Continuous Refactoring in CI: A Preliminary Study On the Perceived Advantages and Barriers

Carmine Vassallo, Fabio Palomba, Harald C. Gall,
University of Zürich, Switzerland
vassallo@ifi.uzh.ch, palomba@ifi.uzh.ch, gall@ifi.uzh.ch

***Abstract*— By definition, the practice of Continuous Integration (CI) promotes continuous software quality improvement. In systems adopting such a practice, quality assurance is usually performed by using static and dynamic analysis tools (e.g., SonarQube) that compute overall metrics such as maintainability or reliability measures. Furthermore, developers usually define quality gates, i.e., source code quality thresholds that must be reached by the software product after every newly committed change. If a quality gate fails (e.g., a maintainability metric is below a certain threshold), developers should refactor the code possibly addressing some of the proposed warnings. While previous research findings showed that refactoring is often not done in practice, it is still unclear whether and how the adoption of a CI philosophy has changed the way developers perceive and adopt refactoring. In this paper, we preliminarily study—running a survey study that involves 31 developers—how developers perform refactoring in CI, which needs they have and the barriers they face while continuously refactor source code.**

***Index Terms*—Continuous Integration; Quality Assurance; Refactoring.**

I. Introduction

Continuous Integration (CI) is a development practice aiming at *continuously* building new software [7]. When developers push changes to a shared repository, CI requires to compile, test, and quality assure the modified software product, so that developers can immediately know how their changes fit with the software being developed. One of the most promising features of CI concerns the possibility to make the identification of bugs as well as the improvement of source code quality easier [6]. Specifically, developers and stakeholders have the possibility to define the so-called *quality gates* [23], namely a set of constraints on the quality of the built software that are expressed through thresholds on certain metrics (e.g., the number of code smells [8] at each build should be lower than a predefined threshold).

Such quality gates are de-facto the way to control for maintainability degradation [22]: if a newly committed change fails a quality gate, the developer needs to take action to let the build pass again. One of the possible actions to improve source code quality is *refactoring* [8], which consists of changing the internal structure of source code while preserving the external behavior of the system [8].

In the last decade, a number of researchers have investigated refactoring practices from several perspectives [3], [11], [15], [18], [19], [20], [24], [28], finding that its application not only positively affects adaptability, maintainability, understandability, re-usability, and testability [1], but also developers' productivity [16]. However, despite its usefulness refactoring is often perceived as a fault-prone activity [2], [11] or simply not frequently performed by developers when improving source code quality [3].

While the studies conducted so far provided some important insights on how developers perform refactoring and what are the main concerns leading them to not do it, there is still a lack of knowledge on whether and how CI has changed the way developers adopt refactoring. Indeed, CI advocates to perform continuous code quality improvement, meaning that developers *might be willing to apply refactoring operations at every performed change*. Thus, it is worth analyzing whether such *Continuous Refactoring* is an actual practice adopted and needed in CI. As Continuous Refactoring is not listed yet among the CI best practices [7], our goal is to study its applicability, usefulness, and its possible introduction as a best practice in CI, with the aim of stimulating new research in supporting developers while doing it.

In this paper, we provide a preliminary overview around the way refactoring is applied in CI, performing a survey study that collects the opinions of 31 developers. In the context of CI, we investigate (i) how frequently developers refactor, (ii) whether developers need to perform refactoring continuously (i.e., Continuous Refactoring), and (iii) which are the current barriers faced by developers while refactoring at every build.

Key findings of our study report that most of the surveyed developers tend to perform refactoring continuously as they feel the need for it to improve maintainability and testability of source code. At the same time, however, our participants revealed the presence of several barriers while doing refactoring in CI context: these are not only related to time-constraints or the fear to break a build, but also concern team-related aspects. Finally, developers clearly point out the need for reliable refactoring tools as well as highly-effective test suites to use when testing for regression.

II. Methodology

The *goal* of this preliminary study is to understand (i) how developers perform refactoring and (ii) what are the pros and cons of adopting Continuous Refactoring through a survey study involving developers that actually adopt CI. Hence, the *context* of our study includes (i) as *subjects* the participants of our survey (more details about them in the next sub-sections) and (ii) as *objects* the specific refactoring practices that they adopt when developing software.

Our study is structured around three research questions:

- **RQ$_1$** *How do developers perform refactoring in CI?* In the first research question, we are interested in investigating the frequency of refactoring tasks. Specifically, we aim at understanding whether refactoring is applied continuously.
- **RQ$_2$** *Do developers need to refactor continuously?* This research question studies to what extent developers consider Continuous Refactoring as a crucial practice and why.
- **RQ$_3$** *Which are the barriers to Continuous Refactoring?* The last research question is about the barriers faced by developers to apply Continuous Refactoring.

