What Is the Right Time for Just-In-Time Quality Insurance?

Mathieu Nayrolles, Abdelwahab Hamou-Lhadj
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
{mathieu.nayrolles, wahab.hamou-lhadj}@concordia.ca

Emad Shihab
DAS Lab, CSE Dept, Concordia University
Montréal, QC, Canada
eshihab@cse.concordia.ca

***Abstract*—abstract goes here**

I. Introduction

Software quality insurance encompasses different activities (e.g., defect detection and prediction, clone detection, source code inspection, unit testing, implementing new requirements, etc.) that play an important role in producing high quality software. It exists several classes of maintenance (adaptive, perfective, corrective, preventive) which, when combined, represent up to 70% of the overall cost of any given project [1]. The international organization for standardisation (ISO) defines software maintenance to be the modification of a software product after delivery to correct faults, to improve performances or other attributes, or to adapt the product to a changed environment [2]. Researchers, however, have proposed approaches and tools to support practitioners in doing these actions before the delivery. Developers can leverage these approaches to, for examples, detect anti-patterns, clones, defects or, performance bottlenecks before shipping a new version of their software. More specifically, developers have access to tools and IDE-plugins that supports the detection of anti-patterns, clones and defects patterns. Quality assurance team have access to approaches that generates lists of packages or files that are likely to contain defects.

In recent years, several research areas (mining bug repositories, bug analysis, patch analysis, bug reproduction) related to software maintenance have been exposed to a novel idea: "Just-In-Time Software Quality Insurance." "Just-In-Time Quality Assurance" is the concept that software maintenance (or quality insurance) cannot be a separate task of developing software but interleaved with day-to-day programming [3]–[5]. Indeed, software project and repositories not only became large, but they are also now ultra-large [6]. Thinking about software quality insurance activities as a dedicated chunk of times where the code is reviewed, risky package analyzed, anti-patterns refactored and code clones removed is impractical because the mental models and thoughts processes that led to these changes or design decisions are long gone. Consequently, "Just-In-Time Software Maintenance" aims to perform these activities as soon as a change is made, so the decisions motivating the chances are still fresh in the mind of practitioners.

In this paper, we first propose a short historic of the just-in-time concepts. Then, we present a comparison of the different moments where just-in-time maintenance can be applied. We do so to characterize what would be the right time, in terms of productivity and resulting quality, to apply just-in-time software maintenance approaches.

II. Just-In-Time: From Toyota Plants to Software Quality

The term Just-In-Time (JIT) have first been associated with manufacturing in the 1970s. More specifically, Just-In-Time manufacturing was first developed and perfected within the Toyota manufacturing plants by Taiichi Ohno as a means of meeting consumer demands with minimum delays and waste (i.e., time and resources) [7]. To do so, JIT manufacturing relies on several management philosophies and tools: continuous improvement (product-oriented layout of plants, division of systems, simplicity), elimination of waste (overproduction, waiting time, inventory waste, transportation, product defects), Hausukipingu (clean workspaces), Kabans (pulling the right number of items from the right shelves at just the right time), Jidoka (autonomous machines with judgment capabilities) and Andons (signal problems for corrective action).[1]

Some of these ideas have long been transposed to software engineering. For example, continuous improvement is one of the cornerstones of the agile manifesto [8], integrated development environment (IDE) are Jidokas in a sense that they auto-complete us and auto-correct us based on past behavior and, unit tests, bug report management systems and quality assurance (QA) bots are Andons. Hausukipingu and Kabans are more difficultly transposable to software manufacturing. The last principle, elimination of waste, is one that is the more closely related to JIT software maintenance. Indeed, in JIT software maintenance and JIT quality assurance, researchers want to reduce the number of defects, clones, and bad design while

[1]We kept the Japanese version of most of the words as they are used without translation in the literature.

they are still fresh in the mind of developers. Consequently, product defects are lowered and time is saved.

III. Types of Just-In-Time Quality Insurance

In this section, we describe three different moments where JIT software maintenance could take place: real-time, commit-time and integration time. For each, we will divide our analyze on four parts: description, influential papers and, advantages and pitfalls.

A. Real-Time

Real-Time software quality insurance tools operate directly inside developers' IDE using IDE plugins. The rationale behind IDE-plugin and real-time software quality insurance is that it allows warning developers of potentially hazardous code as they write it and, consequently, save time and strengthen the overall quality of the software.

The adoption of such tools is, however, limited in the industry.

