Continuous Delivery Practices in a Large Financial Organization

Carmine Vassallo[1], Fiorella Zampetti[1], Daniele Romano[2], Moritz Beller[3],
Annibale Panichella[3], Massimiliano Di Penta[1], Andy Zaidman[3]
[1]University of Sannio, Italy, [2]ING NL, Amsterdam, The Netherlands, [3]Delft University of Technology, The Netherlands

***Abstract*—Continuous Delivery is an agile software development practice in which developers frequently integrate changes into the main development line and produce releases of their software. An automated Continuous Integration infrastructure builds and tests these changes. Claimed advantages of CD include early discovery of (integration) errors, reduced cycle time, and better adoption of coding standards and guidelines. This paper reports on a study in which we surveyed 152 developers of a large financial organization (ING Netherlands), and investigated how they adopt a Continuous Integration and delivery pipeline during their development activities. In our study, we focus on topics related to managing technical debt, as well as test automation practices. The survey results shed light on the adoption of some agile methods in practice, and sometimes confirm, while in other cases, confute common wisdom and results obtained in other studies. For example, we found that refactoring tends to be performed together with other development activities, technical debt is almost always "self-admitted", developers timely document source code, and assure the quality of their product through extensive automated testing, with a third of respondents dedicating more than 50% of their time to do testing activities.**

***Index Terms*—Continuous Delivery, Continuous Integration, DevOps, Agile Development, Technical Debt, Refactoring, Testing, Test-Driven Development**

I. Introduction

Continuous Integration (CI) was originally introduced by Grady Booch in 1991 [1], and came into fashion as one of the twelve Extreme Programming practices in 1997 [2]. Fowler defines CI as [3]:

> *A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*

CI has multiple assumed benefits, for example, that integration errors among different components of a software application can be detected earlier, easier, and with less manual effort [4]. At the heart of CI stands a testing phase, possibly in multiple integration environments, in which unit, integration, system, and even acceptance tests can automatically be executed [5], [3]. This is complemented by running Automated Static Analysis Tools (ASATs), *e.g.,* FindBugs, Checkstyle, or JSHint as part of the CI can augment the dynamic testing phase [6]. In addition to these checks of code and system quality, CI is said to improve release frequency and predictability [7], increase developer productivity [8] and improve communication [9], hence reducing the time-to-market and allowing users to benefit from continuous updates of their software. Continuous Delivery (CD) is the development practice that enables frequent releases by help of a CI process [10]. Ståhl and Bosch observed that CI, and by extension CD, have become increasingly popular in software development [11].

However, Ståhl and Bosch observed that there is not one homogeneous practice of continuous integration, indeed there are variations points with the term continuous integration acting as an umbrella for a number of variants [11]. Moreover, they showed that there is no clear insight into how the practice of CD influences other aspects of the development process.

The **goal of this paper** is thus to shed light on the interaction between CI and CD from the aspect of (i) the general development process, (ii) managing technical debt, (iii) testing activities, (iv) technical questions about the CI infrastructure.

To bootstrap this investigation, one of the authors spent three months as an intern in a large financial organization, namely ING Netherlands (https://www.ing.nl, in the following referred as ING NL) and observed how their newly adopted CD environment enables developers to run their own operations, called DevOps [12]. Based on these inside observations by an outsider to ING NL, we have designed a survey in which we asked developers about various practices they adopted in the CD pipeline. By consulting and embedding an external technical expert without domain knowledge, ING NL wanted to gain an independent understanding of their process and identify potential areas of improvement with regard to testing and managing technical debt.

Paper Structure. Section II provides an overview of the CD pipeline in ING NL. Section III defines the study, formulates its research questions, and details its planning. Then, Section IV reports and discusses the study results. Threats to validity of the conducted studies are then discussed in Section V, while Section VI discusses related literature on CD and build-release management. Finally, Section VII concludes the paper.

II. Continuous Delivery in ING NL

ING is a large financial organization with about 94,000 employees and over 67 million customers in more than 40 countries.

Nine years ago, ING NL realized the need to fundamentally change the organization of its Information Technology (IT) department. The main rationale was to bridge the gap from the

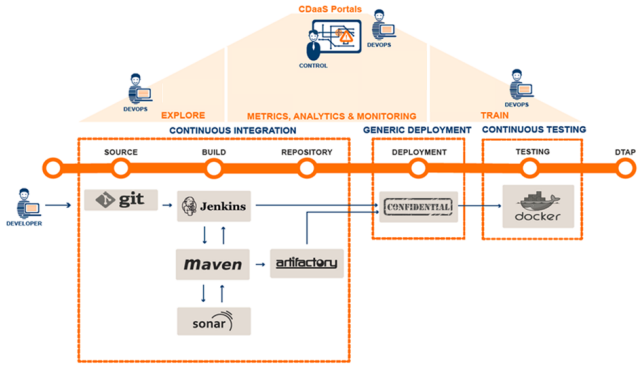


Fig. 1. Continuous Delivery pipeline.

IT and ING NL's core business. Before that, the IT activities were mainly outsourced, which created managerial effort and costs, while taking resources away from the development.

Moreover, the previously adopted development process exhibited a communication gap between the department aimed at "changing the business", *i.e.,* changing its software, and the department aimed at "running the business', *i.e.,* operating and maintaining the software. Such a gap was mainly bridged by complex processes and procedures for managing changes. This rigor was mainly introduced to ensure stability of the software systems being developed. To "change the business, the focus was on guaranteeing short release cycles. This created conflicting objectives between developers ("Devs") whose goal it was to meet deadlines, and operators ("Ops") whose goal it was to reduce the risk of runtime incidents.

The development process changed when ING NL decided to introduce a mobile application for online banking, since that long development cycles would have led to an outdated application. For this reason, development activities were changed from the previous outsourcing model to a development process in which the development was internal to the company.

When changing the development process, DevOps teams have been introduced. Such teams take charge of the application over its whole lifetime, *i.e.,* during development and operations. The next step was the introduction of a CD pipeline enforcing an agile development process to reduce the testing and deployment effort and duration, especially because such activities were used to be mainly manual work for two separate teams.