A. Survey Design

Our questionnaire starts clarifying some terms (e.g., "Continuous Refactoring"). It consisted of 14 questions, which include 5 multiple choice (MC), 6 checkboxes (C) and 3 open questions (O). In case of checkboxes, our participants could leave further comments. Table I reports the specific questions, which were grouped into four topics: (i) Background (omitted in Table I), (ii) Current Practice, (iii) Need for Continuous Refactoring, and (iv) Barriers to Continuous Refactoring.

In the Background section, we profiled our participants according to (i) development experience, (ii) domain of the organization where they work, (iii) CI expertise, and (iv) adopted tools (i.e., build and static analysis tools integrated in their CI pipeline). The Current Practice section concerns the usage of refactoring in CI context. We were interested in understanding how frequently developers perform refactoring and pay attention to code quality (Q1.1, Q1.2, Q1.3) and which are their motivations for doing (Q1.4) refactoring. In the section Need we encourage our participants to tell us whether, in their opinion, there is a need for Continuous Refactoring (Q2.1, Q2.2) and which are the factors or needs empowering its applicability (Q2.3). In the last section (Barriers), we explore why developers tend to avoid Continuous Refactoring.

B. Recruitment and Survey Participants

Our survey was implemented using GOOGLE FORMS.[1] To recruit participants, we posted our questionnaire on REDDIT[2], targeting three specific sub-forums dedicated to LearnProgramming, DevOps, and JavaScript: we selected these communities as they (i) allow users to post surveys (e.g., suitable communities as Java forbid questionnaires), (ii) have a large number of active subscribers, thus increasing our potential audience (e.g., the JavaScript subreddit has approximatively 300 members that are online daily), and (iii) possibly include members having experience with CI. Due to time constraints, we kept the survey accessible for one week. In the end, 31 developers answered the entire survey. 32.3% of them started to develop more than 10 years ago. 48.4% have between 5 and 10 years' development experience and 19.4% less than 5 years. Furthermore, our respondents come from various domains (we collected 18 different domains). 25.8% of them work in software development consulting, 12.9% in organizations focusing on content platform providers. The same percentage (9.7%) deal with telecommunications and framework development. Regarding to their CI experience, the majority of our participants (48.4%) started adopting CI between 2 and 5 years ago, 32.2% of them even more than 5 years ago. The two mostly used CI build tools are MAVEN (29%) and GRADLE (22.6%) (in line with previous work [26]), while ESLINT (51.6%) is the most adopted static analysis tool.

[1] https://docs.google.com/forms/u/0/

[2] https://www.reddit.com/

C. Limitations and Threats to Validity

The methodology adopted to conduct our study might suffer of some limitations. These are mainly related to the recruitment process: we obtained 31 answers to the survey, despite the potential audience available on REDDIT. While the limited time period set to gather answers might have played a role in the response rate obtained, it is worth noting that generally most of the answers to scientific surveys are collected in a short-time period [14], [25]. This means that, overall, we are pretty confident of the fact that we gathered most of the potential answers we could have obtained.

Another possible issue might be related to the generalizability of the opinions obtained by the study participants. On the one hand, we collected answers from people having different backgrounds and working in different development environments: thus, we can partially argue that the diversity of the considered population is quite large to allow us to provide insights on the way developers perform refactoring in CI. On the other hand, we are aware that this study highlights some preliminary observations that should definitively be confirmed by further quantitative and qualitative analyses.

Finally, the focus of our analysis was consciously targeted on the developers' perception of the phenomenon: we are aware that the opinions collected might differ from the actual practices performed by the participants. Further studies testing our findings in a quantitative fashion are already part of our future research agenda.

III. CURRENT PRACTICE OF REFACTORING IN CI

In the first place, our findings suggest that in CI, accordingly to its best practices [6], developers tend to assess code quality continuously (53.3%) or at least before merging their feature branch (i.e., a branch opened to implement a specific development task) into the master branch (56.7%).

Such a continuous code quality assessment is *often* (33.3%) or *always* (20%) accompanied by a refactoring, according to the participants' point of view. In other words, our preliminary findings seem to highlight that refactoring is rather spread in the CI environment, differently from what is usually done in other contexts [3], [11]. While further studies should be carried out to verify our findings, a key emerging result is that *CI has the potential to change the way software code quality assessment and refactoring can be applied in practice*.