Johnson *et al* found that static analysis tools to find bugs produce too many warnings, do not provide corrective actions and are perceived as black-boxes by users [9]. Take, for example, FindBugs [10], a popular bug detection plugin. This plugin 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. Their findings have since been confirmed many times [11]–[13].

In the particular case of clone detection, Latoza *et al.* [14] found that there exist 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-based local detection techniques can be quite disturbing [15]. Finally, using IDE-plugins can lead to context and workspace switching which can hinder the productivity of developers [16], [17].

An overwhelming majority of approaches target the Eclipse IDE despite the fact that eclipse is only the fifth most used development environment. Indeed, in 2016 the StackOverflow developers survey[2] found that usage of development environment among 46,613 responses were as follows: Notepad++[3] (35.6%), Visual Studio[4] (35.6%), Sublime Text[5] (31.0%), Vim[6] (26.1%) and Eclipse[7] (22.7%). Developers were allowed to choose multiple answers, hence the over 100% total. The average developer, still according to the StackOverflow survey, uses between two and three development environments. The second most investigated IDE when it comes to building real-time quality insurance plugins is Netbeans[8]. Netbeans ranks 11th with 8.1%.

Arguably, Eclipse and Netbeans are targeted because they are open-source and easy to develop for. An interesting point, however, is the fact that among the top-5 development environments, three (Notepad++, Sublime Text and, Vim) are text editors. Text-editors, at the opposite of IDE, treats code as text with only few added features like coloration or the ability to call an external build suite. The declining use of specialized IDE is correlated to the rise of a particular pedigree of developer: Full Stack Developers. Full Stack Developers, as their name suggests, manipulate each layer of their software solutions stack. Consequently, they can found themselves editing files written in a data description language for their databases, a backend language such as Php, Ruby or Java, a front-end language such as Typescript and diverse configuration files in JSON or XML. Instead of mastering as many specialized IDE as they use languages, full stack developers turned to the least common denominator and use one text editor that can read and edit any types of text files.

It is, in our opinion, fascinating that researchers focus on tools, approaches and techniques that could fit into developers' IDE without considering that (a) developers are generally reluctant to the use of such tools and, (b) the majority of developers do not use IDEs.

B. Integration-Time

Another time where just-in-time quality insurance have been applied is at integration-time. Integration-time refers to the times where new code modifications reach a central repository. Performing just-in-time quality insurance at integration-time is the path of least resistance when it comes to change developer processes with the dual aim of saving time and improving software quality. Indeed, integration-time approaches monitor the central code repository and perform a quality evaluation based on code or process metrics. When the quality evaluation is complete, a report is emitted, generally by email. Consequently, developers do not have to change their processes

[2]http://stackoverflow.com/research/developer-survey-2016#technology-development-environments

[3]https://notepad-plus-plus.org/
[4]https://www.visualstudio.com/
[5]https://www.sublimetext.com/
[6]http://www.vim.org/
[7]https://eclipse.org/
[8]https://netbeans.org/

because they do not install any tools, plugins and still use text-editors. Another advantage of integration-time just-in-time quality insurance approaches is that do not execute themselves on developers' workstation. Indeed, they can monitor repositories from a remote server with adequate specifications in terms of computational power. Real-Time approaches are bound to the performances of developers' workstations.

Commit Guru is a popular example of integration-time just-in-time quality insurance approach [18]. Commit guru monitors Github repositories and builds statistical models based on code metrics to determine, for each project, thresholds above which a code modification is likely to introduce a defect into the code. Also, many commercial tools propose integration-time just-in-time quality insurance. Codeclimate,[9] Codacy,[10] Scrutinizer[11] and Coveralls[12] are some examples. These tools will perform various tasks such as executing unit test suites, computing quality metrics, performing clone detection and, provide a report by email.

In our opinion, the problem with integration-time just-in-time quality insurance approach is that the detection occurs too late in the development process. Once the code reaches the central repository, they can be pulled by other members of the development team, further complicating the removal and management of potential defects, clones or anti-patterns. Moreover, the asynchronous way in which integration-time just-in-time software quality maintenance operates can lead to the same context-switching and workspace switching as real-time approaches [16], [17]. Indeed, email reports will be received by developers in an asynchronous fashion. Consequently, it is likely that developers would have started a new task and will have to build back the thoughts processes that led risky design decisions.

