The Impact of Automated Code Quality Feedback in Programming Education

Julian Jansen[1,2,3], Ana Oprescu[2], and Magiel Bruntink[3]

[1]julian.jansen@gmail.com
[2]University of Amsterdam
[3]Software Improvement Group

CEUR-WS.org/Vol-2070/paper-04.pdf

Abstract

While some university-level programming courses focus on software quality, often in introductory courses code quality is little touched upon due to time constraints. Students usually get feedback on code quality after the grading of their assignment, feedback that cannot be used on that same assignment. Our aim is to improve students' skills for code quality during the evolution of a students' programming assignment, while keeping the overhead low for teaching staff as well as for students. *Better Code Hub* is a service that checks code quality according to ten guidelines. We employ *Better Code Hub* as a formative assessment and feedback tool enabling students to monitor their progress on code quality. Our findings indicate that there is an improvement in the code quality of the students' assignments over the period the tool is used. Our experiments show that students benefited the most from feedback on unit length, unit complexity, and code duplication.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org). 07-09 June 2017, Madrid, Spain.

1 Introduction

An important programmer's skill is the ability to write high quality code. Code of low quality can cause maintainability, security, performance, and reliability problems. Education is a means to fill the gap between the supply and demand of skilled programmers.

As Ala-Mutka states [1, p. 84], programming courses are often large in size and cause a heavy workload for the teacher. We observed that in introductory programming courses there can be little to none systematic teaching in the rationale behind code quality. Ala-Mutka also states, that "[a]ssessing and providing feedback on computer programs is time-consuming, because there are many aspects relating to good programming that need to be considered [1, p. 84]." We observed that most of the educational effort is spent on the aspects of learning the fundamentals of programming, new languages, and development environments. Given the time constraints and the focus on these three aspects, students might get elaborate feedback on their code quality only at the end of the assignment. Therefore it cannot be used during the assignment itself. Often, the grade depends (partially) on code quality criteria.

This research focuses on measuring the impact of a tool that helps novice programmers improve their code quality. Better Code Hub (BCH) is such a tool, developed by the Software Improvement Group (SIG) based on their maintainability model [7]. BCH uses ten guidelines as shown in Table 1 to provide the user with code quality feedback. The tool indicates if the user is compliant with the guidelines as set by BCH. It presents refactoring candidates and a short text of *how* and *why* the refactorings should be done. The

BCH website (`www.bettercodehub.com`) provides an explanation per guideline.

1.1 Motivating examples

At the University of Amsterdam (UvA), students enrolled in a programming minor require no a-priori programming knowledge. During one semester, students start by learning the fundamentals of programming and later on specializes in making mobile applications or data visualisations.

The programming minor consists of six courses. A course can be four weeks full-time (average of 40 to 45 hours per week of total workload), or eight weeks part-time. "Programming 1" and "Programming 2" are partially based on Harvard's CS50, teaching the students the basics of computer science. "Programmeertheorie" focuses on heuristics and solving cases as a team. "App Studio" focuses on building a mobile application for *Android* or *iOS*. "Data Processing" focuses on gathering and visualizing data. "Programmeerproject" builds on "App Studio" and "Data Processing", where students work four full-time weeks on an assignment of their own choice.

"Programmeertheorie" has no systematic teaching in code quality, albeit code quality requirements like decomposition, style, and comments influence the final grade. "App Studio" and "Programmeerproject" courses use a grading rubric [14] to assess ten code quality criteria. Stegeman *et al.* [13] set out to formulate an empirically validated model for code quality assessment in introductory programming courses. The model is based on statements about code quality, sourced from three popular software engineering handbooks, and three instructors. From the model ten criteria are formulated in the rubric, that can each be graded on a scale from one to four. Based on the rubric, students get feedback on code quality after the grade is determined, meaning they cannot use the feedback *during* the assignment.

Again, although these courses have little to none systematic teaching in the theory and rationale behind code quality, the final assessment is partially based on code quality criteria.

1.2 Research questions

Our goal is to improve students' skills for code quality during the evolution of a student's programming assignment, while keeping the overhead low for teaching staff as well as for students. Naturally, we require a method to measure the impact of different strategies.

RQ 1: How can we measure the impact of an automated code quality feedback tool on the code quality of students' assignments?

RQ 2: What kind of impact does an automated code quality feedback tool have on the code quality of students' assignments over the span of a learning unit?

Our hypothesis is that the introduction of an automatic code quality feedback tool like BCH, improves the code quality of students' assignments. Our research contributes methods to measure the impact of introducing automated code quality feedback tools in programming education. It also provides insight into the impact of automated code quality feedback tools on students' code quality of assignments in programming education.

2 Background

This section presents research underpinning our work. First, we describe how code quality is part of programming education. Next, we describe how feedback can be provided. Finally, we elaborate on different code quality assessment tools.

2.1 Code quality in programming education

Our research is focused on teaching students about code quality as part of the larger topic of software maintenance, and is agnostic on impacting grading. Teaching practical software maintenance can be done in different ways. The *Software Evolution* course of the UvA's *Software Engineering* master has student teams analyse open source code with tools they develop themselves based on the SIG maintainability model [7].

Szabo studied enhancing group-based maintenance assignments, usually consisting of small code bases of very good quality in which artificial problems were introduced [15]. The author proposes to use old medium-sized student projects with real software bugs. These courses are maintenance-centric, and are aimed at students that are somewhat experienced programmers.

Keuning *et al.* [8] studied code quality issues in student programs. They investigate the frequency of issues related to program flow, choice of programming constructs and functions, clarity of expressions, decomposition and modularization in a large body of *Java* programs of novice programmers from the *Blackbox* database [3]. They found that novice programmers write programs with a substantial amount of code quality issues, and they hardly fix code quality issues, in particular those related to *modularization*. Code quality issues were most prevalent in the *expressions* and *decomposition* categories.

Table 1: 10 guidelines for code quality divided into three categories.

Code	Architecture	Way of Working
Write Short Units of Code	Separate Concerns in Modules	Automate Tests
Write Simple Units of Code	Couple Architecture Components Loosely	Write Clean Code
Write Code Once	Keep Architecture Components Balanced	
Keep Unit Interfaces Small	Keep Your Codebase Small	

2.2 Providing feedback

Hattie and Timperley conceptualize feedback as "information provided by an agent (e.g., teacher, peer, book, parent, self, experience) regarding aspects of ones performance or understanding [6]." The authors state that feedback is one of the most powerful influences on learning and achievement, but the impact of feedback can be either positive or negative [6]. Feedback can be accepted, modified, or rejected by the student. The authors found that studies with the highest effect sizes relating to feedback effects, involved students receiving information feedback about a task, and how to do it more effectively [6, p. 84]. Also, "the most effective forms of feedback provide cues or reinforcement to learners; are in the form of video-, audio-, or computer-assisted instructional feedback; and/or relate to goals [6, p. 84]." And, "feedback is more effective when it provides information on correct rather than incorrect responses and when it builds on changes from previous trails [6, p. 85]." Feedback appears to have the most impact when "goals are specific and challenging but task complexity is low [6, p. 85-86]." Hattie and Timperley [6] arrived at a model for feedback to enhance learning. It states that effective feedback must answer three major questions: "Where am I going? (the goals)", "How am I going?", and "Where to next?".

We distinguish between two forms of providing feedback: formative and summative. Formative feedback can be defined as "information communicated to the learner that is intended to modify his or her thinking or behavior to improve learning [12, p. 154]." This is the feedback that can be used during an assignment, or used in the next assignment within a course. We understand summative feedback, as feedback that is provided after the assessment of the last assignment of a course. Or as Sadler defines it, summative assessment "is concerned with summing up or summarizing the achievement status of a student, and is geared towards reporting at the end of a course of study especially for purposes of certification [11, p. 120]." As stated by Shute, the crux is how feedback can be delivered correctly to significantly improve the learning process and outcomes [12, p. 154]. According to Sadler the key premise is, that "for students to be able to improve, they must develop the capacity to monitor the quality of their own work during actual production [11, p. 119]."

2.3 Code quality feedback tools

We distinguish between tools that can provide automated feedback on code quality, and tools that fully depend on manual work, either by the student or teaching staff. This is not a division in formative and summative tools. Also, we are not aiming to automate the grading process. However, automated grading tools for student assignments do exist (i.e. *Web-CAT*, *Autolab*, and *INGInious*).

Automated code quality assessment tools. As Ala-Mutka states, the undeniable benefits of automated assessment tools include objectivity, consistency, speed, and 24-hour availability [1, p. 96]. Several commercial and open source tools are available that perform static code analysis, and provide code quality feedback in some form. Examples are: *Better Code Hub* (`www.bettercodehub.com`), *Code Climate* (`www.codeclimate.com`), *Codebeat* (`www.codebeat.co`), *Codacy* (`www.codacy.com`), *PMD* (`pmd.github.io`), and *SonarQube* (`www.sonarqube.org`).

These tools can differ in the way they are implemented in the process. Some provide the feedback within the Integrated Development Environment (IDE), but are limited to this IDE. Others are integrated into version control (i.e. *GitHub*), and can't be used with other version control systems. Lastly, the tools differ in the programming languages that are supported and the set-up needed per language (i.e. the need for separate engines for different stacks).