Fig. 1 depicts the CD pipeline that has been put in place in ING NL. As the figure shows, the pipeline is composed of two layers. A base layer (depicted in the bottom), which is a typical CD pipeline, and an additional layer (top) which deals with continuous monitoring. As soon as the developer pushes a commit, this is detected by the CI server, Jenkins [13], and triggers the software build. Its main task is to run build scripts, mainly Maven scripts, but also, for a minority of projects, Ant, Gradle and other build scripts.

Similar to most Open-Source CI builds [5], builds at ING NL are considered broken for a number of reasons, ranging from traditional compiling errors to failing test cases, up to software quality problems – *e.g.,* the presence of a code smell

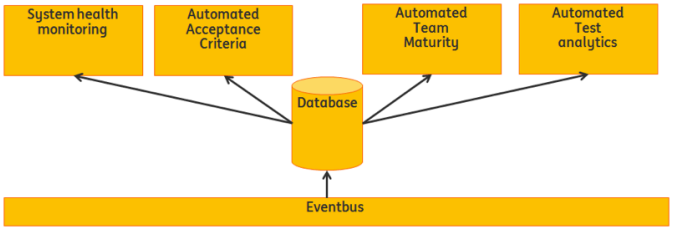


Fig. 2. Monitoring layer in the CD pipeline.

like too high McCabe cyclomatic complexity – detected by ASATs. In ING NL, such the ASAT of choice is SonarQube [14].

In case the build succeeds, the artifacts are stored in the *Repository* stage using the Artifactory [15]. This introduces several advantages, such as the possibility of implementing caching mechanisms for rapid application re-deployment. Once the *Repository* stage is reached, the application is ready to be deployed in different environments, *i.e.,* DEV (development), TST (testing), ACC (acceptance), and PRD (production).

The monitoring layer in the pipeline collects (top part in Fig. 1) a series of metrics for evaluating the CD pipeline performance. This comprises the three phases of (i) instantiating a CD pipeline, (ii) performing measurements on the pipeline, and (iii) learning from such measurements to further improve the pipeline.

The monitoring layer is detailed in Fig. 2. It is composed of one *event bus*, implemented using Apache Kafka [16], and aimed at collecting events (*e.g.,* build failures or successes) from the pipeline and storing them in a *database*, implemented using MongoDB [17]. Then, the information stored in the database is utilized by different monitoring tools, shown in the top part of Fig. 2.

The *system health monitoring tool* monitors the pipeline's software and hardware resources and its primary purpose is ensuring the pipeline's availability. The *automated acceptance criteria tool* aims at checking whether the release meets the acceptance criteria defined by the organization, before promoting it to the ACC or PRD stage. The *automated team maturity* and *test analytics* tools inform teams about releases (*e.g.,* mean cycle time a team is able to handle) and statistics about test execution, such as the percentage of failed tests.

The whole monitoring approach reflects the Lean cycle [18], in which DevOps engineers continuously learn by observing metrics and adapt the pipeline when needed.

ING NL has monitored the effect of CD adoption in terms of costs, productivity, and customer satisfaction. In three years, from 2011 to 2013, ING NL has increased the number of delivered function points by 300% and reduced the cost of a single function point to one third. Additionally, between 2012 and 2014, the release frequency has doubled, reaching one release every four days.

III. Study Design

The *goal* of this study is to better understand the implementation of CD practices in industry, by surveying how

software engineers use relevant methods and tools during the development process. The *context* is the CD pipeline of a large financial organization (ING NL).

More specifically, the study aims at addressing the following four research questions:

- **RQ_1:** *What are general development practices within the Continuous Delivery pipeline?* This research question is preliminary to the ones going deeper into the CD process, and mainly aims at investigating to what extent developers share a common development methodology, and how they plan, schedule, and monitor their development activities.
- **RQ_2:** *What are the practices adopted to manage technical debt?* This research question aims at understanding how developers manage technical debt by commenting source code, by reviewing it, and by performing any sort of static analysis or metric extraction.
- **RQ_3:** *What are the testing practices adopted within the Continuous Delivery pipeline?* This research question aims at understanding how testing is framed within the software development process, *e.g.,* whether DevOps adopt a Test-Driven Development approach [19].
- **RQ_4:** *How is Continuous Integration performed?* This research question investigates on the developers' attitude to coordinate changes through the CD infrastructure, including the use of private builds and the priority given to fix build breakages.

A. Context Selection

As a population of candidate participants to the survey, we selected a total of 176 DevOps engineers belonging to various development teams of ING NL. Such participants have been identified through the projects' mailing lists.

B. Survey Design

The four research questions have been addressed by means of a survey. The survey has been designed by the authors observing the development activities (by looking at the life-cycle of user stories and participating in daily stand-up meetings), and talking with developers to get insights about the CD pipeline and the way it has been implemented in ING NL.

The survey also addresses specific knowledge needs at ING NL, triggered by one of the authors who is affiliated with ING NL. The survey is organized into four sections, plus a preliminary section aimed at investigating demographic characteristics of the respondents (age, years of experience, years in ING NL, and technical skills). Overall, it consists of 48 questions, plus five demographics questions. The questionnaire allowed the respondent to select among one or more answers (in most cases multiple answers were allowed), and if needed to provide a textual answer (*i.e.,* by selecting "Other" among the options). In Tables 1–4, we give an abbreviated summarization of the questions we asked developers.[1]

Table I reports the questions aimed at addressing $\mathbf{RQ}_1$. As it can be noticed, besides the first question, mainly aimed at understanding whether DevOps engineers share the methodology being adopted, all other questions clearly refer to agile development practices and in particular to Scrum [20]. For example, we ask questions about sprint planning and user story progress monitoring, but also specific questions about how DevOps manage issues and schedule/perform refactoring actions. We asked specific questions about refactoring as in this study we were particularly interested to understand activities related to technical debt management.

Specific questions about managing technical debt – reported in Table II – compose the second part of the survey, aimed at addressing $\mathbf{RQ}_2$. We ask questions about (i) how developers document source code by means of comments, (ii) how they perform code review, (iii) what kinds of problems do they detect by means of code review and using automated smell detection tools, as well as how they remove problems by means of refactoring, and (iv) whether they perceive that smells are usually introduced because of deadline pressure.

