Towards an Empirically Validated Model for Assessment of Code Quality

Martijn Stegeman
University of Amsterdam
martijn@stgm.nl

Erik Barendsen
Radboud University Nijmegen
Open Universiteit
e.barendsen@cs.ru.nl

Sjaak Smetsers
Radboud University Nijmegen
s.smetsers@cs.ru.nl

ABSTRACT

We present a pilot study into developing a model of feedback on code quality in introductory programming courses. To devise such a model, we analyzed professional standards of code quality embedded in three popular software engineering handbooks and found 401 suggestions that we categorized into twenty topics. We 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. 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 create criteria for the model. We used the instructor statements as well as 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. As such, this process leads to an overview of code quality criteria and levels that can be very useful for constructing a standards-based rubric for introductory programming courses.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms

Design, Human factors

Keywords

Programming education, Code quality, Feedback, Assessment, Rubrics, Empirical validation

Koli Calling '14, November 20–23 2014, Koli, Finland.

ACM 978-1-4503-3065-7/14/11 ...$15.00.
http://dx.doi.org/10.1145/2674683.2674702

1. INTRODUCTION

1.1 Code quality

Software quality has been a topic of interest to professional software engineers, software engineering researchers and computer science educators. The notion of quality can be applied to all aspects of creating software, from the software process itself to the requirements.

We consider *code quality* to be an aspect of software quality that can be determined by just looking at the source code, i.e., without any form of testing, or checking against specification. In the software quality model from Boehm et al. [3], this aspect can be identified as the *understandability* characteristic, which itself comprises five sub-characteristics: consistency, self-descriptiveness, structuredness, conciseness, and legibility. Note that this precludes concepts like efficiency, portability, and conformance to a specification. This distinction is comparable to what is called 'internal' versus 'external' quality in professional software engineering [12].

In many programming courses, aspects of code quality are part of the learning goals. For example, students are expected to be able to choose an appropriate type of loop, or to develop a consistent coding style [1]. It seems natural that emphasis on structural quality is present even in early programming courses, because attention to such aspects can lay the groundwork for more advanced treatment in later software engineering courses.

More importantly, introductory courses often aim to present the fundamentals of computer science as a whole: learning a programming language and building software is used to teach students about computing concepts for the first time [1]. Instructors of such courses tend to regard low code quality ('clumsy code') as a possible sign of students' fundamental misconceptions or of an unstructured design process. This provides additional motivation to review novices' code systematically.

1.2 Feedback

Providing feedback is known to be crucial to effective learning in general [6, 7]. In a characterization of useful feedback, Sadler [15] proposed, that 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 [15]. Several studies support the added value of specific feedback [16] that support these student needs.

A number of instructors have published grading schemes used for assessing the quality of student code and provid-

ing feedback. These grading schemes decompose expected performance into several criteria, usually linked to language features designed to support the understandability of code (e.g. comments, modularization).

Instructors from Jacksonville University documented their informal consensus-based process to come to a number of criteria [5]. In their scheme, execution, design and documentation are covered. The criteria are not explicitly defined.

Howatt's [8] grading scheme specifies several levels for a number of criteria, and definitions are not only provided, but also intended to be shared with students. The evolution of some criteria is also documented: 'design' was separated from 'style', as students neglected to devise a plan up-front; 'specification conformance' was separated from 'execution', as students stopped implementing the remaining requirements as soon as they got part of the program working correctly.

Becker's [2] scheme is described in great detail: there are many criteria, and for each of those, three levels of accomplishment have been defined. There is particular attention to specific design choices, such as the use of magic numbers and global variables. Here too, the definitions are shared with students. Smith [17] provides a similarly elaborate scheme.

Comparing feedback practices based on grading schemes is complex. From the grading schemes we found, all but [8] are provided without any systematic justification for the criteria. In practice, it seems reasonable to assume that instructors generally come to the same scores regardless of grading method and philosophy [4]. However, from these schemes we conclude that instructors give feedback at different levels of specificity and have varying expectations of code quality: it is especially notable that no single grading scheme above covers all of the criteria that are used by these instructors. This is why we see an opportunity to create a specification of code quality for introductory courses in a more systematic way.

1.3 Aim of the study

Our goal is to construct a model for feedback on code quality based on empirical data from *professional standards* and *instructional practice*. Such a model is intended to support development and analysis of feedback instruments for code quality. In line with [15], feedback should be provided in two dimensions: *criteria*, i.e., the relevant aspects of code quality, and *achievement levels* for each of these criteria.

This paper describes a first exploration of systematically eliciting criteria and levels that are relevant to code written in introductory programming courses. We characterize such courses by adopting the common topics derived by Tew and Guzdial [19]. These are: fundamentals (variables, assignment, etc.), logical operators, selection statement (if/else), definite loops (for), indefinite loops (while), arrays, function/method parameters, function/method return values, recursion, and object-oriented basics (class definition, method calls).

Our research questions are as follows.

1. What criteria for code quality, relevant to introductory programming courses, are present in professional software engineering literature?

