How Junior Developers Deal with Their Technical Debt?

Fabian Gilson
University of Canterbury
Christchurch, New Zealand
fabian.gilson@canterbury.ac.nz

Miguel Morales-Trujillo
University of Canterbury
Christchurch, New Zealand
miguel.morales@canterbury.ac.nz

Moffat Mathews
University of Canterbury
Christchurch, New Zealand
moffat.mathews@canterbury.ac.nz

ABSTRACT

Technical debt is a metaphor that measures the additional effort needed to continue to add more features in a software due to its inherent decrease in code quality. Most software systems suffer from technical debt at some point so that dedicated tools and metrics have been developed to monitor such debt. Alongside tools, appropriate engineering practices must be put in place by the development team to keep that debt at an acceptable level. In this empirical study, we observed and surveyed Scrum development teams composed of experienced students in order to understand their quality-related processes on a year-long academic project. We found that (1) students do use static analysis tools of many forms, but their actual usage is limited due to time pressure; (2) retrospective and non-constraining feedback on code quality has little to no effect, even when given regularly during the course of the project; and (3) junior developers value composite quality indicators (*e.g.*, maintainability, reliability in *SonarQube*), even if they do not fully understand their meaning. From our findings, we propose a series of recommendations, both technical and methodological, on how to train junior developers to understand and manage technical debt.

CCS CONCEPTS

• **Software and its engineering** → **Agile software development**; **Maintaining software**; • **Social and professional topics** → *Software engineering education*;

KEYWORDS

software quality metrics, static code analysis, empirical study

ACM Reference Format:
Fabian Gilson, Miguel Morales-Trujillo, and Moffat Mathews. 2020. How Junior Developers Deal with Their Technical Debt?. In *International Conference on Technical Debt (TechDebt '20), October 8–9, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387906.3388624

1 INTRODUCTION

Alongside ensuring a program behaves as expected, software engineers must take care of the quality of the software to guarantee it can evolve with new requirements or technologies and does not fall apart as it grows [3]. In order to keep track of the quality of a code base (*e.g.* programs potentially written in many different programming languages, configuration files, tests, code documentation), rules and metrics have appeared in order to characterise and detect quality problems in the form of *"antipatterns"* at the design level [31] or *"code smells"* at the implementation level [22]. These quality issues have been recognised to have a detrimental effect on software systems and even software development projects in general [8]. The amount of necessary work to clear these quality issues is often referred to as the *"technical debt"* of a software [18].

Furthermore, earlier empirical studies have demonstrated the relationship between program comprehension (*i.e.* the ability to understand the goals of a program from its implementation code) and antipatterns [1] as well as their impact on the fault-proneness of a software (*i.e.* the likelihood that bugs can be introduced into the code base) [20]. However, Ramasubbu *et al.* identified that three interconnected dimensions come into play while making decision to *"optimise"* the technical debt, *i.e.* customer satisfaction, software reliability and technological disruption, such that clearing all the debt is not always feasible or economically viable [25].

Therefore, learning about technical debt is also learning about the economic trade-offs together with the internal quality of a software product [15]. However, from our experience, teaching about code quality metrics to students in a theoretical way is not effective for two intertwined reasons: (a) when developing software systems in their assignments or small-scale projects, students tend to prioritise *"quick and dirty"* solutions to well designed ones as soon as the program produces the expected output; and (b) software solutions the students are asked to implement are usually of a small scale, span across a very limited period of time and often correspond to a one-time effort instead of a longer term activity.

Within the Software Engineering degree at the University of Canterbury, we designed a year long team-based project course where students develop a mid-scale product in multiple iterations following the Agile's Scrum method [6, 26]. In this paper, we are interested in exploring students' perception of code quality metrics and practices as well its evolution over time. Therefore, the aforementioned project course offers an interesting controlled environment to evaluate how junior developers deal with technical debt over a longer period than typical one-off coding assignments. The contributions of this paper are the following:

(1) an evaluation of the usage of code quality metrics, tools and practices by junior developers over a year-long software development project;
(2) a discussion of an effective teaching strategy for code quality assurance techniques from both a technical and methodological perspective.

In this paper, we discuss similar studies, including teaching of techniques to deal with the technical debt in Section 2. We introduce

TechDebt '20, October 8–9, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7960-1/20/05…$15.00
https://doi.org/10.1145/3387906.3388624

our research questions in Section 3 and give more details on the population under study, the setting of the project and the typical tasks the students are doing, mimicking an industrial environment in Section 4. We introduce our threefold process to gather and analyse data in Section 5 and present the results in Section 6. We discuss these results in Section 7 before concluding in Section 8.

2 RELATED WORK

Alves *et al.* reviewed existing techniques and tools to manage and visualise technical debt [2]. Similarly, Sharma and Spinellis conducted a literature review on software smells looking at their characteristics, their origin, their effects and their detection techniques [27]. Fontana *et al.* discussed different technical debt indices available in existing tools [17]. Ernst *et al.* investigated what processes and tools are used in the industry to manage softwareâĂŹs technical debt [13]. Digkas *et al.* looked into what kind of repayment actions are taken by practitioners [11]. In this research, we investigate both the perception of software smells and their metrics as well as the course of actions taken by junior developers to manage the technical debt in a controlled academic but realistic setting.

Simpson and Storer report on a similar project course as we depict in Section 4 [29]. However, their teaching, assessment and reporting approach are very different to ours as they involve different industry partners for all teams (which inherently induces a higher pressure on product delivery than learning activities) and do not report specifically on the evolution and perception of technical debt by students. Tonin *et al.* address the awareness of technical debt in an eXtreme Programming project course using a specifically designed *"technical debt board"* the students were encouraged to use [30]. In this research, we refrain to specifically focus on technical debt issues proactively and observe the behaviour of junior developers in a realistic environment where the quality of each code base naturally decrease. We want to avoid explicitly and artificially require students to document their technical debt to limit cognitive biases in students' development activities. Bai *et al.* present a platform integrating static code analysis reports (SonarQube) in the feedback sent to students [4]. However, the authors do not evaluate the effects of this feedback on students' behaviour. Haendler and Neumann experimented with a serious game dedicated to teach about refactoring activities [19]. Our strategy is to let students experiment about technical debt in a realistic, yet academic and supervised environment within a software development project rather than a dedicated learning activity. Last, Kuhrmann *et al.* conducted an online survey about the current state of practice in software engineering education [21]. However, the authors did not investigate static code analysis or technical debt specifically.

3 RESEARCH QUESTIONS

In this research, we are interested in evaluating the position junior software developers take regarding code quality metrics and technical debt. To this end, we want to investigate:

RQ1 *How regularly junior developers use software analysis tools to monitor the evolution of the technical debt in their software development project?*

As a first step, we investigate the frequency at which junior developers look at quality metrics. By frequency, we intend to evaluate both the time span between subsequent checks as well as when in the development process these checks occur, *i.e.* sequence of activities that were executed right before and right after the quality check.

