Designing a Rubric for Feedback on Code Quality in Programming Courses

Martijn Stegeman
University of Amsterdam & Open University, The Netherlands
martijn@stgm.nl

Erik Barendsen
Radboud University & Open University, The Netherlands
e.barendsen@cs.ru.nl

Sjaak Smetsers
Radboud University, The Netherlands
s.smetsers@cs.ru.nl

ABSTRACT

We investigate how to create a rubric that can be used to give feedback on code quality to students in introductory programming courses. Based on an existing model of code quality and a set of preliminary design rules, we constructed a rubric and put it through several design iterations. Each iteration focused on different aspects of the rubric, and solutions to various programming assignments were used to evaluate. The rubric appears to be complete for the assignments it was tested on. We articulate additional design aspects that can be used when drafting new feedback rubrics for programming courses.

CCS Concepts

• **General and reference → Design;** • **Social and professional topics → CS1; Student assessment;** *Software engineering education;*

Keywords

Programming education, Code quality, Feedback, Assessment, Rubrics

1. INTRODUCTION

Code quality is an aspect of software quality that concerns directly observable properties of source code. Software engineers and researchers have shown a particular interest in improving the internal quality of software, assuming that this will have a positive effect on the external quality as experienced by users [10]. Teachers in the field of computer science, in turn, have increasingly provided students with feedback on the quality of their solutions to programming assignments. However, enrolments for introductory programming courses began to surge as early as two decades ago [18], with time to produce feedback growing proportionally. This has prompted the introduction of teaching models where teaching assistants play a prominent role; for example, [24].

Koli Calling 2016, November 24 - 27, 2016, Koli, Finland

© 2016 Copyright held by the authors. Publication rights licensed to ACM. ISBN 978-1-4503-4770-9/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2999541.2999555

These factors have engendered active research among teachers to develop tools that systematize and automate feedback, aiming to ease teaching loads and to provide consistency and timeliness in grading, as well as in feedback. One approach is to create detailed grading schemes, to be used by any combination of teachers, assisstants, and students. Several grading schemes have been published [5, 7, 2, 21, 3], but while all of these focus on similar aspects of code quality, such as readability, style and decomposition, they are very diverse in form as well as in content. Our goal is to understand how to create such a grading scheme in a systematic fashion and what design principles emerge from this process. In earlier work [22], we have created a detailed model of code quality as it applies to early programming courses (using languages from the imperative paradigm). That model is based on empirical data: the contents of several books on code quality, as combined with norms in use by teachers of introductory programming courses. In this study, we describe the first steps towards using that model to design a rubric that helps articulate feedback for students. We also describe the design aspects that appear to be relevant for this type of rubric.

2. FEEDBACK AND RUBRICS

Feedback given to students in a course can be used to promote learning [6]. In order to learn, a student needs to know three things: what good performance on a task is; how their own performance relates to good performance; and what to do to close the "gap" between those [20].

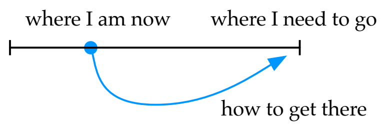

Figure 1: Feedback components from [20]. The horizontal line forms a scale of achievement.

A rubric is a tool that helps the assessment of student work by defining a set of criteria, a number of levels of accomplishment, and optionally verbal descriptors that explain the various levels [19]. Such a rubric can be used to calculate grades, to provide feedback, or both [9]. Rubrics were once developed to perform a summative assessment by quickly ranking or scoring, without providing content-related feedback [4]. However, they can be augmented to function as a tool for formative assessment, where they are used to provide feedback to students who can use it to improve their performance. Andrade [1] argues for an instructional rubric, primarily designed as a teaching tool instead of a scoring tool. Such rubrics prominently feature the verbal descriptors, which become the focal point of the assessment process. The key idea here is that students can use the rubric to understand what is important for good performance on the task at hand: it provides transparency [8]. This is achieved by designing criteria, levels and descriptors in such a way that students can use the rubric to figure out the three things they need to know to learn (cf. Figure 1). Introducing rubrics may not directly cause the desired learning effect, as there are many mediating and moderating factors. Still, many positive results have been reported after introducing of rubrics in the classroom [15].

2.1 Rubric design principles

Literature provides best practices for several aspects of creating rubrics. We list principles for three important aspects.

Rubrics can be either general purpose or task-specific. General purpose rubrics can be used across a range of assignments or even courses, while task-specific rubrics are fully aligned to the particular requirements of an assignment. A general purpose rubric can be used as a long-term learning tool, assuming that understanding will develop as the rubric is used repeatedly by students [23]. Messick [12] suggests to aim for a middle ground that is representative for a *class* of tasks.

