in vast open source and commercial software repositories. Analysis of this trove of data should replace the tradition of deductively declaring new silver bullets to improve software productivity and quality. The methodological infrastructure and the wide availability of data make it feasible to build a large-scale extraction and characterization of micro-practices.

The main idea behind the concept of micro-practice is that it reflects what developers actually do in specific circumstances, not necessarily what they say they do or they think they do. In particular, VCS, ITS, communication media, came into being because these tools were needed by developers (not mandated by their management) to cope with the complexity of their development and maintenance work. It is, therefore, likely that these repositories reflect how people actually complete their tasks in a bewilderingly varied set circumstances of their regular work.

We believe that some fundamental questions can be answered only by considering the entire universe of publicly available source code and its history [15] and, therefore, we propose to utilize large-scale software repository data to discover and categorize the micro-practices across projects. However, to investigate a relative new phenomenon, we may want to achieve an initial understanding through analysis of a small number of cases in restricted contexts. Without a solid research base to date, analysis on a small number of cases could be deemed appropriate as such "revelatory cases" [23] may provide the required rich insight.

2. WHY MICRO-PRACTICE?

The practices in software projects were discovered starting by direct observation of the actual software projects. A typical case is Brooks' observations based on his experiences at IBM while managing the development of OS/360 [4]. He had added more programmers to a project falling behind schedule, a decision that he would later conclude had, counter-intuitively, delayed the project even further. Later qualitative methods such as interviews and surveys were extensively employed to study project practices. For example, Curtis et al. [6] interviewed personnel from 17 large projects and found that the thin spread of application domain knowledge, fluctuating and conflicting requirements, and communication bottlenecks and breakdowns, affected software productivity and quality through their impact on cognitive, social, and organizational processes. There has been interest in finer-scale practices as well, e.g., personal software process (PSP) [11] proscribes certain behaviors and suggests ways to keep track of them. It does not appear to be used by the vast majority of developers, perhaps because of the overhead of data gathering and discipline needed to follow PSP.

There have been attempts to conduct controlled experiments investigating developer activity patterns. For example, Perry et al. reported on two experiments to discover how developers spend their time [19]. They described how noncoding activities can use up development time and how even a reluctance to use e-mail can influence the development process.

The lessons we could draw from these early studies are that software development is a knowledge intensive activity and that a large number of potentially confounding factors exist, especially related to human elements [2, 6, 5]. Perhaps that's why software engineering is often found to be different from traditional science or engineering disciplines. In particular, the course of action in most sciences when faced with a question of opinion is to obtain experimental verification, but software engineering disputes are not usually settled that way [2]. Besides the above mentioned reasons, the expense of attempting to do controlled studies in an industrial environment involving medium or large-scale systems accounts for the reliance on opinions [2]. Moreover, despite the effort to conduct controlled studies, the data from experiments rarely apply to the bigger questions of what works in practice. For example, a multiple case study found very large cost and process differences among four companies implementing an identical small web document management system, but the outcomes did not suggest a positive correlation between cost, process, and outcomes [1].

Fortunately, today the large volume of operational data accumulated in software repositories allows us to answer a number of important questions without the excessive cost of conducting experiments. Each development activity typically leaves a digital trace, therefore all the traces recorded in these repositories constitute the digital history of software development. In particular, many of the digital traces represent artifact-mediated communication, as illustrated in, e.g., [25]. There is, therefore, a good chance that we could understand how people work with each other through mining these artifacts.

Learning the practices from the historical data is an ongoing activity and it needs to expand and become more comprehensive. A variety of metrics were derived from operational data to measure project practices. For example, Herbsleb et al. found that the time it takes to complete distributed tasks is almost three times longer than for co-located tasks [10]. We found earlier that it took three years for a developer to become fluent at central tasks in a big project [24]. However, almost invariably, published significant relationships tend to become less significant or disappear once more data is collected [12]. While such studies have not been done in software engineering, the results are likely to be similar because of the complex nature of software development which involves human factors that are difficult to measure and greatly vary between projects [20].

