Understanding code quality for introductory courses

Martijn Stegeman

Faculty of Science, University of Amsterdam
`martijn@stgm.nl`

Abstract. An exploration of the meaning of 'code quality' for introductory programming courses has led to a model that brings together teacher feedback and suggestions from professional handbooks. We translated this model into a rubric for giving feedback to students in their first courses. Using this rubric in teaching has encouraged us to reflect on the *intent* of programming language features and how these can be used appropriately. We now propose future work: how can 'approriate use' be more systematically integrated into the model? Furthermore, we propose that using the derived rubric in teaching may very well confront students with unnoticed misconceptions and help them eliminate these. Is this a reasonable assumption? How can we test it?

Keywords: POP-VI-E. computer science education research, POP-V.B. phenomenology, POP-III-C. procedural/object oriented, POP-I-A. learning to program

Supervisors: Erik Barendsen and Sjaak Smetsers of Radboud University, Nijmegen, Netherlands

1 Theory: rubrics

In the past, teachers have used various tools to systematically assess student work. A very simple case would be a *checklist*, where a number of criteria are listed and can be checked off to indicate performance. *Rubrics* add to this by defining a number of levels of accomplishment, and optionally verbal descriptors to explain each combination of criterion and level (Sadler, 1985).

Checklist and rubrics have often been used as scoring tools, initially to gain reliability in the calculation of grades. In contrast, Andrade (2005) argues for the idea of an *instructional rubric*, primarily designed as a teaching tool instead of a scoring tool:

> "A rubric that is cocreated with students; handed out; used to facilitate peer assessment, self-assessment, and teacher feedback; and only then used to assign grades is an instructional rubric. It is not just about evaluation anymore; it is about teaching. Teaching with rubrics is where it gets good."

Defined as such, a rubric is composed primarily for use by students instead of teachers. By studying that rubric, their own work, and the work of others, students are encouraged to form a conception of expected (and current) quality.

2 Previous work: patterns of feedback in introductory classes

In previously published work, we have examined the feasibility of creating a rubric for introductory programming courses[1]. Our focus was on deriving criteria and descriptors in a systematic fashion by studying the practice of teachers and professional software engineers. We first analyzed standards of code quality embedded in three popular software engineering handbooks and found 401 suggestions that we categorized into twenty topics. We also recorded three instructors who performed a think-aloud judgment of student-submitted programs, and we interviewed them on the topics from the books, leading to 178 statements about code quality.

[1] This paragraph presents work that has been previously published; it is partially based on the abstract of Stegeman, Barendsen, and Smetsers (2014).

The statements from the instructor interviews allowed us to generate a set of topics relevant to their practice of giving feedback, which we used to select criteria for the model. We then used these instructor statements together with the book suggestions to distinguish three levels of achievement for each criterion. This resulted in a total of 9 criteria for code quality. The interviews with the instructors generated a view of code quality that is very comparable to what was found in the handbooks, while the handbooks provide detailed suggestions that make our results richer than previously published grading schemes.

This process has led us to a model of code quality criteria and accompanying levels that we have used to construct a preliminary rubric for introductory programming courses, as included at the end of this paper. This rubric seems to be much more complete and less arbitrary than previously published rubrics or grading schemes for introductory programming courses (Hamm, Henderson, Repsher, & Timmer, 1983; Howatt, 1994; Becker, 2003).

3 Next up: help students understand the role of language elements

When using the rubric with students, we gradually started motivating its contents by explaining that programming languages embody a certain *intent*. This is because programming languages are presumably created by programmers to solve certain problems that pop up when programming. Features in these languages are thus expected to be used in a certain way, and, as time passes, are used to solve other problems as well. In short, it seems that the question that the rubric wants to answer is this: *"How do I appropriately use the tools that the language provides?"* These 'tools', then, are indeed the language features that we find as the nine criteria in the first column of the rubric: comments, decomposition, etcetera. In this light, it becomes clear that the rubric tries to convey a common-sense approach to using such features. This immediately presents us with the question: can we validate or even select the criteria in an even more systematic fashion, by studying the theory and history of programming language design?

3.1 Reframing the model

In the current version of the rubric, all parts, including the language features, have emerged from a single bottom-up coding process. However, by studying programming language publications and history, we should be able to construct an up-front inventory of these features. To make the data analysis more systematic, and possibly to come to a more complete rubric, it would seem appropriate to make this part more theory-based. Doing this can then help us re-focus the data analysis on patterns in the use of programming language features; for example, in the first version of the rubric, we have already seen that for a whole family of formatting features (indentation, white space use), it is usually *consistency* that is asked for. Aren't there many more of these patterns? Using those to select and formulate criteria may help us answer the "How do I appropiately..." question more clearly in the rubric.

