Software Maintenance at Commit-Time

Mathieu Nayrolles
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
mathieu.nayrolles@concordia.ca

***Abstract*—The maintenance and evolution of complex software systems account for more than 70% software's life cycle. Hundreds of papers have been published with the aim to improve our knowledge of these processes in terms of issue triaging, issue prediction, duplicate issue detection, issue reproduction and co-changes prediction. All these publications gave meaning to the millions of issues that can be found in open source issue & project and revision management systems. Context-aware IDE and think tank in open source architecture open the path to approaches that support developers during their programming sessions by leveraging past indexed knowledge and past architectures. In this report, we present four approaches: BUMPER, JCHARMING, PRECINCT, BIANCA. When combined these approaches (i) provide the possibility to search related software artifacts using natural language, (ii) accurately reproduce field-crash in lab environment, (iii) prevent the introduction of clones / issues at commit time.**

I. Introduction

Software maintenance activities such as debugging and feature enhancement are known to be challenging and costly [1]. Studies have shown that the cost of software maintenance can reach up to 70% of the overall cost of the software development life cycle [2]. Much of this is attributable to several factors including the increase in software complexity, the lack of traceability between the various artifacts of the software development process, the lack of proper documentation, and the unavailability of the original developers of the systems.

More than three decades of research in different fields such mining software repository, default prevention, clone detection, program comprehension or software maintenance aimed to understand better the needs of and challenges faced by developers when entertaining a program. Researchers have proposed hundreds of approaches and tools to improve the maintenance processes. Yet, the large adoption of these tools by the practitioners remains limited [3]–[7]. Factors that prevent such an adoption are still an open question. Nevertheless, based on developers interviews, several hypotheses have been formulated. For example, developers are known to have trust issues with statistical models based on process or code metrics. Another hypothesis for the limited adoption of software maintenance approaches is the lack of integration with developers' workflow. Finally, developers also express concerns regarding the numbers of warnings, the general heaviness of information provided by software maintenance tools and the lack of clear corrective actions to fix a given warning.

In this thesis, we propose to address some of the issues mentioned above by focusing on developing techniques and tools that support software maintainers at commit-time. As part of the developer's workflow, a commit marks the end of a given task or subtask as the developer is ready to version the source code. Commits are bite-sized units of work that are potentially ready to be shared with the rest of the organization [8].

We propose a set of approaches in which we intercept the commits and analyze them with the objective of preventing unwanted modifications to the system. By doing so, we do not only propose solutions that integrate well with the developer's work flow, but also there is no need for software developers to use any other external tools. More precisely, we propose the following contributions: (a) an aggregated bug report/bug fix repository system, (b) a clone prevention technique at commit-time, (c) a risky change detection technique at commit-time and, (d) a bug reproduction technique based on directed model checking and crash traces.

In the rest of this paper is organized as follows: In section II we describe the preliminary works we conducted to support our contributions. Then, in section III we present the way we plan to lower the resistance to the adoption of software maintenance tools by developers. Finally, in section IV we summarize our contributions.

II. Preliminary Work

Our preliminary work aimed to understand the rationale behind the low adoption of software maintenance tools by practitioners. To do so, we conducted a systematic literature review of several research fields. Indeed, to understand the problem, we had to survey not only the literature related to software maintenance but also the literature related to developers themselves: How developers work? How developers thinks? What are the thoughts processes and mental models behind software maintenance activities?

In this section, we report the findings that led us to believe that, while current method and tools are efficient to improve the software maintenance activities, they do not interact with the developers at the right time in the development process and lack some key features.

First of all, there is a lack of integration of developers workflow of most approaches ([9], [10] are some notable examples). If developers want to improve their processes using these approaches, they will have to download, install and understand them to achieve a given task. Having an external tool to perform software maintenance or creation activities do not fit into the workflow of developers (i.e., coding, testing, debugging, committing). Moreover, tools are specialized for a given tasks (i.e., feature location with a command line tool, development and testing code with an IDE, development, and testing front end code with another IDE and a browser, etc.). Using them lead to context and workspace switching which can hinder the productivity of developers [11], [12].

