Does Static Analysis Help Software Engineering Students?

Reinhold Plösch
Business Informatics – Software Engineering
Johannes Kepler University Linz
Linz, Austria
reinhold.ploesch@jku.at

Cornelia Neumüller
Business Informatics – Software Engineering
Johannes Kepler University Linz
Linz, Austria
cornelia.neumueller@jku.at

ABSTRACT
Research on the impact of static analysis tools on software quality is often targeted towards practitioners or open source projects in general. Research in the field of software education concentrates on the usefulness of static analysis in introductory courses to programming. Contrary, we want to find out, whether students doing their first larger programming project (projects of 3000 to 5000 LOC) can benefit from applying static analysis tools. We therefore prepared a SonarQube based quality profile with 448 coding best practices and set up an environment that helped us to analyze code submitted by the students throughout a semester. Students were asked to frequently have a look at the provided data (using the SonarQube dashboard) and to fix those violations of best practices where they thought it makes sense. There were no incentives or penalties for fixing or not fixing these violations of best practices. The case study shows that there are substantially different kinds of violations of best practices depending on the experience level of the student teams. Additionally, while high experience and moderate experience student teams learn quickly and substantially during a semester, students with low experience have difficulties in understanding the underlying problems of the reported violations of best practices.

CCS Concepts
•General and reference→Surveys and overviews • Applied computing→Computer-assisted instruction • Applied computing→Interactive learning environments

Keywords
static analysis; SonarQube; software quality improvements; learning programming; static analysis and student experience

1. INTRODUCTION AND OVERVIEW
In research, operational quality models help formalizing the abstract notion of quality, as they refine the concept along with measurable properties and entities. The Quamoco quality model [1] and the ISO 25020 model [2] are just two examples of quality models that are widely accepted in research and industry. An operational quality model does not only structure the term quality but also provides concrete instruments, i.e., tools that provide concrete metrics and best practices, often for specific programming languages. A *metric* calculates some number on basis of properties of a piece of software, e.g., Halstead Complexity [3]. *Rules* can be viewed as *best practices*, i.e., conventions a good programmer typically would not break. In the remainder of the paper, we will concentrate on best practices and their violations by programmers, in particular students.

Typical examples of tools that help developers are SonarQube (http://www.sonarqube.org), PMD (http://pmd.github.io) and FindBugs (http://findbugs.surceforge.net) for Java and Lint (http://www.gimpel.com) for C++, just to give a few examples. In previous research projects, we developed operational base quality models for Java [4], embedded quality models [5] as well as quality models for object-oriented design [6]. All this research work is targeted toward scientists and practitioners and had – up to know – not been used in the software engineering and programming education of students.

In this research, we were interested to find out, whether coding and best practices that are automatically measured in the context of students doing their first large software engineering project is helpful for them.

In Section 2 we discuss the related work with a clear focus on teaching in programming and software engineering. Section 3 gives an overview of the structure of the case study as well as of the research questions. Section 4 discusses the results of the study, Section 5 discusses the threats to validity of the study and Section 6 derives conclusions and discusses ideas for future work in this field.

2. RELATED WORK
There are a number of publications available that deal with the impact of applying static analysis for introductory courses to programming, like [7] with an emphasis on finding out whether enhancing the messages provided by compilers or static analysis would increase the understanding for the students (see Denny et al. [8] and Pettit et al. [9]).

The focus of our work is on applying general-purpose static analysis tools commonly used in industry. This approach is quite similar to that presented by Edwards et al. (see [10], [11]), with two major differences: (1) Edwards et al. concentrate on one individual tool (FindBugs - http://findbugs.surceforge.net) which is known to be a bug hunting tool but is less suitable for identifying violations of coding best practices, style issues, etc. We, on the contrary use SonarQube with a large number of measures (448) that are also used by our industry partner Siemens as the default set of measures in industry projects. These selected

ICEIT 2020, February 11–13, 2020, Oxford, United Kingdom

ACM ISBN 978-1-4503-7508-5/20/02…$15.00
https://doi.org/10.1145/3383923.3383957

measures were hand-selected in a cooperation project and also incorporates feedback from industry projects. Thus, this set of measures can be considered as important in an industry context. (2) Edwards et al. [11] concentrate on small exercises for beginners, while we focus on one product that has to be implemented by a team of students within a semester with typical sizes between 3000 to 5000 lines of Java code. The typical effort spent by each student for this project is between 125 and 150 working hours per semester.

