Teaching Clean Code

Linus W. Dietz
Department of Informatics
Technical University of Munich, Germany
linus.dietz@tum.de

Johannes Manner and Simon Harrer
Distributed Systems Group
University of Bamberg, Germany
firstname.lastname@uni-bamberg.de

Jörg Lenhard
Department of Mathematics
and Computer Science
Karlstad University, Sweden
joerg.lenhard@kau.se

***Abstract*—Learning programming is hard – teaching it well is even more challenging. At university, the focus is often on functional correctness and neglects the topic of clean and maintainable code, despite the dire need for developers with this skill set within the software industry. We present a feedback-driven teaching concept for college students in their second to third year that we have applied and refined successfully over a period of more than six years and for which received the faculty's teaching award. Evaluating the learning process within a semester of student submissions (n=18) with static code analysis tools shows satisfying progress. Identifying the correction of the in-semester programming assignments as the bottleneck for scaling the number of students in the course, we propose using a knowledge base of code examples to decrease the time to feedback and increase feedback quality. From our experience in assessing student code, we have compiled such a knowledge base with the typical issues of Java learners' code in the format of before/after comparisons. By simply referencing the problem to the student, the quality of feedback can be improved, since such comparisons let the student understand the problem and the rationale behind the solution. Further speed-up is achieved by using a curated list of static code analysis checks to help the corrector in identifying violations in the code swiftly. We see this work as a foundational step towards online courses with hundreds of students learning how to write clean code.**

I. Introduction

Many computer science graduates lack programming proficiency when starting their first software engineering job. Programming at university is traditionally taught in lectures followed by practical exercises that usually revolve around theoretical concepts like algorithms, data structures or specialized aspects of programming paradigms (i.e., object-orientation or functional programming). Such courses challenge the students to develop software to prove that they can apply the theoretical concepts. These prototypes, however, are usually trashed after the end of the course. Consequently, students have few incentives to self-educate themselves in writing code that not only works, but is also of high quality.

Given the high demand for computer scientists with college degrees in industry, many students choose this challenging course of studies. From 2009 onwards, the faculty for Information Systems and Applied Computer Sciences at the University of Bamberg grew from 340 students to 1200 in 2017. It is a huge challenge for lecturers to keep up the quality of teaching, since they inevitably will have less time for providing individual feedback to the students. This is problematic, as teaching programming well is time-consuming [1]. Therefore, e-learning, especially Massive Open Online Courses (MOOCs) [2] are an emerging teaching method. However, the automated assessment of programming assignments is challenging [3] and most often only focuses on functional correctness instead of code quality. A study among 227 IT professionals finds that the most important skill is the *"ability to read, understand and modify programs written by others"* [4]. Since low code quality has a direct effect on maintainability, we argue that clean code should be an integral part of the programming education. Consequently, the motivating questions we seek to answer in this paper are:

- How to teach clean code at university?
- How to teach clean code at scale?

We present a didactic concept that describes how to effectively teach programming to undergraduates with an emphasis on writing and reviewing code. Identifying the correction of assignments, (i.e., providing high-quality feedback) as the bottleneck to scale the participants in the course, we use a knowledge base [5] and propose automated static code analysis to speed up assessment while preserving the quality of feedback. The interplay between these two components is promising, as it enables teachers with little prior experience to give students the necessary information to improve their coding style.

The driving idea behind this paper is to assess what academia can learn from industry to achieve high code quality in teaching. Therefore, we briefly discuss the concept of code quality and how it is achieved in industry in the next section. On this basis, we present and evaluate our didactic concept based on writing and reviewing code in Section III. In Section IV, we present the knowledge base and a tool for automated static code analysis for code quality in an educational context. We conclude our findings and point out future work in Section V.