The third part of the survey aims at addressing $\mathbf{RQ}_3$ and features questions about testing activities, as shown in Table III. After having asked a question aimed at understanding whether DevOps engineers use TDD, we asked questions about the effort spent on writing test cases and to what extent test cases are kept up-to-date. Also, we ask questions about information and strategies being used to derive test cases for different testing levels. Then we ask questions about test execution (*i.e.,* to what extent is this done within private builds or on the CI server), and how developers assess test effectiveness and deal with low coverage.

Finally, the fourth part of the survey addresses $\mathbf{RQ}_4$ and is composed of questions (see Table IV) about (i) promotion policies[2], (ii) how DevOps engineers handled build failures, (iii) how they used branches and (iv) how frequently they pushed their changes.

C. Survey Operation

The survey questionnaire was uploaded onto a survey management platform internal to ING NL, and the candidate participants were invited using an invitation letter explaining the general goals of the survey, its length and estimated time to complete, and highlighting how its results have the purpose of understanding the CD process within ING NL, also in order to identify directions for its improvement.

Respondents had a total of three weeks to participate to the survey, and a reminder to those who did not participate yet was sent every week. In total, we obtained 152 filled questionnaires, *i.e.,* we achieved a return rate of 85%. We left respondents the choice not to answer a question. The number of answers for each question is reported in the last column of the tables enumerating the questions. Overall, the median number of responses per question was 129 for $\mathbf{RQ}_1$

[1]The original survey with all questions is available at https://figshare.com/s/fa8c4e11fc9fa4b8f8cb

[2]A promotion entails the selection of a release candidate and subsequent deployment to the correct environment [21].

TABLE I
DEVELOPMENT PROCESS - QUESTIONS (S/M/R STANDS FOR SINGLE, MULTIPLE, OR RANKING ANSWER QUESTION).

#	Summarized Question	S/M/R	# of Resp.
Q1.1	What is your software development methodology?	S	150
Q1.2	Is the product vision always clear to you? Why? Why not?	S,M	149
Q1.3	Do you prefer to use a physical board or an electronic one? Why?	S,M	125
Q1.4	During a sprint why do you add some tasks to the already planned ones?	R	138
Q1.5	Which is the main topic you address during the sprint retrospective?	S	138
Q1.6	Which is the average percentage of completed user stories at the end of a sprint?	S	138
Q1.7	Which Scrum metrics do you usually collect?	M	128
Q1.8	Which is the main reason why a "done" user story comes back to "in-progress"?	S	130
Q1.9	Do you consider non-functional requirements as definition of "done" of a user story?	S	130
Q1.10	Which kind of non-functional requirements do you consider as definition of "done" of a user story?	M	120
Q1.11	You detect a defect that was previously resolved: how to deal with it?	S	129
Q1.12	Do you usually schedule refactoring tasks? Why?	S	129
Q1.13	Which priority do you usually assign to refactoring tasks?	S	128
Q1.14	How frequently are refactoring tasks included in other tasks?	S	128
Q1.15	Which is the average percentage of scheduled refactoring tasks that are completed at the end of a sprint?	S	123

TABLE II
MANAGING TECHNICAL DEBT - QUESTIONS (S/M/R STANDS FOR SINGLE, MULTIPLE, OR RANKING ANSWER QUESTION).

#	Summarized Question	S/M/R	# of Resp.
Q2.1	To what extent do you introduce method and class level comments?	S	116
Q2.2	To what extent do you introduce statement level comments?	S	116
Q2.3	To what extent do you update code documentation/comments?	S	116
Q2.4	Do you perform code review? Why?	S,M	116
Q2.5	How do you usually detect code smells?	M	110
Q2.6	Which of those problems do you usually detect? (null pointers, interface misuse, memory leaks, unreachable code, unused variables, uninitialized variables)	M	116
Q2.7	Which of these bad design/implementation choices do you usually detect during code reading? (function having huge size, method with many responsibilities, high module coupling, module exposing its attributes)	M	116
Q2.8	Which source code metrics do you usually look at?	M	116
Q2.9	Do you sometimes do poor implementation choices because of near deadline?	S	116
Q2.10	Do you usually use a tool in order to do code refactoring? Why?	S	116

TABLE III
TESTING - QUESTIONS - QUESTIONS (S/M/R STANDS FOR SINGLE, MULTIPLE, OR RANKING ANSWER QUESTION).

#	Summarized Question	S/M/R	# of Resp.
Q3.1	Do you use TDD (Test Driven Development)? Why/why not?	S,M	125
Q3.2	Which percentage of your time do you spend on writing tests?	S	124
Q3.3	How frequently do you review and (if necessary) update the tests for every change to production code?	S	124
Q3.4	Do you usually test the code written earlier by others? Why (not)	S,M	124
Q3.5	Which strategy do you usually use to categorize inputs for each test case?	S	122
Q3.6	Which information do you need in order to perform Unit Testing?	M	122
Q3.7	Which information do you need in order to perform Integration Testing?	M	122
Q3.8	Do you usually automate the generation of the test cases?	S	122
Q3.9	In which kind of testing do you usually automate the generation of the test cases?	M	21
Q3.10	Which kinds of testing are executed automatically? Why (not)?	M	120
Q3.11	Where do you test code?	M	120
Q3.12	Which percentage of written tests are executed?	S	120
Q3.13	Do you always run all test cases together? Why?	S	120
Q3.14	How frequently do tests pass?	S	120
Q3.15	Which types of code coverage do you measure?	M	107
Q3.16	Which is the average percentage of code coverage that you usually score during unit testing?	S	103
Q3.17	How do you deal with low coverage?	S	103
Q3.18	Which of those test metrics do you find useful?	M	116
Q3.19	How do you react to a failure?	R	116

TABLE IV
CONTINUOUS INTEGRATION - QUESTIONS - QUESTIONS (S/M/R STANDS FOR SINGLE, MULTIPLE, OR RANKING ANSWER QUESTION).

#	Summarized Question	S/M/R	# of Resp.
Q4.1	Promotion policies: what do you do when you are ready to push code on the master branch?	S	112
Q4.2	How do you deal with failures at building/packaging time?	S	112
Q4.3	Branching issues: how do you deal with parallel development?	S	112
Q4.4	When do you usually push your changes?	S	112

TABLE V
RESPONDENTS' DEMOGRAPHICS: AGE, YEARS OF DEVELOPMENT EXPERIENCE, AND YEARS SPENT IN ING NL.