The available literature mentioned above, typically is targeted towards novice programmers and does not consider different experience and knowledge levels of students. On the contrary, as our case study is carried out with students that are at least in the third semester of their business informatics bachelor and should already have passed the introductory courses in programming and algorithms, we have to deal with students with different programming literacy. With our case study, we are particularly interested, whether the improvement or learning effect within a semester is impacted by the experience level of the students.

3. CASE STUDY DESIGN

Students of Business Informatics at the Johannes Kepler University mandatorily have to attend the course "Software Engineering Project". They typically visit this course in the fourth or fifth semester of their bachelor studies. The goal of this course is the application of the knowledge students gained in earlier courses on algorithms, introduction to programming and software engineering in the context of a larger software development project. Besides core programming tasks, students are also responsible for analyzing and documenting requirements, software architecture development as well as testing and project management.

For this case study, we required the students to develop a cash management application or a health management application, using core Java technologies and Java Swing or Java FX for the user interface part of the application. In total, we had 8 teams, where 4 teams developed the health management application and 4 teams developed the cash management application. Both applications are quite similar in size and complexity. The average team size is 4 students, with a total of 31 students participating in this case study over a period of four months.

We analyzed the student projects on a daily basis with static code analyzers and provided the analysis data to the students via a SonarQube server. Weasked them to fix problems identified by the tools in those cases where they felt it makes sense. We did not setup any threats or rewards for fixing or not fixing the identified problems. Using this unsupervised process, we aimed at answering two research questions:

- **RQ 1:** Which best practices are the most commonly violated by students and do they fix these violations?
- **RQ 2:** Does applying static analysis help students improving their code quality?

For all research questions we also wanted to find out, whether the programming experience of the students has an impact on how they deal with violations of best practices found by the static code analyzers.

The case study was carried out in three phases that are discussed in more detail in the subsequent sections.

3.1 Preparation and Setup

The major preparatory work to be done was the selection of the rules that should be checked for the student projects. We relied here on a quality profile (set of rules) we developed together with our research partner Siemens, and that is used as a standard quality profile in industry projects. Table 1 gives a quantitative overview of the quality profile, that consists of 448 rules, i.e. best practices.

Table 1. SonarQube quality profile for student projects.

Category	*Blocker*	*Critical*	*Major*	*Minor*
Bugs	8	9	67	18
Vulnerabilities	11	18	2	9
Code Smells	18	71	106	111
Total	*37*	*98*	*175*	*138*

Furthermore, we set-up Git repositories for each student team and provided some tools that automatically build and analyze the student systems and that provide the analysis data via our SonarQube server that is accessible for the students. A major goal here was that the students have nothing to do with this process in order to reduce any dependencies that could have an impact on the output of the case study, because of troubles students might have in the integration of their work with the static analysis tooling.

At the beginning of the semester, we showed the students the available SonarQube tooling and told them how to analyze results, how to mark SonarQube findings as false-positive, how to assign the fixing of identified problems to a team member, etc. Furthermore, during this setup-phase, we conducted a survey to find out about the programming experience of the individual teams by means of a self-assessment. Basically, we asked each participant about the programming experience in general (in years) and the programming experience with Java (in years). The overall experience of a team was calculated as average value of the programming experience and the Java programming experience. Table 2 shows the average values for all teams as well as the derived experience level.

Table 2. Experience levels of participating teams.