RQ2 *What are the triggering events leading junior developers to take corrective actions in order to reduce the technical debt?*

We analyse the reasons why junior developers take actions to decrease the technical debt. We are interested in analysing if milestones (*e.g.*, client demonstrations, formal feedback) have an incentive effect on reducing the debt or if other qualitative factors come into play such as maintainability problems, error-proneness of badly-designed code or subjective thresholds on a certain amount of technical debt.

RQ3 *What metrics junior developers value the most when evaluating the technical debt of their software development project?*

After having a better idea of the timeline of quality-related activities, we investigate what metrics junior developers are looking at and which are mostly disregarded. As a side effect, we will also gather feedback on potential missing metrics from existing tools used throughout the project as well as visualisation the developers would be interested to have.

4 POPULATION AND CONTEXT

4.1 Population under study

The population under study is composed by 60 students enrolled in the *Software Engineering Group Project* at the University of Canterbury. This class is composed by 3^{rd} year students enrolled in a Bachelor of Engineering (with Honours) in Software Engineering (BE(Hons) SENG) for which this course is compulsory, accompanied by 3^{rd} year students in Bachelor of Science in Computer Science (BSc CS). BE(Hons) SENG students have a prior software development experience in an academic project and typically 400 hours of internship in the industry. BSc CS students usually have a lower development experience but faced small-scale coding assignments in their curriculum.

For the purposes of this study, a junior developer is a software developer with little practical experience who exhibits a tendency to develop software with the main goal of *"making it work"* rather than creating good quality software.

4.2 Project Course Setting

The software engineering project is a year-long development project (over 24 lecture weeks) mimicking a professional environment where students work in teams of eight members. Throughout the year, each team delivers seven versions of their product and use a set of tools to manage their source code and its quality.

The teams follow a Scrum methodology with all typical ceremonies (*i.e.* planning 1 and 2, stand-ups, product reviews and retrospectives). All students are developers and they are coordinated by a Scrum Master being a tutor[1]. Each member of the teaching staff involved in that course takes a different role in the project: a Product Owner (PO), a technical expert and a process and quality expert.

[1]Tutors are either staff members or experienced students that performed outstandingly in the course earlier.

The course has little formal lectures, but technical workshops are organised on top of scheduled Scrum ceremonies.

The project runs over seven sprints of around 3 to 4 weeks (*i.e.* lecture weeks only). At first, all teams develop the same product based on the same backlog that is enriched as the first half of the year goes. Typically after the fourth sprint, each team is encouraged to bring its own ideas into its own product in order to differentiate it from the others. Since the product reviews are preceded by public presentations of the latest features, all teams can observe the progress of each other.

4.3 Typical Tasks

Members of a team are considered equal, all being developers under the Agile terminology. Students are required to log the time spent on the project on a digital board and are expected to hit around 300 hours at the end of the year. Each student takes part actively in all software development activities.

4.3.1 Planning 1 and 2. Planning sessions are conducted by students solely, but they have access to the PO during planning 1 to clarify unclear stories or advise when making trade-offs. The output of planning sessions is monitored on the team's Scrum board.

4.3.2 Development. Each student is expected to implement a significant part of the system and is required to participate in design and documentation activities. Students are also encouraged to pair-program and use dedicated tags in their commit messages for the teaching team to monitor the contribution of each developer. Students are expected to follow a similar branching model to Driessen's gitflow [12] with peer-reviewed merge requests. The development is monitored using custom mining tools on the team's *GitLab*[2] repository locally hosted at the University.

4.3.3 Quality assurance. Students are required to write unit, acceptance and manual tests that need to be logged against predefined tags in the digital Scrum board. They also comply to a continuous integration pipeline where all automated tests are expected to be run. Students are asked to use static code analysis tools throughout the project, usually starting from the third sprint when the students are taught about technical debt and code quality metrics.

4.3.4 Product review. After each sprint, a presentation of the latest features is done in front of all other teams and is followed by a formal product review conducted by another team such that each product is evaluated by another team and the couples *reviewer-reviewee* change every sprint. The review consists in both a black box testing of the delivered stories and a white box testing of the code quality (*e.g.,* test coverage, test readability, code structure), supported by looking at the aforementioned analysis tools. The teaching team conducts another in-depth product review (*e.g.*, delivered stories, code documentation, wiki).

4.3.5 Continuous learning. Students are engaged in a continuous improvement cycle. After each sprint, students write and receive (anonymized) self/peer-feedback and the teaching team sends individual and team feedback regarding team-based behaviour, the progress of the software and its overall quality.

[2]See https://gitlab.com.

4.4 The 2019's Instance

In 2019, the students built a web-based system to manage trips mixing trip planning and social media. Among common features, we asked for various types of user permissions, the ability to create trips, and interface with third party APIs (e.g., geo-localisation). From that common basis, distinctive functionalities were added by different teams so that all products were targeting a different audience (*e.g.,* music bands, business travellers, web *influencers*).

5 DATA GATHERING

5.1 Extraction of Code Quality Metrics

Teams were asked to deploy a *SonarQube*[3] instance in order to monitor the quality of their code base on top of *Sonarlint*, a plugin from *IntelliJ IDEA*[4]. This requirement was made clear as a technical story in all teams' backlogs early in the project (third sprint). We opted for *SonarQube* for the following reasons:

(1) it is open source, cross platform and supports a wide range of programming languages;
(2) it is web-based, allowing students and the teaching team to access it from everywhere, with a relatively simple and straightforward deployment process with extensive official and unofficial documentation; and
(3) it offers off-the-shelf integration with *GitLab* for merge requests and *Sonarlint/IntelliJ* for built-in quality metrics.

Students are also taught about code metrics, technical debt and quality assurance in a companion course being a co-requisite to that project course. Additionally, we deployed our own instance of the tool to monitor the evolution of the quality after each sprint and give qualitative feedback on that aspect. Using the tool, we can have a first set of insights regarding the two first questions:

RQ1 significant decreases in the amount of code smells will likely be indicators of manual actions taken by team members to manage the technical debt instead of side effects of development activities.

RQ2 if significant decrease peaks appear close to milestones (product review) or in a short period after receiving qualitative feedback from the teaching team, we can postulate assessment deadlines and feedback have an impact on the quality-related activities.

We can combine the evolution of quality metrics as described above with a deeper analysis of the code pushed together with the commit messages to identify if these commits are dealing with new features (adding new semantics) or fixing quality issues (semantically equivalent) in order to validate our quantitative observations.