TABLE I
SURVEY QUESTIONS ON CONTINUOUS REFACTORING. (MC: MULTIPLE CHOICE, C: CHECKBOXES, O: OPEN ANSWER)

Section	ID	Question	Type	Answer
Current Practice	Q1.1	How frequently do you perform refactoring?	MC	Never — Rarely — Sometimes — Often — Always
	Q1.2	When do you assess the code quality?	C	At every committed change. Before a release. Before merging a feature branch in the master branch. Never. At the end of a sprint or development iteration.
	Q1.3	When do you perform refactoring?	C	I perform refactoring at every change (when it is needed). I perform refactoring when some quality gates fail. I perform refactoring when I have time. I perform refactoring after assessing code quality (if it is necessary). An IDE warning suggests to do it.
	Q1.4	Which are your motivations for refactoring?	C	Refactoring helps me to improve the overall code quality. Refactoring increases the code comprehension. Avoid to fail quality gates (and cause build failures). I'm forced to do refactoring. I do not perform continuous refactoring.
Need	Q2.1	In your opinion, is Continuous Refactoring needed?	MC	Yes — No — Maybe
	Q2.2	Why do you think Continuous Refactoring is needed or not?	O	*Open Answer*
	Q2.3	Which are the factors that can be used to properly schedule refactoring tasks?	O	*Open Answer*
Barriers	Q3.1	Which are your motivations to avoid Continuous Refactoring?	C	Refactoring is a tedious task and/or I am not paid for doing it. Lack of knowledge about refactoring tactics. Lack of proper automated-refactoring tools. Not enough time during development. Absence of a proper test suite that catch possible bugs introduced by refactoring.

As a further evidence of this point, the majority of our participants (63.3%) reported to refactor (if necessary) at every committed change or generally after assessing code quality (46.7%). The motivations for refactoring can be summarized in three points: *code quality improvement* (86.7%), *better code comprehension* (76.7%), and *green* (i.e., passed) *quality gates* (33.3%). This result is still only partially aligned with previous findings: indeed, recent studies showed that developers either tend to not remove design issues from source code [3] or apply refactoring just to make the development task they should work on easier [20], [24]. Instead, our participants explicitly pointed out that the improvement of code quality and comprehension represents the main motivation for doing refactoring. This further strengthens the conjecture that CI can represent a precious practice to allow developers to perceive the value of code quality improvement.

At the same time, it is interesting to note that 33.3% of respondents declared that the main motivation to refactor is to avoid quality gates' failures. Continuous refactoring is perceived as a practice that should be done at every new change to avoid to fail quality gates. Automated Static Analysis Tools (ASATs) have the potential of being the tools that suggest developers refactoring operations [29]. Thus, given the fact that they are invoked during the CI process [27], ASATs could be a useful instrument for suggesting developers refactoring tasks at every new build.

When asked about the factors making possible refactoring scheduling, our respondents provided us with several insights confirming the usefulness of static and dynamic analysis. For instance, S11[3] reported that she usually schedules refactoring in presence of "warnings raised by static analysis tools". Perhaps more interestingly, some developers are motivated to do refactoring when there is "lack of test coverage" (S8) when the aim is to remove unused code that might decrease the test coverage indicator. Similarly, S18 argued that she starts to look for refactoring opportunities when "code complexity" is high. Thus, they use such tools to schedule refactoring tasks. Likely, our findings go toward the direction of having more contextual and/or developer-oriented quality-related warnings to enable developers to just deal with those that can make sense in a certain situation and avoid false alarms [26].

[3]The respondents are numbered S1 to S31.

IV. THE NEED FOR CONTINUOUS REFACTORING

The majority of our respondents (71%) agree with the fact that Continuous Refactoring is needed. As mentioned by S3[3], Continuous Refactoring is crucial because "code is not the best in the first version". Thus, after implementing a certain code change, developers should immediately reason in terms of the refactoring opportunities that might be applied. In that way, developers "can constantly address tech debt" (S19) and keep the quality gates always green, without failing them.

Interestingly, S14 advocated Continuous Refactoring in order to improve the overall "testability of production code". In other words, according to the opinions of our participants, refactoring can have a positive influence on "software reliability" (S11), that is one of the main goal of CI [6]. Indeed, the CI pipeline provides early feedback about defects included in the codebase and also in case of refactoring "one should fix issues as near to a problem and as quickly as possible" (S25). Furthermore, most of the respondents agreed on the positive effects that refactoring might have on the overall structure of the source code. For instance, S30 argued that "code is like a building site and refactoring is cleaning up the rubble".