Age		Years of experience		Years spent in ING NL	
< 30	24	< 1	2	<1	24
30-39	55	1	5	1	27
40-50	60	2-5	19	2-5	56
> 50	13	6-10	32	6-10	19
		>11	94	>11	26

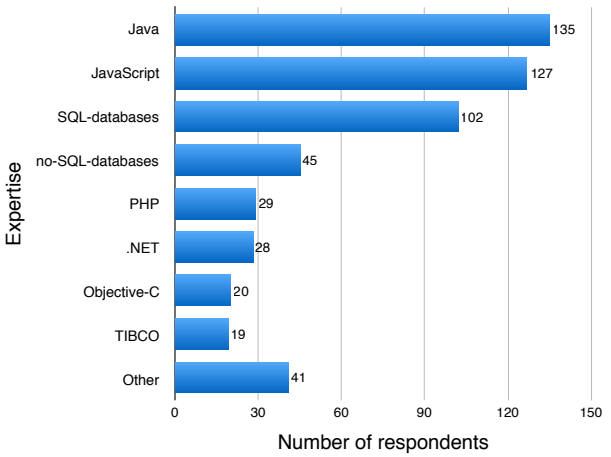


Fig. 3. Technological Knowledge.

questions, 116 for $\mathbf{RQ}_2$, 120 for $\mathbf{RQ}_3$ and 112 for $\mathbf{RQ}_4$. Only for one question (Q3.9, dealing with specific aspects of test automation) the number of answers was below 100, *i.e.,* 21. Both the overall return rate and the return rate for the single answers are higher than typical return rates for software engineering surveys conducted in industry, which often range between 10% and 25% [22], [23]. The high return rate gives us confidence that our survey accurately reflects the opinion of the sampled developers.

IV. STUDY RESULTS

In this section, we highlight key results of our study that directly address the research questions from Section III.

A. Respondents' demographics

Table V and Fig. 3 report demographics information about the study respondents, and namely their age, years of experience, years spent in ING NL, and their main skills (multiple answers were allowed). Most of the respondents are relatively senior both in term of age and development experience (the majority of them has an age between 30 and 50, and over 11 years of experience). The main technological expertise they possess are related to Java or JavaScript programming, and both relational and NoSQL databases.

B. RQ_1: What are general development practices within the Continuous Delivery pipeline?

Methodology. When we asked about the kind of methodology being adopted in the development process (Q1.1) almost all developers (97%) mentioned they use on Scrum as development methodology. At the same time, the product vision (Q1.2) is clear to 68% of the respondents only. One important reason for the lack of clarity is due to frequent changes, which are pretty common in agile development.

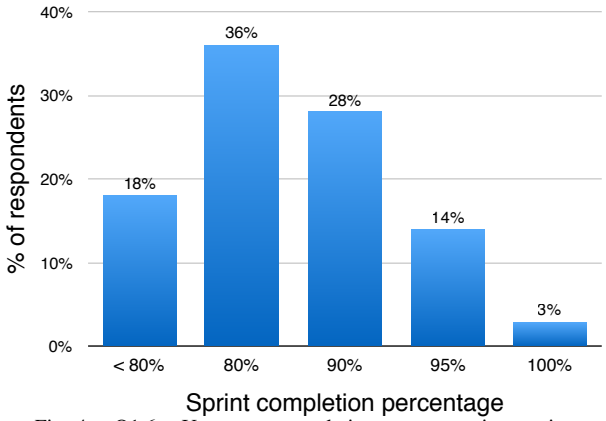


Fig. 4. Q1.6 – User story completion percentage in a sprint.

Interestingly, while most of the respondents (69%) prefer to use an electronic Scrum board (Q1.3), there is a quite high percentage (31%) still preferring a physical Scrum board[3]. On the one hand, they say that an electronic Scrum board facilitates distributed team work (84%), and provides automated calculation of sprint progress metrics (59%). On the other hand, a physical board is always visible in the room (90%), and improves the team cohesion (64%).

Sprint Management. Developers declared that, during a sprint, they add some tasks to the already planned ones (Q1.4). As a main reason for that, 60% of them indicate bug fixing, followed by missing detailed requirements (33%) and only 7% mentioned high-level, business requirements missing during the planning.

During the sprint retrospective (Q1.5), *i.e.,* the meeting in which the sprint activities were discussed in order to understand what went well, what went wrong, and how things can be improved for the next sprint, developers mainly discuss and try to harmonize the way they work (88%). Few responses concern bad implementation (1%), the product not meeting functional (1%) or non-functional (1%) requirements, and other issues (7%).

Fig. 4 reports the average percentage of completed user stories at the end of a sprint (Q1.6). In most cases, respondents agree that no less than 80% of user stories are completed. Other than dealing with functional requirements, user story completion concerns with dealing with different kinds of non-functional requirements, where developers consider as high priority requirements security (89%), reliability (86%) and maintainability (82%). The main monitoring mechanisms for the sprint progress (Q1.7) are the sprint burn-down (60%, tracking the sprint completion), and the velocity, *i.e.,* the number of story points [24, page 87] per hour (58%). A small percentage of respondents consider the number of defects postponed (3%), or the technical debt occurred (7%) as important indicators which are able to influence the completeness of a user story.

[3]https://en.wikipedia.org/wiki/Scrumban#/media/File:Simple-kanban-board.jpg

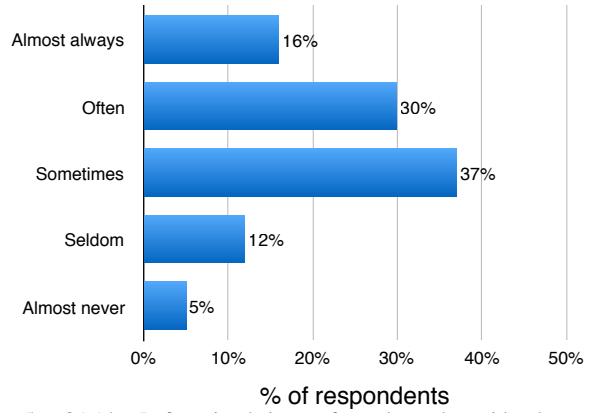


Fig. 5. Q1.14 – Refactoring being performed together with other tasks.