BCH is a tool that provides feedback along ten guidelines when integrated into *GitHub*. This means it can be integrated into the existing work-flows commonly used in education. It is possible that at every commit and push, the code is automatically analysed. Via the BCH website, the analysis report can be inspected. No installation on the students' computer is required. Optionally a configuration file can be created to scope the project to exclude code that should not be included in the analysis (e.g. libraries). No extra set-up per programming language is required, and the feedback is provided in an uniform way per guideline independent of different programming languages. BCH makes use of "quality profiles" to allow for a certain degree of code quality violations. A quality pro-

file divides metrics in distinct categories, ranging from fully compliant code to severe violations [16]. This is based on the principle that some violations are worse than others. With quality profiles, moderate violations, can be distinguished from severe violations [16]. A task-list can be made by selecting refactoring candidates from the list of presented violations. Students can see which tasks have the highest impact on the guideline. The quality profile gives a prediction when a refactoring candidate is selected. This allows students to focus on the highest impact refactoring tasks.

Manual code quality assessment tools

Becker proposes to use rubrics in programming education [2]. Becker's approach is to create two rubrics for each assignment: one focused on general style and design issues, the other on specific elements of the particular problem being solved.

Stegeman *et al.* [14] discuss the design of a feedback rubric for introductory programming courses, focused on code quality. Their rubric is based on criteria from their code quality model [13], and describes *what* is expected from these criteria. The rubric can also be used as a manual formative assessment tool, to provide feedback to students to improve their performance. For example, students can review each others code, or use it for self reflection. Within the UvA programming minor, the rubric is used for the code quality grading of the students' programming assignments. The model for assessment of code quality of Stegeman *et al.* is based on nine criteria for code quality, and three levels of achievement per criterion. The rubric we employed is a later iteration using ten criteria, and four levels of achievement. The design decision to use four levels of achievement mitigates the bias of graders to a middle level [14]. Also, as explained by Stegeman *et al.*, the "flow" criterion "seemed to encompass two different parts: the control of complexity and the appropriate use of control structures and library functions; as these goals were already formulated in a very isolated fashion, we split off the latter into a separate criterion named 'idiom' [14]".

3 Research method

We take the philosophical stance of critical theory. So, as stated by Easterbrook *et al.*,"there is effectively a moral imperative to intervene to solve the problem. Therefore, no attempt is made to establish a control group: the moral imperative implies that it would be unethical to withhold the intervention from some groups. Instead, the emphasis is on identifying useful lessons that help others who wish to pursue a similar change agenda [5, p. 302]." This philosophical stance we take, has an impact on the way we set-up our experiments. We cannot make a control group of students by denying them access to the automated code quality feedback tool, while students in the same iteration of the course do have access. We are unable to perform a controlled experiment, since there are variables which we cannot control that might affect the experiment. One of these is the fact that between courses the group of students differ. We are also introducing learning effects, since students might be in multiple courses that we use for our experiments.

Most research methods attempt to observe the world as it currently exists, where we aim to intervene in the studied situation to improve it [5]. We are introducing an artefact in a real-life context to solve a defined real-world problem. This leads us to Action Research (AR), that has been pioneered in education, where major changes in an educational strategy cannot be studied without implementing them [5, p. 301]. AR is "an approach in which the action researcher and a client collaborate in the diagnoses of a problem and in the development of a solution based on the diagnosis [4, p. 709]." We take into account the positivists concern of careful comparison of the "before" and "after" situations [5, p. 302], by analysing courses before the research started to compare against.

We are looking for organizational problems we can solve with this artefact, such as scaling the provisioning of timely feedback to a large number of students without increasing the teaching load. For this approach, we use a specific form of AR method, the Technical Action Research (TAR) method [18, 17], since this is an artefact-driven approach. TAR is an approach to validate new artefacts under conditions of practice. TAR starts with an artefact that is tested under conditions of practice by solving concrete problems with it [18, p. 220]. In this case, the artefact is BCH and the practice is in an educational setting. TAR bridges the relevance gap between "idealizations made when designing the artefacts and the concrete conditions of practice that occur in real-world problems [18, p. 220]."

The goal of the researcher is "to develop this artifact for use in a class of situations imagined by the researcher [18, p. 221]." The artefact is first tested on toy-problems under idealized conditions. Afterwards, it is scaled up to conditions of practice and more realistic problems are solved with it. This is done until it can be tested by using it in one or more concrete client organizations to solve concrete problems [18, p. 221]. In action research an intervention in a social situation is done in order to both improve this situation and learn from it [18, p. 220]. The entire TAR exercise is based on the assumption that what the researcher learns in this particular case, will provide lessons learned that will be usable in the next case [18, p. 232].

4 Design of experiments

As we set out to understand how to measure the impact, and what the impact of an automated code quality feedback tool is, we need data on how running BCH analyses impacts the code produced by students. Therefore we design two experiments where data is gathered each time BCH is used to analyse code. The first experiment is designed to understand how to measure the impact of introducing BCH. The second experiment is designed to understand the impact of introducing BCH in different variations. By establishing methods to measure the impact, we can formulate what kind of impact BCH has over the span of a learning unit. To increase generality, there is variation in the experiment parameters, such as course duration, size of student-teams, and the period the tool is used.

4.1 Experiment 1: Programmeertheorie

"Programmeertheorie" is an eight week part-time programming course (see Table 2). The course focusses on heuristic techniques to find solutions to problems, where the state-space cannot be completely explored by a computer. Students can choose between cases with different kind of problems. The course starts with the formation of a team of two to three students. Afterwards, the team commits to a case to work on for the remaining weeks. The solutions must be implemented in the *Python* programming language. The grading partially depends on if the code is readable / understandable: divided into functions / classes / modules, consistent style and comments are present where needed.

As students know they will be graded on code quality, there is an incentive to use the feedback of BCH. The use of BCH is mandatory, but the teams are free to use the feedback in anyway they see fit. The teams are instructed from week 2 to activate the "Push & Pull request analysis" functionality of BCH. Every time the team commits and pushes code to *GitHub*, an analysis is run on the code. Feedback from BCH appears in the commit history of *GitHub*, where they can browse to the full analysis report. The introduction lecture about maintainability and BCH is not part of this course. However, some students also participate in "App Studio", where the lecture is given. Our aim is to see how the teams pick up using the tool, without active encouragement from the teaching staff. A version of our BCH manual is made available, and the accompanying book [16] is available in the classroom.

4.1.1 Data analysis

We start by providing insight into the use of BCH, this time focused on the "Push & Pull request analysis" functionality. As the teams work for eight weeks on a single assignment, it is relatively long compared to the other courses of the programming minor. This enables us to explore how the repositories grow in volume.

We use the data generated by BCH in this experiment to explore different metrics, such as the volume trend and the guidelines (see Table 1). Because testing and software architecture are outside of the scope of this introductory course, we will focus on the guidelines of the "Code" category, and the "Write Clean Code" guideline.

We look into the compliance of these guidelines over time and the final compliance score of the last analysis. This last analysis allows us to compare the student teams against each other. Since each team started using BCH at a different moment in time, we need to align in time their respective first and last analysis of their repositories in order to compare their respective compliance trends. To normalize for the times BCH is run, and mitigate its effect on the trend line, we linearly interpolate the data points.

4.2 Experiment 2: Programmeerproject

"Programmeerproject" is a four-week full-time programming course, where students create a mobile application (the *Android* or *iOS* track) or a web visualization (the *Data processing* track using the *D3.js* library) with some requirements. Programming languages used are *JavaScript* and *Python* for a web visualization, *Java* for an *Android* app, or *Swift* for an *iOS* app. The assignment will be graded on five aspects including code, using the Stegeman *et al.* rubric [14]. As the students know they will be graded on code quality, there is an incentive to use the feedback of BCH.

The course is divided into four weeks. The first week opens with an introduction lecture. This week students write a proposal and design document, and deliver a prototype. By the end of the second week, students need to deliver an incomplete alpha version. By the end of the third week, they must finish a fully functional beta version. In the last week, the students should not work on adding new features, rather on finishing their application and meeting the grading criteria. The course ends with a product demonstration.

All set-ups based on this course are detailed in Table 3.

Set-up 3.1: The baseline. For this set-up we consider an instance of the "Programmeerproject" course that took place before this research started. This set-up has no involvement of BCH as indicated in Table 3. Thus, this is the BCH-agnostic set-up which we use as a baseline. This baseline set-up yields the compliance level against which BCH-enabled set-ups are compared. The students code base has been stored in repositories.

Table 2: Programmeertheorie.

Set-up	Week count	Enrolled students	BCH
2.1	8	53	Week 2 to 8

Table 3: Programmeerproject.

Set-up	Week count	Enrolled students	BCH
3.1	4	40	No BCH
3.2a	4	66	Week 4
3.2b	4	66	No BCH
3.3	4	29	Week 1-2-3-4

Set-up 3.2. In this set-up of the "Programmeerproject", students are encouraged to use BCH, however it is not mandatory. Compared to set-up 3.1, the student now can run their code through BCH. The difference between set-up 3.2a and 3.2b organically appeared by students that wanted to use the tool (3.2a), and students that ignored the tool (3.2b). To comply with ethics we could not withhold the tool from a part of the students.