Team	avg. programming experience	avg. Java programming experience	avg. experience	experience level
CM 1	5,00	5,00	5,00	HE
CM 2	2,00	2,00	2,00	LE
CM 3	2,75	2,50	2,63	ME
CM 4	1,00	1,00	1,00	LE
HM 1	4,00	4,00	4,00	HE
HM 2	2,50	1,75	2,13	LE
HM 3	2,75	2,00	2,38	LE
HM 4	4,00	3,00	3,50	ME

The team column indicates the team, where the abbreviation CM stands for cash management and HM stands for health management. Teams with an average experience of more than 2.5 average experience years have medium experience (ME), teams with at least four average experience years have high experience (HE). Teams with a maximum of 2.5 average experience years have low experience (LE). According to this analysis, we have 2 teams with high experience, 2 teams with medium experience and 4 teams with low experience. This grouping of teams was essential to us, as we also want to analyze, whether the experience

of the students has an impact on the results. In the remainder of this paper we will not address the results of individual teams but always aggregated according to the experience level.

3.2 Student Project Mentoring

During the semester each student team had to provide three versions of their software at defined milestones. During our bi-weekly meetings, students gave progress reports, demos of their software, etc.

During these meetings, we reminded the students not to forget to have a look at the SonarQube dashboard in order to enhance their code quality. We gave no further advice, which programming problems have to be fixed, which ones need no fix etc. On a daily basis, our tooling checked whether the project could be properly built and as soon as code changes were available they got analyzed and uploaded to our SonarQube server. Analysis of the source code was done once a day. In case we encountered problems during the build process (bad maven configuration, missing libraries), etc. we fixed this without contacting or discussing it with the project team.

3.3 Closing

At the end of the semester, we pre-analyzed the available data and identified (for each team) the most important violations of best practices, typically fixed problems and those problems that seem to be important but were marked by the students as "won't fix".

In the closing session, we also had a small survey where we wanted to get feedback about the reasoning, why specific violations of best practices were fixed or not fixed. This is discussed in more detail in the results section.

Furthermore, we aggregated the available data to have an overview of all teams as well as the aggregated data grouped by the experience level of the teams.

4. RESULTS

In Section 4.1 we concentrate on the typical programming problems students encounter during their software project. The pre-analysis showed that there a significant difference between teams of different experience levels. We therefore present the results grouped by experience levels. In Section 4.2 we had a closer look on the quantitative development (increase or decrease) of violations of best practices in order to see whether providing the static analysis results has a positive impact on quality of the source code, i.e., to see whether static analysis helps students to improve over time.

4.1 RQ 1 – Typical Programming Problems and their resolve status

For this research question, we selected from all experience levels the top 10 rules of the violations of best practices. These 10 rules represent about 44 to 50 percent of the total number of violations of best practices. The total number of violations for all teams is 8464.

Table 3 shows the violations of the best practices of the teams with *low experience*, the relative violation count (RVC) and their resolve status (fixed, open or won't fix) of violations in percentage. In total, these teams had 4220 violations. The results in Table 3 are in descending order, i.e. the best practices with the most violations are on top of the list. It is well recognizable, that these teams have to deal with very low-level violations, such as missing braces, naming conventions, or declaring multiple variables on the same line. It also shows that teams with a low experience rarely fixed these frequent violations, or even marked them quite often as won't fix, although many of the same violations have already been fixed. This leads to the assumption that the students simply did not understand these best practices. E.g., over 55 percent of the violations from the best practices *Exception handlers should preserve the original exceptions* were marked as won't fix or false-positive.

In our retrospective survey in the closing part of the experiment (see Section 3.3), we asked the students, why they don't won't fix these important (major) violations. The answers varied from "I don't understand the rule", "I think this rule doesn't make sense" to "the implementation would have been too complex". The last two statements clearly show that the coding best practice was simply not understood.

Table 3. Violations in low experience level.