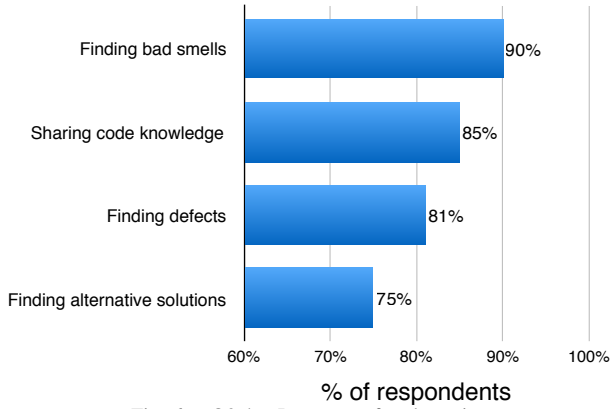


Fig. 6. Q2.4 – Purposes of code review.

In some cases, a completed user story may be rolled back to "in-progress" (Q1.8), but mainly because developers realize that functional (34%) or non-functional (25%) requirements are not completely implemented. Only in 22% of the cases does this occur because of changes in users' expectations. 7 respondents (5%) explicitly specified that in case they realize changes in requirements, *e.g.,* because of changed users' expectations – they rather open a new user story than reopening a previously closed one. One respondent even clarified that a "done" user story should be considered to be in production already, and therefore should not be reopened again.

When a previously resolved defect occurs again (Q1.11), 52% of the respondents indicate that they open a new issue anyway. This can either indicate a careful approach in which developers try to keep the new occurrence of the defect separated from the previous one.

Refactoring activities. When we asked about refactoring tasks (Q1.12), 64% of respondents indicated that refactoring is usually properly scheduled. The main reasons for refactoring include improving program comprehension (87%), allow making changes easier (77%), and help to find bugs (24%). Those who not schedule refactoring tasks, they do it either because they are too time consuming and take effort away from feature implementation tasks (27%), or because they do not clearly perceive refactoring advantages (9%). A large proportion of respondents (64%) indicate other reasons. For example, they mentioned that "refactoring is just performed as it pops up", that they "naturally consider refactoring as part of other development tasks", or that "code should be made maintainable right away". Also, some respondents indicated planning reasons, *i.e.,* part of the user story effort calculation. Last, but not least, someone indicates that all depends on the size of the refactoring activity to be performed is, *i.e.,* small refactorings are performed together with development, whereas larger ones are kept separate.

When being scheduled (Q1.13), refactoring tasks often have a medium priority (70%) than other tasks, with 9% assigning a high priority and 23% a low priority. Indeed, 42% of respondents indicate that more than 80% of the planned refactorings within a sprint are actually completed (Q1.15).

Differently from what Fowler reported [25], refactoring tasks are often performed together with other tasks, as shown in Fig. 5 (Q1.14). Only 5% of respondents declare that they clearly separate refactoring from other tasks.

C. RQ_2: What are the practices adopted to manage technical debt?

Source code comments. The first block of questions we asked about managing technical debt concerned the way and the extent to which developers comment source code. Respondents said they almost always (23%), often (34%), and sometimes (24%) introduce class-level and method-level comments (Q2.1). Instead, as expected only 3% and 15% of the respondents introduce statement-level comments always and often, respectively (Q2.2). Still, 38% of the respondents introduce them sometimes.

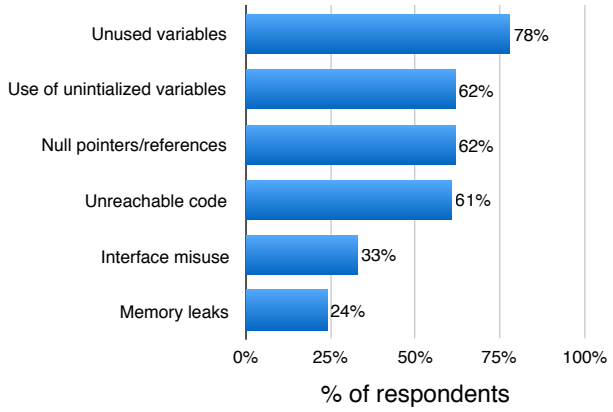
In line with the CD process, and with the aim of preserving program understanding, 79% of the respondents' update comments immediately when changing the source code, while 13% postpone such changes to a specific phase aimed at producing/updating documentation (Q2.3).

Code reviews. Code review (Q2.4) is adopted by almost the whole set of respondents (95%) and, as shown in Fig. 6, the obvious purposes are detecting bad smells (90%) and finding defects (81%). However, code review is also used a lot to share code knowledge (85%), or to find alternative ways for implementing a feature (75%). These results are partially in line with the observations on the code review process at Microsoft [26] and on open-source projects [27]. At Microsoft, finding defects was the most important motivation, followed by code improvement and finding alternative solutions, while sharing code ownership was only ranked seventh.

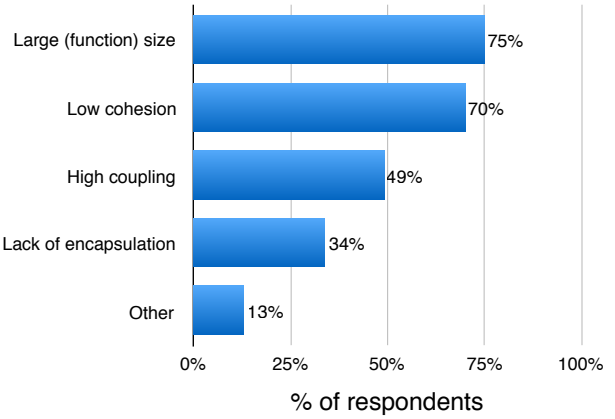
Analysis of bad code smells. Respondents indicate (Q2.5) that code reviews are the premier way for detecting code smells (92%), while 63% of the respondents also use static analysis tools[4]. The main problems detected (Q2.6) either by means of automated or manual code review are reported in Fig. 7 (a): the majority indicated as main problems detected unused (78%) or uninitialized (62%) variables, null pointers[5]

[4]Due to confidentiality reasons, we cannot disclose the list of tools being used.

[5]Including null references in languages not directly using pointers, *e.g.,* Java.