The first week opens with a lecture that is also partly about maintainability and BCH. In the third week, the goal is to develop a fully functional beta version, which should be analysed by BCH by the end of that week. In the fourth week, the goal is to refactor the code and fix bugs. Thus, in this set-up, students are encouraged to use BCH only in this last week of the course. The accompanying book [16] is available in the classroom. With this set-up we can gather data from deploying BCH over a short period in the last week of the course.

Set-up 3.3. In this set-up the students must use BCH from the prototype version on. Compared to set-up 3.2, where the students are only encouraged to use BCH after they had to run it once at the last Friday of week three. The second week starts with a guest-lecture about maintainability and BCH. From the prototype version on, the students have to use BCH with "Push & Pull request analysis" enabled. We monitored which students did not yet start using BCH, so we could inform the Teaching Assistants (TAs) to help the student get started. Every time the student commits and pushes code to *GitHub*, an analysis is run on the code. Feedback from BCH appears in the commit history of *GitHub*, from where they can browse to the full analysis report. At every push to *GitHub*, the code is automatically analysed with BCH. We visited the students a few times after the guest-lecture, to help with questions about BCH. The accompanying book [16] is also available in the classroom. With this set-up we can gather data from an active approach of deploying BCH over a longer period.

4.2.1 Data collection

Preparation of the manually analysed repositories. We cleaned the repositories that were not yet analysed to make sure we only analyse the code written by the students, not third-party code used by the students. We also manually searched the repositories for duplicate files that were created as local version control (different versions put in separate folders in the repository) and deleted them, so this has no effect on the analysis. Some students changed the name of the repository during the course that caused the repositories to not match with the BCH data. This was manually fixed.

Set-up 3.1: Generating data. The projects are downloaded from *GitHub* and analysed at different points in time to be able to compare them to the data from set-up 3.2 and 3.3. The last week of the course was, just like in set-up 3.2 and 3.3, devoted to refactoring the code. To compare the data, we performed two analyses with BCH after course completion. BCH was not used or mentioned at all during the course. The first analysis is performed on the code closest to the last Friday of week three of the course, but no later than that day. The second analysis is performed on the code closest to the presentation deadline of the last Friday of the course, but no later than that. We removed two students that handed in code from a previous iteration of this course. One student had no commits after week three, so was omitted.

Set-up 3.2: Generating a test group. Out of the 66 students, 56 students used BCH. This organically gave us a group of 10 students that did not use BCH. By running BCH a-posteriori, we can measure the improvement over the last week of the course, similarly to set-up 3.1. The first analysis is done on the code closest to the last Friday of week three, but no

later. The second analysis is performed on the code closest to the presentation deadline of Friday in the last week of the course, with a margin of one day. Some repositories needed cleaning up (e.g. libraries), to make sure we only analysed the code of the students. Students that had no commits close to both our deadlines were omitted. This gave us 6 students from the set of 10 students we could generate data for. Two belonged to the *Android* track, and four to the *Data processing* track. We will call this set of 6 students that did not use BCH themselves "set-up 3.2b".

Set-up 3.3: Setting up a questionnaire. In addition to the quantitative data we gather with BCH, we also have students fill in a questionnaire about their experience with BCH to gather qualitative data. Our goal is to evaluate the usability of BCH feedback. Unfortunately we could not enforce the structure of students committing before and after BCH use, to see how students reacted upon BCH feedback.

To set-up a questionnaire we use principles as stated by Kitchenham and Pfleeger [9]. An ordinal scale of "strongly disagree", "disagree", "agree", and "strongly agree" is used to limit the time needed to respond to the closed questions. The typical "neutral" column is left out to force the students to make a decision between positive or negative. Also, to prevent a "middle option" that could act as a neutral choice, we have an even number of options for the closed questions. The closed questions are placed after the open questions, to limit the influence of first on the latter. They were handed out when the students used BCH around 2,5 weeks in the ideal case.

4.2.2 Data analysis

To test our hypothesis that the introduction of an automatic code quality feedback tool like BCH improves the code quality of students' assignments, we gathered BCH analysis data from set-up 3.1, 3.2a, 3.2b, 3.3. We use the same rationale as in experiment 1 for choosing the BCH guidelines. So, we will focus on the guidelines of the "Code" category, and the "Write Clean Code" guideline. We can directly compare set-ups 3.1 and 3.2 on the five guidelines. We cannot directly compare set-up 3.3 with set-up 3.1 and 3.2, since set-up 3.3 was measured over a longer time span by starting the measurements earlier in the four week course.

Per set-up we align in time the first and last analysis of each repository. Hereby we improve the depiction of the start and end compliance, and the compliance over the span of the analyses. To normalize for the times BCH is run, and mitigate its effect on the trend line, we linearly interpolate the data points. We plot the trend lines and corresponding confidence bands with a confidence interval of 95%. If the confidence bands do not overlap, we can state their is a statistical significant improvement in code quality when using BCH for the selected guidelines.

We will look at the compliance trends per guideline over time, where we cluster *Android* and *iOS* under *Native* applications to obtain a sizeable group. The web visualization projects will be the *Web* group. Also, the division between *Web* and *Native* was chosen, because the coding style used in *Web* follows a fundamental different approach than BCH's *Write Short Units of Code* guideline. The final compliance scores per guideline allow us to compare all three set-ups, including set-up 3.3.

We processed the data from the questionnaire by encoding the scale from "strongly disagree", "disagree", "agree", and "strongly agree", to 1, 2, 3, 4. Entries in-between circles are marked with ,5. The responses are labelled to prevent double or missing entries.

5 Results

Our results are grouped per experiment.

5.1 Experiment 1

Figure 1 shows the number of analyses per team. With the "Push & Pull request analysis" functionality of BCH enabled, the teams on average ran BCH 82 times. When we exclude outliers that run less than five analyses, we find an average of 112,77 runs, as measured over a period of seven weeks. Table 4 summarizes the final compliance per team per guideline. Figure 2 shows the volume trend per team, and the spread of the analyses over time. Figure 3 depicts the trends of compliance per guideline.

5.2 Experiment 2

With baseline datasets (set-up 3.1 and 3.2b) and datasets of the groups that used the tool, the datasets can be plotted against each other.

Our approach is the synchronize the first and last analysis of every student by mapping the first analysis and last analysis on a relative time line, to improve the depiction of start and end compliance with the selected guidelines. To normalize for the effect that number of times the tool is run has on the trend line, the data must be normalized. Our approach was to linearly interpolate the data points. A diagram that shows two trend lines for each dataset can be produced. In order to state there is a significant statistical improvement, the confidence bands (with a 95% confidence interval) of the corresponding trend lines should not overlap.

Figure 4a depicts the difference in measured code quality of the accumulation of five guidelines, for all three tracks (*Android*, *iOS* and *Data processing*) combined. The figure shows the improvement on the four

"Code" guidelines and the "Write Clean Code" guideline over time. Figure 5 shows the trends for set-up 3.1 and set-up 3.2a per platform (here the *Android* and *iOS* track are combined). Figure 6 shows the improvement on the combined five guidelines for set-up 3.3. The assignments in "Native" and "Web" applications are more homogeneous in "Programmeerproject", compared to the cases of "Programmeertheorie" (experiment 1), so we are able to cluster the assignments. With this platform clustering we can improve the visualization of the compliance per guideline diagrams, as shown in Figure 7 and Figure 8. Table 5 allows us to directly compare the difference in the final compliance scores of each set-up.

6 Discussion

We examine how we can measure the impact of an automated feedback tool, and what kind of impact the introduction of a automated feedback tool has on the code quality of students' assignments. We start with examining how to measure the impact by discussing the two experiments, and answer the research question of how this can be done. In the second section we examine what kind of impact the introduction of a automated feedback tool has, based on the results of the second experiment.

6.1 How to measure the impact of an automated feedback tool

We performed two experiments, each with different set-ups. In the first experiment we had just one set-up where BCH was used for seven weeks by teams with one assignment. In the second experiment BCH was used for a full-time four week assignment.

6.1.1 Experiment 1

This experiment provided us with data that was distributed over the span of seven weeks. Figure 2 shows that several teams have a wider spread over time in the distribution of the analyses over time. It also shows that some teams have performed few analyses (like team 3, 6, 15, 16 and 17). Using also Figure 1, we gain insight for the data filtering: it might be the case that students use the tool incorrectly and skew the data. Thus, data needs to be filtered to accurately measure the impact of BCH. This can be done manually or by helping the students correctly scope their project files.

Due to the diversity of the assignments, we cannot aggregate the measurements of all the teams into one diagram. We use Table 4 to summarize the final compliance per team per guideline. This provides insight into where the most compliance is per guideline. What we learned is that methods that created Figure 3 and Table 4 contribute to measuring the impact of distinct parts of the tool at a smaller granularity.

6.1.2 Experiment 2

Intuitively, in order to examine if an improvement in code quality can be attributed to an automated code quality tool, we need a baseline dataset. This baseline dataset is not straightforward to generate. A moral imperative might hinder establishing a control group, and generating data from an already finished course requires extensive cleaning.

We chose to visualize the combined compliance of a selected number of guidelines. Trend lines indicate how the students comply overall on the selected guidelines. In Figure 4a the confidence bands overlap, due to the lower number of students in the set-up 3.2b dataset. In order to state there is a significant statistical improvement, the confidence bands (with a 95% confidence interval) of the corresponding trend lines should not overlap. With a larger sample size, as used in set-up 3.1, we improved the confidence bands.