Coding standards in professional literature are not necessarily framed in a way that is useful for introductory programming courses, even when we constrain the selection to strictly relevant topics. Therefore, we will establish the relevant aspects of code quality in teaching practice:

2. What kind of feedback do instructors of introductory programming courses give to students on programming assignments?

Finally, we will combine the insights we gain from answering the above questions to construct a model of feedback for code quality.

3. What levels of code quality can be distinguished in professional standards and instructional practice?

Communicating expectations to students usually requires more than listing criteria and levels [22]. A common approach is to provide students with a *rubric* [21] accompanied by descriptive statements or exemplars. Rubrics can be used to calculate grades, to provide feedback, or both [9]. We intend to use the proposed model to construct such feedback instruments for our own courses, but actual development is beyond the scope of the present study.

2. METHOD

2.1 Analysis of professional handbooks

In order to sample professional standards, we selected a compact set of renowned professional handbooks concerned with code quality. As a first set of candidate titles, we took the top-15 books of the user-generated list of 'programming' books, as published by the Goodreads website[1]. From this list, we selected those books that focus on code quality in general: neither tied to a specific programming language, nor mainly focused on the software engineering process. We discarded books that are primarily concerned with constructs going beyond the usual scope of introductory courses [19], e.g. design patterns or advanced algorithms. The handbooks in the final selection were (A) *The pragmatic programmer* by Andy Hunt and Dave Thomas [20]; (B) *Code complete* by Steve McConnell [12]; and (C) *Clean code* by Robert Martin [11].

Within these books, we selected the normative statements on code quality (to be called 'suggestions' from now on), restricting these to the list of common topics in introductory courses [19] and the sub-characteristics of 'understandability' [3]. We observed that these are defined on an abstract methodological level. To prepare for the construction of a feedback model defined in more concrete, observable terms (like in the grading schemes discussed earlier), we performed a qualitative analysis using an analytic coding [18] procedure. We took the sub-characteristics as initial categories and labeled the suggestions in terms of specific elements or properties of program code. Then we merged similar labels until a set of pairwise disjoint code aspects was obtained. After this reduction, a more in-depth analysis of the suggestions within the resulting code aspects was performed, focusing on content, priorities, and variation.

2.2 Instructor interviews

To elicit instructors' views and feedback practice, we conducted interviews with three experienced Dutch lecturers from different universities, each having at least fifteen years of teaching experience in introductory courses in higher education:

[1]Top-15 programming books, as fetched on February 24, 2014 from `https://www.goodreads.com/shelf/show/programming`.

- Instructor 1 teaches a programming course for science students in general, given in Java.
- Instructor 2 teaches a course for computer science freshmen, also given in Java.
- Instructor 3 teaches a course for applied computer science freshmen, given in C.

We held two rounds of semi-structured interviews: the first consisted of individual sessions with each of the instructors, and the second was a focus group with all three together.

In the first round, we started the interview with a think-aloud feedback session, based on a programming assignment taken from the third week of an introductory course at the first author's university. We selected three different solutions from students who had taken the course. Each instructor was asked to give feedback about the three solutions. In order to capture both the applied criteria and the instructors' reasoning, we asked the instructors to think aloud while articulating their feedback—focusing on characterizing the quality of the code, giving suggestions for improvement, and comparing to feedback they usually give.

Then, instructors were presented with each of the topics we derived from the professional programming handbooks, and asked to respond by giving examples of concrete feedback from their own courses. We did this to make sure our generated view of code quality would become as complete as possible and not be limited by the assignment used.

The second round had a similar structure, but was held as a focus group to benefit from the interaction between the three instructors. In order to discover more advanced criteria, the think-aloud interview in this round was based on the final assignment in a Java course covering the topic lists of [19]; the instructors gave feedback on three student solutions to this assignment. The results of the handbook analysis were discussed for a second time, but this time together.

All sessions were recorded and transcribed verbatim and the data were subjected to a qualitative analysis. In an initial open coding [18] phase, we identified the statements about quality that are observable in program code. Subsequently we performed an analytic coding [18] procedure, summarizing similar statements using labels such as "comments should not repeat the code" or "choosing the right variable scope". Finally, the labels were categorized into more general topics in terms of distinct aspects of code.

2.3 Construction of the model

Here we combined the results from the textbook analysis and the instructor interviews. Figure 1 summarizes the way we derived the model from our analyses. The *criteria* were initially taken from the topics that we generated in the analysis of the instructor interviews. The *levels* for each criterion are based on the corresponding statements on code quality from both interviews and handbooks.

During the instructor interviews, we noted that the instructors tended to give three kinds of feedback, which we used as a preliminary breakdown of levels:

- remarks about serious shortcomings;
- remarks relating code to the expected quality;
- remarks about quality rising beyond expected levels.

This resulted in a possibly partial set of criterion-level combinations. Wherever possible, we added descriptions taken from the in-depth analysis of the handbooks to fill gaps in the model not covered by the instructors' statements.