	RVC	Fixed	Open	Won't fix
Member variable visibility should be specified	6,33%	79,40%	13,86%	6,74%
Exception handlers should preserve the original exceptions	5,64%	36,55%	8,40%	55,04%
Unnecessary imports should be removed	5,43%	95,63%	4,37%	
Standard outputs should not be used directly to log anything	5,10%	87,44%	3,26%	9,30%
Strings literals should be placed on the left side when checking for equality	4,27%	86,11%	8,89%	5,00%
Control structures should use curly braces	4,24%	64,25%	8,38%	27,37%
Throwable.printStackTrace(...) should not be called	4,17%	31,82%	1,14%	67,05%
Multiple variables should not be declared on the same line	3,25%	40,88%	28,47%	30,66%
Private fields only used as local variables in methods should become local variables	2,65%	60,71%	34,82%	4,46%
Local variable and method parameter names should comply with a naming convention	2,18%	26,09%	53,26%	20,65%

An overview of those best practices with the most violations in the teams with *moderate experience* is shown in Table 4. The total violation count of these teams is 2618. These teams still have to struggle with the visibility of variables, code duplications, and unused code. The best practice *"static" members should be accessed statically* is typically violated by these teams but less violated by the other teams. Only about 47 percent of these violations were fixed until the final release. That is interesting because fixing would be easy and not time-consuming.

Table 4. Violations in moderate experience level.

	RVC	Fixed	Open	Won't fix
Standard outputs should not be used directly to log anything	8,79%	97,83%	2,17%	
"static" members should be	7,34%	46,88%	53,13%	

accessed statically				
Unnecessary imports should be removed	5,04%	69,70%	30,30%	
Exception handlers should preserve the original exceptions	4,43%	62,07%	11,21%	26,72%
Methods should not be empty	4,43%	62,93%	37,07%	
Member variable visibility should be specified	4,13%	91,67%	8,33%	
String literals should not be duplicated	3,32%	68,97%	17,24%	13,79%
Sections of code should not be commented out	3,17%	66,27%	13,25%	20,48%
Unused local variables should be removed	3,17%	79,52%	19,28%	1,20%
Dead stores should be removed	2,64%	81,16%	17,39%	1,45%

The violations weren't marked with won't fix either. This may indicate that they were running out of time in the end. Again, we asked the teams why they won't fix the best practice *Exception handlers should preserve the original exceptions* and they simply answered, that they do not understand the rule.

Table 5 gives an overview of those best practices with the most violations in the *high experience* level teams. The total violation count of these teams is 1626. These teams already have to deal with more specific rule violations, such as lambdas or generics (diamond operators). An interesting fact is that the best practice *Short-circuit logic should be used in Boolean contexts* is often violated by both teams with high experience, but never violated by the other teams. All these blocker violations were fixed before the final release.

Nevertheless, these students also have to deal with low-level violations, such as *Unnecessary imports should be removed* or *Sections of code should be commented out*. However, these violations were fixed to almost 100 percent. Furthermore, some best practices have a significant number of won't fixes, such as *Mutable members should not be stored or returned directly* or *Classes and enums with private members should have a constructor*. In the evaluation of our survey, we found that the latter best practice was not understood either.

Table 5. Violations in high experience level.

	RVC	Fixed	Open	Won't fix
Unnecessary imports should be removed	7,84%	100,00%		
Sections of code should not be commented out	7,09%	99,19%		0,81%
Short-circuit logic should be used in boolean contexts	4,32%	100,00%		
Types should be used in lambdas	3,80%	53,03%		46,97%
Exception handlers should preserve the original exceptions	3,52%	96,72%	1,64%	1,64%
Mutable members should not be stored or returned directly	3,00%	34,62%		65,38%
Strings literals should be placed on the left side when checking for equality	2,48%	90,70%	4,65%	4,65%
Avoid unused getters in classes with many getters	2,48%	55,81%		44,19%
The diamond operator ("<>") should be used	2,19%	97,44%		
Classes and enums with private members should have a constructor	2,13%	27,03%	18,92%	54,05%

It is interesting to note that all experience teams violate the best practice *Unnecessary imports should be removed*, which is normally done automatically by the programming environment. However, if we look at the time of occurrence, it can be seen that the number of violations decreases over time in the high experience teams and it was completely fixed until the final release. This is not the case with the other teams. With these, the decrease is only slight or even an increase of violations and also they were not completely fixed until the final release.

Another violated best practice that can be found in all teams is *Exception handlers should preserve the original exceptions*. And just like the previous best practice, it can be seen that the number of violations decreases in the teams with the high experience and increases, or only slightly decreases in the other teams. The teams with high experience fixed almost all the violations, while the teams with moderate experience fixed only about 62 percent and the teams with low experience only fixed 37 percent of these violations.