Deliberately choosing the number of achievement levels in a rubric is especially relevant to instructional rubrics. Sadler [20] argues that many levels have to be defined. This should help student motivation by allowing them to see the results of their progress, compared to simply getting a pass or fail result. A more practical matter is that the number of criteria should preferably be even, as graders tend to bias to a middle level when the rubric is used by multiple graders [25]. For a rubric to be used by many teachers, four levels is recommended by Walvoord [25].

Rubrics can be augmented or replaced by exemplars; these are partial products that represent a particular level of accomplishment. Rubrics without exemplars seem to improve learning more than when using both, or solely exemplars [11].

2.2 Rubric quality

Rubric quality can be evaluated using concepts and techniques derived from psychological test construction. There, reliability shows whether a test can consistently provide the same results, and validity shows to what extent a test covers what it is intended to measure. Moskal and Leydens [14] propose how to apply this to rubrics. Content-related evidence should show that the rubric covers all relevant aspects of the domain; construct-related evidence should show that results of assessment using the rubric are representative for student performance and do not, for example, take irrelevant aspects into account; criterion-related evidence should show that student performance would translate to an outside setting. Besides that, interrater and intrarater reliability should show that no unwanted variation in the assessment is present. Jonsson and Svingby [9] show that these aspects are evaluated in various ways in rubric studies.

3. METHOD

The model of code quality from our earlier work [22] aims to be applicable to university programming courses. Our question is: how can we use this theoretical model to construct a rubric that is useful for assessing code quality in introductory programming courses? The method for answering this question is an implementation of *educational design research*. Like other design strategies, this methodology follows an iterative path [16]. During multiple cycles, educational interventions are developed and the results of testing the interventions are studied. Any cycle brings forth two types of results: a tentative product, and tentative design principles. During further cycles, these principles are tested in the development of new versions of the product.

In the case of this preliminary study, we aim to find a first set of design principles that help construct a valid rubric. From the start, one part of validity is derived from the underlying model of code quality: it is assumed to be reasonable complete for introductory programming courses. However, condensing the model into a rubric compels us to make choices about what to include and what to leave out.

Broadly speaking, one can distinguish four phases in a design cycle: (i) problem identification and analysis (ii) collecting initial design guidelines and creating a prototypical intervention (iii) several iterations of testing and refinement (iv) deriving design principles [17]. We have described the first phase in previous paragraphs, and will now describe how we approached the design.

We created an initial prototype of the rubric using design principles from literature. In three iterations, the rubric was subjected to trial assessments and results were discussed with teachers (Table 1). In all cases, the rubric was used with student work, written in response to programming assignments. Each iteration had a different focus: the first iteration was a basic understandability check, with a single teacher who had not used the rubric before; the second iteration aimed to test completeness, with three teachers who mostly had no experience with the rubric, thereby providing content-related evidence for validity; the third iteration aimed to find structural problems, possibly originating in the underlying model, thereby providing construct-related evidence for validity. This final iteration was performed with help of a teacher who had extensive experience using an earlier version of the rubric. We used different platforms and assignments, some allowing for decomposition and modularization (Table 2). After the three iterations, we analyzed the changes made to the rubric to propose aspects that are relevant for further development. Further detail on the method used in each step is provided in the next section.

round	# participants	experience
1	1	none
2	3	none
3	1	extensive

Table 1: Evaluation participants by iteration.

round	platform	decomposition	modularization
1	C	no	no
2	C	yes	no
3	iOS	yes	yes

Table 2: Assignment language by iteration.

4. RUBRIC DESIGN

Initial design

We constructed a rubric by a defining a set of criteria, formulating a number of levels of accomplishment, and writing verbal descriptors that explain the various levels. We derived the nine criteria from our code quality model [22]. We chose to define four levels of accomplishment, given that we aim for the rubric to be usable in many programming courses. To create a simple progression, we chose the following definitions for each: (i) problematic features are present (ii) core quality goals not yet achieved (iii) core quality goals achieved (iv) achievement beyond core quality goals. To create an instructional rubric that can facilitate learning, we wrote descriptors that help understand differences between levels of achievement[1].