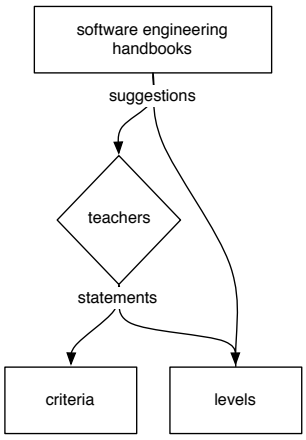

Figure 1: Model generation process

For each criterion, we checked whether the identified levels were mutually *compatible*, i.e., could be expressed in terms of the same sub-criteria. In the case of incompatible levels, the corresponding criterion was split into two (or more) sub-criteria.

3. RESULTS

3.1 Analysis of professional handbooks

Here we describe the results of our analysis of professional handbooks. Selecting suggestions from the three books by using introductory topics [19] and characteristics of understandability [3] resulted in 401 individual items. Labeling all suggestions using the understandability characteristics resulted in the frequencies listed in Table 1.

conciseness	44
consistency	38
legibility	249
self-descriptiveness	13
structuredness	57

Table 1: Frequency of suggestions for each of the characteristics of understandability in software.

Analytic coding of the suggestions within each characteristic resulted in the labels described in Table 2 on the next page, and the categorization of these same labels into a set of code aspects is described in Table 3. In this set, we separated some of the aspects that are independent of the overall code structure (comments, formatting, layout, and naming) and then split higher-level structure (decomposition, modularization) from expressiveness of smaller parts.

The 21 suggestions from The Pragmatic Programmer (A) that we selected, were all either connected to the 'Don't repeat yourself' principle, or suggestions about good documentation and structure. The 213 suggestions from Code Complete (B) and 167 from Clean Code (C) both span most of the criteria that emerged from the inductive grouping; the only exception is that (C) has no suggestions specifically about the appropriate use of idiom.

conciseness	44
dead code	6
duplication	4
fragmentation	2
minimalist comments	32
consistency	38
appropriate idiom	28
formatting consistency	3
naming consistency	7
legibility	249
affinity	3
clarification	30
clear control flow	27
comment content	6
expressive formatting	18
formatting	21
module size	1
naming	108
order	11
routine size	3
type signature	21
self-descriptiveness	13
comment content	13
structuredness	57
abstraction	19
focus	38

Table 2: Frequencies of labels generated during analytic coding

comments	51
comment content	19
minimalist comments	32
formatting	42
expressive formatting	18
formatting	21
formatting consistency	3
layout	20
affinity	3
dead code	6
order	11
names	115
naming	108
naming consistency	7
structure	88
abstraction	19
duplication	4
focus	38
module size	1
routine size	3
type signature	21
fragmentation	2
expressiveness	85
appropriate idiom	28
clarification	30
clear control flow	27

Table 3: Frequencies of labels, re-grouped by code aspect

Below, we describe the in-depth analysis of the handbook contents organized by the aspects from Table 3.

Comments — Suggestions on comments can be separated into two themes: arguments on having as little commenting as possible, and suggestions about appropriate content. A theme that is present in all books is to only comment if strictly needed:

> Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often. (C)

The books use different perspectives for minimizing comments. Code should preferably be self-documenting by use of good names and a clear structure (B,C). Comments can be redundant as they repeat information that can be derived from the code (A,B,C). There is also information that is often listed in header comments but can be usually found in, for example, a source control system: authorship, revision history, the name of the current file, etc. (A,C). The repeated argument for minimizing comments is that comments can get obsolete quickly (B,C):

> The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. The reason is simple. Programmers can't realistically maintain them. (C)

However, there are still categories of information that are complementary to the code, so there are many suggestions for including appropriate information in comments. McConnell summarizes:

> The three kinds of comments that are acceptable for completed code are information that can't be expressed in code, intent comments, and summary comments. (B)

In contrast to this, (C) suggests that even summary comments should hardly be needed, as routines and classes should be of minimal length. However, (B) recognizes the value of such summaries, allowing readers to quickly scan the code. Also, header comments should describe at least how to use routines and classes (A,B).

All books describe small decisions and problems that can be highlighted by using comments, such as exceptions in control flow or significant data type declarations.

Finally, (C) wants comments to be precise and spelled correctly.

Formatting — The first major theme in formatting is consistency. This theme is present in (C) and (B):

> You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. (C)

The second theme is the need for the formatting to mimic the structure of the program:

> The Fundamental Theorem of Formatting says that good visual layout shows the logical structure of a program. (B)

Other suggestions in (C) and (B) link specific types of formatting to that goal: related statements should be grouped and separated by a blank line; indentation should consistently follow the scope of the statements; multi-line statements should be split at a point that makes clear that they are unfinished; white space and parentheses should be used to emphasize the expected evaluation order of expressions; and brackets should be used to emphasize flow control where it is not self-evident.

> Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines. (C)

Finally, line length should normally not be more than 80–100 characters, but long lines should be the exception, and not the rule (B,C).

Layout — For the layout of code in files, we found three themes: the idea of putting related parts close together, the order in which to put parts, and the presence of old code.

Putting related parts together in the code is called 'affinity' by (C). It requires, for example, that related routines are placed close together in a source file. (B) adds that each class should be in a separate file.

Ordering of code in a file can be optimized for readability (C), for example by putting the most used routines at the top and related routines directly below. An alternative is to order routines alphabetically (B). Consistency in a project also plays a role here; for example by always putting variable declarations at the top (C).