RQ 1: How can we measure the impact of an automated code quality feedback tool on the code quality of students' assignments?
To measure the impact of an automated code quality feedback tool we found that data is needed on how the students are performing on code quality. Data needs to be collected directly from the tool in a sufficient volume, with a certain interval over a defined period of time. For example, at least have the students commit and push their code every day, so an analysis is run in the background, over the period of the course where code is developed by the students. External reasons might prohibit the researcher from collecting enough data. By planning in advance, the researcher can mitigate scenarios that might be of influence. Students might ignore the tool, than the researcher should elicit information from the students to find out what the reasons are.

Data cannot always be aggregated due to the diversity of students' programming assignments, or different set-ups of courses. As a consequence, meaningful statistics cannot be performed because of the small sample size. However, the data can still be explored to gain insight on a more granular level than overall compliance scores produced by the tool. By visualizing the distribution of analyses over time, and the number of analyses per student/team, insight can be gained to base the filtering of data on. Outliers can be spotted and filtered out of the data. Guidelines of the tool can be selected that show a meaningful trend. Filtering is needed to improve the quality of the data and produce meaningful visualizations and calculations.

Table 4: Programmeertheorie final compliance scores per team. Zero indicates guideline not met, one indicates compliance with the guideline.

Team	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Write Short Units of Code	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
Write Simple Units of Code	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Write Code Once	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
Keep Unit Interfaces Small	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	1
Write Clean Code	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

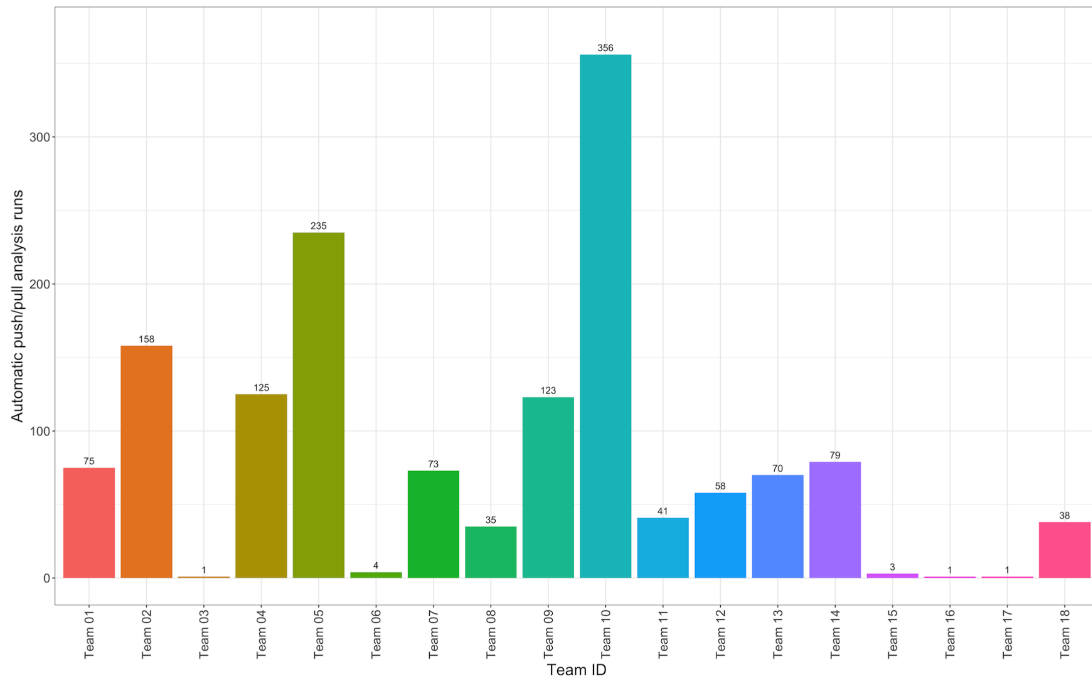


Figure 1: **Set-up 2.1**

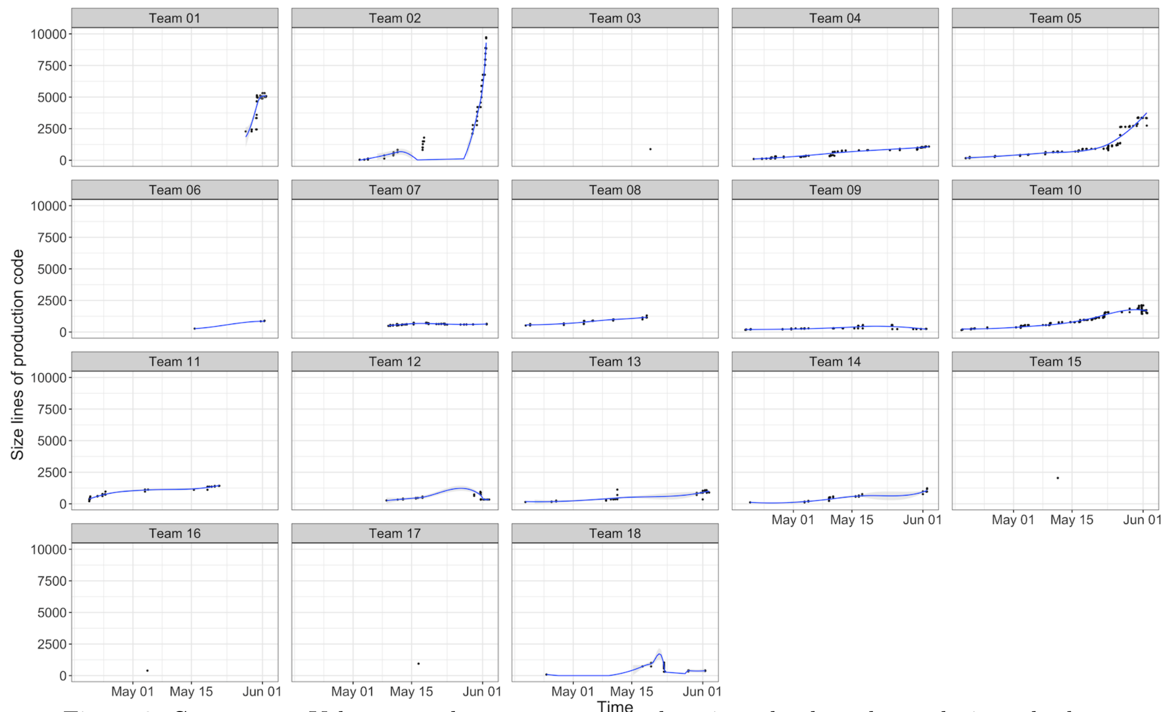


Figure 2: **Set-up 2.1** Volume trend per team, mapped against the date the analysis took place.