On the other hand, it is worth mentioning that 29% of the participants were less convinced on the need for continuously refactoring the code. More specifically, 16.1% of them said that Continuous Refactoring is *maybe* needed and 12.9% that is *not* a crucial activity at all. There are two major motivations behind these answers: (1) risks associated to the re-structuring of a portion of source code and/or (2) effort required to actually apply the transformation. As an example, S4 reported that "continuous program transformations can decrease the understandability of the overall architecture of the system": this seems to be particularly true in presence of development teams working together on a portion of source code, as in that cases it might be critical to modify too much the structure

because it can decrease its understandability, as also clarified by S6: "Not the best approach but it works on small teams with large software to build". This finding seems in line with what reported by Hall et al. [9] on the risks associated with, for instance, the big-bang re-modularization.

S6 also reported that "refactoring-as-you-go" can be beneficial but "takes time and requires re-testing": "it's really more a matter of trade-offs and balancing risk and reward". Similarly, S23 explained that the effort required for "remov[ing] tech debt may be higher than available for a given task". Thus, some developers believe that refactoring should not be necessarily continuous, but performed only when the gain (in terms of quality/understandability/etc.) is higher than the pain (e.g., excessive complexity of a refactoring solution). In our opinion, this is an important insight, as it somehow represents a call for new methodologies and techniques able to recommend or prioritize refactoring operations taking into account effort-related and community-related information.

V. Barriers To Continuous Refactoring

Despite the findings discussed so far, 35.5% of developers still *sometimes* perform refactoring, mainly when they have time. In these cases, developers tend to refactor only if serious quality gates fail (30%). Thus, while CI can naturally contribute to improve source code quality, there are still open issues that the research community should face. Specifically, from the answers obtained we could observe that both developers who frequently and not frequently refactor face several barriers while applying refactoring.

The main barrier (50%) is related to the *lack of time*. As expected and as happens in traditional development processes, developers always give higher priorities to functional tasks rather than source code quality improvement because of time pressure, high workload, or upcoming deadlines. This is especially true in agile projects: when developers are pushed to deliver new changes fast, there is not so much time left for activities considered at a lower priority.

The second barrier is *absence of a proper test suite* (36.7%). Refactoring code that work can be dangerous. As showed by Bavota et al. [2], it might induce new bugs: to avoid their introduction, developers need a strong test suite as well as to keep it updated to be sure that no errors can be introduced during refactoring actions. This finding further motivates the research on how to make test suites more effective.

The third barrier is the *lack of tools for automatic refactoring* (16.7%). This is in line with previous findings achieved in other contexts [11]. In any case, CI advocates automation: developers see the benefits of performing refactoring continuously, however they want support while doing it because it might be dangerous. Similarly, another barrier cited by some developers (10%) is the *poor knowledge about refactoring tactics*. This suggests not only the need for tools able to automate refactoring tasks, but more importantly for novel techniques that can explain to developers the rationale and the benefits behind a refactoring recommendation.

VI. Discussion and Implications

Despite preliminary, our study revealed some relevant findings and provided a number of implications.

Continuous Refactoring as a CI best practice. Compared to the results of previous work, it seems that CI is the right context for making Continuous Refactoring possible to apply. Indeed, our respondents indicated it as a useful methodology to keep under control the increasing complexity of the changes applied in a CI scenario. This recalls the possibility to include Continuous Refactoring among the patterns [7] to follow in order to fully benefit from the adoption of CI.

Avoid to fail quality gates. A clear research opportunity is preventing instead of fixing serious quality issues. Starting from just-in-time techniques to promptly spot design issues in source code [21] until approaches for scheduling refactoring operations. The research in those directions still require notable advances to be able to support developers when working in a CI context.

Developer-oriented static analysis tools. As observed in our study, developers would like to consider the output of static analysis tools while deciding whether to refactor. However, static analysis tools still generate false alarms [10], preventing them from being fully trusted by developers. This immediately recalls the need of improving the way such tools provide suggestions: it is important to assist developers by raising the "right" warnings that can be actually perceived as symptom of need for refactoring. To this aim, developer-oriented static analysis tools should be conceived.

Effort- and Community-aware refactoring recommenders. We revealed how CI developers need to perform refactoring continuously. However, such an operation can be very time-consuming and/or dependent on the surrounding development team allocation. Thus, refactoring recommenders and prioritization approaches should exploit effort- and community-related factors when suggesting which refactoring operations are suitable in a given development context.

Summarization of Refactoring Opportunities. Some of our study participants declared a lack of knowledge about refactoring tactics. We clearly envision further research opportunities, aimed at finding solutions to make the reasons why a certain recommender is suggesting to perform a refactoring operation understandable to developers. For instance, this might go in the direction of refactoring summarization, as also initially explored by Bavota et al. [4].