C. Commit-Time

Commit-Time just-in-time software quality insurance is the last time at which just-in-time software quality insurance can operate. Each time a developer makes a commit, the changes are intercepted using a pre-commit hook. Pre-commit hooks are custom scripts set to fire off when certain important actions of the versioning process occur. There are two groups of hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, whereas server-side hooks run on network operations such as receiving pushed commits. These hooks can be used for all sorts of reasons such as checking compliance with coding rules or automatic run of unit test suites. The pre-commit hook runs before the developer specifies a commit message. It is used to inspect the modifications that are about to be committed. To the best of our knowledge, we are the first to conceptualize commit-time for just-in-time software quality insurance.

By design, pre-commit hooks treat code modification as text and kicks in seamlessly during the versioning process. Consequently, every approach that currently takes place at real- or integration-time could be migrated at commit-time without significant effort. Also, commit-time just-in-time software quality insurance would provide recommendations interactively during the versioning operations. Providing recommendations during versioning would not interfere with developers thoughts processes as commits are bite-sized units of work that are potentially ready to be shared with the rest of the organization [19]. Commits usually mark the end of a given task or subtask as the developer is ready to version the source code. Thus, undesirable side-effects of real-time software quality insurance would be avoided while retaining its benefits. Also, commit-Time just-in-time software quality insurance intervenes before the code reaches the central repository and became pullable by other members of the organization. Finally, the quality analysis happens synchronously and prevent developers to begin new tasks. Their design decisions would still be fresh in their mind presented with the results of the quality analysis at commit-time.

IV. Conclusion

In this paper, we presented three different times at which just-in-time software quality insurance can be implemented: real-time, integration-time and commit-time. While it exists real-time and integration-time approaches and tool, to the best of our knowledge, we are the first to conceptualize what a commit-time approach would look like.

Real-time approaches interrupt the developers with many warnings and recommendations. Moreover, they assume that developers use IDE and the majority do not. Integration-time approaches allow developers to keep their processes as they seamlessly monitor the central repository and perform asynchronous quality analysis when new changes are received. When the report of integration-time approaches is received by developers, they are likely to have started new development tasks and will have to build back the thoughts processes that led risky design decisions.

Commit-Time just-in-time software quality insurance approaches would be an efficient trade-off between real- and integration-time as they would address major factors that contribute to the slow adoption of quality assurance tools. Indeed, they would fit in the day-to-day workflow of developers (i.e. coding, testing, debugging, committing) and will not hinder their productivity by yielding context or workspace switches. In addition, commit-time

[9] https://codeclimate.com/
[10] https://codacy.com/
[11] https://scrutinizer-ci.com/
[12] https://coveralls.io/

approaches do not rely on IDE but the versioning system. Following the rationale that full stack developers turned to text editor that can read and edit any types of text files, commit-time approaches would rely on another text-based technology that any current development team uses: code versioning. Finally, developers are used to having an interactive versioning process as, for example, they can be notified that they do not have the latest version of the source-code locally in the case that another member of the organization committed modifications. In such a case, developers would have to retrieve the modifications and merge them to their own before being allowed to share their work with the organization. We that providing quality insurance recommendations following the same process would not impact significantly developers while achieving the quality objectives of just-in-time quality insurance. Commit-time *is* the right time for just-in-time software quality insurance.

As a future work, we plan to conceptualize, implement and test commit-time approaches for risky changes identification, clone detection, anti-patterns identification, and other related quality insurance activities.

References

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

[2] ISO/IEC-14764:2006, *Software Engineering – Software Life Cycle Processes – Maintenance*. 2006, p. 44.

[3] Y. Kamei, E. Shihab, A. Ihara, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. I. Matsumoto, "Studying re-opened bugs in open source software," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[4] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," in *2015 ieee international conference on software quality, reliability and security*, 2015, pp. 17–26.

[5] P. Tourani and B. Adams, "The Impact of Human Discussions on Just-in-Time Quality Assurance: An Empirical Study on OpenStack and Eclipse," in *23rd international conference on software analysis, evolution, and reengineering*, 2016, pp. 189–200.

[6] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and Others, "Ultra-large-scale systems: The software challenge of the future," DTIC Document, 2006.

[7] K. Suzaki, *New manufacturing challenge: Techniques for continuous improvement*. Simon; Schuster, 1987, p. 255.

[8] M. Fowler and J. Highsmith, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.

[9] 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.

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

[11] 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.

[12] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *Proceedings of the 19th international symposium on software testing and analysis - issta '10*, 2010, p. 241.

[13] 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.

[14] T. D. Latoza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceeding of the 28th international conference on software engineering - icse '06*, 2006, pp. 492–501.

[15] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[16] 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.

[17] 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.

[18] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the10th joint meeting on foundations of software engineering*, 2015, pp. 966–969.

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