II. Foundations

From a pragmatic point of view, code quality is strongly linked to understandability: How easy is it for other developers to understand a piece of code and how well can it be extended and reused in other contexts? The concept is referred to in several books, most prominently in *Clean Code* [6], *Code Complete* [7], *Effective Java* [8], *The Pragmatic Programmer* [9] and *Refactoring* [10]. From a business perspective it can also be seen as a function of the maintenance costs, which typically amounts to 40–80% of the total project costs [11].

To lower such costs, the software industry has introduced many ways to improve the coding process, most of them fitting under the hyped term *'agile'* [12]. For instance, Li et al. report increased software code quality of a team using Scrum in a longitudinal study [13]. Code reviews and feedback play a vital role in agile methods. In pair programming [14] the review is done simultaneously with a partner. Mob programming [15] even extends this to a group of people. Besides pairing, companies also use *'pull requests'* with continuous integration (CI), where the proposed patch for the upstream is automatically tested and needs to be signed off by another developer. In addition, companies often use static code analysis in their CI pipeline to ensure that the code adheres to a predefined style.

A number of static code analysis tools are freely available, e.g., Checkstyle[1], PMD[2] or SpotBugs[3]. They operate on source or bytecode level and automatically detect common code smells and careless mistakes that are not easy to spot. To use them efficiently, they need to be fine-tuned by an expert. A more user-friendly solution are online code quality services like SonarQube[4] or Codacy[5]. They build upon the mentioned static code analysis tools, however, they require in-depth integration into the build pipeline.

III. Teaching Clean Code at University

The availability of a mentor who supports the learner is a huge benefit. We argue that high-quality code in industry is created when several people work together and review each other's code, either simultaneously in pair programming sessions or when assessing each others code in pull requests. From this insight, the following didactic concept has a strong emphasis on *reviewing code* in various ways.

A. Didactic Concept

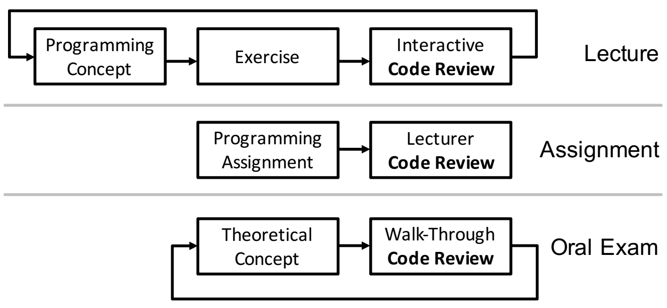

Fig. 1. Didactic Concept Based on Code Reviews

A sketch of the didactic concept is depicted in Fig. 1 and it covers three parts: lecture, assignments, and the examination. Note that at the end of each part, a different type of code review is done. In the following, we detail each part.

1) Lecture: Each course session is an alteration between lecture time and practice. After having received a 10–30 minute introduction to a programming concept, the students are asked to solve a 20–40 minute exercise, in which they are to apply this concept. After finishing the task, at least one solution is presented to the plenum of students, who review and improve upon the solution with help and input from the lecturers. These discussions on improving the code step-by-step are central to the learning outcome, as this is where the students observe the refactoring process, learn about the requirements in the programing assignments, and experience the transformation of a first working solution to clean code.

2) Assignment: During the semester, students work on multiple assignments in groups. The assignments require the application of the previously introduced and practiced programming concepts from the lectures, but put them in a larger scope to address more realistic and complex problems (e.g., programming a reference manager or an issue tracker). The assignments have to be submitted within a given timeframe using Git and are graded by the lecturers. The students receive a detailed textual code review of their solution with a focus on code quality, and refactoring opportunities, since they usually get most functional aspects right. Besides the individual feedback, the lecturers also publish a collection of common issues found in the assignments to the course. These common issues serve as a knowledge base for the course and have resulted in a text book, *Java by Comparison* [5].