Iteration 1

Assignment and participants — We subjected the initial version of the rubric to a basic test by asking one teacher to assess 5 student-submitted C programs using it. The goal was to eliminate any basic mistakes in the design of the rubric. All programs were produced in response the same assignment; no code framework was given to students, but hints and tips on the approach were given in the description; because of the nature of the assignment, students were generally not expected to perform any decomposition or modularization; the solutions were anonymized; and the teacher was experienced with this assignment, but had not graded these particular solutions before.
Method details — We first asked the teacher to think aloud while assessing the programs without using the rubric and to write down 2—3 suggestions for improvement for each student. We then asked them to use the rubric for a second assessment of all programs and indicate the perceived level for each criterion. Finally, we talked through all results. The teacher was asked to reflect on similarities and differences between their own feedback and the feedback that they indicated in the rubric.
Results and changes — As expected, the modularization criterion was not used for any of the solutions. Although there was also no realistic opportunity for decomposition in this assignment, the teacher was able to classify the performance at lower levels. There was a need for some corrections and clarifications, but no change in levels or arrangement of criteria. For *names*, the word "fuzzy" could be removed as its meaning was unclear and irrelevant; also for *names*, the concept of consistent casing was added (this concept was indeed present in the empirical data from the model, but had not been added to the rubric); for *formatting*, the goal of consistency was added explicitly; for *flow*, the term "choice of libraries" was explained more clearly as "use of library functions".

Iteration 2

Assignment and participants — To more thoroughly evaluate the second version we asked three teachers to use the rubric to assess two solutions to a programming problem. This problem again asked for a solution in C, but this time a basic framework was offered to students; the students were expected to decompose their solution code when needed, but there was no opportunity for modularization; the solutions were anonymized; one of the teachers was the teacher that also participated in the first round; and the teachers were all experienced with the assignment, but each had not graded the provided solutions before.
Method details — In this evaluation, we asked the teachers to write down feedback on the program without discussing and without use of the rubric; we also asked them to provide two important code quality goals for the student based on their assessment. Then we discussed the rubric by talking through each separate criterion and providing clarifications when asked. Then, the teachers assessed both student solutions using the rubric. Finally, the teachers discussed if their important goals were also present in the rubric, and were asked to identify any further problems that they encountered.
Results and changes — As in the first iteration, the modularization criterion was not used. This iteration provided input for a limited number of significant changes to the rubric. Lines of code that are too long appeared to be seen as a problem with *formatting* and not overall *layout*; we moved this item to the formatting criterion; in the rubric, layout was said to be about the "arrangement of code in source files", but it appeared quite unclear what was meant by that phrasing; we replaced this by using "positioning of elements in source files"; the teachers noted and agreed on a separation between having *old code* that is "commented out" and code that is simply unreachable given the structure of the program; these two were separated, moving unreachable code to the *flow* criterion; on the other hand, 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".

Iteration 3

Assignment and participants — For the final in-depth evaluation of the rubric, one teacher was asked to assess seven student projects written in Swift using the iOS framework. The students were expected to use decomposition and modularization, and solutions were generally composed of 10 or more classes; the projects were anonymized, and this was the first assessment of the projects; the teacher had used the initial version of the rubric about 6 months earlier to assess different student projects. Our goal was to elicit information about potential problems in all criteria and all levels of the rubric.
Method details — Here, the teacher was asked to perform an in-depth review of the code, indicate the perceived level of the code for each criterion, and provide elaborate written feedback, pointing out specific problems by referencing the code. The teacher was interviewed by first asking about each box in the rubric in particular detail, and then discussing the feedback that was written in relation to the rubric assessment.
Resulting changes All criteria of the rubric were used for all assignments. Not all levels were used for all criteria, especially the lowest and highest level. For *names*, the "clear word boundaries" needed clarification in the previous round and again in this round; this led us to rephrase as "consistent use of casing", referring to the common parlance of "camel casing". For *headers*, it was left implicit that header comments are generally required to be present; this has been made implicit in the new version; also, the lack of accuracy