3.2 Potential questions

1. Is there existing theory of programming language features?
2. ...that includes stylistic features such as use of whitespace?
3. How do these relate to existing models of code quality?
4. If needed, how can we create a complete overview of programming language features?
5. How is 'appropriate use' defined in literature and/or historical documents?
6. What separates 'appropriate use' from concepts like guidelines, patterns, etc.?
7. How useful is the rubric in follow-up courses that deal with more elaborate use of object-oriented features?

4 Later: confronting and eliminating misconceptions

Apart from teaching students to write ‘good code’, there is another possible motivation for giving systematic feedback on code quality. We think that bad code may well be a sign of remaining problems with learning to program.

4.1 Varying difficulties

Difficulties of learning to program have been studied extensively. Researchers have found, for example, that some students lack the skill to interpret and understand the problem statement (McCracken et al., 2001); some have misconceptions of the notional machine that a programming language represents (Utting et al., 2013); some produce semantically correct fragments but have difficulties combining those (Soloway, 1986); and some lack the skill of mentally simulating the runtime behavior of programs (Lister et al., 2004). In these studies, the authors discovered the difficulties as a result of studying *bugs* in the programs that students produce. However, Du Boulay (1986) described that even without bugs, there may still be difficulties:

> “I have often been surprised at the bizarre theories about how the computer executes programs held even by students who have successfully ‘learned to program’.”

Berges, Mühling, and Hubwieser (2012) studied this phenomenon. The authors asked students to create concept maps as proof of understanding, and compared those with programs the students had written while making use of those same concepts. In several categories, the students appeared to be able to apply concepts of which they showed no clear understanding. More recently, Teague and Lister (2014) found evidence that some students are able to trace programs without being able to explain them.

Something else that may go unnoticed without the presence of bugs is the difficulty with combining correct fragments of code into a working program. Dorn and Guzdial (2010) studied professional web designers. Most developers that needed new information in order to make progress used a process that they describe as *trial-and-error*: writing code, checking the results and finding appropriate information to make the code work. The authors compare this to the opportunistic programming described by Brandt, Guo, Lewenstein, Dontcheva, and Klemmer (2009), where developers construct functional programs while sometimes actively avoiding to learn the complicated syntax for future use.

4.2 Going forward

Students being at least able to construct a working program according to specifications is arguably a fine result of an introductory programming course. However, isn’t it possible to further the understanding of students by providing feedback that addresses the hidden difficulties they may still have? When assuming that low quality code may actually be the result of the hidden difficulties that we discussed, we can speculate that giving feedback on those aspects of code might indeed confront students with the underlying difficulties they are still having, and as such have the potential to encourage learning. One course of action for us would be to use a concept-test and combine this with systematic formative feedback on code quality, in order to find if positive effects can be seen.

4.3 Potential questions

1. Do students ask more conceptual questions when they are systematically given feedback on code quality?
2. Do students perform better on conceptual test when they are systematically given feedback on code quality?
3. Should we use a more qualitative study to gain insight into student understanding in this regard?

5 Conclusion

There is quite a bit of work embedded in the proposed refinement of the method of creating a code quality rubric, as well as in the proposed evaluation of its usefulness in the introductory programmer's learning process. The above is a first try at defining future work and integrating the various potential strands of research. Suggestions on the proposed questions, on related theory and on the proposed methods are very welcome.

References

Andrade, H. G. (2005). Teaching with rubrics: The good, the bad, and the ugly. *College Teaching*, *53*(1), 27-31. Retrieved from `http://www.tandfonline.com/doi/abs/10.3200/CTCH.53.1.27-31` doi: 10.3200/CTCH.53.1.27-31

Becker, K. (2003, June). Grading programming assignments using rubrics. *SIGCSE Bull.*, *35*(3), 253–253. Retrieved from `http://doi.acm.org/10.1145/961290.961613` doi: 10.1145/961290.961613

Berges, M., Mühling, A., & Hubwieser, P. (2012). The gap between knowledge and ability. In *Proceedings of the 12th Koli Calling international conference on computing education research* (pp. 126–134). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2401796.2401812` doi: 10.1145/2401796.2401812

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 1589–1598). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1518701.1518944` doi: 10.1145/1518701.1518944

Dorn, B., & Guzdial, M. (2010). Learning on the job: Characterizing the programming knowledge and learning strategies of web designers. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 703–712). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1753326.1753430` doi: 10.1145/1753326.1753430

Du Boulay, B. (1986). Some difficulties of learning to program. In (Vol. Studying the Novice Programmer). Baywood.