5.2 Survey

A survey (mostly Likert-type scale) has been designed to understand many aspects of our three research questions and cross-check the investigation of the data extracted from the static code analysis tool. The full survey is presented in Table 1.

We mapped the survey questions to our RQs as follows:

[3]See https://www.sonarqube.org
[4]See https://www.jetbrains.com/idea and https://www.sonarlint.org

Table 1: Survey questionnaire (detailed multi-valued options are visible in Figures 1 to 9)

	Questions	Scale type
	Prior knowledge of Software Quality Assurance	
1.	Prior to taking *SENG302*, to what extent did you know about Technical Debt?	Likert scale
2.	Prior to taking *SENG302*, which of these practices did you used either at university, on personal projects or in industry?	Multi-select list
3.	Prior to taking *SENG302*, how would you describe your level of knowledge in Programming (e.g., Java, Javascript)?	Likert scale
	Software Quality Management Practices	
4.	From the following practices, which one did your team put in place for each merge requests during your *SENG302* project?	Multi-select list
5.	From the same practices as above, could you rank them in order of importance in your own opinion (1=most important, 6=less important)?	Rank order
6.	What process or tool did your team used to check the quality of the code in general during your *SENG302* project?	Multi-select list
	Usage of Static Code Analysis Tool	
7.	How often were you looking at a static analysis tool report during your *SENG302* project?	Likert scale
8.	When you used a static analysis tool during your *SENG302* project, what metrics did you look at?	Multi-select list
9.	From the same metrics as above, could you rank them in order of importance in your own opinion.	Rank order
10.	To your own opinion, when looking at a static code analysis report with poor values at one or more metrics, which ones of these metrics were triggering a code refactoring task to lower the technical debt of your *SENG302* product?	Multi-select list
11.	When looking at a static analysis tool report during your *SENG302* project, what actions were you typically taking if any metric you reckon as important had a poor value (e.g., low maintainability, high cyclomatic complexity)?	Likert scale
	Final thoughts	
12.	After completing this questionnaire, is there anything else you would like to add?	Open question

RQ1 questions 7 and 11 deal with **timing**-related behaviour and actions regarding technical debt;

RQ2 questions 4 and 10 deal with **event**-related behaviour and actions regarding technical debt;

RQ3 questions 6, 8 and 9 deal with specific quality-assurance practices/metrics and their ranking by students.

Questions 4 and 5 are control questions to evaluate if students actually put in place quality-assurance practices and if static code analysis tools were actually used. Additionally, we collected data on any preexisting knowledge and experience (questions 1 to 3) and left an open question for final thoughts (question 12).

5.3 Focus Groups

As a last step, a focus group has been organised in the following weeks after the survey to dig deeper in the previous findings of both the analysis of the historical data and the survey, mainly to answer *"why"* questions. The focus group was prepared following guidelines proposed by Breen [7] and addressed the following Focus group Questions (FQ):

FQ1 Could you describe the process your team was following when accepting merge requests, specifically regarding quality assurance practices and the usage of static code analysis tools?

From the survey, a ranking of quality-assurance practices (question 5) and critical metrics (question 11) can be identified, but we want to investigate if the ranking and metrics are linked to a sequential execution of those practices (**RQ1**). We also expect to get more insights regarding the evolution of technical debt during the project (**RQ2**).

FQ2 What impediments were you facing while preparing or handling merge requests that prevented you to use a static analysis tool systematically?

From the survey, the team's policy and practices related to quality-assurance can be investigated, including the usage of static analysis tools during the project's lifecycle (question 4, 6 and 7). We therefore want to discuss the impediments that prevented teams to systematically apply quality assurance practices such as testing the new features or using static analysis tools (**RQ1** and **RQ2**).

FQ3.1 Static analysis tools offer many quality metrics (*i.e.* composed *e.g.*, reliability, maintainability or simple, *e.g.*, test coverage, comments). What metrics did you find the most or less insightful and why?

FQ3.2 As a follow-up question, what metrics or visualisation were missing in current tools (*e.g.*, *SonarQube*, *SonarLint*)?

Where, in the survey, we investigate metrics valued by students, we want to understand why junior developers found those metrics useful or not, or what were they missing (**RQ3**).

6 DATA ANALYSIS

In this section, we summarise the analysis of extracted data from *SonarQube* reports, detail the results of the survey and report the main outcomes of the focus group.

Table 2: Code Smells and Final SonarQube Indicators for All Projects.

Teams	Alpha	Bravo	Charlie	Delta	Echo	Foxtrot	Golf	Hotel
Feedback 1	186	190	60	226	35	19	498	182
1 day after	192	190	61	223	34	19	469	267
3 days after	168	219	61	240	36	19	489	152
1 week after	176	198	59	244	54	19	525	152
Feedback 2	294	297	105	309	134	61	368	375
1 day after	200	292	102	320	145	21	380	361
3 days after	200	291	102	320	145	21	380	361
1 week after	200	290	103	321	145	29	382	361
Feedback 3	300	294	104	329	145	19	381	362
1 day after	300	291	102	365	145	18	388	361
3 days after	301	304	148	211	149	12	381	377
1 week after	163	314	141	216	149	12	410	407
Feedback 4	198	371	147	325	151	20	510	443
1 day after	204	367	140	329	159	12	493	442
3 days after	161	72	160	380	174	6	536	458
1 week after	159	79	159	382	185	8	547	471
Reliability	C	A	C	C	C	A	D	E
Security	B	A	E	B	B	A	B	B
Maintainability	A	A	A	A	A	A	A	A
Lines of code	9303	19635	9779	20424	23452	12853	14389	15265
Duplication rate	1.3%	2.1%	0.9%	18.3%	6.5%	1.7%	2.1%	2.3%

6.1 Static Code Analysis Reports

As mentioned in Section 4.3.3, we required teams to deploy their *SonarQube* instance during the third of the seven sprints. We could then formally refer to the output of the tool in our sprint feedback. Table 2 reports on the number of smells calculated by our *SonarQube* instance before and after each feedback sent to students[5]. All statistics were calculated on a daily basis on each team's main branch only. For the sake of completeness, we also supply the latest values for the main quality indicators calculated from *SonarQube*[6], the total number of lines of code and the code duplication rate for each projects.

As can be seen in Table 2, no visible effect can be observed for 5 out of 8 teams where the technical debt globally increased. By the latest sprints, Team Alpha spent some time to reduce their debt, mostly one week after Feedback 3 showing a decrease of 45% of the number of smells (from 301 to 163), all fixed in a merge request commented as code quality improvement.

A similar, but earlier and more long-term quality improvement action has been taken by team Foxtrot in two stages: after feedback 2 (65% decrease) and by the end of the project (another 60% in multiple steps), to end with all *SonarQube* indicators evaluated at A and a minimal debt of 8 smells, all of these decreases being also identifiable by explicit commit messages. That particular team kept their technical debt very low throughout the project since their first code quality-focused feedback.