To avoid context and workspace switching researchers have heavily resorted to IDE plugins. IDE plugins are extensions to developers IDE that performs specialized tasks and reports its findings directly into the IDE. Unfortunately, developers report that these IDE plugins do not provide the corrective actions that would circumvent a given warning. Take, for example, FindBugs [13], a popular bug detection tool. This tool detects hundreds of bug signatures and reports them using an abbreviated code such as CO_COMPARETO_INCORRECT_FLOATING. Using this code, developers can browse the FindBug's dictionary and find the corresponding definition *"This method compares double or float values using pattern like this:* $val1 > val2\ ?\ 1 : val1 < val2\ ?\ -1 : 0$*"*. While the detection of this bug pattern is accurate, the tool does not propose any corrective actions to the developers that can help them fix the problem. Moreover, it has been reported in the literature that the output of existing maintenance tools tends to be verbose at the point where developers decide to simply ignore them [14]–[16]. Another example, related to clone detection, found that they are six different reasons that trigger the use of clones (e.g., copy and paste of code examples, a reimplementation of the same functionality in a different language, etc.). Developers are aware that they are creating clones in five out of six situations. In such cases, warnings provided by IDE plugins will only be disturbing for developers.

Finally, developers perceive statistical models aiming to improve the maintenance processes as untrustworthy black-boxes. In risky changes detection, where approaches warn developers about changes to the software that are likely to introduce a new bug, statistical model are frequently based on code metrics (i.e. number of line added/deleted, numbers of files/package modified, etc). Then a statistical model is built with the changes that are known to have introduced bugs in the software, and if a new change is over the fit, a flag is raised. Developers do not trust statistical model mainly because they do not provide examples but are only based on model fit.

According to these different findings, we believe that software maintenance tools and approaches should interfaces with developers at commit-time and, in addition, provide corrective actions for each warning. Interacting with software developers at commit-time avoid to resort to external tools that induce context switching or IDE plugins that can be distracting. Also, software maintenance at commit-time has the advantage to intervene before the code actually reaches the central repository and become pullable by the other members of the organizations. Indeed, approaches consisting in monitoring central code repository for quality check exist. These approaches provide email reports to developers. We argue that this is not practical because clones, defects, and mistakes can be synchronized by other team members, which may lead to challenging merges if corrective actions are undertaken.

Also, it is our opinion that the *right* time for the "Just-In-Time Quality Assurance" [17] movement—where defect prediction models identify risky changes as they get committed—is commit-time.

III. Research Approach

Our preliminary works have been used to conceptualize tools on approaches that could fit into commit-time software maintenance. The approaches we proposed do not hinder the productivity of developers by interrupting them yet, they integrate themselves seamlessly into the day-today workflow of developers.

To create approaches that are perceived as trustworthy by developers, we need to provide, in the case of risky changes preventions, examples on why the change is considered risky and present corrective actions to the developers. To do, so we propose BUMPER (An Aggregate Bug-Fix Repository for Developers and Researchers) in section III-A in combination with PRECINCT and BIANCA presented in section III-B both. Finally, if despite commit-time maintenance, defects reach the central repository and thereafter release to production, it will lead to bug reports. One of the most crucial piece of information to fix a bug is the step to reproduce [18]. Consequently, we propose an approach named JCHARMING to reproduce the bug.