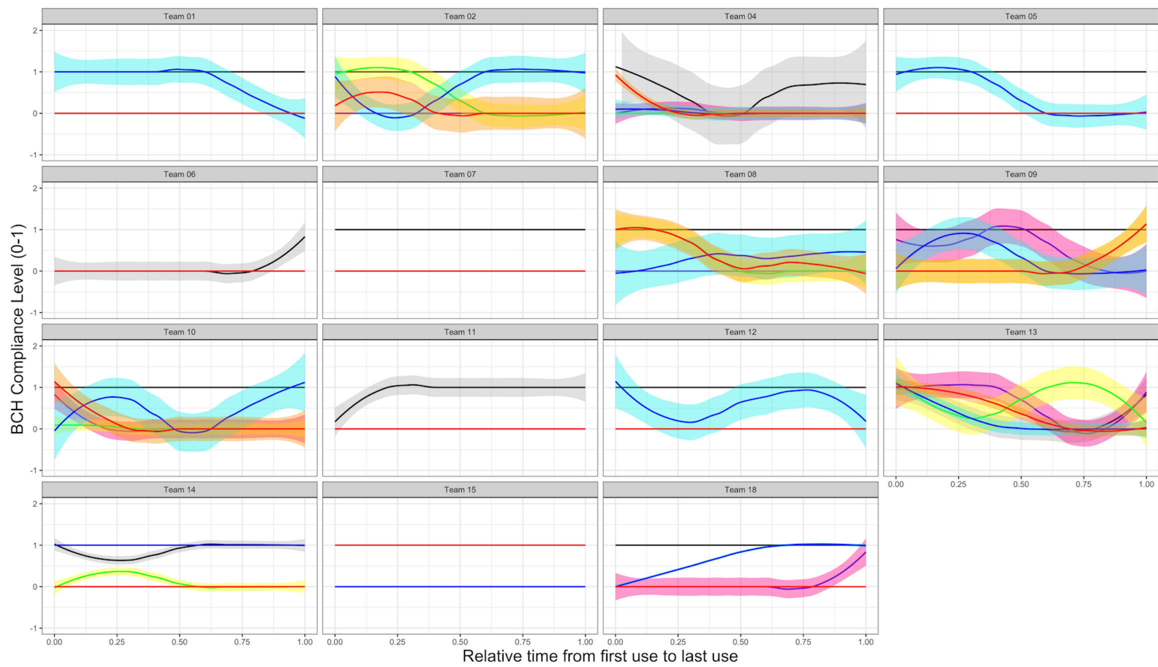
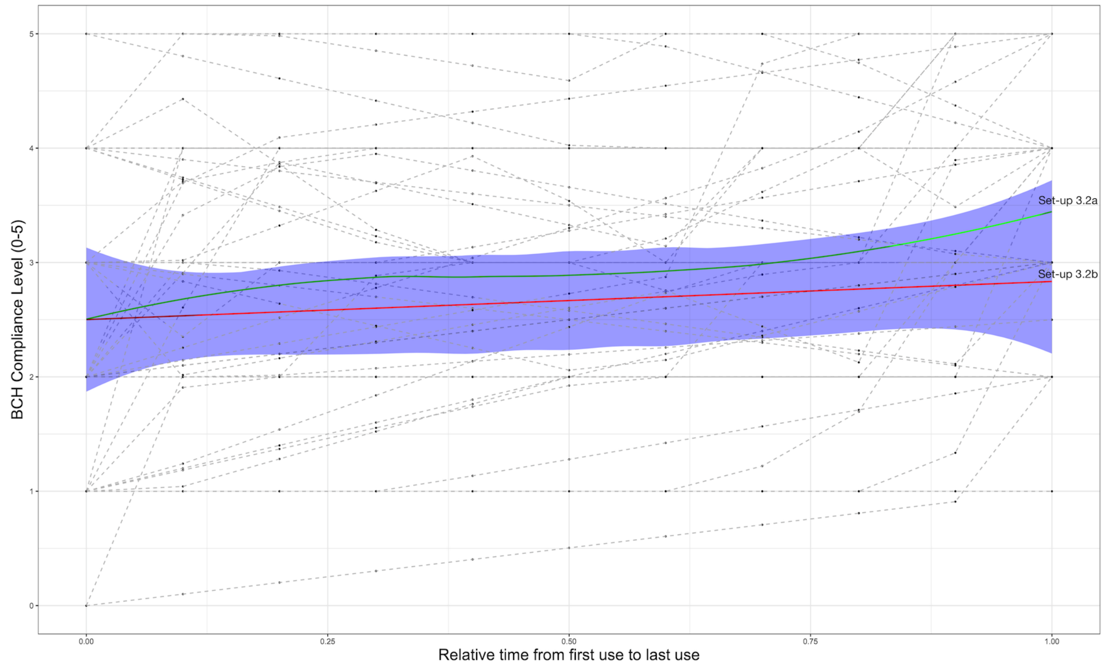
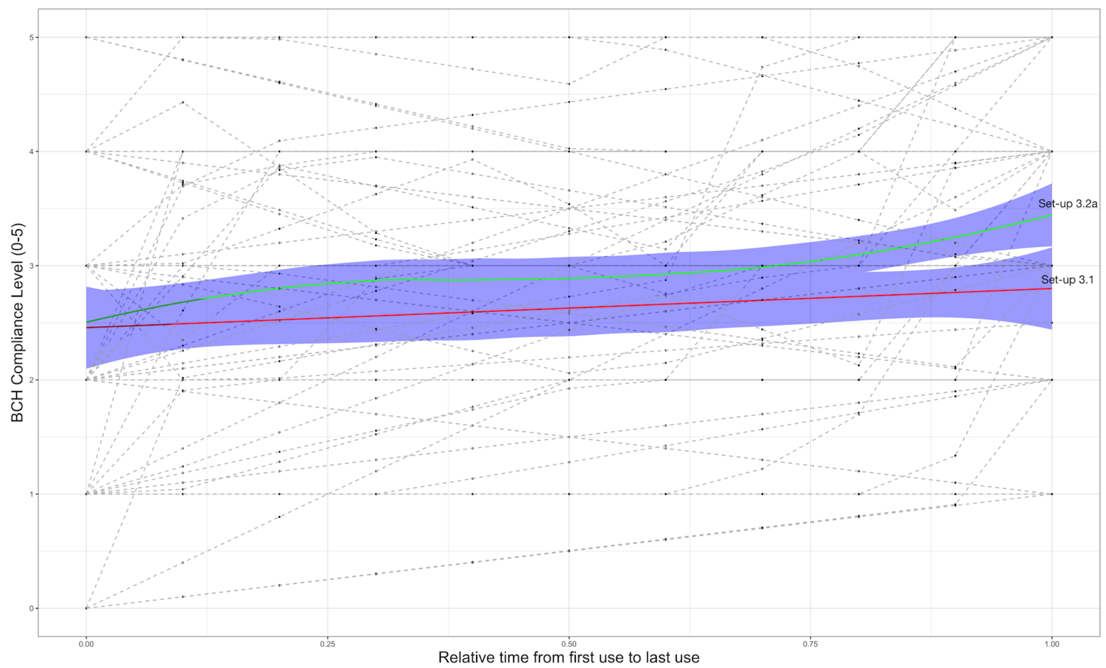


Figure 3: **Set-up 2.1** Improvement on the four "Code" category guidelines, and the "Write Clean Code" guideline per team. First and last analysis are synced up. 0 indicates that no compliance is met with the guidelines. 1 indicates compliance with the guideline. -1 and 2 are by-products of the confidence bands. Since we use trend lines, the lines might not end exactly on zero or one. See Table 4 for the final compliance scores. Teams with one measurement are excluded.



(a) **Set-up 3.2** Set-up 3.2a are the students that used BCH. Set-up 3.2b are the students that did not use BCH. Set-up 3.2b shows a straight line because we performed two measurements per repository.



(b) **Set-up 3.1 & 3.2a** Set-up 3.1 shows a straight line since we performed two measurements per repository.

Figure 4: Improvement on the four "Code" guidelines and the "Write Clean Code" guideline. First and last analysis are synced. Zero indicates that no compliance is met with the guidelines. Five indicates that all five selected guidelines are met. Each dotted line is a repository.

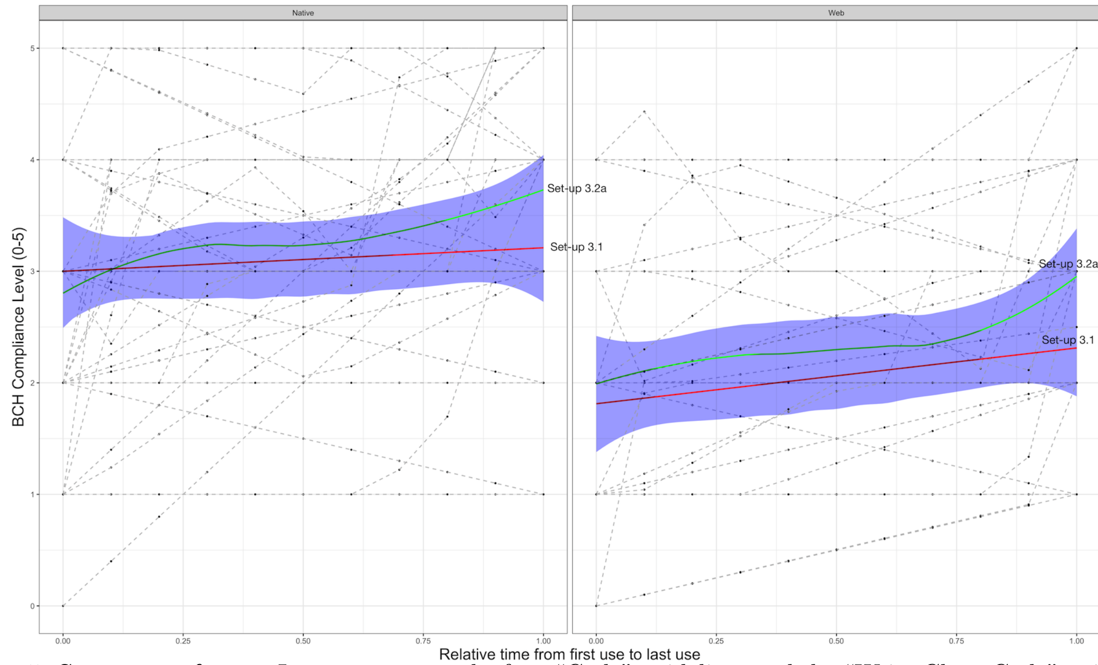


Figure 5: **Set-up 3.1 & 3.2a** Improvement on the four "Code" guidelines and the "Write Clean Code" guideline. First and last analysis of each repository are mapped on a relative time line. Zero indicates that no compliance is met with the guidelines. Each dotted line is a repository. Set-up 3.1 shows a straight line because we performed two measurements per repository.