Last, team Bravo seems to have cleaned up their code from most of their smells by the end of the project with a decrease from 367 to 72 smells, representing an 80% decrease (with a slight increase to 79 smells for the final deliverable), therefore moving all *SonarQube* indicators back to A.

We also looked at the evolution of the debt before sprint reviews in a similar fashion as presented in Table 2 (not reproduced here for space reason). The analysis shows no clear effect but in most cases the debt was increasing the few days prior a sprint review, sometimes steeply, probably due to last minute development.

6.2 Survey Results

The survey was implemented in an online tool[7] and a participation link was sent to the whole cohort of students enrolled in the aforementioned project course. The survey was opened from 9^{th} to 24^{th} October 2019 and the participation was voluntary, without any external incentive and after the projects were graded. On a total of 60 students, 25 responses were collected (41.67% overall response rate). The respondents were coming from all 8 teams (ranging from 1 to 5 respondents per team).

6.2.1 Prior knowledge. We wanted to ensure the students did not know much about technical debt prior taking the course, but had some experience with other quality-assurance practices and programming in general.

As shown in Figure 1, no respondents indicated to have an extensive knowledge of technical debt and 11 (44%) indicated to have *"heard about the term only"*. This result was expected since technical debt is discussed for the first time in the companion course.

[5] Team names have been changed and randomised to guarantee anonymity.

[6] *"Reliability"* represents the potentiality to trigger unwanted behaviour; *"security"* represents any kind of well-known vulnerabilities (*e.g.*, cleartext storage of vulnerable information, cross-site request forgery); and *"maintainability"* represents the total number of code smells.

[7] See https://www.qualtrics.com

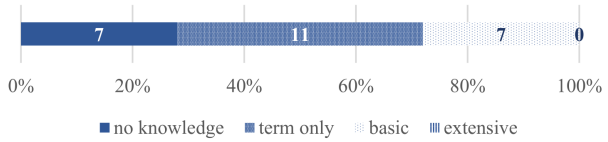

Figure 1: Q1 - Prior knowledge of technical debt.

Other practices related to code quality-assurance are part of the students' curriculum prior taking that project course, such as unit and acceptance testing. We therefore asked the students to indicate which of these practices were known to them. As visible in Figure 2, all students indicated being familiar with unit testing, but only two with regression testing and static analysis tools.

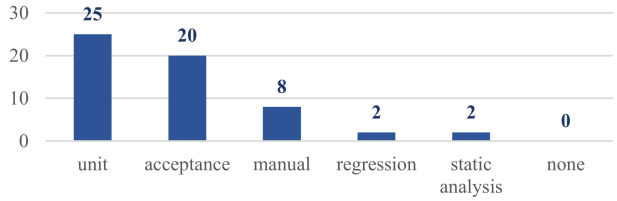

Figure 2: Q2 - Prior knowledge of quality-related practices.

As a control question for external validity, we asked the students to self-assess their programming experience in order to evaluate the applicability of our results to similarly-trained junior developers. Despite simple self-assessment questions are widely used and known as reliable in general [28], we focused on the amount of written code instead of a subjective level of proficiency as we have access to the students results in that particular project course which give us another insight of their coding proficiency. As results, 1 student (4%) indicated having little programming experience (few assignments), 14 (56%) having participated to small-scale projects (under 10.000 LOC) and 10 (40%) having been involved in larger-scale projects (more than 10.000 LOC).

6.2.2 Software Quality Practices. The second set of questions concerned the engineering practices related to quality-assurance applied by the students and their team during the project. On top of *compiling and building*, we asked about important aspects of common best practices including testing [5], coding standards [9] and merge request management [10].

As can be seen in Figure 3, successful *building* of code and the presence of meaningful in-code *documentation* (including comments) were identified as team's best practices by 24 out of 25 respondents (96%). *Testing* and *coding standards* were also checked by respectively 21 (84%) and 19 (76%) of the respondents. However, more advanced quality assurance practices such a merge request checklists or the usage of static analysis tools were not so popular with respectively 13 (52%) and 12 (48%).

When looking at the individual answers, no respondents from the same team fully agreed on the practices used inside the team, but members from 7 out of 8 teams mentioned the usage of static

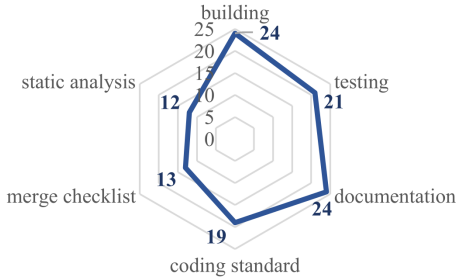

Figure 3: Q4 - Quality-related practices applied in projects.

analysis tools and respondents from 6 teams mentioned the usage of merge checklists.

From the same practices, we asked the students to rank them from the most to the less important according to their own opinion, as depicted in Figure 4.

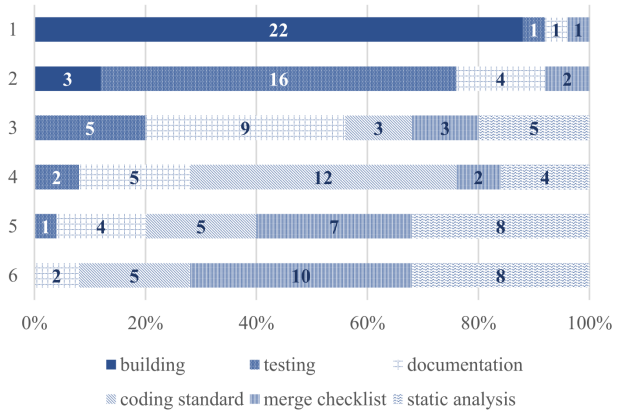

Figure 4: Q5 - Ranking of quality practices, from most (1) to less (6) important by respondents.

Similarly to Figure 3, *building* is clearly ranked first with 22 respondents (88%), but *testing* has been ranked second by 16 respondents (64%) where *documentation* comes mostly in third position with 9 respondents (36%). Starting from the third place, the ranking is more disputed where no practices obtain an absolute majority. *Merge checklist* and *documentation* are the only practices having been ranked at all six places by the respondents. The usage of static analysis tool is ranked from the third to last place, being the most picked one at fifth place (8 respondents, 32%).

The last question investigated the usage of practices and tools to ensure the quality of the written code during the project. The results, ordered from the most to the less used practices and tools are presented in Figure 5.