Figure 1 shows the relative distribution of the violations, whether they were closed by fixing the violation, still open violations or explicitly marked as won't fix or false-positive. This figure shows that the higher the experience, the more violations were fixed. However, a very interesting point is that about 20 percent of the violations were marked as won't fix in the high and low experience teams. The violations of the moderate teams just remained open violations.

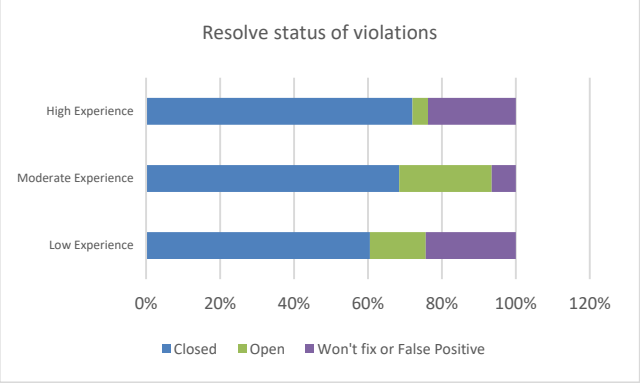

Figure 1: Resolve status of violations.

Based on the previous finding, it can be concluded that the best practices that are marked as won't fix, are those rules that they do not understand or they just do not understand the underlying issues of these best practices. To better support the students with static analysis, it may be necessary to improve the descriptions of the best practices. Additional description of consequences could help the students understanding the negative consequences of certain practices.

4.2 RQ 2 – Static Analysis and Improvement

With this research question, we wanted to find out whether violations of best practices decrease over time. Decrease over time is an indicator for us that students established a good

understanding of specific best practices and are able to proactively produce better code.

In order to calculate this decrease, it was necessary to normalize the violations of best practices with the lines of code of the project at that time. As the student teams work with different speed and effort over time, as well as with different strategies for committing source code changes to the code repository, we calculated the violations of the coding best practices once per month (March to July, i.e. 5 values) as the arithmetic mean value of the commit data of a team (for each commit we measured the violations of best practices). Based on these data we calculated the normalized violations of best practices in total as well as split up into the categories *bugs*, *blocker*, *critical*, *major*, and *minor*. Analysis of the individual team results showed us, that team CM 2 hardly provided any source code until the end of May. The major implementation work (more than 50 % of the total lines of code) was provided in July. Therefore, the data of this project team is outside the typical pattern of the other student projects and can hardly show learning effects. We therefore decided to not include the data of the CM 2 team for answering this research question.

We aggregated the individual data of the teams to the three experience levels *high experience* (2 teams), *moderate experience* (2 teams), and *low experience* (only 3 teams for this research question).

Figure 2 shows the results of the *low experience teams*.

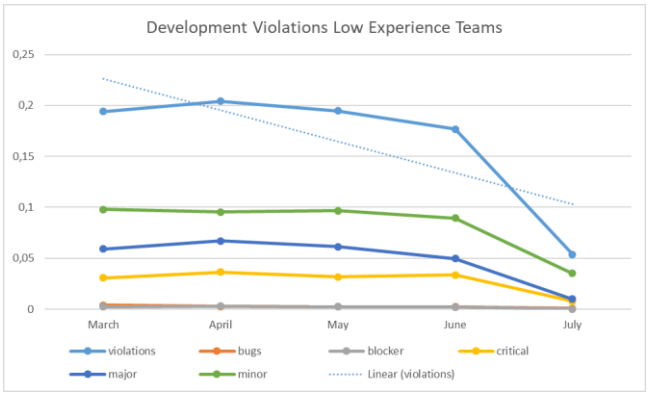

Figure 2. Violations of Low Experience Teams.