(a) Q2.6 – Software defects



(b) Q2.7 – Bad design choices

Fig. 7. Problems detected by automated and manual code review.

(62%), and unreachable code (61%). In terms of bad design choices (Q2.7) (Fig. 7 (b)) as expected respondents mainly deal with large function size (75%). Surprisingly, they focus more on low cohesion (71%) than high coupling (49%), although in previous studies [28], [29] the latter has been perceived by developers as a negative factor for software maintainability and comprehensibility.

The majority of respondents (58%) rejected the common wisdom that poor implementation choices occur because of deadline pressure (Q2.9), confirming previous results obtained in the open source [30]. Interestingly, almost all respondents (88%) annotate these poor implementation choices: hence the principle of self-admitted technical debt – previously investigated in open source [31], [32] – is pretty well applied in ING NL. When time allows, developers try to refactor such smells using some automated tool support: 71% use tools automatically enacting refactoring actions, such as the Eclipse refactoring infrastructure, and not tools recommending refactorings (*i.e.,* tools such as JDeodorant [33]), while 29% do it manually. The latter indicate as main reason for manual refactoring the lack of adequate tools (76%) but also the lack of trust in automated refactoring tools (15%), confirming results studies showing the dangers of using automated tools for applying refactorings [34].

Metric collection. Other than identifying specific defects,

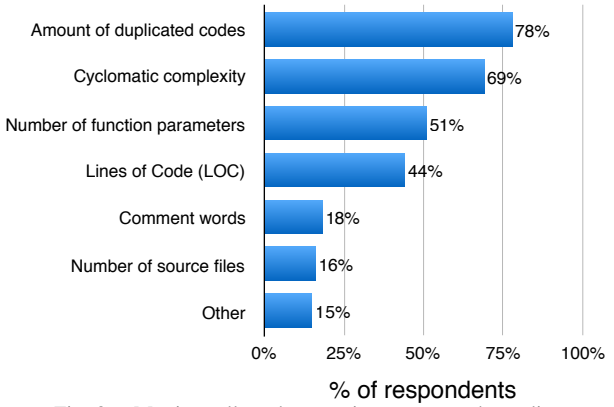


Fig. 8. Metrics collected to monitor source code quality.

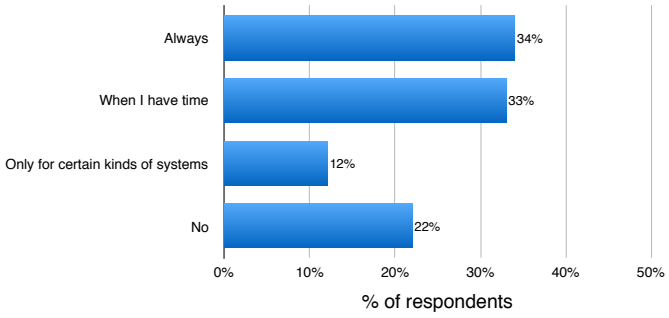


Fig. 9. Q3.1 – Adoption of Test-Driven Development.

developers collect a series of metrics to monitor source code quality (Q2.8). The main metrics used are reported in Fig. 8. Surprisingly, the most important metric is the amount of duplicated code (78%) which traditionally is considered as a kind of bad smell too. Other than that, the cyclomatic complexity (69%, again, indicator of some code smells such as Complex Method) and number of function parameters (51%, indicator of Long Parameter List bad smell). Only 44% of respondents mention LOC.

D. RQ_3: What are the testing practices adopted within the Continuous Delivery pipeline?

Test-Driven Development (TDD) and Testing in general. TDD is the practice of "driving development with tests" [35]. As reported in Fig. 9, 34% of the respondents say they always use TDD (Q3.1). 33% answered they use TDD for certain kinds of (sub) systems, and 12% use it when time pressure allows. 22% do not use TDD at all. Respondents reported to adhere to a TDD style when they can create or have existing unit (96%), integration (53%), acceptance (25%), or performance (15%) tests for the functionality they are about to implement. Reasons for not using TDD are mainly related to TDD not being directly applicable for many types of code changes, *e.g.,* when developing graphical user interfaces (59%), which triggers the need for other kinds of tools, such as capture-replay tools. Another important reason was TDD's time consuming nature (33%).

Regarding testing in general, 47% of the respondents allo-

cate between 25% and 49% of their time for testing (Q3.2), and 31% more than 50% of their time. Developers in the WatchDog study [35] estimated to spend on average around 50% of their time on automated, codified testing, very closely resembling the estimates in our study.

One may wonder how accurate developers' self-estimations are and whether developers who claim to use TDD do indeed apply it. Beller *et al.* [35] found in their WatchDog study that developers spent a quarter of the work time on testing (instead of half, which they originally estimated), and that, even when they reported that they were using TDD, developers practically never applied it strictly [35]. A similar observational study on developers' testing habits could identify whether and how these findings apply in our given context. Casual evidence from another context (not at ING NL) suggests that, some developers were referring to acceptance testing with the Framework for Integrated Testing (FIT) [36] as TDD, but meant Behavior-Driven Development (BDD) [37]. Generally, our survey answers suggest that quality assurance through testing is a crucial concern at ING NL. A significant amount of manual work is required for TDD in particular and testing in general. Automated tool support, including test case generation, might help further reduce it. When asking a specific question on automation of test generation (Q3.8, Q3.9), 17% of the respondents indicated they use some techniques and tool to automate test case generation.

A factor that highlights the cost of testing and that TDD may indeed be followed is the answering to the question of continuous updating of test suites for every change (which is in line with the idea of CI). Most of the respondents claim they almost always (58%) or often (28%) update tests when changing production code (if necessary).

Testing strategies and criteria. We found that developers make use of specific testing strategies such as black box testing relatively seldom (Q3.5). 52% of the respondents say they do not use any strategy. As regard black box testing, only 20% and 19% use equivalence class testing and category partitioning [38] criteria respectively. Regarding white box testing, the main criteria being used (Q3.15) are statement coverage (94%), branch coverage (84%), multiple condition coverage (68%), and in some cases path coverage (42%). Most of the respondents picked multiple options indicating that depending on the feature under test, they choose whichever strategy is most suitable.