Manually or automatically testing the code was the most common process to ensure the quality (23 responses, 92% of respondents). Next, *SonarQube* was picked by 21 respondents (84%), right before manually screening the code that is selected by 19 respondents (76%). *Sonarlint* was used by 17 respondents (68%). While looking

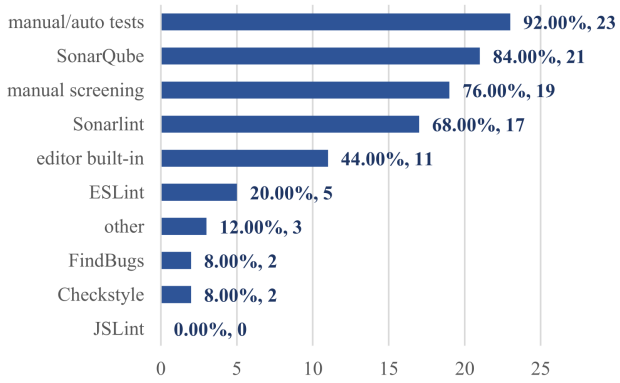

Figure 5: Q6 - Processes and tools used by teams in project.

at the individual responses, one respondent did not mention any tool, but picked *manual screening* and *test* as answers.

6.2.3 Usage of Static Code Analysis Tools and their Metrics. The last series of questions were related to the usage of static code analysis tools within the context of the project course. We wanted to understand how the students integrated the usage of these tools in their development activities. Figure 6 shows how often during the project the students were referring to static analysis tools.

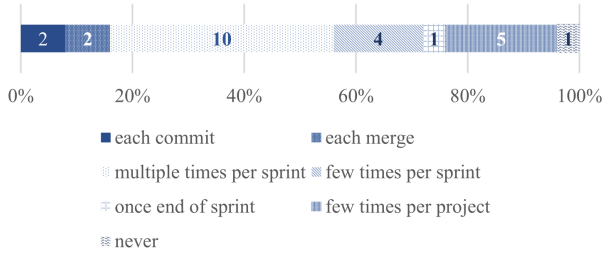

Figure 6: Q7 - Usage frequency of tools during project.

A majority of surveyed students (19 out of 25) were using tools at least *once a sprint*, corresponding to 76%, from which 4 have integrated the tools as part of either their *commit* or code *merge* review processes. Most of the reported usage of tools was *multiple times per sprint*, representing 40% of the answers. The one respondent that mentioned not using any static analysis tool selected *ESLint*[8] and builtin tools as answers to question 6, such that this answer seems to be a misinterpretation of what static analysis tools are since *ESLint* belongs to that category.

In two separate questions, we asked the students to indicate their level of interest in different quality metrics available in typical static analysis tools and more particularly in *SonarQube* since students were using it in their project. The list contained a mixture of composite indicators and more simplistic metrics, all being taught or discussed in either the companion course or the project course itself. We compare the metrics of interest (see Table 1 question 8) and the metrics that were triggering corrective actions when showing poor values (question 10) in Figure 7 where the metrics are sorted on the number of responses at question 8.

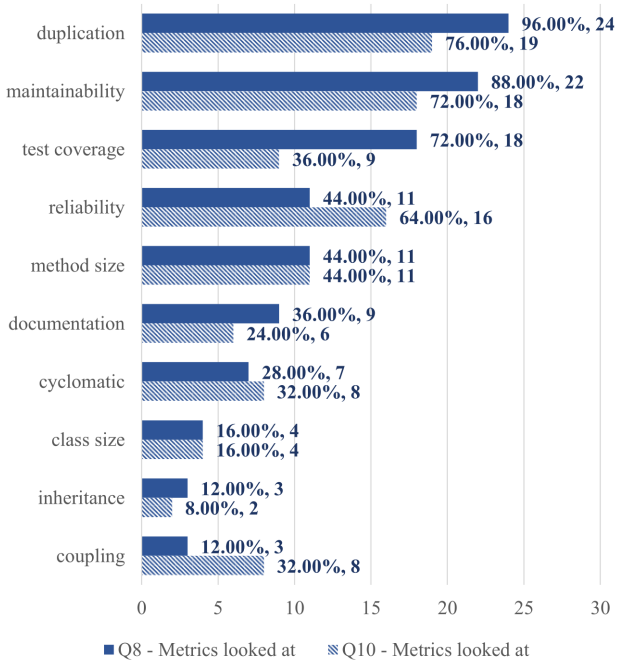

Figure 7: Comparison of results from Q8 and Q10 on consideration of quality metrics.

As can be seen in Figure 7, *duplication* gathers the highest results in both questions (resp. 24 out of 25 respondents for question 8 and 19 for question 10) followed by the *maintainability* index (22 and 18 respondents), gathering all code smells (*e.g.,* hardcoding of configuration values, unused import, commented out code). The results for the remaining metrics are sometimes very different in both questions, especially the *test coverage* that is highly regarded as important (18 respondents, 72%), but triggering significantly fewer corrective actions (9 respondents, 36%). An inverse relationship, but to a lesser extent, can be observed regarding the *reliability* index (*i.e.* potentiality to trigger unwanted behaviour) that is looked at by 11 respondents (44%) but was encouraging 16 respondents (64%) to take corrective actions.

Similarly to the ranking of practices presented in Figure 4, we asked the students to rank the same set of metrics from questions 8 and 10. The final ranking is shown in Figure 8.

The two extremes show clear tendencies with the *reliability* metric ranked first by 12 respondents (48%) and *documentation* rate as the last one (13 respondents, 52%). The remaining results are rather mixed with no clear tendencies except *duplication* ranked fourth by 9 respondents (36%), but being mostly present high in the ranking, from the second to the fifth place. Also, *test coverage* was mainly ranked in the top four places (resp. gathering 7, 6, 6 and 5 responses) showing its high importance to the respondents.