Although the general trend (with respect to all violations) is positive, i.e., shows some learning effect. Figure 2 clearly shows that students of low experience teams start relatively late in the development process (begin of June) to improve their code quality until the end of June. The rather steep decrease in July is due to the fact that students knew, that the code quality is one quality aspect to be considered for grading their work. Interestingly, the amount of minor coding best practice violations is higher than for the medium and high experience teams. This is an indicator that low experience student teams struggle with basics like naming conventions and indentations – two aspects that are part of minor coding best practices. The same applies to the critical violations, which is also a (typical) pattern for unexperienced students, as they often do not yet fully understand the negative consequences of certain programming practices.

Figure 3 shows the results of the *medium experience teams*.

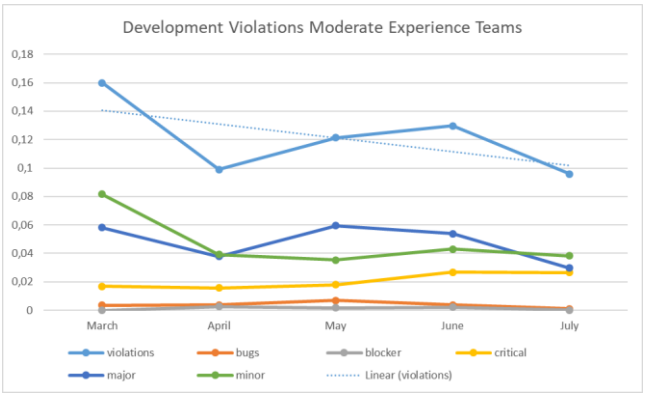

Figure 3. Violations of Moderate Experience Teams.

The major difference to the low experience teams is, that teams with moderate experience start with slightly better code quality (0,16 compared to 0,19) in the beginning and obviously have a better awareness for code quality, as they immediately enhance their code in April. Nevertheless, they do not seem to be so eager about code quality as we can see a slight increase in violations of coding best practices during May and June, followed by cleaning up work in July.

Figure 4 shows the results of the *high experience teams*.

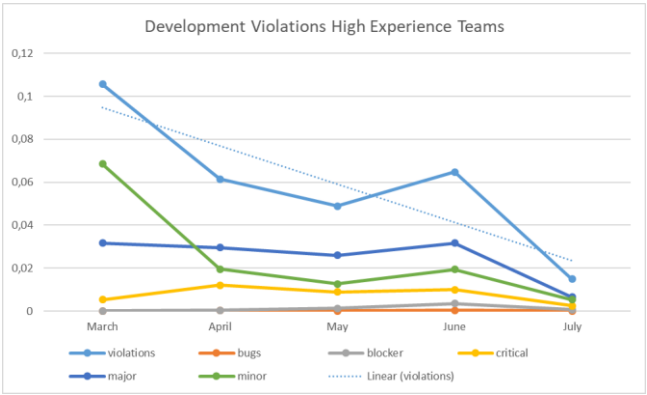

Figure 4. Violations of High Experience Teams.

The high experience team obviously starts with a better code quality (0,10 violations per LOC) and also finishes with considerably higher code quality (only 0,01 violations per LOC compared to 0,09 violations for moderate experience teams and 0,05 for the low experience teams. The high experience teams obviously have a better awareness of software quality, as they quickly and systematically learn and apply the best practices and are able to learn a lot about good programming, as the high level of code quality is maintained until the end of the semester.

During the closing session (see Section 3.3) we conducted a survey. In this survey we asked the question of whether students felt like they were writing better code by applying static code analysis tool, using an ordinal scale from “did not help at all” to “helped enormously”.

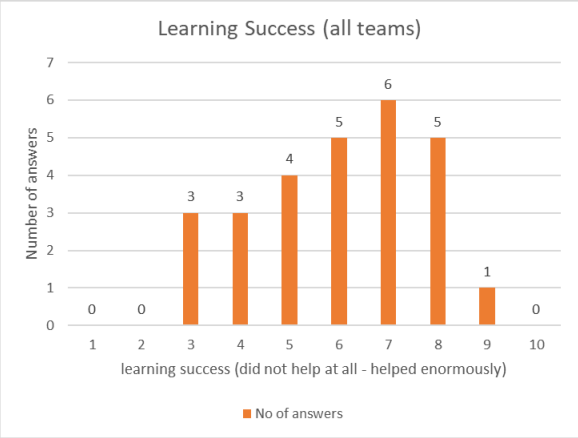

Figure 5. Learning Success (all teams).