In summary, the observed relationship often changes when project context varies, and we never fully control the variations of a project context. In other words, there is no lack of successful practices, what lacks is the capability to quantify and reproduce the practices employed by different individuals and projects.

We propose, therefore, to study how to derive micro-practices from operational data, with an emphasis on quantitative approaches and reproducible mechanisms. We define micro-practices as fine-scale activity patterns used by software projects to accomplish key tasks. We emphasize micro level to guarantee that the practices could be observed and quantified from the operational data. For the similar task, different projects or different individuals may or may not have the similar practice, depending on the specific project context they have. Therefore, first, in order to understand the micro-practices used in projects, it's important to quantify the project landscape and the corresponding practices. Second, in order to test reproducibility of the practices, we need to investigate a large-scale dataset. Without sufficient data, it may be difficult to discover cross-project interaction and interdependence practices (e.g., cross project reuse, developer turnover), and to build models (e.g. effort or code recommendation models) that work well in a variety of projects.

3. RESEARCH TRAJECTORY

In order to quantify micro-practices that could reproduce the success of software projects, it's important to start from collecting the data and understanding the data law, e.g., how the data were generated [16].

Based on the data, we could try a variety of approaches to locate micro-practices depending on the questions we want to address. One is to start from specific projects, and the other is to target the whole universe projects. Both are driven by specific research questions, and have their respective advantages.

3.1 Building Large-Scale Repository

The publicly accessible data recorded in operational support tools of millions of Open-source Software (OSS) projects bring us an opportunity to investigate software engineering best practices across projects. Such scale of data (the so-called big data) offers us the opportunity to understand the world with new tools and new insights along with challenges [14]. For example, if we want to measure productivity using the number of commits, it's important to know how developers commit code in that project. Some developers tend to commit code after a complete test, but others may use an iterative approach. If we don't identify that difference, we may end up adding numbers representing different types of activities bringing misleading results. There are numerous other variations in micro-practices that we need to identify and adjust for when using operational data.

There are a number of attempts to collect and share open source project data, e.g., FLOSSMOLE (http://flossmole.org), GitHub Archive (http://www.githubarchive.org), gittorrent[8]. We are also building a public repository (https://passion-lab.org) that records the history of open source universe.

Building a public repository is a tedious and long-term commitment. There are at least two very basic tasks that have to be completed. The first task is to retrieve data from Internet. The various types of repositories (e.g., cvs, svn, git, and hg for VCS; jira, bugzilla for ITS), the project administrator's policies (e.g., banning the IP addresses that do the data retrieval), the network bandwidth, the huge amount of changes, issues in a project (e.g., GitHub has more than 12 million repositories), and so forth, all make the retrieval difficult. The second task is to standardize data. It is a lot of work to extract the raw data from the operational support tools and to standardize into formats convenient for analysis (an example is in [25]).

3.2 Analysis of a Small Number of Projects

For novel concept such as micro-practice, the relative newness of the concept, leads to a lack of a solid research base on the phenomenon. Bearing this in mind, we may be concerned with initially achieving an increased understanding of micro-practices via a small number of "revelatory cases" [23].

A good example of micro-practice area is bug triage. The goal of bug triage is to harness the incoming bug reports. Triage is of great interest for software projects because it has the potential to reduce developer effort by involving a broader base of non-developer contributors to filter and augment reported issues [22]. However, the value of contributions made by these non-developer triagers may be underestimated. Therefore it's important to understand the micro-practices of these triagers and leverage their strength to improve the project – this has been rarely studied in the literature. To reveal this new phenomenon, we chose two projects, Mozilla and Gnome, to get a deep understanding about their triage practices.