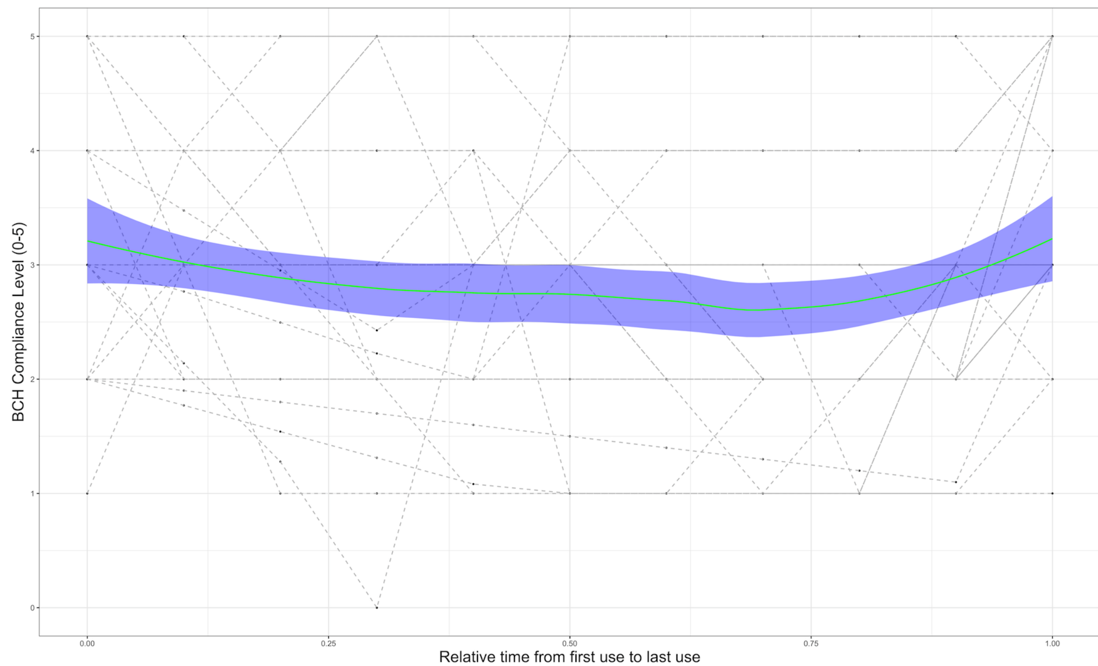
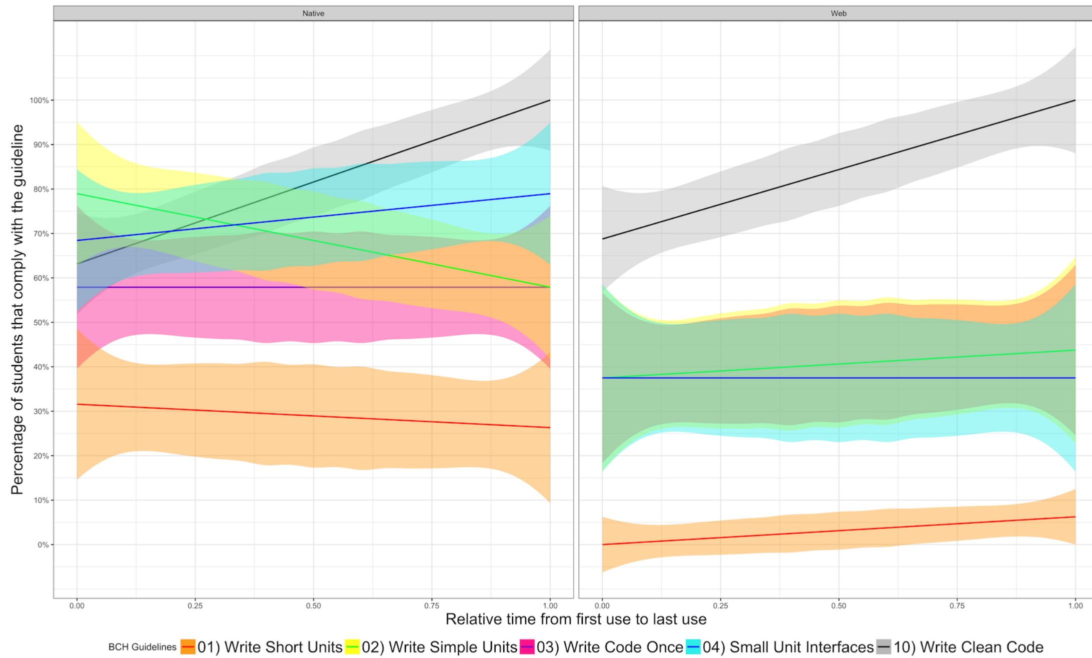
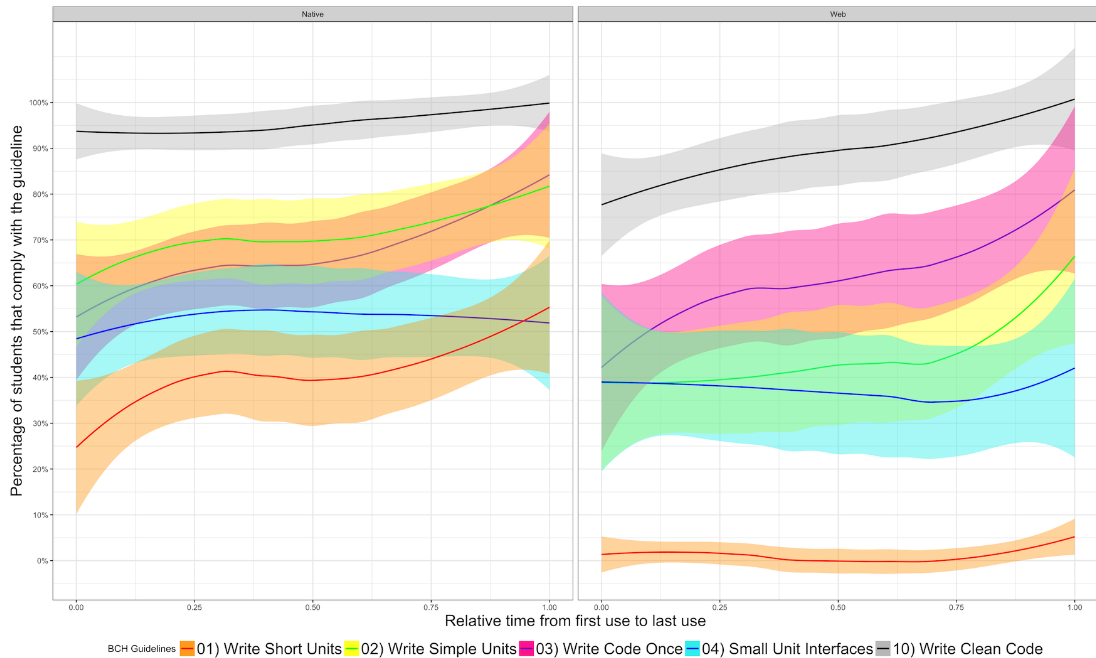


Figure 6: **Set-up 3.3:** Improvement on the four code guidelines and the write clean code guideline. First and last analysis are synced up. Zero indicates that no compliance is met with the guidelines. **The x-axes should not be directly compared to the diagrams of set-up 3.1 and 3.2, since set-up 3.3 was measured over a longer time span by starting the measurements earlier in the four week course.** Each dotted line is a repository.



(a) **Set-up 3.1** The diagrams show straight lines because we performed two measurements per repository.



(b) **Set-up 3.2a**

Figure 7: Improvement on the four "Code" guidelines and the "Write Clean Code" guideline. First and last analysis of each repository are mapped on a relative time line. The y-axes indicate the percentage of students meeting the guideline. Zero means that no compliance is met with the guidelines.

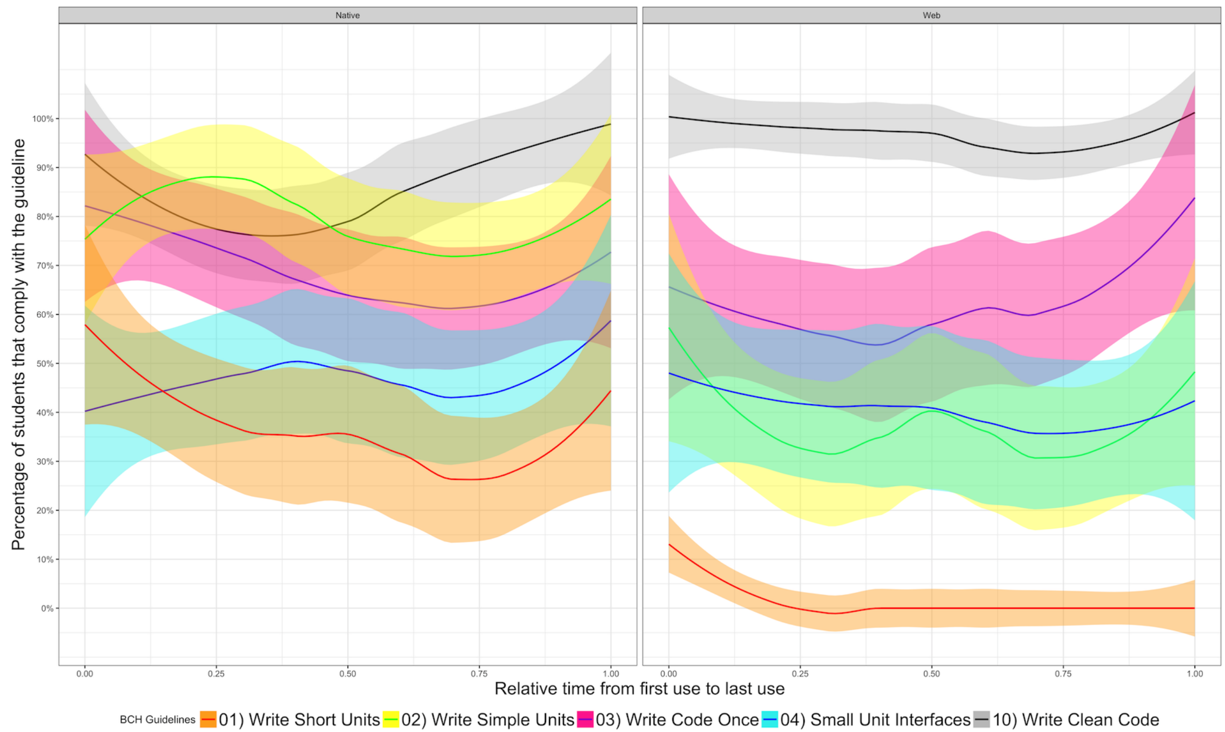
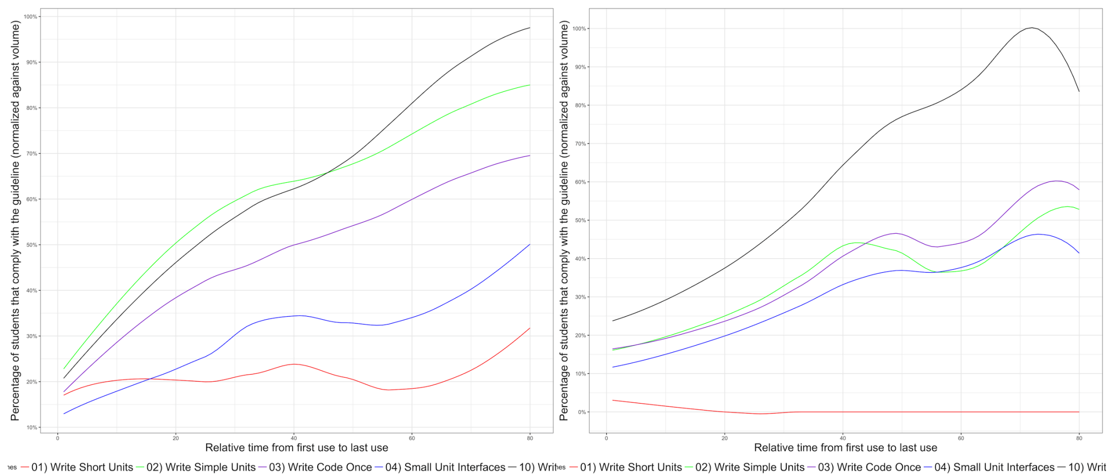