Figure 5 clearly shows that the majority of students think that static code analysis helped them to improve their programming skills. The result of this self-assessment is in line with the data presented in Figure 2 to Figure 4. Aggregating the individual answers to the low experience, moderate experience, and high experience clusters leads to the results shown in Figure 6.

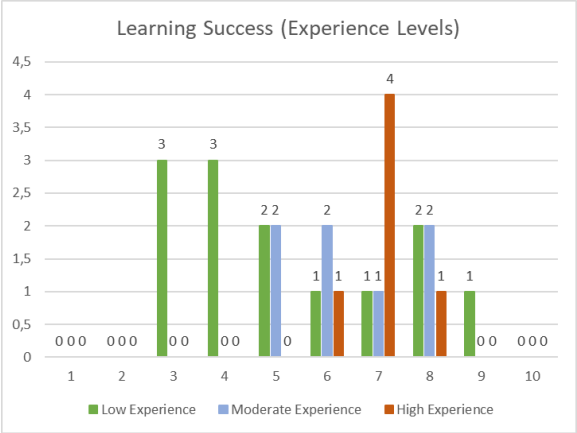

Figure 6. Learning Success (experience levels).

The higher the experience level of the teams the more valuable the judge the application of static analysis tools. The average score for the low experience teams is 5,31, for the moderate experience teams 6,43 and for the high experience teams 6,86.

5. THREATS TO VALIDITY

In any study, some factors influence the findings and represent threats to validity [12]. In more detail, threats to *internal validity* concern any confounding variables that could have influenced the outcome. We respect the aggregation of data from individual teams to the low, moderate and high experience datasets as an internal threat. To mitigate this threat, we a priori (i.e., before getting to know the students) decided on the mode on how to assign teams to these datasets as well as on the thresholds used. Furthermore, the ex post analysis of the data showed quite similar patterns of violation occurrences over time among teams of the same experience cluster. This might be seen as an indicator, that the assignment of teams to experience clusters is correct.

Another threat to internal validity is the prior knowledge of students about static analysis in general and SonarQube in particular. No specific actions were necessary here, as no member of any team ever had experience with static analysis at all. Therefore, the presentation of how to work with SonarQube was equally new to all participants.

Another threat to internal validity is the availability of local static analysis tools that could help students to improve their work before we could get hold of the original data. Students could not influence when their code was checked. Checking took place after each commit to the code repository. Furthermore, the students had no local tools available that would allow them to pre-check their source code and they could also not initiate a SonarQube analysis or delete available measurement data.

The different projects (cash management, health management) could have an impact on the complexity of the project and therefore on the results of static analysis. To mitigate this risk the provided user requirements for both applications were closely aligned and contained the same type and number of requirements. Furthermore, by providing the user requirements from our side, all applications solve the same functionality. So they have the same complexity from point of view of the requirements to fulfill.

Threats to *external validity* concern the ability to generalize the results [12]. The major threat to external validity is the limited number of student teams. Although we had 8 student teams with a total of 31 students this is not yet representative. Nevertheless, the experiment can and will be repeated with student teams in the next semester in order to broaden the data basis.

6. CONCLUSION AND FURTHER WORK

From answering RQ 1 we can clearly conclude, that students with different experience levels have different needs in their support for static analysis. Students with low experience still struggle with low-level problems and it would possibly help them to work with a different (smaller) set of best practices in order to be able to cope with them better. On the other hand, experienced teams use other programming language concepts (e.g., lambdas in Java) which leads to different violations of best practices. Again, also these students have problems in understanding the underlying problems of these more complex coding best practices. Although SonarQube provides detailed information on individual coding best practices, we do not know, whether this more detailed information was read and studied by the students or whether the simply worked with the short messages provided by SonarQube.