> If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use. [...] This makes it easy to find the called functions and greatly enhances the readability of the whole module.

Old code, such as commented-out code or routines that are never called, should always be removed according to (C). (B) adds that variables, even though declared, can remain unused.

Names — The books emphasize three themes for naming: expressiveness, readability and consistency. (C) summarizes the first goal as follows:

> Choosing names that reveal intent can make it much easier to understand and change code.

All books support this idea and provide ways to achieve expressive names. Names should cover the complete abstraction (B,C), be concise (C) and distinctive (B,C). For distinctiveness, many examples are given: the use of very generic words such as `flag` for names is pointed out as potentially problematic (B). The books also suggest what to avoid here: intentional misspellings (B,C) or multiple similar names with number postfixes (B). In contrast to expressiveness, names can be meaningless in many ways:

> Variable names, of course, should be well chosen and meaningful. `foo`, for instance, is meaningless, as is `doit` or `manager` or `stuff`. Hungarian notation (where you encode the variable's type information in the name itself) is utterly inappropriate in object-oriented systems. Remember that you (and others after you) will be reading the code many hundreds of times, but only writing it a few times. Take the time to spell out `connectionPool` instead of `cp`. (A)

Some names can be plainly misleading, for example when abusing generic pre- and postfixes:

> The word "list" means something specific to programmers. If the container holding the accounts is not actually a List, it may lead to false conclusions. (C)

There are also suggestions for the readability of names: for example, having clear word boundaries in names by using underscores or camel casing, using positive boolean names, and using easy to pronounce names (B,C).

Consistency in names is mostly related to vocabulary (B,C). The books suggest that modules have a noun name or noun phrase name in line with platform conventions. There also also names that are common in programming, for example `done`, `error`, `found` and `success` (B).

Structure — For routines, we found that size should mostly be limited (B,C). This is reflected in many suggestions that propose to constrain the focus of each routine. (C) advises to create extremely small routines that are just two, three or four lines long, while (B) speaks of individual routines that could be allowed to grow to a size of 100–200 lines. However, this is deemed an exception. In general, functions should do one thing, as emphasized by (B):

> Functional cohesion is the strongest and best kind of cohesion, occurring when a routine performs one and only one operation.

If the routines still have multiple tasks, (B) argues to separate these into parts as much as possible. This includes variables (B,C).

> As a general rule, the variables you initialize before the loop are the variables you'll manipulate in the housekeeping part of the loop. (B)

(A,C) argue to limit the amount of variables that are shared between routines. This can be contrasted to having not too many parameters, which is also argued by (C). (B), while acknowledging the problems programmers have with parameters, focuses on putting parameters in a natural or consistent order and giving intention-revealing names.

Finally, removing repetition in routines is an explicit goal in all books. Just as with comments, (A) argues against such duplication:

> Avoid similar functions: often you'll come across a set of functions that all look similar — maybe they share common code at the start and end, but each has a different central algorithm. Duplicate code is a symptom of structural problems. (A)

For modules, we find that they should have well-defined subjects (B,C). (C) cites the 'single responsibility principle' to support this. (B) calls it "presenting a consistent level of abstraction." All books argue for defining modules such that communication between them is limited.

At the highest level of design we find a trade-off between keeping modules small and preventing fragmentation of the system as a whole. (C) argues that fragmentation is of lower priority, however. The same trade-off happens between preventing fragmentation of individual modules, thus having many routines, and the preference to keep routines small.

Expressiveness — We found three larger themes: having a simple control flow, using appropriate idiom, and having simple expressions.

Control flow should be kept simple by avoiding nested structures (B,C) and keeping structures like loops short (B,C). Furthermore, (B) advises to feature the nominal path to the code most prominently, for example by always keeping the expected case in a selection statement in the `if` clause and not in the `else` clause. Also, do not use too many `return` and `break` statements to jump out of the normal flow (B). However, (B) says that sometimes, it is actually the right thing to do:

> In certain routines, once you know the answer, you want to return it to the calling routine immediately. If the routine is defined in such a way that it doesn't require any further cleanup once it detects an error, not returning immediately means that you have to write more code. (B)

Finally, both (B) and (C) suggest not to do more than one thing per line, especially if it is considered a side-effect.

For the use of language idiom, only (B) gives many examples of choosing the right structure, and of using structures in a misleading way. For example, when choosing a loop, prefer a `for` loop when it's appropriate, such as when looping over a known range or a certain number of times. Otherwise, use a `while` loop. Also use a `while` loop any time you need to jump out of the middle of the loop (B). Custom use of control structures can be very misleading to other programmers:

> It's bad form to use the value of the loop index after the loop. The terminal value of the loop index varies from language to language and implementation to implementation. The value is different when the loop terminates normally and when it terminates abnormally. Even if you happen to know what the final value is without stopping to think about it, the next person to read the code will probably have to think about it. It's better form and more self-documenting if you assign the final value to a variable at the appropriate point inside the loop. (B)

For expressions, we found three themes: keeping them simple, using the right data types, and naming all constants. In general, for keeping expressions simple, the books suggest performing logical simplifications, using intermediary variables, and making implicit or explicit comparisons (B,C).