3) Examination: Finally, during examination, both the theoretical concepts of the course itself and clean code skills are evaluated. The students are examined individually in an oral examination. During this examination, a few initial theoretical questions are used to check if the students understood the general concepts. Then, the students are asked to explain how they solved specific aspects in their own assignments' code. This is to evaluate whether they can make the transfer from the theoretical concepts to practical knowledge and also to assess whether they have contributed sufficiently to the group's submissions. Finally, the students are asked to review a small snippet of unknown code regarding bugs and code smells.

B. Application

Since 2011, this concept has been used in two practical programming courses with 3 ECTS for undergraduate computer science students at the University of Bamberg: *'Advanced Java Programming' (AJP)* covering XML serialization, testing, and MVC-based GUIs, and *'Introduction to Parallel and Distributed Programming'* covering systems communicating through shared memory and message passing on the JVM.

These courses are taught in bi-weekly, four-hour lab sessions. The students need to hand in four two-week assignments solved by groups of three, and pass an individual 15-minute oral examination. In addition, we also provide an optional student help desk and forum support for any course-related questions. Students have rated our courses on average 1.5 (on a Likert scale from 1–very good to 5–very bad) within the last six years in the university's standardized evaluation form. Furthermore, the lecturers were nominated six times for the excellent teaching award of the faculty and won it once.

C. Assignment Evaluation

In the following, we present our findings from analyzing student code submissions using static code analysis. First, we

[1]Checkstyle, http://checkstyle.sourceforge.net
[2]PMD https://pmd.github.io
[3]SpotBugs on Github: https://github.com/spotbugs/spotbugs
[4]https://www.sonarqube.org/
[5]https://www.codacy.com

TABLE I
MOST FREQUENT VIOLATIONS

Name of Violation	Frequency
MagicNumberCheck	1382
AbbreviationAsWordInNameCheck	452
ParameterAssignmentCheck	150
PreserveStackTrace	144
UnnecessaryConstructor	143
UselessParentheses	124
VisibilityModifierCheck	120
ConfusingTernary	107
HideUtilityClassConstructorCheck	96
SingularField	93

TABLE II
CHANGE OF VIOLATIONS

Name of Violation	Normalized Frequency	Change in %	
CyclomaticComplexityCheck	43 → 0	−100	Disappear
UnnecessaryFinalModifier	37 → 0	−100	
ModifierOrderCheck	33 → 0	−100	
EqualsAvoidNullCheck	23 → 0	−100	
CollapsibleIfStatements	13 → 0	−100	
AvoidFieldNameMatchingMethodName	128 → 3	−98	Decrease
NeedBracesCheck	67 → 2	−97	
UselessParentheses	226 → 9	−96	
AvoidInstantiatingObjectsInLoops	37 → 2	−95	
PrematureDeclaration	16 → 1	−94	
UnnecessaryConstructor	67 → 63	−6	Stable
LogicInversion	3 → 3	0	
MultipleVariableDeclarationsCheck	6 → 6	0	
AvoidCatchingNPE	10 → 10	0	
MagicNumberCheck	580 → 633	+9	
SingularField	27 → 82	+203	Increase
HiddenFieldCheck	13 → 50	+284	
AbbreviationAsWordInNameCheck	43 → 200	+365	
VariableDeclarationUsageDistanceCheck	13 → 72	+453	
VisibilityModifierCheck	6 → 54	+800	
InnerAssignmentCheck	0 → 8	+∞	New
ClassFanOutComplexityCheck	0 → 10	+∞	
SignatureDeclareThrowsException	0 → 17	+∞	
UncommentedEmptyMethodBody	0 → 19	+∞	
CompareObjectsWithEquals	0 → 19	+∞	

provide an overview of the most frequent violations and then we analyze the relative change of code quality violations. The analysis is based on a curated list of static code analysis checks described in Section IV-B.

Table I lists the ten most frequent code style violations. The magic number check is an outlier, as students did not encapsulate all numbers into `final static` fields in the beginning, and it was violated frequently in the last two assignments (testing and GUI programming). The others are typical problems of unprofessional code: bad naming, unnecessary elements, and bad habits such as re-assigning parameters or not preserving the stack trace when re-throwing exceptions. Some, like the usage of the ternary '`?`' operator, might be opinionated, but we argue that it is nevertheless a good didactic exercise to think about the use of such constructs.