Overall, about statement coverage (Q3.12), 84% of the respondents indicated they try to achieve a coverage level of at least 80%. Other than that, as it is shown in Fig. 10, developers rely on a number of different metrics, mostly the number of failed/passed/blocked test cases (77%) but, for example, also related on how well test cases cover user stories (27%).

For unit testing purposes, test cases are often written using (Q3.6) requirements for black box testing (78% of respondents) and source code for white box testing (80%). Only 24% of respondents rely on models. As for integration testing (Q3.7), code is less used (43%) while developers mainly rely on module interfaces (66%).

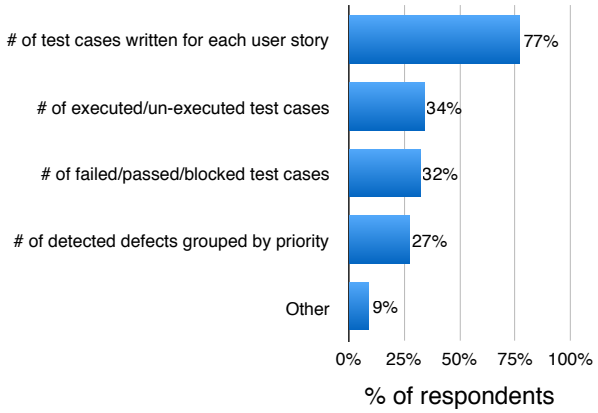


Fig. 10. Q3.18 – Test metrics.

E. RQ_4: How is Continuous Integration performed?

The first question we asked (Q4.1) was about the use of testing in private builds before opening a pull request. As one can expect, results indicate how the use of CI changes the promotion management policies one may adopt. While in principle [39] one can be tempted to promote code as long as it compiles, with CI developers are encouraged to perform some tests (*e.g.,* unit testing) in the private builds. Indeed, 97% of the respondents indicated they actually do it, while only 3% let the CI perform all tests when builds are performed.

In case of build breaking changes (Q4.2), 96% of the developers confirmed that they interrupt their implementation activities and focus on fixing the build.

To minimize conflicts, the majority of respondents (62%) create a feature branch and merge it later in the master branch, even if only 22% of them perform a daily merge (Q4.3). Regarding the frequency of pushing changes in the master branch (Q4.4) results indicate that 60% of developers push changes whenever a small piece of a task is completed, while 30% do it only when a whole task is completed. Only few respondents (10%) push changes more than one time in a week.

V. Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. In a survey, such threats may mainly occur because respondents could possibly interpret a question in a different way than it has been conceived, possibly producing misleading results. For example, when answering to Q3.1, and as explained in Section IV-D, it is possible that developers believe they are applying TDD, while this is not the case. Whenever possible, the quantitative findings obtained with the survey were confirmed by the observations made by one of the authors, who observed the ING NL development process for three months. Possibly, the most suitable way of complementing the survey would have been a follow-up live interview or a longitudinal study, which is plan for future work.

Threats to *internal validity* concern with factors that could have influenced our results. One such factor could be the evaluation apprehension [40]. For example, answers to Q2.9 indicated that deadline pressure is not a major cause for poor

implementation choices. Another threat is related to the survey return rate. We have shown that the overall return rate is quite high (85%), and generally higher than other surveys conducted in the area of software engineering.

Threats to *external validity* concern the generalization of our findings. The obtained findings are clearly and intendedly confined to the specific case of ING NL, and may or may not generalize to other organizations, even within the same domain. In some cases, *e.g.,* for the use of code reviews, we have shown how our results confirm what seen in other organizations [26].

VI. Related Work

In recent years, researchers have conducted different studies on the adoption of CI and CD in industry and open source.

Experience reports. Laukkanen *et al.* [41] interviewed 27 developers at Ericsson R&D to understand their perception of CI. They observed that developers face many technical and social challenges when adopting CI, such as the test infrastructure. An industrial experience report from Kim *et al.* [42] details a CI setup at the package level, rather than at source code line level, hence increasing the responsibility of package maintainers. Ståhl and Bosh [11] conducted a literature review on CI practices and found that different software development projects use different CI implementations because of several contextual factors such as size, longevity, budget, competences, organizational structure, or geographical distribution. This suggests that contradicting elements in the results of our survey when compared to other studies can possibly be explain by variations in context.

Build failures. A challenge in CI is dealing with build failures, which might negatively impact developers' productivity. Thus, researchers have investigated the most common causes of these failures. For example, Miller [8] at Microsoft reported that, for the Service Factory system, build failures are mainly due to compilation failures, failing tests, static analysis tool issues, and server failures. Seo et al. [43] at Google found that most failures are due to dependency-related issues between software components. In contrast, Beller et al. [5] analyzed build failures due to test executions. In particular, they found that testing is an important part in CI and it is also the most frequent reason for build failures.

Benefits of CI practices. Other researchers have investigated the effect of CI on code quality and developers' productivity. For example, Miller [8] reported that for the Service Factory system the CI cost was about 40% of the cost of an alternative (non-CI) process achieving the same level of quality. Deshpande and Riehle [44] analyzed commit data from open source projects and found that, differently from industrial development, in open source the adoption of CI has not yet influenced development and integration practices. However, Vasilescu *et al.* [45] mined GitHub projects and found that CI makes teams more productive and improves the likelihood of pull request mergers, without sacrificing the projects' quality.

Tools and techniques. Brandtner *et al.* [46] focus on improving common CI practices, in particular, they developed a platform that dynamically integrated data from various CI-tools and tailors the information for developers. In other work, Brandtner *e*t al. [47] propose a rule-based approach to automatically profile stakeholders based on their activities in version control systems and issue tracking platforms. platform, namely SQA-Mashup, which dynamically integrates data from various CI-tools and tailors the information for developers. Elbaum et al. [48] presented regression test selection techniques to make continuous integration processes more cost-effective.

While the studies described above focused on CD experience itself or introducing new tools and techniques, our survey conducted in ING NL focuses more on the development practices within the CD pipeline, with a particular emphasis on how DevOps engineers manage technical debt and perform testing.