> Instead of merely testing a boolean expression, you can assign the expression to a variable that makes the implication of the test unmistakable. (B)

Only (B) provides suggestions for choosing the right data type, and focuses on often-overlooked enums and structures. All three books comment on the use of unnamed constants, or "magic numbers."

> This is probably one of the oldest rules in software development. I remember reading it in the late sixties in introductory COBOL, FORTRAN, and PL/1 manuals. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants. (C)

3.2 Instructor interviews

Below, we describe the results of two rounds of interviews with instructors. We combined the remarks they made during the think-aloud judgment of student submissions with the remarks that were prompted by describing criteria derived from the handbooks. This led to a first set of topics that span code quality in introductory courses.

We labeled the 178 statements about observable code quality from the instructor interviews. Grouping the statements using analytic coding generated eight topics, all supported by a variety of statements from interviews (Table 4).

Documentation	Presentation	Algorithms	Structure
names	layout	flow	decomposition
comments	formatting	expressions	modularization

Table 4: Topics derived from interviews

The frequencies of statements provided by each instructor are listed in Table 5. Of the individual topics, relatively few statements were on 'layout': this aspect was mentioned in only 5 statements by instructors 2 and 3. 'Names' and 'modularization' were represented by 11 statements each, while all other topics were represented in 21 or more statements.

instructor	1	2	3	total
Documentation	13	21	21	48
Presentation	4	16	9	29
Algorithms	26	14	19	59
Structure	20	12	10	42
	63	63	52	

Table 5: Frequencies of coded statements

In terms of differences between the first and second round of interviews, we find that especially instructor 1 made many more statements on 'decomposition' in round two. There were also a few more statements on 'modularization'. In contrast, there were less statements on 'flow', 'formatting' and 'comments'.

Looking at the differences between prompted and spontaneous statements, we find that 'modularization' was only mentioned when prompted. 'Naming' generated more statements when prompted than spontaneously, while all other topics were represented more often in spontaneous statements than in prompted statements.

We will now discuss the statements that the instructors provided for each of the eight topics.

Names — The instructors gave feedback on the appropriate names of routines, modules or variables. Instructor 2 said: "The name of a class should precisely indicate what that thing has or does." There were some negative examples: one instructor found that a longer name did not actually describe what the routine did, and two instructors saw problems with short unexplained variable names.

> One could say "it's a small program and it does not really matter that you violate the rules," but even then I think it would be wise to teach them to pick good names. So now that I think about it, I would probably deduct a point for that[2]. (Instructor 1)

Instructors 1 and 2 also commented on names that they found hard to read:

> I would prefer that variables, if they contain multiple words, use camel casing, so I can easily read them. (Instructor 1)

Comments — All instructors stated that comments should add meaning to the code: instructors 1 and 2 made this explicit, and instructors 2 and 3 gave feedback on redundant comments that repeat what is in the code.

> The comments should be at a higher level of abstraction. You should describe the intent behind what you are doing, not provide a textual version of what follows. (Instructor 1)

[2]All quotes in this section have been translated from Dutch by the authors.

Two instructors expressed problems with having too many comments: instructor 1 and 3 stated that comments in code should often not be necessary, provided that the code is simple. On the other hand, instructors 1 and 2 noted that routines and modules should normally have header comments. Instructors 2 and 3 also gave feedback on some occasions where comments were missing.

Layout — Instructors 2 and 3 both remarked on the presence of old code and the ordering of code in a file. Instructor 2 said that old code would not necessarily result in points deducted, but that it would be commented on. Instructor 2 stated that students having old code happens very often. Instructor 3 noted that one solution had instance variables at the bottom of a module, and instructor 2 concluded that this ordering should be consistently applied.

Formatting — All instructors mentioned some form of formatting. The methods that were discussed: grouping with blank lines (2 and 3), extra brackets (1), indentation (1, 2 and 3), controlling line length (2) and spacing (2 and 3). Two goals for formatting were stated by the instructors: making code structure explicit (all), and emphasizing similarities and differences (2):

> I think that is very nice, very symmetrical. I like symmetric code; if you do similar things they should be similarly formatted. (Instructor 2)

All instructors noted that formatting choices should be consistently applied. Instructor 3 mentioned two cases were he said formatting was misleading.

Flow — The instructors all listed reasons why they thought the flow control could be too complex: deep nesting (1 and 3), choice of control structures (all), performing more than one task per line (1 and 3), having exceptions in the flow (1 and 2), having large blocks of code in a conditional (all), and library use (all). Instructor 3 gave feedback on the misleading use of idiom. Instructor 1 said about the choice of control structure:

> I think it's really neat that he uses enhanced `for` loops. (Instructor 1)

Expressions — All instructors commented on the redundancy of some expressions for loops and conditionals, where the expression partially retests a previous condition. Instructors 1 and 2 also noted various tests that were completely duplicate.

> This is not DRY, as you have the same test: one in the `if` statement, and the other in the `while`. (Instructor 2)

All instructors also commented on using hardcoded literals in expressions (magic numbers). They all said that this should be avoided and replaced by named constants.