Figure 8: **Set-up 3.3** Improvement on the four "Code" guidelines and the "Write Clean Code" guideline. First and last analysis are synced. Zero means that no compliance is met with the guidelines. The y-axes indicate the percentage of students meeting the guideline. **The x-axes should not be directly compared to the diagrams of set-up 3.1 and 3.2, since set-up 3.3 was measured over a longer time span.**



(a) Native platform

(b) Web platform

Figure 9: **Set-up 3.3:** Improvement on the four "Code" guidelines and the "Write Clean Code" guideline, trend lines normalized with code volume. First and last analysis of each repository are mapped on a relative time line. Zero indicates that no compliance is met with the guidelines. The y-axes indicate the percentage of students meeting the guideline. **The x-axes should not be directly compared to the diagrams of set-up 3.1 and 3.2, since set-up 3.3 was measured over a longer time span.**

From a cleaned dataset, trend line visualizations mapped over time and tables with final compliance scores can be produced to gain insight in the performance of the students over time. To be able to state that an improvement can be attributed to the introduction of a tool, we need to compare against baseline data. One way to obtain such a dataset is by establishing a control group that does not have access to the tool. Moral imperatives might withhold the researcher from establishing control groups. In some cases, a group of students refuse to use the tool and data can be generated from this group afterwards. If this is not the case, the researcher can generate a baseline dataset from a previous iteration of the same course, where the tool was not used. In both cases cleaning of the data is required to make sure that the measurements represent the code quality of the students, not that of external libraries and frameworks.

We were not able to directly map the guidelines of BCH to the criteria of the rubric. So we did not use the rubric to measure the impact of BCH.

6.2 The kind of impact of an automatic feedback tool

We first examine the measurable impact of an automatic feedback tool. Next, we address how the students experienced the tool, after using it for a period of four weeks. Finally, we answer research question 2.

6.2.1 The measurable impact

In Figure 4a, the trend line of set-up 3.2a shows a higher quality over the complete span off the measured period. This indicates that there is an improvement in the code quality of the students' assignments over the period BCH was used. We cannot state that there is a statistical significant improvement over set-up 3.2b, due to the sample size of the control group that organically appeared. In Figure 4b we observe the same positive trend for set-up 3.2a compared to the BCH-agnostic trend line. This time we do observe that the confidence bands do not overlap. Therefore, we can state that there is a statistical significant improvement in the code quality of students' assignments over the period BCH was used.

In Figure 5, the "Web" platform starts about one guideline lower than the "Native" platform. Further inspection in Figure 7a and Figure 7b show a noticeable difference between the platforms on the "Write Short Units" guideline. This can be explained by the programming style used in the "Web" platform. Functions tend to become large, because these functions in turn can hold other functions and variables. Since BCH correctly detects the longer outer functions with its variables, the guideline becomes harder to comply with using that programming style. This explains why the "Web" platform starts noticeably lower than the "Native" platform.

In Figures 6 and 8, set-up 3.3 shows an interesting difference when compared to set-up 3.1 and 3.2, where BCH was used to measure in the last week only (manually by the researcher in the case of set-up 3.1). Where set-up 3.1 and 3.2 only show improvement over time, we observe in set-up 3.3 first a decrease in code quality, before improving again. We suspect that this is caused by the increase in volume of the project, and the focus in the last week of the course where it shifts towards fixing bugs and improving the code. To explain the decrease in code quality in set-up 3.3, we examine the hypothesis that code quality will drop when the volume of code increases, until no more features are added to the application (a slower increase in volume). We extract overall volume trend lines per platform. The volume trend lines are used to normalize the guideline trend lines as shown in Figure 9. We filtered out data points above the 2500 volume threshold, to exclude data points that are skewed by incorrect configuration files (e.g. the project is scoped in a way that third party code is included, like frameworks).

For the "Native" platform, the "Write Clean Code", "Write Simple Units", and "Write Code Once" guidelines tend to have a more linear trend of improvement. For the "Small Unit Interfaces", and "Write Short Units" we only see a steady improvement in the last part of the time span of the course. This might be explained by the last week, where the students fix bugs, and improve the code quality. For the "Web" platform, we see, as expected, that the "Write Short Units" guideline is not met at all by the students. The other guidelines tend to become more linear, but with a small decrease at the end. This might be explained by the volume that decreased in the end, by removing duplicate code.

Compared to set-up 3.1 all guidelines improve except for the "Keep Unit Interfaces Small" and "Write Clean Code" guidelines (see Table 5). From our visits in the classroom during set-up 3.3, we learned that the students off the "Native" platform found it hard to meet the "Keep Unit Interfaces Small" guideline because of the *Firebase* framework. A response we got in the questionnaire from an *Android* track student is, that it is found to be "impossible" to improve on the standard functions with four parameters. This might be explained by a risen popularity of *Firebase* being used by the students. The "Write Clean Code" guideline is met by every student in all the set-ups on every platform. Though, on the "Native" platform in set-up 3.2 the students start of higher on this guideline. We cannot explain this observation from our data. We do conclude that the use of BCH has no significant im-

pact on the "Write Clean Code" guideline. This conclusion is also supported by the final compliance scores of "Programmeertheorie", as compiled in Table 4.

Though we see improvements in code quality after introducing BCH, it is not a steep improvement overall. This might be caused by the main focus of the course, which is not on code quality. In a discussion with the TAs, one remarks that some students do not understand why the BCH feedback is useful to them. Students are focused on functionality, not on code quality. Some students are still working in the last week on functionality, instead of focusing on the code quality. We observe differences in students performance on the individual guidelines. "Native" platform students appear to benefit most from the "Write Short Units" guideline. In our baseline data we observed a decrease over time on the "Native" platform, without using the tool. In both the 3.2 and 3.3 set-up we measure nearly double the percentage of students meeting the guideline. This guideline could not be successfully complied with by the "Web" platform students, due to the programming style. Both platforms improve with the tool on the "Write Simple Units" guideline, in one case an extra 30% of students. All students improved with the use of the tool on the "Write Code Once" guideline, the "Web" platform students with over a doubling of the percentage. Students of both platforms overall did not benefit from the tool on the "Keep Unit Interfaces Small" guideline. Also, on the "Write Clean Code" guideline all students did not need any extra support from a tool, since they already scored 100% in all set-ups, in both the "Programmeertheorie" and "Programmeerproject" courses.

6.2.2 Experience of the students

We gathered qualitative data with a questionnaire about the experience of the introduction of BCH from the students in set-up 3.3. The results show that the students agree that BCH helped them improving their code. They mostly agree that BCH supported them in the learning process. A bad score on BCH did not demotivate the students, and they did not have the feeling they had to endlessly improve their code. Overall, the students agree that BCH has a place within the programming minor, and believe it should be used again in the course. The book "Building Maintainable Software" was hardly used during the course.