VII. Conclusions

This paper reported results of a survey – conducted with 152 developers of a large financial organization (ING Netherlands) – about their use of Continuous Delivery. The survey featured questions about (i) the development process and task management, (ii) managing technical debt, (iii) testing, and (iv) Continuous Integration activities. The main findings of the survey suggest that:

- While refactoring is properly scheduled, contrarily to both common wisdom and to Fowler stated [25], it is often performed together with other development activities, as it is considered as part of a user story effort, and this prevents to release poorly maintainable source code.
- Respondents tend to "self-admit" technical debt when writing source code, in order to be able to fix it when possible. Instead, they reject the hypothesis that such smells are introduced because of deadline pressure. Then, they use both code reviews and automated tools to identify and refactor code smells.
- The majority of developers mention they use TDD, although we do not know whether they are strictly applying TDD. At the same time, quality assurance in the form of (manual) testing requires a significant portion of the allocated time for a sprint.
- The use of a Continuous Integration infrastructure encourages developers to test their changes using private builds, and to give very high priority to fix build breakages.

In conclusion, our survey-based study shows how practices such as TDD or the identification and refactoring of bad smells (with the help of automated tools) are put in practice in a large organization as ING NL, sometimes confirming common beliefs, sometimes contradicting them. This study requires replications in other organizations, and needs to be complemented with other studies, *e.g.,* case studies, controlled experiments and longitudinal field studies, in which developers' activities can be closely observed to have a better understanding of their behavior when working within a CD pipeline.

ACKNOWLEDGMENTS

The authors would like to gratefully thank all the study participants as well as all developers from ING NL that provided precious inputs for the planning of this study.

REFERENCES

[1] G. Booch, *Object Oriented Design: With Applications*. Benjamin Cummings, 1991.

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] M. Fowler and M. Foemmel, "Continuous integration," 2006. http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf.

[4] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[5] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of travis CI builds with GitHub," *PeerJ PrePrints*, vol. 4, 2016.

[6] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proc. Int'l Conf on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 470–481, IEEE, 2016.

[7] D. Goodman and M. Elbaz, "It's not the pants, it's the people in the pants — learnings from the gap agile transformation," in *Agile 2008 Conference*, pp. 112–115, 2008.

[8] A. Miller, "A hundred days of continuous integration," in *Agile 2008 Conference*, pp. 289–293, 2008.

[9] J. Downs, B. Plimmer, and J. Hosking, "Ambient awareness of build status in collocated software teams," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 507–517, IEEE, 2012.

[10] P. Debois, "Just enough documented information. Agile 2008, Toronto, Canada http://www.jedi.be/blog/2008/10/09/agile-2008-toronto-agile-infrastructure-and-operations-presentation/," 2008.

[11] D. Stahl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[12] F. Erich, C. Amrit, and M. Daneva, "A mapping study on cooperation between information system development and operations," in *Product-Focused Software Process Improvement - 15th International Conference, PROFES 2014, Helsinki, Finland, December 10-12, 2014. Proceedings*, pp. 277–280, 2014.

[13] CloudBees, "Jenkins – https://jenkins.io," June 2016 (last accessed).

[14] SonarSource, "SonarQube – http://www.sonarqube.org," June 2016 (last accessed).

[15] JFrog, "JFrog Artifactory – https://www.jfrog.com/artifactory/," June 2016 (last accessed).

[16] A. S. Foundation, "Apache Kafka – http://kafka.apache.org," June 2016 (last accessed).

[17] MongoDB, Inc., "MongoDB – https://www.mongodb.com," June 2016 (last accessed).

[18] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[19] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[20] K. Schwaber, *Agile Project Management with Scrum*. Microsoft Press, 2004.

[21] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson, 2010.

[22] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 12–23, ACM, 2014.

[23] E. K. Smith, R. T. Loftin, E. R. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 89–92, IEEE, 2013.

[24] M. Cohn, *User Stories Applied: For Agile Software Development*. Addison Wesley, 2004.

[25] M. Fowler, *Refactoring: Improving the design of existing programs*. Addison Wesley, 1999.

[26] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 712–721, IEEE, 2013.

[27] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: which problems do they fix?," in *Proceedings of the working conference on mining software repositories (MSR)*, pp. 202–211, ACM, 2014.

[28] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 692–701, IEEE, 2013.

[29] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *Proceedings of the joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 354–364, ACM, 2011.

[30] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proc. Int'l Conf. Softw. Engineering (ICSE)*, pp. 403–414, 2015.

[31] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pp. 91–100, IEEE, 2014.

[32] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pp. 315–326, ACM, 2016.

[33] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proc. Int'l Conf. on Software Engineering (ICSE)*, pp. 1037–1039, 2011.

[34] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 185–194, ACM, 2007.

[35] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proc. joint meeting European Software Engineering Conf. and Int'l Symposium on Foundations of Softw. Engineering (ESEC/FSE)*, pp. 179–190, ACM, 2015.

[36] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, 2005.

[37] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *Proceedings Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 383–387, IEEE, 2011.

[38] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[39] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java (3rd Edition)*. Pearson, 2009.

[40] N. B. Cottrell, D. L. Wack, G. J. Sekerak, and R. H. Rittle, "Social facilitation of dominant responses by presence of others," *Journal of Personality and Social Psychology*, vol. 9, no. 3, pp. 245–250, 1968.

[41] E. I. Laukkanen, M. Paasivaara, and T. Arvonen, "Stakeholder perceptions of the adoption of continuous integration - A case study," in *Agile Conference (AGILE)*, pp. 11–20, 2015.

[42] S. Kim, S. Park, J. Yun, and Y. Lee, "Automated continuous integration of component-based software: An industrial experience," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pp. 423–426, IEEE, 2008.

[43] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *Proc. Int'l Conference on Software Engineering (ICSE)*, pp. 724–734, ACM, 2014.

[44] A. Deshpande and D. Riehle, *Continuous Integration in Open Source Software Development*, pp. 273–280. Boston, MA: Springer US, 2008.

[45] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proc. joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 805–816, ACM, 2015.

[46] M. Brandtner, E. Giger, and H. C. Gall, "Supporting continuous integration by mashing-up software quality information," in *2014 Software Evolution Week — Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 184–193, IEEE, 2014.

[47] M. Brandtner, S. C. Müller, P. Leitner, and H. C. Gall, "Sqa-profiles: Rule-based activity profiles for continuous integration environments," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 301–310, IEEE, 2015.

[48] S. G. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proc. Int'l Symp. on Foundations of Software Engineering (FSE)*, pp. 235–245, 2014.