Using issue tracking data and interviews with experienced contributors we investigate ways to quantify the impact of non-developer triagers. We find the primary impact of triagers to involve issue filtering, filling missing information, and determining the relevant product. While triagers were good at filtering invalid issues and as accurate as developers in filling in missing issue attributes, they had more difficulty accurately pinpointing the relevant product, leading to a substantial fraction of incorrect assignments. In particular, Mozilla has over 85 products with strong interdependencies, e.g., the product "Core" provides base API to Firefox and Thunderbird. "Follow the stack-trace to locate the problematic product" may be a too sophisticated skill for an average non-developer, resulting in over 21% mistaken product assignments in Mozilla (with the developer error rate of 18% and non-developer error rate of 29%).

Based on this understanding, we proposed product assignment recommender (PAR) [21] to estimate the odds that a product assignment value in the ITS is incorrect. PAR learns from the past practices in ITS and performs prediction using a logistic regression model. PAR helps developers to focus on fixing real problems, and could also be used to improve data accuracy in ITS by crowd-sourcing non-developers to verify and correct low-accuracy data.

We expect that this work will highlight the importance of bug triage in software projects and will help design large multi-project studies to understand and improve triage micro-practices. In particular, it's still not clear how these micro-practices are related to the specific project context, and if these micro-practices could be reproduced in other projects. For example, while PAR is applicable in Gnome, it does not work well in Mozilla. We speculate that the reason for the difference is that Mozilla has a bigger volunteer base than Gnome (so Gnome triage tasks are mostly accomplished by its developers). Therefore, to understand the best practices across projects, it's important to understand the project landscape. For example, GitHub is being used for free storage and as a Web hosting service, and the majority of the projects are personal and inactive [13]. Any researcher may want to bear that in mind when approaching the data in GitHub and discovering micro-practices associated with personal, storage, or web-hosting projects could allow appropriate filtering for the desired analysis.

3.3 Analysis on Universe Projects

Considering the potential of large-scale data, we could target all the projects in the universe, and understand the prevalence of various practices in that universe. Below we suggest the code reuse as an example idea of how to proceed.

Source code reuse, as an important aspect of software reuse, is widely practiced in OSS projects [15]. In particular, what to select for reuse and how to get reputation through others' reuse attract developers' attention [9]. It would be good to find out what are those popular projects to help address these concerns.

As mentioned in Section 3.1, we have been collecting version control repositories for years. We retrieved repositories from code hosting sites, e.g., GitHub, Googlecode, Sourceforge, BitBucket, Launchpad, OSS communities, e.g., Apache, GNU, Eclipse, Mozilla, and other sites.

To deal with the heterogeneity of version control repositories, we use file-versions for observations[1]. When writing this paper, we have over a billion of file-versions.

Based on the file path of these file versions, we use heuristic to locate the frequently reused projects [26]. We find that frameworks and application components constitute the majority of frequently reused projects (as shown in Table 1). In particular, web frameworks, e.g., Ruby on Rails, testing frameworks like Cucumber, and editor component like TinyMC are recurring. This indicates important role of software frameworks and basic application components for software reuse.

Some tools, e.g., the one-click-forking in GitHub, may change the conventional micro-practices. For example, the task of finding good reusable components that address developers' needs well and that have proven stable and well-supported may be simplified by the forking information. In particular, the number of existing forks may indicate component's quality, and the author's reputation (judged by the number of projects, watchers, and collaborators) may serve as an indicator of stability.

Considering the extensive extent of code reuse in OSS communities, it's of interest to devote research effort to the co-evolution problems across projects, e.g., feature implementation and bug fix merging among the projects that use the same code.

[1]a file's different versions in its lifetime are all included and marked a postfix of version number, e.g. main.c/1.1 and main.c/1.2.

Table 1: Frequently reused repositories

Co-occur pair	Project	Application area
lib & mm, include & mm	linux kernel	OS
config & script, config & public	Ruby on Rails	web framework
js & langs, css & langs	TinyMCE	editor component
lib & feature	Cucumber	test framework
lib & spec	Rspec	test framework

4. CONCLUSIONS

The software community pays a lot of attention to the technological aspects of software development, sometimes at the expense of the organizational and social aspects. One often cited reason is the difficulty of quantitatively measuring people factors. Accumulated operational data produced by operational support tools provide the possibility to observe and measure software development, and we propose to mine the fine-scale practices from operational data. These micro-practices may then serve as building blocks to create better ways of using operational data.