[8]See https://eslint.org

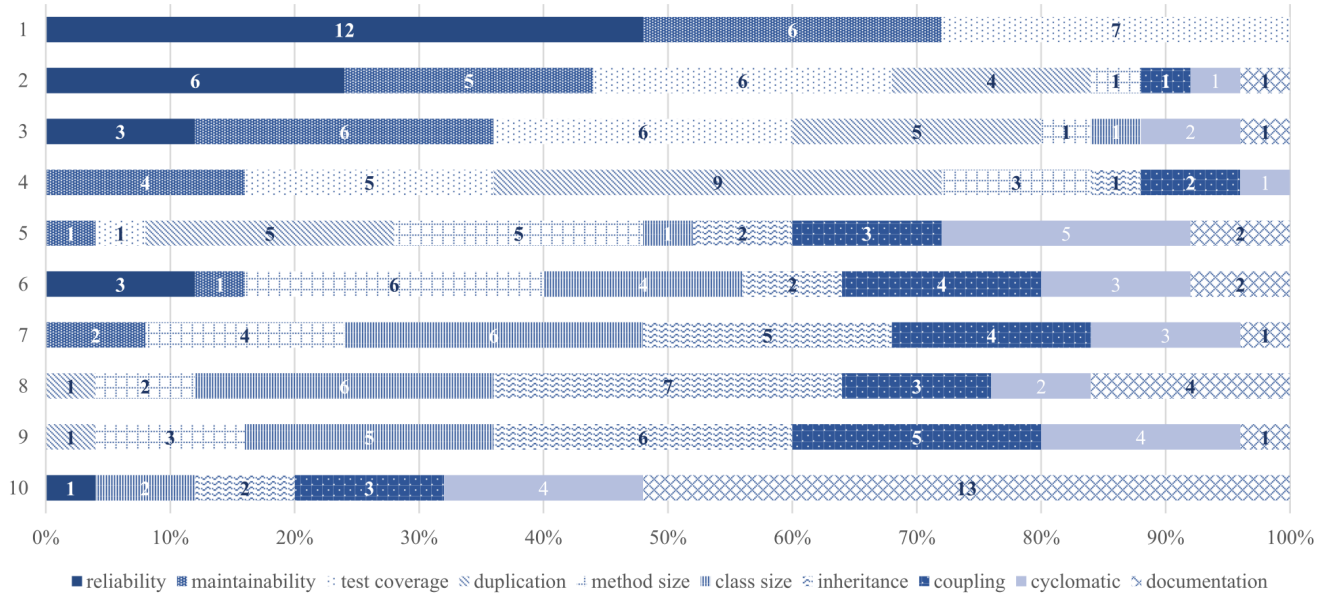

Figure 8: Q9 - Ranking of quality metrics from most (1) to less (10) important by respondents.

Cyclomatic complexity [23] shows mixed results despite its relative simplicity and intelligibility where it achieves its best result in fifth place with 5 responses (20%), being the same result as the *duplication* metric. Composite indices seem to be of particular interest with the *reliability* and, to a lesser extent, the *maintainability* indices that were mostly ranked in the top four places.

Last, we investigated the typical behaviour when students were witnessing poor values to metrics of interests. Figure 9 shows the course of actions taken after identifying quality problems such as an increase in the amount of technical debt.

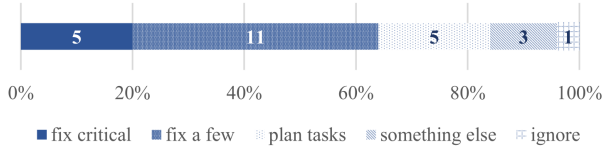

Figure 9: Q11 - Actions taken when poor values were shown in static analysis tool.

Immediate actions into the code were taken according to 16 respondents either to fix the identified *critical* issues (5 respondents, 20%) or *a few* (11 respondents, 44%). Another 5 respondents were creating dedicated tasks in their digital Scrum board or issue tracker for the team to be aware of the problem and potentially schedule corrective actions. From the 3 respondents answering *other*, 2 mentioned they were either fixing all critical or a subset right away, or they were logging the issues in the team's issue tracker. The remaining one from that lot mentioned the time pressure was too high to concentrate on any code quality aspects.

6.3 Focus Group

Participation to the focus group was voluntary and 6 students from 5 teams have been recruited based on their willingness expressed verbally or in the survey, 3 of them not having answered to the survey. The focus group took place on 2nd December 2019, was facilitated by the authors and the discussions were audio-recorded and transcribed for further analysis following an iterative approach [24]. We summarise the main findings per focus group questions rather than presenting the full category-based formalisation since such report of a deeper analysis goes beyond our initial research questions and the scope of this paper.

6.3.1 FQ1 - Process followed when accepting merge requests. The participants reported having followed and adapted aforementioned gitflow [12] with protected main branch and merge requests, as expected. They also mentioned the review process evolved throughout the project from a looser to a more stringent and procedural code review process and, finally entrusting their team members to apply the right level of scrutiny in their review as everyone's experience had increased. Reviews were typically conducted by one or two other team members, often one that was not familiar on that part of the software to offer a critical eye on the code under review. Two participants noticed significant variances in the level of depth of code reviews across team members and one noticed *"lots of people [being] nice [with each other]"*.

All participants mentioned their team had defined and documented guidelines early on during the project, typically during the second or third sprint. Those guidelines focused mainly on code style (*e.g.*, usage of camel-case, maximum nesting levels, unit tests), but little systematic use of or referral to static analysis tools. All participants mentioned they often ignored warnings from within their IDE, especially closer to end of sprints, corroborating our

last observation in Section 6.1 regarding increasing technical debts close to sprint deadlines.

One participant mentioned their team relied on pair-programming to ensure the code was written at the expected level of quality, lowering the need for in-depth code review and therefore speeding up the development pace.

6.3.2 FQ2 - Impediments preventing the usage of static code analysis tools. All participants mentioned facing technical difficulties while setting up *SonarQube*. The most missed feature was the ability to run an analysis on any branch prior merging[9]. Participants were also missing the interaction with their continuous integration pipeline or development tools, despite the ability to do so[10].

Participants were also annoyed by too many *"false positives"* regarding string constants in error messages *"which would often end up making horribly unreadable code"*. But they also recognised that they could learn about features of the programming language they were not aware of as well as discovering error prone code.

Last, all participants mentioned the pressure to add new features to the code base prevented them to spend too much time on fixing smells. Two participants mentioned their respective teams dedicated time-boxed periods to fix smells here and there and two other participants reported their (common) team decided to work preventively to keep the debt low, making it easier to fix a lower number of smells regularly.

6.3.3 FQ3 - Insightful and missing metrics from static code analysis tools. All participants agreed the tools they had access to (*e.g.*, *SonarLint*, *SonarQube*) offered sufficient metrics for their day-to-day development work, such that the low level of usage was related to technical difficulties and a late realisation on *"how much [they] should be maintaining [their] code"* and that they *"didn't think that the code had to be that maintainable"*. All participants also advocated for dedicated workshops to be trained on properly configure and use those tools, apart from and alongside the project itself.

Among the metrics the participants particularly mentioned, we noted the size of classes, complexity of functions, level of nesting, code duplication rate and test coverage. Interestingly, three participants mentioned they did not fully understand *SonarQube* composite indices because of a lack of time, but another participant mentioned *"[they] were so happy when [they] got that green tick"*.

7 DISCUSSION

Following well-known guidelines [32], we report the results of our empirical study from both a software engineering and a teaching perspective, and review its limitations.