Decomposition — The instructors all said that routines should be limited in length and perform a limited amount of tasks.

> Inverting lists and printing them, we don't do that. That is when we need to have a separate function for inverting and a separate function for printing. (Instructor 3)

Instructors 1 and 2 also commented that if a routine performs multiple tasks, they should be clearly separated within the routine. The same instructors gave feedback on having loop counters as class-wide variables.

> That list of instance variables contains `i` and `index`. That is a big problem. It really violates every rule in the book. (Instructor 2)

Modularization — All comments about modularization were made during the discussion of suggestions from the handbooks, and not during the think-aloud judgment. Instructors 1 and 2 emphasized separation of concerns, where instructor 1 stated that "you rarely have too many classes," and instructor 2 said that classes should not be longer that a page of code. They also put a limit on the separation; both stated that classes should not be artificially separated.

3.3 Construction of the model

Here we describe the results of combining the statements made by instructors and the suggestions in the handbooks to create a set of levels for our model.

Formulating criteria and levels.

We based our initial set of eight assessment criteria on the topics generated during the analysis of the instructor interviews. To determine appropriate levels of achievement, we consulted the relevant book suggestions for each topic. Doing so led us to split the 'comments' topic into two, as the book suggestions reinforced the idea from instructors that comment headers and comments in the code have differing goals. As one instructor said:

> If I'm writing a program for solving a quadratic equation then I should put that at the top.

This mandatory aspect of explaining what larger parts of the code do, is also present in the books. This in contrast to comments *in the code*, where instructors and books agree that these should only be included when strictly needed. These incompatible sub-criteria led us to separate 'headers' from 'comments', resulting in a total of 9 criteria:

Documentation	Presentation	Algorithms	Structure
names headers comments	layout formatting	flow expressions	decomposition modularization

Table 6: Final set of criteria.

Notation.

Below, we list the sources we used for the defining the levels in the model. We distiguished the following levels. Ⓔ signifies descriptions of how to reach the *expected quality* of each criterion, elaborating on how to use features of the programming language to help make the code better. Ⓢ signifies descriptions of *serious shortcomings* in code; sometimes this is a violation of simple heuristics (e.g., old code still present), sometimes it indicates non-effort (e.g., meaningless variable names). Finally, Ⓑ signifies descriptions of code that is optimized *beyond* expected quality; reaching this level would involve experimentation and more advanced trade-offs.

Names — Instructors and books emphasize that names should describe the *intent* Ⓔ of the underlying code. Both contrast this to *meaningless* Ⓢ names; for example, one- or two-letter variable names that have no apparent connection to the intended meaning. The books add *misleading* Ⓢ names. Furthermore, the books list ways in which names can be more meaningful in the context of a program: complete, distinctive (supported by one instructor) and concise.

The books mention that names should be readable. One instructor stated a way to achieve this: having clearly separated parts by way of camel-casing. Examples in the books concern other easy to fix problems, such as needless abbreviation or using hard-to-distinguish characters (`11`). We therefore added *unreadable* Ⓢ names as a shortcoming.

In the books, there are many examples of having a *consistent vocabulary* Ⓑ. As this requires experimentation and some experience to get right, we see this as an optimization goal.

Headers — Although the books in general mention that comments should hardly ever be needed, *because* routines should be kept small; as this is a separate goal, we take the position that header comments should usually be present.

Instructors state that routines and modules should normally have header comments. These usually *summarize* Ⓔ the goal of each part of the program and explain parameters, i.e. *how to use* Ⓔ the part. Using correct spelling can be a part of this, as supported by one instructor and one book. Instructors and books both emphasize that *redundant* Ⓢ comments are a problem. One book and instructors noted the goal of *spelling* Ⓔ correctly. The books add that descriptions can be *obsolete* Ⓢ, and instructors noted that headers can be completely *missing* Ⓢ.

The books list several types of comments that are redundant because the information is available outside of the code. Providing only *essential* Ⓑ information can thus be seen as a optimization goal. To allow students to progress from missing headers to only essential information, we added *incomplete* and *wordy* descriptions as intermediary stages.

Comments — The books explain that good comments in the code provide elaboration of *decisions* Ⓔ and *potential problems* Ⓢ. As with header comments, these can be *redundant*, *obsolete* Ⓢ, *missing* Ⓢ or use *mixed languages* Ⓢ.

Instructors and books both state that comments in the code should usually not be necessary, provided that the code is simple. We defined *where strictly needed* Ⓑ as a optimization goal. Between missing comments and only commenting where strictly needed, we added *wordy* and *concise* descriptions as intermediary stages.

Layout — The books suggest that optimizing layout for *readability* is a goal Ⓔ, and specify two aspects of this: grouping code and ordering it. The latter is also supported by two instructors. Together, we call these "arrangement". Doing this *consistently* Ⓑ between files in emphasized by one instructor and by the books.

Two instructors offered that old code was regularly present, but they did not say this was a big problem. Because the books argue that having *old code* Ⓢ is easily avoided, we list this as a problem.