There are numerous software development models that range from techniques to determine effectiveness of software methods, to ways to estimate developer productivity and predict software quality that rely on operational data. More diverse and more detailed repositories are bound to appear with corresponding methods and tools to analyze them in the future. The research in this area will make it possible to measure aspects of project practices in more detailed and more relevant ways and that will open new possibilities to understand and improve project productivity and software quality to satisfy the increasing demands of the rapidly changing, multicultural, and global software development.

5. ACKNOWLEDGMENT

Thanks the National Natural Science Foundation of China Grants 91118004 and 61432001, and the National Basic Research Program of China Grant 2015CB352200.

6. REFERENCES

[1] B. C. Anda, D. I. Sjøberg, and A. Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE TSE*, 35(3), May/June 2009.

[2] V. Basili and D. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–737, 1984.

[3] B. W. Boehm, B. Clark, E. Horowitz, and et al. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1(1):1–24, November 1995.

[4] F. P. J. Brooks. *he Mythical Man-Month.* Addison-Wesley, 1975.

[5] B. Curtis. Human factors in software development. Technical report, ITT Programming Technology Center, 1986.

[6] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, Nov. 1988.

[7] L. A. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *CSCW*, pages 1277–1286, 2012.

[8] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean ghtorrent: Github data on demand. In *MSR'2014*, pages 384–387, May 31-June 1, 2014.

[9] S. Haefliger, G. Von Krogh, and S. Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[10] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally-distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, June 2003.

[11] W. S. Humphrey. *Managing the Software Process. SEI series in software engineering. Reading, Mass.* 1989.

[12] J. P. A. Ioannidis. Why most published research findings are false. *PLoS Med*, 2(8):e124, August 30 2005.

[13] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian. The promises and perils of mining github. In *MSR'2014*, pages 92–101, May 31-June 1, 2014.

[14] K. Michael and K. W. Miller. Big data: New opportunities and new challenges [guest editors' introduction]. *Computer*, 46(6):22–24, 2013.

[15] A. Mockus. Amassing and indexing a large sample of version control systems: towards the census of public source code history. In *MSR'2009*, May 16–17 2009.

[16] A. Mockus. Engineering big data solutions. In *FOSE, ICSE 2014*, Hyderabad, India, June 1–6 2014.

[17] A. Mockus, R. F. Fielding, and J. Herbsleb. A case study of open source development: The Apache server. In *22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 4-11 2000.

[18] I. Nonaka. A dynamic theory of organizational knwledge creation. *Organizational Science*, 5(1):14–37, February 1994.

[19] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, pages 36–45, July 1994.

[20] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, and R. Paige. Large-scale complex it systems. *Commun. ACM*, 55(7):71–77, July 2012.

[21] J. Xie, Q. Zheng, M. Zhou, and A. Mockus. Product assignment recommender. In *ICSE'2014 Research Demonstration*, pages 556–559, Hyderabad, India, June 1–6 2014.

[22] J. Xie, M. Zhou, and A. Mockus. Impact of triage: a study of mozilla and gnome. In *ESEM 2013*, pages 247–250, Baltimore, Maryland, USA, Oct 10–11 2013.

[23] R. K. Yin. *Case Study Research: Design and Methods. Fourth Edition.* SAGE Publications, California, 2009.

[24] M. Zhou and A. Mockus. Developer fluency: Achieving true mastery in software projects. In *ACM SIGSOFT / FSE*, pages 137–146, Santa Fe, New Mexico, November 7–11 2010.

[25] M. Zhou and A. Mockus. What make long term contributors: Willingness and opportunity in OSS community. In *ICSE 2012*, pages 518–528, Zürich, Switzerland, 2012.

[26] J. Zhu, M. Zhou, and A. Mockus. The relationship between folder use and the number of forks: A case study on github repositories. In *ESEM'14*, Sep 18–19 2014.