Hamm, R. W., Henderson, K. D., Jr., Repsher, M. L., & Timmer, K. M. (1983, February). A tool for program grading: The Jacksonville University scale. *SIGCSE Bull.*, *15*(1), 248–252. Retrieved from `http://doi.acm.org/10.1145/952978.801059` doi: 10.1145/952978.801059

Howatt, J. W. (1994, September). On criteria for grading student programs. *SIGCSE Bull.*, *26*(3), 3–7. Retrieved from `http://doi.acm.org/10.1145/187387.187389` doi: 10.1145/187387.187389

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., . . . Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. In *Working group reports from iticse on innovation and technology in computer science education* (pp. 119–150). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1044550.1041673` doi: 10.1145/1044550.1041673

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from iticse on innovation and technology in computer science education* (pp. 125–180). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/572133.572137` doi: 10.1145/572133.572137

Sadler, D. R. (1985). The origins and functions of evaluative criteria. *Educational Theory*, *35*(3), 285–297. Retrieved from `http://dx.doi.org/10.1111/j.1741-5446.1985.00285.x` doi: 10.1111/j.1741-5446.1985.00285.x

Soloway, E. (1986, September). Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, *29*(9), 850–858. Retrieved from `http://doi.acm.org/10.1145/6592.6594` doi: 10.1145/6592.6594

Stegeman, M., Barendsen, E., & Smetsers, S. (2014). Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling international conference on computing education research* (pp. 99–108). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2674683.2674702` doi: 10.1145/2674683.2674702

Teague, D., & Lister, R. (2014). Blinded by their plight: Tracing and the preoperational programmer. In *25th Anniversary Psychology of Programming annual conference (PPIG), Brighton, England, 25th-27th June.*

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., … Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports* (pp. 15–32). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2543882.2543884` doi: 10.1145/2543882.2543884

LEVEL	1	2	3	4
		Documentation - is the code well-annotated to ensure rapid understanding?		
names	names appear unreadable, meaningless or misleading	names accurately describe the intent of the code, but can be incomplete, fuzzy, lengthy, misspelled	names accurately describe the intent of the code, and are complete, distinctive, concise, correctly spelled	all names in the program use a consistent vocabulary
headers	headers are missing or descriptions are redundant or obsolete; or use mixed languages	headers summarize the goal of parts of the program and how to use those, but may be incomplete or misspelled	headers accurately summarize the role of parts of the program and how to use those, are spelled correctly, may be wordy	headers contain only essential explanations, information and references
comments	comments are generally missing, redundant or obsolete; or use mixed languages	comments highlight important decisions and potential problems, but may be wordy or misspelled	comments highlight important decisions and potential problems, are concise and spelled correctly	comments are only present where strictly needed
		Presentation - is the code visually organized for a quick read?		
layout	old code is present	arrangement of code within source files is not optimized for readability	arrangement of code within source files is optimized for readability	arrangement of code is consistent between files
formatting	formatting is missing or misleading or lines are too long to read	indentation, line breaks, spacing and brackets highlight the intended structure but erratically	indentation, line breaks, spacing and brackets fully highlight the intended structure	formatting makes differences and similarities clearly visible
		Algorithms - is each part of the code as simple as possible?		
flow	there is deep nesting; code performs more than one task per line; control structures are customized in a misleading way	flow is complex or contains many exceptions; choice of control structures and libraries is inappropriate	flow is simple and contains few exceptions; choice of control structures and libraries is appropriate	flow prominently features the expected path
expressions	expressions are repeated or contain unnamed constants	expressions are complex; data types are inappropriate	expressions are simple; data types are appropriate	expressions are all essential for control flow
		Structure - is the code organized for quick understanding of parts and the whole?		
decomposition	most code is in one or a few big routines; variables are reused for different purposes	most routines are limited in length but mix tasks; routines share many variables; parts of code are repeated	routines perform a limited set of tasks divided into parts; shared variables are limited; code is unique	routines perform a very limited set of tasks and the number of parameters and shared variables is limited
modularization	modules are artificially separated	modules have vague subjects, contain many variables or contain many routines	modules have clearly defined subjects, contain few variables and a limited amount of routines	modules are defined such that communication between them is limited

- highlight features from all levels that are present in the code, starting at the lowest
- for each criterion, circle the level that is most representative of the features that are present
- no need to circle a level that is not relevant to the assignment
- level 2 implies that the features in level 1 are not present, level 4 implies that the features in level 3 are also present

Level 1: problematic features are present
Level 2: core quality goals not yet achieved
Level 3: core quality goals achieved
Level 4: achievement beyond core quality goals