Formatting — In the books as well as in the interviews we found that many syntactic features can be used to make code easier to read. Using indentation, blank lines, spacing and brackets to consistently *highlight* Ⓔ the intended structure of the code are named as most important factors. We defined it as a problem when these are *missing* Ⓢ or plainly *misleading* Ⓢ, as indicated by the instructors.

A more complex goal is highlighting *similarities and differences* Ⓑ between lines of code by formatting those in consistent fashion. One instructor in particular valued this use and the books give several examples, so we include this as a optimization goal.

One instructor commented that *line length* Ⓢ prevented the code to fit on the screen in one case. As the books support this by suggesting that line length be limited, and it is easily controlled, we add it as a problematic feature.

Flow — Instructors and books define *simplifying* Ⓔ and *limiting exceptions* Ⓔ as goals for the the control flow. Choosing *appropriate* Ⓔ control structures and libraries is named by all instructors and in one book.

Instructors have problems with *deep nesting* Ⓢ and performing *more than one task* per line Ⓢ. They and the books also mention cases where customizations of control structures are *misleading* Ⓢ

Prominently featuring the *expected* or nominal Ⓑ path was named by all instructors and as an important topic in one book. Because this requires refactoring we add this as a optimization goal.

Expressions — The books define a goal of having *simple* Ⓔ expressions. Because it is prominently featured, we put it into the rubric. We also adopt choosing *appropriate* Ⓔ data types as a goal from the books. The instructors in particular named *duplication* Ⓢ of (partial) expressions and the use of *unnamed constants* Ⓢ as problems. Having only *essential* Ⓑ expressions, thus not covering a subset of another expression, can be seen as a complex goal, as it requires a good grasp of data types and edge cases.

Decomposition — Instructors and books would like to see a decomposition that results in most routines having a *limited set of tasks* Ⓔ, and *duplication* Ⓔ that is eliminated. Within routines, instructors and books would like different tasks to be separated into *parts* Ⓔ. Because the instructors generally placed a strict limit on tasks, we considered it a problem to put most code in *one or a few routines* Ⓢ.

A related goal named by both instructors and books is having a limited amount shared *variables* Ⓔ, i.e. reducing scope. Reusing variables for multiple purposes is a related violation Ⓢ that was reported by instructors and books.

The associated optimization goal is to have routines perform *very limited* Ⓢ sets of tasks while at the same time limiting the number of shared variables and parameters.

Modularization — Based on the statements from one instructor, and supported by the books, we included having a *clearly defined subject* Ⓔ as a goal for modularization. Elaborating on this, having a limited amount of routines was requested by one instructor and all books, and having a limited amount of variables only by the books. The books in general go beyond this by requiring a more advanced separation of concerns that *minimizes communication* Ⓑ between modules. We included this as a optimization goal. On the other hand, instructors noted that students sometimes perform *artificial separation*, which can be seen as a problem Ⓢ; this is also supported by the books, in arguing to limit the amount of modules.

4. CONCLUSIONS AND DISCUSSION

4.1 Conclusions

Our research questions were as follows:

1. What criteria for code quality, relevant to introductory programming courses, are present in professional software engineering literature?
2. What kind of feedback do instructors of introductory programming courses give to students on programming assignments?
3. What levels of code quality can be distinguished in professional standards and instructional practice?

As to Question 1, using the chosen sample of books as our research material, we found that selecting only suggestions relevant to introductory programming courses still led to a broad and richly described view of code quality: in our inductive analysis we found 20 topics, each having many related suggestions.

As to Question 2, our interviews with instructors provided us with a variety of statements on code quality. By offering the instructors our selection of topics from handbooks, we made sure the results were not limited to the instructors' usual way of giving feedback. Our subsequent analysis generated eight topics focusing on different aspects of program code. Notably, most of the statements about 'modularization' were collected only when instructors were prompted with the topics from the book analysis.

As to Question 3, from our analysis of the empirical data a classification emerged consisting of nine criteria, each having many sub-criteria that are divided into three levels.

4.2 Reflection on the methods used

We discuss the present study in terms of the framework for validity of qualitative research by Lincoln and Guba [10].

Supporting the *credibility* of this research is the prolonged engagement of the interviewer within the domain of introductory programming courses courses. The interviewer has taught several introductory programming courses using a variety of open educational resources, based on different views of code quality. This allowed the interviewer to keep an open mind. The results of interviews and book analysis have also thoroughly been discussed with a peer not directly involved in teaching introductory programming courses. Finally, combining book suggestions to the model derived from the interviews allowed us to critically review interpretations done during the analysis of the interviews.

Our selection of data sources influences the *transferability* of the results. We have based our handbook analysis in this pilot on three popular programming books that all focus on code quality. It does not seem likely that our selection of criteria would have been broader if we had selected more handbooks on code quality: all introductory topics from [19] are covered by the suggestions. It does seem relevant to include introductory programming textbooks, to provide even richer descriptions. We did severely constrain our selection of applicable books and suggestions by limiting these to common introductory topics [19] and by excluding handbooks that cover a specific programming language. This means that we have not considered some specific topics that come up when using object-oriented, functional or logic languages. We see extending the scope of this study as future work, but instructors should be able to accommodate specific requirements in one of the criteria or in an additional specification.