We do see some differences between the "Data processing" students responses compared to the other two tracks. The "Data processing" tend to agree more that they would have ended up with the same code quality without BCH, and some believe it unnecessarily cost them time. Time they would rather have spend with a Teaching Assistant (TA). They also think BCH is less of a valuable addition to the programming minor, compared to the *iOS* and *Android* tracks. Also, the "Data processing" found it to be less clear where the biggest quality problems in their code were. And, when asked to speculate if they would have used the tool on recommendation as well, the "Data processing" students lean towards the negative side. Overall, the "Data processing" students were more negative compared to the other two tracks. This might be explained by the "Write Short Units" guideline, where the students struggled with because of their programming style. From the responses of the open questions we can state that the students believe BCH to not be working correctly with the *D3.js* framework they used, despite correct measurements. From the open questions we notice that the students want to be able to meet all the guidelines on BCH, but because not all guidelines are directly related to their projects, they are not able to meet all of them (e.g. testing is not part of the course).

In our interview after set-up 3.3 with the TAs, we asked them how they think that the students experienced BCH. They believed there were students that did not fully understand the usefulness of a tool like BCH. They focused more on functionality, not on code quality. There were also people that took it seriously, and worked to meet the guidelines. They sometimes got angry when guidelines like "Automate Test", and "Write Short Units" could not be met, despite their best efforts. Questions arose if these kind of guidelines would be punished in the grading.

RQ 2: What kind of impact does an automated code quality feedback tool have on the code quality of students' assignments over the span of a learning unit?

We can state that there is a statistical significant improvement in the code quality of students' assignments when BCH is used. There are differences in the performance on individual guidelines. "Native" platform students appear to benefit most from the "Write Short Units" guideline. We measured nearly double the percentage of students meeting the guideline (see Table 5). "Web" platform students struggled with successfully using this guideline because of their programming style. Both platforms improve with the tool on the "Write Simple Units" guideline, in one case an extra 30% of students. All students improved with the use of the tool on the "Write Code Once" guideline, where the "Web" platform students appear to benefit most, with over a doubling of the percentage. Students of both platforms overall did not benefit from the tool on the "Keep Unit Interfaces Small" guideline. Also, on the "Write Clean Code" guideline all students did

Table 5: Final compliance as percentage of students that complied with the guideline (rounded to next integer).

Set-up	3.1	3.2	3.3	3.1	3.2	3.3
Platform	Native	Native	Native	Web	Web	Web
BCH usage	No BCH	Week 4	Week 1-2-3-4	No BCH	Week 4	Week 1-2-3-4
BCH Student count	19	34	16	16	22	13
Write Short Units of Code	26%	50%	50%	6%	5%	0%
Write Simple Units of Code	58%	79%	88%	44%	64%	54%
Write Code Once	58%	79%	75%	44%	77%	92%
Keep Unit Interfaces Small	79%	50%	63%	35%	50%	38%
Write Clean Code	100%	100%	100%	100%	100%	100%

not need any extra support from a tool, since they already scored 100%.

Students agree that BCH helped them improving their code, and mostly agree that BCH supported them in the learning process. A bad score on BCH did not demotivate the students, and they did not have the feeling they had to endlessly improve their code. Overall, the students agree that BCH has a place within the programming minor, and believe it should be used again. Though, an accompanying book was hardly used. A noticeable difference is that the students of the “Data processing” track responded more negatively compared to the other tracks.

7 Limitations

- The students between the set-ups differ, so the perceived differences might be influenced by them. We mitigated this by maximizing the number of students per set-up.
- We used relative small samples to base our statistics on. We chose to only use the statistics that made sense for our size of samples.
- Human error might be introduced by the researcher when handling the data.
- Some of the students that used BCH struggled with the configuration file (used to define the scope of which files are analysed), or did not scope in the beginning, despite our best efforts to instruct the students. This might have an effect on the measured code quality. We believe that the scoping was mostly set correctly, and nearly all of them towards the end. This observation is supported by inspecting individual volume trends. We saw the volume drops, most likely caused by changing the configuration file, happen in the middle of the time span. When generating analysis data (as in set-up 3.1 and 3.2b), we manually inspected the repositories, and deleted the code that did not belong to the students (e.g. external libraries and frameworks).
- Inherent to the nature of our research method, we only tested BCH within the context of the programming minor. Not all results might be generalizable and we can only hypothesise about the usage in different contexts.
- Researcher bias might be introduced by presenting the data in a favourable way. We mitigated this via critical reflection in the discussion.

8 Related work

In our work we focussed on employing an automated code quality feedback tool in a formative way to the students. As Ala-Mutka [1, p. 95] states, several authors reported that assessment tools with resubmission possibilities helped the students with their programming assignments. The automatic feedback guides the students, and, they note, that the type of feedback naturally affects the students' working strategy. Though, the authors referenced by Ala-Mutka focus on errors, erroneous code fragments, and/or the correctness of the program. Ala-Mutka concludes that it is unfortunate that the present assessment tools in 2005 are developed for local use, and only for a certain type of assignment. Adding, that inter-operable tool approaches would offer new and concrete co-operation possibilities for teachers for sharing knowledge. With the rise of continuous integration tools like BCH, we can contribute to the building of knowledge about approaches on the assessment of students code, that is widely applicable.

Keuning *et al.* [8] examined the differences in the occurrence of code quality issues between students who use code analysis extensions compared to students who do not. The extensions they selected are: *Checkstyle* for checking code conventions, *PMD* for static analysis to detect bad coding practices in *Java* programs, and *PatternCoder* for support with implementing de-

sign patterns. Their conclusion is, that the use of code analysis tools by students has little effect on the occurrence of issues. They state that educators should pay more attention to code quality in their courses, and automated tools need to be improved to better support students in understanding and solving code quality issues [8]. Their research is limited to *Java* programs, where we look at multiple different programming languages. A tool like *PMD* can overwhelm students with violations, where BCH works with quality profiles, meaning not everything has to be fixed, some percentage of violations are justified. *PMD* might present issues that are out of scope for novice programmers (e.g. *Law of Demeter* [10]). We are of the opinion that the way in which feedback is presented to the students, and the selection of feedback has an impact on the learning results.

9 Conclusion

In this research we set out to examine a proposed tool that may be used to mitigate the problem of the lack feedback regarding code quality during an introductory programming course. Code quality feedback is given after the grading and cannot be used during the assignment itself. Often, the grade depends (partially) on code quality criteria. Our goal was to improve students' skills for code quality during the evolution of a student's programming assignment by introducing an automated code quality feedback tool. We performed experiments for half a year within the programming minor to examine what benefits can be gained from introducing an automated code quality feedback tool in programming education.

We contributed methods to measure the impact of introducing automated code quality feedback tools in programming education, and provided insight into the impact of such a tool on students' code quality of assignments in programming education.

Our findings indicate that there is an improvement in the code quality of the students' assignments over the period an automated code quality feedback tool is used.

9.1 Future work

Apart from the measurements performed with an automated code quality analysis tool, we propose to automatically analyse the students' repositories on a daily basis. If the students daily commit and push their work to an on-line repository, a code quality performance trend can be generated, even if the students do not use an automated code quality analysis tool themselves. When data can be gathered with this method, it might allow the researchers to track students over multiple courses and assignments. And, when more uniform data is collected, we can further examine the hypothesis that if the tool is used more by the students, the higher the final code quality compliance. We did not gather enough data with our method to produce a meaningful visualization. It would also be interesting to gather data on the refactoring candidates that are being resolved, and allow us to examine the actionability of the feedback.

We explored more ways to be able to state claims with statistical significance. We found that for several tests for significance we did not have sufficient sample sizes. An interesting test to examine is a T-Test, and check if the groups statistically significant improved per guideline.

Lastly, the tool we used could not cover all the criteria of code quality where the students are graded on in the programming minor. It might be interesting to test a combination of tools, like tools that force style and conventions of programming languages. This might introduce to much overhead, and distinct tools might be needed for all the programming languages used.

Acknowledgements

We would like to thank Martijn Stegeman (UvA), Gracia Redder (UvA), Michiel Cuijpers (SIG), Marco Di Biase (SIG), Paco van Beckhoven (UvA / SIG), and Nevena Lazarevic (SATToSE shepherd) for their feedback.

References

[1] Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] Katrin Becker. Grading programming assignments using rubrics. In *ACM SIGCSE Bulletin*, volume 35, pages 253–253. ACM, 2003.

[3] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 223–228. ACM, 2014.

[4] Alan Bryman. *Social research methods*. Oxford university press, 2015.

[5] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[6] John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007.

[7] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[8] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. 2017.

[9] Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Software Engineering Notes*, 27(2):20–24, 2002.

[10] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE software*, 6(5):38–48, 1989.

[11] D Royce Sadler. Formative assessment and the design of instructional systems. *Instructional science*, 18(2):119–144, 1989.

[12] Valerie J Shute. Focus on formative feedback. *Review of educational research*, 78(1):153–189, 2008.

[13] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 99–108. ACM, 2014.

[14] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, pages 160–164. ACM, 2016.

[15] Claudia Szabo. Student projects are not throwaways: teaching practical software maintenance in a software engineering course. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 55–60. ACM, 2014.

[16] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.", 2016.

[17] Roel Wieringa. Technical action research. In *Design Science Methodology for Information Systems and Software Engineering*, pages 269–293. Springer, 2014.

[18] Roel Wieringa and Ayşe Moralı. Technical action research as a validation method in information systems design science. In *International Conference on Design Science Research in Information Systems*, pages 220–238. Springer, 2012.