To obtain an impression of the overall learning process between the beginning and the end of the course, we compare the number of violations between the first and the last assignment. Since the assignments differ in size, we normalize the number of violations to make them comparable. There are several metrics for normalizing code sizes, e.g., the lines of code, the number of methods/classes, etc. We use the number of non-final method parameters per assignment, since lines of code would give the fourth assignment with GUI programming too much weight. The non-final method parameters are better than just counting the number of methods, as this metric also accounts for how complex a method is. Finally, the metric was already computed using the *MethodArgumentCouldBeFinal* rule and we knew from the manual correction that our students did not mark method parameters as `final`.

Table II is subdivided into five categories that describe the amount of change that happened to the violations. The first part lists violations that disappeared entirely from the first to the last assignment, followed by violations in which numbers decreased strongly. Then, the table lists violations in which numbers remained stable, followed by increasing and new violations. Due to space constraints, we just show five violations per category. Overall, the result is satisfying: Out of a total of 107 rules, 18 were not violated at all. The number of violations of three rules was stable, and 44 decreased of which 21 were not violated anymore. Of the 42 that increased, 32 did not occur in the first assignment, so they can be seen as rather specialized.

In summary, there is an improvement in the structure of the code in the last assignment: problems regarding the cyclomatic complexity diminish, and only few variables are prematurely declared. Braces are almost always placed, and most unnecessary parentheses are removed. Furthermore, bad habits, like checking references with `equals()` and instantiating objects in loops plummeted.

On the increasing side, we note that there are still issues with naming and declaring variables. The number of fields only used in one method tripled, the number of local variables shadowing a field went up by 284%, and variables were often defined too far from their usage. We partially attribute these issues to the topic of the last assignment, GUI programming, where UI classes can be cluttered due to the UI framework.

D. Validity Concerns

These findings are meant to be understood as a tendency, rather than strong empirical evidence. The reason for this is that the data is from one semester with only 18 groups. Also, we cannot attribute the effects to the teaching concept only, since this is only a case study without a control group of submissions that have not participated in the course. Nevertheless, the groups did not know that their assignments would be analyzed with static code analyses, so they did not program to conform with a standard.

IV. TEACHING CLEAN CODE AT SCALE

Giving valuable feedback in programming is time consuming. Marking about two dozen assignments that students without prior experience can solve within two weeks usually requires about a week of full-time work. We present two approaches to reduce this time and improve the review quality.

A. Knowledge Base

During the assessment of assignments, we found that students make similar mistakes that result in the same feedback. With a knowledge base of issue, one can simply provide links to the issues, thereby saving time and relieving the corrector from repetitive actions. It also leads to shorter code reviews. We propose the following structure for issues in the knowledge base:

Name A concise name capturing the solution to a code quality problem as an action, for example *"Avoid Negations"*.

Code Two code snippets within the same context, one containing the highlighted problem and the other one the solution. For example, a snippet containing a negation named `!done` and a snippet with the refactored solution like `inProgress`.

Text Two detailed explanations making an argument explaining the problem and supporting the solution, e.g., negations are harder to understand and the refactored solution reads much easier.

At first, we created a knowledge base for each assignment in the form of a markdown document named *"common issues"*. We noticed a time reduction in marking itself, more consistent and comparable markings, and, therefore, a reduction in the number of questions regarding the markings and comments. Furthermore, the knowledge base created a common terminology for the discussions in the courses. We have turned these *"common issues"* per assignment into a larger knowledge base in the form of a book [5], covering the most common *"common issues"* with high-quality code and text.

B. Automated Didactic Code Review

With a universal knowledge base in place, the correction is essentially reduced to spotting the problematic parts in the code and referring the students to the corresponding items. In practice, this is not trivial, since the corrector must determine the functional correctness before assessing code quality. For this we recommend using a large set of integration tests.