A. An Aggregate Bug-Fix Repository for Developers and Researchers

In this work, we introduce BUMPER (BUg Metarepository for dEvelopers and Researchers), a web-based infrastructure that can be used by software developers and researchers to access data from diverse repositories using natural language queries in a transparent manner, regardless of where the data was created and hosted [19]. The idea behind BUMPER is that it can connect to any bug tracking and version control systems and download the data into a single database. We created a common schema that represents data, stored in various bug tracking and version control systems. BUMPER uses a web-based interface to allow users to search the aggregated database by expressing queries through a single point of access. This way, users can focus on the analysis itself and not on the way the data is represented or located. BUMPER supports many features including: (1) the ability to use multiple bug tracking and control version systems, (2) the ability to search very efficiently large data repositories using both natural language and a specialized query language, (3) the mapping between the bug reports and the fixes, and (4) the ability to export the search results in JSON, CSV and XML formats.

Importantly, BUMPER differs from other approaches such as Boa [20] because (a) it updates itself every day with the new closed reports, (b) it proposes a clear and concise JSON API that anyone can use to support their approaches or tools.

B. An Approach for Preventing Risky Changes and Clone Insertion at Commit-Time

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of the code in a given software system [21], [22]. Nevertheless, clones are considered a bad practice in software development since they can introduce new bugs in code [23]. If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused to fix the bug in each place.

In this research, we present PRECINCT (PREventing Clones INsertion at Commit-Time) that focuses on preventing the insertion of clones at commit time, i.e., before they reach the central code repository. PRECINCT is an online clone detection technique that relies on the use of pre-commit hooks capabilities of modern source code version control systems. A pre-commit hook is a process that one can implement to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an existing code segment. In fact, PRECINCT, itself, can be seen as a pre-commit hook that detects clones that might have been inserted in the latest changes with regard to the rest of the source code.

Similar to clone detection, we propose an approach for preventing the introduction of bugs at commit-time. Many tools exist to prevent a developer to ship *bad* code [24]. However, these tools rely on metrics and rules to statically and/or dynamically identify sub-optimum code. Our approach, called BIANCA (Bug Insertion ANticipation by Clone Analysis at commit-time), is different than the approaches presented in the literature because it mines and analyses the change patterns in commits and matches them against past commits known to have introduced a defect in the code (or that have just been replaced by better implementation).

C. A Bug Reproduction Technique Based on a Combination of Crash Traces and Model Checking

When preventing measures have failed, and bugs have to be fixed; the first step is to reproduce what happened on sites. Crash reproduction is an expensive task because the data provided by end users is often scarce [25]. It is, therefore, important to invest in techniques and tools for automatic bug reproduction to ease the maintenance process and accelerate the rate of bug fixes and patches. Existing techniques can be divided into two categories: (a) On-field record and in-house replay [26], and (b) In-house crash explanation [27].

We propose an approach, called JCHARMING (Java CrasH Automatic Reproduction by directed Model checkING) that uses a combination of crash traces and model checking to reproduce bugs that caused field failures automatically [28], [29]. Unlike existing techniques, JCHARMING does not require instrumentation of the code. It does not need access to the content of the heap either. Instead, JCHARMING uses a list of functions output when an uncaught exception in Java occurs (i.e., the crash trace) to guide a model checking engine to uncover the statements that caused the crash. Such outputs are often found in bug reports.

IV. Contributions

Based on our preliminary works, we conceptualized four approaches (BUMPER, PRECINCT, BIANCA and JCHARMING) that allows the aggregation of bug-fix, the prevention of clone insertion and risky and changes before they reach the central repository and the reproduction of bug, namely.

Our approaches work at commit-time and are efficient trade-offs between external tools, IDE-plugin and remote approaches for clone detection, risky changes detection, and bug reproduction and, as such, we believe that it addresses major factors that contribute to the slow adoption of software maintenance approaches by practitioners.

We believe that software maintenance at commit-time is a non-intrusive yet effective way to improve software quality altogether as it will combine state-of-the-art approaches in clone detection, risky change detection and bug reproduction without the context switching or distraction they provoke according to developers. Indeed, a commit marks the end of a particular programming task which is consequent enough to be shared with the organization. The task at hand being completed there is no risk of interrupting the thoughts processes of developers and hindering their productivity. In addition, we argue that intervening at commit-time is better than, for example, remote approaches that analyse the quality of the code after they reach the central repository and send email reports because mistakes can be pulled by other members of the organization further complexifying their removal. Finally, emails report comes in an asynchronous manner and developers, in case of warnings, have to reconstruct the mental models they were in for the suspected tasks.