[1]All versions of the rubric are available on `stgm.nl/quality`

in level 2 was made explicit. A lack of consistency between the *headers* and *comments* criteria was noted, and both were changed to require correct spelling and consistent use of natural languages in all but the lowest level. For *comments*, the phrasing "comments highlight important decisions" required some explanation; to make the criterion better understandable, we changed this to the slightly more abstract "comments explain code". For *layout*, the amount of lines in "lines are too long to read" was not clear, as it seemed to be a very strict statement; we changed it to specify "generally too long to read". For *layout*, it was noted that positioning of elements is not only supposed to be consistent, but also in line with platform conventions (e.g. in Swift, member variables connected to the user interface should be at the top of a class); this requirement was added to the fourth level to reflect the fact that this asks for some initiative and research by the student. For *formatting*, the phrasing "differences and similarities" was not clear; it was changed to state that similar parts of code should be similarly formatted (i.e., consistency). For *flow*, it was not clear that "exceptions" could point to any kind of exception, not only the exception handling feature of some languages; to broaden the applicability, the word "jumps" was added. For *flow*, a clarification was added for the highest level, because it was not immediately clear how to link it to other levels. For *idiom*, based on the assignment at hand, the use of library functionality seemed to be a higher level of achievement than an appropriate choice of control structues; it was moved up one level. For *expressions*, a clarification was needed that long expressions can be seen as a variant of complex expressions. For *decomposition*, a clarification was added that sharing variables between routines can be contrasted to using paramenters. For *modularization*, the same phrasing as in *decomposition* was added, marking a lack of trying as problematic. For *modularization*, the word "subject" needed clarification, and the more common term "responsibilities" was introduced instead. For *modularization*, in level three, the word "somewhat" was added to convey a more lenient mode of assesment.

5. DISCUSSION

In the previous section we have documented the changes made during the development of the rubric. Although we based these changes on interactions with a limited amount of teachers, we already observe some patterns that may be translated into design principles, to be used for designing new versions of the rubric. Below, we group the observed patterns into four areas of interest.

Abstractions

We note that condensing all information in the code quality model for use in the rubric sometimes leads to unclear generalizations or missing concepts. In some cases, the levels of certain subcriteria needed to be amended. Some subcriteria were moved to different criteria in order to better fit the mental models of the teachers. One criterion was even split into two: the criterion had already been formulated as two virtually separate parts, separated by a semicolon. There was also one instance of a missing subcriterion (casing). Evaluating with more teachers and finding common patterns seems to be needed to improve on the choice of abstractions in the rubric.

Goals

Criteria in the model are linked to certain goals. During the design iterations, it appeared that making these goals explicit helped teachers' understanding of the rubric. We added several explicit goals during the design iterations: "consistency" in, for example, the use of casing; "convention" as it applies to layout; and the "lack of trying" that is often (implicitly) part of the lowest achievement level. There is a clear opportunity in drawing explicit parallels between the goals in different criteria by choosing similar words to describe these. For example, it seems to be possible to categorize goals as either optimizing or maximizing. For example, decomposition asks for subtle optimization, not the creation of a limitless amount of tiny methods, while formatting aims to follow the structure of the program as much as possible.

Terminology

In several cases, descriptions in the rubric did not resonate with the participants. Two variants were found. The first concerned terminology that is too contrived and not clearly linked with the domain: "clear word boundaries" was too text-technical, while programmers call this "casing". The other variant concerned cases where the chosen terms seemed to point at specific instances, where a more general concept was intended; for example where the use of "exceptions" was changed to specify "exceptions and jumps". Creating a general or local glossary of common terminology in introductory courses, or simply comparing the rubric to other course materials, should contribute to a better choice of words.

Phrasing

In many cases, the phrasing used in the rubric was not explicit or precise enough. Teachers appeared to need more guidance to make better decisions. In several cases, key words were added to the descriptions in order to make them more recognizable. Sometimes they signaled a contrast, or added precision. There were some examples where we felt the need to soften the boundaries between levels somewhat, because interpretation appeared to be overly strict. We did this by adding moderating words, such as "generally" for lines that are too long. The rubric has explicit instructions that level 1 features (that is to say, problems) generally shouldn't be present in work of level 2. There was a tendency to make this explicit in the rubric itself by adding statements in all relevant level descriptors. For example, for *headers* we explicitly added their required presence to levels 2–4, while earlier, only their absence was stated at the lowest level. Doing this systematically for all criteria seems to be a good way to bring clarity and consistency to the rubric.

6. CONCLUSIONS AND FUTURE WORK

We have created the first version of a rubric that is based on a model of code quality as it applies to introductory programming courses. The rubric seems to be reasonably complete, given that we tested with several different types of programming assignments, while no substantial parts of code quality appeared to be missing. This provides evidence for the content validity of the rubric. For construct validity, we see that many different kinds of changes were made to the rubric in the three iterations: at the current stage of development, this is still a weak point. Finding important design aspects and principles should help redesign the rubric more systematically and help understand the nature of good abstractions. This work describes a first set of candidate design principles for that purpose.