7.1 Summary of Findings

7.1.1 RQ1 - Usage of software analysis tools. A vast majority of surveyed students reported the use of static analysis tools (24 out of 25, 96%). Since we required the students to use such tools, this result was expected, but 19 respondents, corresponding to 76%, used static code analysis tools at least once per sprint, indicating regular checks to ensure the quality of their product was on track. However, it appears the usage of analysis tools were more of a personal initiative than a team one. The focus group highlighted that, even if students appear to look at the reports or warning issued from these tools at least once per sprint, their actual usage was rather limited, mainly due to a lack of time and/or knowledge.

7.1.2 RQ2 - Course of actions to reduce the technical debt. A majority of surveyed students indicated that they were taking immediate corrective actions either to fix all critical or a few code smells (23 survey respondents, 92%). During the focus group, we discovered that either teams were enforcing high quality early on the project, therefore handling new debt easily, or ignoring most of the smells because they found it irrelevant (*e.g.*, usage of string constants) or the time pressure was too high on delivering new features. Still, some participants mentioned the tools helped at improving their code quality as they were developing with little hints or suggestions about language features they were not aware of.

From the data mined from *SonarQube*, retrospective feedback from the teaching team seems to have little effect since we could observe some improvements right after written feedback was given for 2 teams out of 8. A third team also reduced significantly his technical debt close to the final deadline. One participant in the focus group confirmed that sprint feedback had incentive effect on their decision to reduce the debt. Another participant mentioned they were conscious of the problem, but could not commit manpower at reducing the debt because of other priorities, hence an overall limited effect of feedback on the code quality.

The focus group also highlighted that all teams suffered from poor decisions and shortcuts taken early on the project and that they did not realise early enough that they needed to hit a higher quality standard to keep implementing new features efficiently.

7.1.3 RQ3 - Quality metrics valued by junior developers. Aggregated metrics (using traffic light flags in *SonarQube*) are attracting the respondents attention while looking at static code analysis tools and this has been partially confirmed by the focus group. However, the time pressure and the lack of a fully dedicated workshop done separately from the project itself prevented students to fully benefit from the usage of these tools.

Among simpler metrics, code duplication was identified with high importance during the survey as well as the level of nesting and class sizes during the focus group. Test coverage is another metric gathering lots of interest and was particularly scrutinised during code review or a mean of increasing the code quality, as discussed with the focus group. On the other hand, complexity metrics were mostly disregarded by students.

7.2 Advice for Teachers

This retrospective study targeted the usage of static code analysis tools in an academic year-long project aiming to reproduce an industrial environment as much as possible. From the data we gathered, we believe the usage of such tools helped teams to keep track of their technical debt and improve the code quality overall as 2 teams had a high level of code quality and the other teams kept an acceptable level of debt, according to the maintainability indices computed by *SonarQube* on the projects' code base.

[9]Branch analysis is a paid feature of *SonarQube* we had not provided the students with.
[10]These features were available in *SonarQube*, but students did not dig into the tool configuration possibilities by themselves as we were expecting them to do so.

However, technical difficulties did have hindered the students' experience and must not be neglected. Despite the possibility to do so and our implicit expectations that the students would invest time to configure their tools beyond what we supplied them with, all students expressed the need for a deeper and even more stringent integration of analysis tools within their software development environment and continuous integration pipeline, reducing the manual burden of the usage of these tools during merge requests.

An interesting, but not surprising observation from the focus group relates to trade-offs students had to make between customer, technology and reliability [25]. In order to recreate an industrial setting, students feel pressured to deliver new features instead of concentrating on the code quality and then realise halfway through the project that shortcuts and low code quality prevent faster development. Despite this is an expected learning outcome of such longer term projects, all participants to the focus group recommended to strictly enforce a high level of quality metrics for all delivered products throughout the year and therefore offer a mixture of *learning-by-doing* and more classical teaching philosophy.

They also recommended to add dedicated technical workshops alongside the project itself to be encouraged to learn more about technical debt outside their development duties in the project itself as well as to understand their technical aspects more deeply.

7.3 Limitations

7.3.1 Conclusion validity. Because of a limited number of participants to the survey (25), the present research is under typical threshold for statistical significance and representability, such that we are mainly looking for general tendencies. Furthermore, in order to increase our confidence in the results, we gathered data from multiple sources for triangulation purposes where (1) we analysed historical data from static code analysis reports to ensure the results from the survey are in line with the actual timeline of the projects in term of technical debt and (2) we targeted our questions during the focus group to crosscheck the findings from both the survey and data mined from *SonarQube*.

7.3.2 Construct Validity. The survey has been beta-tested by researchers in software engineering and implemented in a commercial tool taking care of the initial gathering and analysis of the data. Additionally, the survey and the research proposal have been reviewed by researchers outside the software engineering and received prior approval from the Ethics Committee from the University.

Research involving students may raise a social threat grieving the results. We minimised that potential issue by making the participation to the survey and focus group voluntary without any form of external incentives and after completion of the course.

The deployment of *SonarQube* was made a requirement but no marks were specifically given to encourage its usage on a regular basis, hence the results discussed in Section 7.1.1. Still, students reported facing technical difficulties with the tool which have potentially influenced their user experience as well as our results.

7.3.3 Internal Validity. Prior students' experience with technical debt was limited (see Section 6.2.1 Q1 and Q2), but a majority used static analysis tools in their development activities on a regular basis, making us confident in our results. This aspect has been crosschecked with the focus group, even if the actual level of expertise and usage varied significantly between students and teams.

7.3.4 External Validity. Using students as proxy for junior developers is subject of great debate in the software engineering community [14, 16]. However, in this research we had a double objective:

(1) retrospectively understand the behaviour of students/junior developers facing increasing technical debt in a controlled environment mimicking an industrial setting on a longer period than typical course assignments;
(2) improve our teaching material in a year-long project course (and its companion course) where we train students mostly in a *learning-by-doing* philosophy with regular feedback and continuous learning (*e.g.,* Scrum-based software development, version control, software quality assurance).

We then believe our conclusions can be generalised to similar software development projects using a similar teaching philosophy on similarly trained students or junior developers freshly graduated with a basic to no knowledge of technical debt.

8 CONCLUSION

In this research, we investigated the usage of static code analysis tools within a year-long team-based project in a third year of a software engineering degree. We designed this study around a survey, focus groups and the analysis of historical code quality data to identify (1) how often junior developers use static analysis tools, (2) what events trigger corrective actions in term of code quality and (3) what metrics are of interest to them. We used third year students with a prior experience of academic and industry software development as proxies for junior developers. We observed that (1) students do use static analysis tools of many forms, (2) students do report having suffered from technical debt at some point, (3) feedback from teaching team on technical debt has little effect, mostly because of time pressure (4) students value both composite quality indices together with simple metrics, but they confess being puzzled by some rules or metrics. Additionally, as a side effect, we observed that despite decent understanding of quality measures and methods, few junior developers invest a significant effort into reducing the technical debt on a regular basis, mainly because of time pressure, even if the usage of static analysis tools is encouraged, but not enforced.