References

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* Palgrave Macmillan, 2005, p. 880.

[2] Health, Social and E. Research, "The Economic Impacts of Inadequate Infrastructure for Software Testing," 2002.

[3] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? findings from a google case study," in *Proceedings of the international conference on software engineering*, 2013, pp. 372–381.

[4] S. L. Foss and G. C. Murphy, "Do developers respond to code stability warnings?" in *Proceedings of the 25th annual international conference on computer science and software engineering*, 2015, pp. 162–170.

[5] L. Layman, L. Williams, and R. S. Amant, "Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools," in *Proceedings of the first international symposium on empirical software engineering and measurement (esem 2007)*, 2007, pp. 176–185.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th acm sigplan-sigsoft workshop on program analysis for software tools and engineering - paste '07*, 2007, pp. 1–8.

[7] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 35th international conference on software engineering*, 2013, pp. 672–681.

[8] B. O'Sullivan and Bryan, "Making sense of revision-control systems," *Communications of the ACM*, vol. 52, no. 9, p. 56, Sep. 2009.

[9] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[10] Findbugs, "FindBugs Bug Descriptions." 2015.

[11] T. J. Robertson, J. Lawrance, and M. Burnett, "Impact of high-intensity negotiated-style interruptions on end-user debugging," *Journal of Visual Languages and Computing*, vol. 17, no. 2, pp. 187–202, 2006.

[12] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the sigchi conference on human factors in computing systems*, 2006, pp. 231–240.

[13] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[14] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "A gamified tool for motivating developers to remove warnings of bug pattern tools," in *Proceedings - 2014 6th international workshop on empirical software engineering in practice, iwesep 2014*, 2014, pp. 37–42.

[15] S. Kim and M. D. Ernst, "Which warnings should I fix first?" *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering ESECFSE 07*, p. 45, 2007.

[16] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective Error Ranking for FindBugs," in *2011 fourth ieee international conference on software testing, verification and validation*, 2011, pp. 299–308.

[17] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[18] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th acm sigsoft international symposium on foundations of software engineering*, 2008, p. 308.

[19] M. Nayrolles and W. Hamou-Lhadj, "BUMPER: A Tool to Cope with Natural Language Search of Millions Bugs and Fixes." in *International conference on software analysis, evolution, and reengineering*, 2016, pp. 649–652.

[20] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the international conference on software engineering*, 2013, pp. 422–431.

[21] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the 2nd working conference on reverse engineering*, 1995, pp. 86–95.

[22] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proceedings. ieee international conference on software maintenance*, 1999, pp. 109–118.

[23] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the ieee 31st international conference on software engineering*, 2009, pp. 485–495.

[24] D. Hovemeyer, "FindBugs." 2007.

[25] N. Chen, "Star: stack trace based automatic crash reproduction," PhD thesis, The Hong Kong University of Science; Technology, 2013.

[26] T. Roehm, S. Nosovic, and B. Bruegge, "Automated extraction of failure reproduction steps from user interaction traces," in *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)*, 2015, pp. 121–130.

[27] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "MIMIC: locating and understanding bugs by analyzing mimicked executions," in *Proceedings of the 29th acm/ieee international conference on automated software engineering*, 2014, pp. 815–826.

[28] M. Nayrolles, A. Hamou-Lhadj, T. Sofiene, and A. Larsson, "JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," in *Pro-*

ceedings of the 22nd international conference on software analysis, evolution, and reengineering, 2015, pp. 101–110.
[29] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, “A bug reproduction approach based on directed model checking and crash traces,” *Journal of Software: Evolution and Process*, 2016.