There are other aspects of the rubric's quality that we did not study. Criterion validity should show that student performance would translate to an outside setting. Although part of this stems from the fact that the rubric is generated from a generic model of code quality (and based on professional norms), performance in this respect could be tested by using the rubric with professional software projects and comparing to expert assessment. Apart from that, Messick [13] has proposed a more fine-grained perspective on validity of rubrics, and this should be taken into account as the rubric matures. Finally, reliability of the rubric is an especially important evaluative criterion. In contrast to validity, the rubric's interrater agreement can be tested statistically and serve as evidence for the actual performance of the tool, and at the same time point out weak aspects. This can be tested as soon as improved versions of the rubric will be used in class.

7. REFERENCES

[1] Heidi Goodrich Andrade. Teaching with rubrics: the good, the bad, and the ugly. *College Teaching*, 53(1):27–31, 2005.

[2] Katrin Becker. Grading programming assignments using rubrics. *ACM SIGCSE Bulletin*, 35(3):253–253, June 2003.

[3] Veronica Cateté, Erin Snider, and Tiffany Barnes. Developing a rubric for a creative CS Principles lab. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 290–295, New York, NY, USA, 2016. ACM.

[4] Charles R Cooper. Holistic evaluation of writing. In Charles R Cooper and Lee Odell, editors, *Evaluating Writing: Describing, Measuring, Judging*. National Council of Teachers of English, Urbana, Illinois, 1977.

[5] R. Wayne Hamm, Kenneth D. Henderson, Jr., Marilyn L. Repsher, and Kathleen M. Timmer. A tool for program grading: The Jacksonville University scale. *SIGCSE Bulletin*, 15(1):248–252, February 1983.

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

[7] James W Howatt. On criteria for grading student programs. *ACM SIGCSE Bulletin*, 26(3):3–7, 1994.

[8] Anders Jonsson. Rubrics as a way of providing transparency in assessment. *Assessment & Evaluation in Higher Education*, 39(7):840–852, 2014.

[9] Anders Jonsson and Gunilla Svingby. The use of scoring rubrics: reliability, validity and educational consequences. *Educational Research Review*, 2(2):130–144, 2007.

[10] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: the elusive target. *IEEE software*, 13(1):12–21, 1996.

[11] Anastasiya A. Lipnevich, Leigh N. McCallen, Katharine Pace Miles, and Jeffrey K. Smith. Mind the gap! Students' use of exemplars and detailed rubrics as formative assessment. *Instructional Science*, 42(4):539–559, 2014.

[12] Samuel Messick. The interplay of evidence and consequences in the validation of performance assessments. *Educational Researcher*, 23(2):13–23, 1994.

[13] Samuel Messick. Validity of performance assessments. In Gary W Phillips, editor, *Technical Issues in Large-Scale Performance Assessment*. U.S. Department of Education, 1996.

[14] Barbara M Moskal and Jon A Leydens. Scoring rubric development: validity and reliability. *Practical assessment, research & evaluation*, 7(10):1–11, 2000.

[15] Ernesto Panadero and Anders Jonsson. The use of scoring rubrics for formative assessment purposes revisited: a review. *Educational Research Review*, 9(0):129 – 144, 2013.

[16] Tjeerd Plomp and Nienke Nieveen. An introduction to educational design research. In *Proceedings of the Seminar Conducted at the East China Normal University [Z]. Shanghai: SLO-Netherlands Institute for Curriculum Development*, 2007.

[17] Thomas Reeves. Design research from a technology perspective. In Jan van den Akker, Koeno Gravemeijer, Susan McKenney, and Nienke Nieveen, editors, *Educational design research*, pages 52–66. Routledge, 2006.

[18] Eric Roberts, John Lilly, and Bryan Rollins. Using undergraduates as teaching assistants in introductory programming courses: an update on the Stanford experience. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '95, pages 48–52, New York, NY, USA, 1995. ACM.

[19] D. Royce Sadler. The origins and functions of evaluative criteria. *Educational Theory*, 35(3):285–297, 1985.

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

[21] Lon Smith and Jose Cordova. Weighted primary trait analysis for computer program evaluation. *Journal of Computing Sciences in Colleges*, 20(6):14–19, June 2005.

[22] 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.

[23] Briana E. Crotwell Timmerman, Denise C. Strickland, Robert L. Johnson, and John R. Payne. Development of a 'universal' rubric for assessing undergraduates' scientific reasoning skills using scientific writing. *Assessment & Evaluation in Higher Education*, 36(5):509–547, 2011.

[24] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 93–98, New York, NY, USA, 2011. ACM.

[25] Barbara E. Walvoord and Virginia Johnson Anderson. *Effective grading: a tool for learning and assessment in college*. Wiley, 2011.