For automating the code quality review, we have developed a static code analysis meta tool[6] that can be integrated into existing build setups using Gradle. Currently, it scans the code for 107 code quality violations using PMD and Checkstyle and produces a `.csv` output that contains the violated rule, the identifier of the submission, and the location of the violation (file, line, column). As mentioned before, static code analysis tools need to be fine-tuned to be valuable. We picked these 107 rules based on our knowledge from the course, making them suitable for most learners, without enforcing a specific style of programming and producing too many false positives. Furthermore, the lecturer can adjust them to perfectly fit the needs of the students. Naturally, the rules cannot cover all issues in the code, since some problems are impossible to detect automatically. Nevertheless, the detection of many issues can be improved, without much effort on the side of the corrector.

[6]https://github.com/LinusDietz/QualityReview

V. Conclusions

This paper is motivated by the question of how universities can learn from software companies to improve their programming education. We presented and evaluated a code review-driven course concept for undergraduates that has been executed and awarded at the University of Bamberg. Facing the challenges of scaling the concept to an increasing number of students while keeping up the quality of code reviews, we proposed using static code analysis combined with a book on code quality that is suited for the target group of the learners.

In the future, we aim to empirically evaluate the didactic concept along with the knowledge base and the tool at multiple universities through experiments. Furthermore, we plan to integrate the knowledge base and tool into existing automatic educational code assessment frameworks like ArTEMiS [16].

Acknowledgments

The authors would like to thank Guido Wirtz for giving us the freedom to develop these courses. Moreover, we thank our student assistants Christian Preißinger, Gabriel Nikol, Michael Träger, Henrik Cech, and Tobias Jakubowitz who helped us lecture those courses.

References

[1] A. Vihavainen, M. Paksula, and M. Luukkainen, "Extreme apprenticeship method in teaching programming for beginners," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2011, pp. 93–98.
[2] F. G. Martin, "Will massive open online courses change how we teach?" *Communications of the ACM*, vol. 55, no. 8, pp. 26–28, Aug. 2012.
[3] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, "Towards practical programming exercises and automated assessment in massive open online courses," in *IEEE TALE*, 2015, pp. 23–30.
[4] J. Bailey and R. B. Mitchell, "Industry perceptions of the competencies needed by computer programmers: Technical, business, and soft skills," *Computer Information Systems*, vol. 47, no. 2, pp. 28–33, Jan. 2006.
[5] S. Harrer, J. Lenhard, and L. Dietz, *Java by Comparison: Become a Java Craftsman in 70 Examples*. Pragmatic Bookshelf, 2018.
[6] R. C. Martin, *Clean Code*. Prentice Hall, 2009.
[7] S. McConnell, *Code Complete*. Microsoft Press, 2004.
[8] J. Bloch, *Effective Java*, 3rd ed. Addison Wesley, Nov. 2017.
[9] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison Wesley, 1999.
[10] M. Fowler, *Refactoring*. Addison Wesley, 1999.
[11] R. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, pp. 112–111, May 2001.
[12] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," *CoRR*, vol. abs/1709.08439, 2017.
[13] J. Li, N. B. Moe, and T. Dybå, "Transition from a plan-driven process to scrum: A longitudinal case study on software quality," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM Press, 2010, pp. 1–10.
[14] J. T. Nosek, "The case for collaborative programming," *Communications of the ACM*, vol. 41, no. 3, pp. 105–108, mar 1998.
[15] A. Wilson, "Mob programming – what works, what doesn't," in *Agile Processes in Software Engineering and Extreme Programming*, C. Lassenius, T. Dingsøyr, and M. Paasivaara, Eds. Cham: Springer, 2015, pp. 319–325.
[16] S. Krusche and A. Seitz, "ArTEMiS – an automatic assessment management system for interactive learning," in *SIGCSE*. ACM, 2018.