From answering RQ2 we can conclude, that for students with moderate and high experience the provided support for static analysis had a very positive effect on the quality of the source code. This is not only reflected by a considerable decrease in best practice violations per LOC, but is also reflected by our survey data (see Figure 6). The student teams with low experience also fixed a number of violations at the end of the project, but a closer look on the data (see Figure 2) reveals, that this decrease – at least from our interpretation – is due to the fact, that students knew, that they had to fix some of the issues. As can be seen from Figure 2, the decrease in violations of best practices per LOC starts only at the end of the semester, and there were only small improvements before. We judge this to be an indicator that they could only very late in the project learn something from the static analysis data provided to them.

We conclude, that for high experienced student teams the chosen approach (providing an industry-grade set of best practices) is a good way to go. We will further investigate this in future projects with the addition, that we will help students with more complex coding best practices in order to enhance the understanding of the students of the underlying technical problems.

From the data available we assume, that students with low experience need a different support to enhance their learning achievements. In a future case study, we will therefore use more low-level and smaller quality profiles for these students. We will then try to systematically enlarge the quality profile during the semester in order to support a gradual learning effect. We hope that this approach will lead to better learning improvements of students with low experience.

7. REFERENCES

[1] Wagner S., Lochmann K., Heinemann L., Kläs M., Trendowicz A., Plösch R., Seidl A., Göb A. and Streit J. 2012, The Quamoco product quality modelling and assessment approach, *Proceedings of the 34th International Conference on Software Engineering* (ICSE 2012), Zurich, 2012, S. 1133–1142.

[2] Al-Kilidar H., Cox K., and Kitchenham B. 2005, The use and usefulness of the ISO/IEC 9126 quality standard, *2005 International Symposium on Empirical Software Engineering, 2005.*, 2005, S. 7 pp.-.

[3] Halstead M. H. 1977. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., New York, NY, USA.

[4] Wagner S., Lochmann K., Heinemann L., Kläs M., Trendowicz A., Plösch R., Seidl A., Göb A. and Streit J. 2015, Operationalised Product Quality Models and Assessment: The Quamoco Approach, *Journal of Information and Software Technology (IST), Elsevier, Volume 62, pp 101-123, 2015,* doi:10.1016/j.infsof.2015.02.009.

[5] Mayr A., Plösch R., and Saft M. 2014, Objective Safety Compliance Checks for Source Code, in Companion *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014),* New York, NY, USA, 2014, S. 115–124.

[6] Plösch R., Bräuer J., Körner C., and Saft M., MUSE - Framework for Measuring Object-Oriented Design, *Journal of Object Technology., Vol. 15, Nr. 4, S. 2:1-29*, Aug. 2016.

[7] Ayewah N., Hovemeyer D., Morgenthaler J. D., Penix J., and Pugh W. 2008, Using static analysis to find bugs, *IEEE Software., Vol. 25*, pp. 22–29, Sept. 2008.

[8] Denny P., Luxton-Reilly A., and Carpenter D. 2014, Enhancing syntax error messages appears ineffectual, in *Proceedings of the 2014 Conference on Innovation; Technology in Computer Science Education, ITiCSE '14*, (New York, NY, USA), pp. 273–278, ACM, 2014.

[9] Pettit R. S., Homer J., and Gee R. 2017, Do enhanced compiler error messages help students?: Results inconclusive., in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '17,* (New York, NY, USA), pp. 465–470, ACM, 2017.

[10] Edwards., Kandru N., and Rajagopal M. B. 2017, Investigating static analysis errors in student java programs, in *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER'17*, (New York, NY, USA), pp. 65–73, ACM, 2017.

[11] Edwards S. H, Spacco J., und Hovemeyer D. 2019, Can Industrial-Strength Static Analysis Be Used to Help Students Who Are Struggling to Complete Programming Activities?, *Hawaii International Conference on System Sciences, 2019*.

[12] Wohlin C., Runeson P., Höst M., Ohlsson M., Regnell B, Wesslén A. 2012, *Experimentation inSoftware Engineering*, Springer Berlin Heidelberg, 2012, doi:10.1007/978-3-642-29044-2.