Automating regression testing of refactoring operations. The presence of effective test suites is perceived as a key factor to help developers when doing refactoring. As some refactoring operations might require to update the tests (e.g., when applying an Extract Class refactoring, the tests should reflect the new allocation of responsibilities), it is of a paramount importance to assist developers with ripple-effect-aware regression analyses and methods, such as automatic refactoring of test suites based on the refactoring applied to the production code. At the same time, our findings further

highlight the importance of augmenting existing test suites in a way to make them more effective.

VII. Related Work

Previous studies analyzed why developers perform refactoring. Silva et al. [24] found that refactoring is often driven by changes rather than by the necessity to fix code smells. Our participants confirmed the willingness of doing refactoring at every change instead as a consequence of failed quality gates.

Other researchers have investigated how developers perform refactoring. Among the findings obtained by Murphy-Hill et al. [17], it is interesting to note how developers (i) perform at least one refactoring session in more than 40% of development activities, (ii) rarely (less than 10% of times) configure refactoring tools, and (iii) often perform *floss refactoring* (*i.e.*, interleaving refactoring with other programming activities). Instead, our preliminary results show that refactoring in CI is performed very frequently (i.e., at every build), but confirm how developers perform refactoring while doing functional tasks. Furthermore, our work confirm that developers lack of proper automated tools while refactoring. Kim *et al.* [12] reported a high perceived risk of introducing bugs while refactoring. Our participants confirmed the fear of breaking the code or introduce bugs while refactoring.

Previous work adopted the term "Continuous Refactoring". Lindvall et al. [13] claimed that XP encourages Continuous Refactoring although it is a controversial practice, because it runs contrary to the widespread principle of "if it isn't broken, don't fix it". Chen et al. [5] discussed of the impact of continuous small refactoring on software architecture, but neither define the frequency of such changes or studying them in the CI context. Zazworka et al. [30] analyzed how frequently bachelor students of an XP course performed Continuous Refactoring. They found different results from us, i.e., the majority of them did not continuously refactor. Our findings possibly suggest that experience matters. To summarize, the previous work (i) did not provide a shared definition of Continuous Refactoring, (ii) did not investigate it in CI context, and (iii) provided conclusions only based on novice developers.

VIII. Conclusion

This paper has presented a preliminary investigation of the refactoring practices in CI. Our findings showed that developers tend to perform refactoring at every new build and need Continuous Refactoring. However, developers still face several barriers while continuously refactoring. For this reason, in our future research agenda we plan to (i) strengthen our initial results through interviews and quantitative data from open-source and industrial projects, and (ii) conceive approaches for supporting developers while applying Continuous Refactoring.

References

[1] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

[2] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *SCAM 2012*, pages 104–113.

[3] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[4] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):4, 2014.

[5] L. Chen and M. A. Babar. Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development. In *WICSA 2014*, pages 195–204.

[6] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[7] P. M. Duvall. Continuous integration. patterns and antipatterns. *DZone refcard #84*, 2010.

[8] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[9] M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn. Establishing the source code disruption caused by automated remodularisation tools. In *ICSME 2014*, pages 466–470.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE 2013*, pages 672–681.

[11] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *FSE 2012*, pages 50–61.

[12] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[13] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kahkonen. Agile software development in large organizations. *Computer*, 37(12):26–34, 2004.

[14] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *RN*, 15(01), 2015.

[15] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[16] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer, 2008.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Transactions on Software Engineering*, 38(1):5–18, 2011.

[18] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[19] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *ICSME 2017*, pages 1–12.

[20] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An exploratory study on the relationship between changes and refactoring. In *ICPC 2017*, pages 176–185.

[21] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk. Towards just-in-time refactoring recommenders. In *ICPC 2018*.

[22] D. L. Parnas. Software aging. In *ICSE 1994*, pages 279–287.

[23] G. Schermann, J. Cito, P. Leitner, and H. C. Gall. Towards quality gates in continuous delivery and deployment. In *ICPC 2016*.

[24] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *FSE 2016*, pages 858–870.

[25] K. T. Stolee and S. Elbaum. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *ESEM 2010*, pages 35–44.

[26] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *SANER 2018*, pages 38–49.

[27] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *ICSME 2017*, pages 183–193.

[28] Y. Wang. What motivate software engineers to refactor source code? evidences from professional developers. In *ICSM 2009*, pages 413–416.

[29] F. Wedyan, D. Alrmuny, and J. M. Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *ICST*, pages 141–150. IEEE Computer Society, 2009.

[30] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider. Are developers complying with the process: an XP study. In *ESEM 2010*.