In the future, we plan to conduct a deeper topic analysis of the transcripts of the focus group where students came up with valuable insights outside of the scope of this research. Furthermore, a deeper analysis of the corrective actions per types of smells can be conducted to understand students' behaviour at a finer grained level. Last we plan to reproduce this research on a second cohort where we would have implemented the pedagogical suggestions we discussed above.

Acknowledgement

Thanks to Tim Huber and Erik Brogt from the University of Canterbury for proofreading the research proposal and survey. Thanks

to the students enrolled in the 2019's edition of SENG302 project course for their active participation.

REFERENCES

[1] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*. 181–190. https://doi.org/10.1109/CSMR.2011.24

[2] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de MendonÃğa, Rodrigo O. SpÃŋnola, Forrest Shull, and Carolyn Seaman. 2016. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology* 70 (2016), 100 – 121. https://doi.org/10.1016/j.infsof.2015.10.008

[3] Paris Avgeriou, Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, and Carolyn Seaman. 2016. Reducing Friction in Software Development. *IEEE Softw.* 33, 1 (Jan. 2016), 66âĂŞ73. https://doi.org/10.1109/MS.2016.13

[4] Xiaoying Bai, Mingjie Li, Dan Pei, Shanshan Li, and Deming Ye. 2018. Continuous delivery of personalized assessment and feedback in agile software engineering projects. In *40th International Conference on Software Engineering: Software Engineering Education and Training*. 58–67.

[5] K. Beck. 1999. Embracing change with extreme programming. *Computer* 32, 10 (Oct 1999), 70–77. https://doi.org/10.1109/2.796139

[6] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. 1999. SCRUM: An extension pattern language for hyperproductive software development. *Pattern Languages of Program Design* 4 (1999), 637–651.

[7] Rosanna L. Breen. 2006. A Practical Guide to Focus-Group Research. *Journal of Geography in Higher Education* 30, 3 (2006), 463–475. https://doi.org/10.1080/03098260600927575

[8] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

[9] Andrea Capiluppi, Cornelia Boldyreff, Karl Beecher, and Paul J. Adams. 2009. Quality Factors and Coding Standards âĂŞ a Comparison Between Open Source Forges. *Electronic Notes in Theoretical Computer Science* 233 (2009), 89 – 103. https://doi.org/10.1016/j.entcs.2009.02.063 Proceedings of the International Workshop on Software Quality and Maintainability (SQM 2008).

[10] J. Czerwonka, M. Greiler, and J. Tilford. 2015. Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 27–28. https://doi.org/10.1109/ICSE.2015.131

[11] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou. 2018. How do developers fix issues and pay back technical debt in the Apache ecosystem?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 153–163. https://doi.org/10.1109/SANER.2018.8330205

[12] Vincent Driessen. 2010. A successful Git branching model. (2010). https://nvie.com/posts/a-successful-git-branching-model/ accessed:30/10/2019.

[13] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 50âĂŞ60. https://doi.org/10.1145/2786805.2786848

[14] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering* 23, 1 (2018), 452–489. https://doi.org/10.1007/s10664-017-9523-3

[15] Davide Falessi and Philippe Kruchten. 2015. Five Reasons for Including Technical Debt in the Software Engineering Curriculum. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW âĂŹ15)*. Association for Computing Machinery, New York, NY, USA, Article Article 28, 4 pages. https://doi.org/10.1145/2797433.2797462

[16] Robert Feldt, Thomas Zimmermann, Gunnar R. Bergersen, Davide Falessi, Andreas Jedlitschka, Natalia Juristo, Jürgen Münch, Markku Oivo, Per Runeson, Martin Shepperd, Dag I. K. Sjøberg, and Burak Turhan. 2018. Four commentaries on the use of students and professionals in empirical software engineering experiments. *Empirical Software Engineering* 23, 6 (01 Dec 2018), 3801–3820. https://doi.org/10.1007/s10664-018-9655-0

[17] F. A. Fontana, R. Roveda, and M. Zanoni. 2016. Technical Debt Indexes Provided by Tools: A Preliminary Discussion. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. 28–31. https://doi.org/10.1109/MTD.2016.11

[18] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Signature Series (Fowler).

[19] Thorsten Haendler and Gustaf Neumann. 2019. Serious Refactoring Games. In *52nd Hawaii International Conference on System Sciences (HICSS'19)*, Tung Bui (Ed.). ScholarSpace / AIS Electronic Library (AISeL). https://doi.org/10.24251/HICSS.2019.927

[20] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (01 Jun 2012), 243–275. https://doi.org/10.1007/s10664-011-9171-y

[21] Marco Kuhrmann, Joyce Nakatumba-Nabende, Rolf-Helge Pfeiffer, Paolo Tell, Jil Klünder, Tayana Conte, Stephen G MacDonell, and Regina Hebig. 2019. Walking through the method zoo: does higher education really meet software industry demands?. In *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 1–11.

[22] M. Mantyla, J. Vanhanen, and C. Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance*. 381–384. https://doi.org/10.1109/ICSM.2003.1235447

[23] T. J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (Dec 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[24] Fatemeh Rabiee. 2004. Focus-group interview and data analysis. *Proceedings of the Nutrition Society* 63, 4 (2004), 655âĂŞ660. https://doi.org/10.1079/PNS2004399

[25] N. Ramasubbu, C. F. Kemerer, and C. J. Woodard. 2015. Managing Technical Debt: Insights from Recent Empirical Evidence. *IEEE Software* 32, 2 (Mar 2015), 22–25. https://doi.org/10.1109/MS.2015.45

[26] Ken Schwaber and Mike Beedle. 2001. *Agile Software Development with Scrum* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[27] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173. https://doi.org/10.1016/j.jss.2017.12.034

[28] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19, 5 (01 Oct 2014), 1299–1334. https://doi.org/10.1007/s10664-013-9286-4

[29] R. Simpson and T. Storer. 2017. Experimenting with Realism in Software Engineering Team Projects: An Experience Report. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T)*. 87–96. https://doi.org/10.1109/CSEET.2017.23

[30] Graziela Simone Tonin, Alfredo Goldman, Carolyn Seaman, and Diogo Pina. 2017. Effects of Technical Debt Awareness: A Classroom Study. In *Agile Processes in Software Engineering and Extreme Programming*, Hubert Baumeister, Horst Lichter, and Matthias Riebisch (Eds.). Springer, 84–100.

[31] Bruce F. Webster. 1995. *Pitfalls of Object-oriented Development.* M & T Books, New York, NY, USA.

[32] Claes Wohlin, Per Runeson, Marin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer-Verlag.