The sample of instructors in the pilot study was relatively small. However, their teaching experience spans a variety of courses at considerably different institutions. In spite of these different backgrounds they pointed to very similar features of code when giving feedback. In addition, there were striking similarities between the way instructors talked about code quality and the way the handbooks treat the subject. Instructors regularly recognized topics that are proposed by the books and were able to provide examples, and the handbooks contain many specific suggestions that elaborate on criteria that emerged from the interviews. It seems appropriate, nevertheless, to incorporate more instructors in a follow-up study. Furthermore, it should be interesting to elicit feedback data from instructors involved in advanced courses on software engineering. Finally, although the content and form of the Dutch courses taught by our instructors appears to comply with international curriculum standards [1], including instructors from the international community could enrich the data even more.

There are some issues with the *dependability* of this study. The qualitative analysis of the data was mainly done by the first author of this paper. Possible problematic categorizations were discussed with the other authors until a consensus was reached. Although we expect the reliability of our selection and analysis to be reasonable, there is an opportunity to perform systematic reliability testing and refinement of the codes used in the follow-up to this pilot study.

We have taken measures that contribute to the *confirmability* of this study. All interviews have been transcribed verbatim, and all book suggestions have been archived in full, both stored in combination with applied codes.

4.3 Comparison to earlier schemes

It is interesting to investigate how our results relate to existing grading schemes. The instructors' statements in this study contain a view of code quality that spans the topics that are often present in previously published grading schemes, although none of the previous schemes contain all of the criteria we discovered. As such, we see our study as an important step in creating a more generally usable model for feedback.

The suggestions derived from the handbooks also provide more detail than is available in the earlier grading schemes. We have found, for example, completeness, distinctiveness, conciseness and consistency of names; the highlighting of important decisions in comments, in addition to summarizing; having lines that are too long; arrangement of code in files; spelling errors in comments; misleading customization of control structures; having simple expressions; having no duplicate parts of code. Generally, these sub-criteria add detail to the existing grading schemes, which may provide the student with hints on how to improve the code. We have already seen that learning how to improve is a fundamental part of learning from feedback [15] and that feedback specificity is linked to better learning in general [16].

We did find an aspect of code quality that our data has not accounted for: Becker [2] assesses the presence of initialization for new variables. This is a requirement that does not make sense in languages that always provide clean memory for variables, which is probably why it is not treated

	1	2	3	4
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

Figure 2: Example row of a rubric constructed from the data in this study.

in the handbooks we studied. Such a criterion could be embedded in a customized rubric when used with languages that need this, although we would suggest using a separate, course-specific rubric or checklist.

4.4 Future work

It appears that an empirical analysis of the kind described here has not been done before. The richness of our results and the consistency of the findings from our two empirical sources suggest that our method is fruitful. We intend to extend the pilot by incorporating a bigger and more diverse sample of data sources (handbooks, textbooks, instructors) and by considering the need for extension beyond the topics covered by introductory courses.

Even after gathering the data in a more rigorous fashion, we can only really evaluate the usefulness of our research by putting it into practice for giving feedback. To understand the feasibility of constructing a rubric from the gathered data, we employed some simple rules to derive a prototype version (Figure 2). We defined four levels of achievement, a recommended number to start with for a rubric to be used by many instructors [21]. We put the problematic Ⓢ features in the level 1, the optimization goals Ⓑ in level 4, and the core goals Ⓔ in levels 2 (goal not yet reached) and 3 (goal reached). For each level, we wrote *verbal descriptors* to help students understand what is expected from them. We have already noticed that this requires some interpolation between levels and more importantly, devising formulations that do justice to the progression we expect students to make. Doing this systematically could be a research project in itself.

Evaluating such a rubric in an educational setting typically requires an evaluation of the reliability of the rubric by checking consistency between graders [13]. On top of that, validity can be considered, in particular the suitability for student use of the verbal descriptors [14]. We intend to start evaluation of a fully specified rubric in 2015.

5. REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer science curricula 2013. Technical report, ACM Press and IEEE Computer Society Press, December 2013.

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

[3] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd. International Conference on Software Engineering*, ICSE '76, pages 592–605. IEEE Computer Society Press, 1976.

[4] Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. What are we thinking when we grade programs? In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 471–476. ACM, 2013.

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

[6] John Hattie. What is the nature of evidence that makes a difference to learning?, 2005. `http://research.acer.edu.au/research_conference_2005/7`.

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

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

[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] Yvonna S. Lincoln and Egon G. Guba. *Naturalistic Inquiry*. Sage Publications, Inc., 1985.

[11] Robert C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[12] Steve McConnell. *Code complete*. O'Reilly Media, Inc., 2004.

[13] Barbara M. Moskal and Jon A. Leydens. Scoring rubric development: Validity and reliability. *Practical Assessment, Research & Evaluation*, 7(10):1–11, 2000.

[14] Y. Malini Reddy and Heidi Andrade. A review of rubric use in higher education. *Assessment & Evaluation in Higher Education*, 35(4):435–448, 2010.

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

[16] Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[17] 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.

[18] Anselm Strauss and Juliet M. Corbin. *Basics of qualitative research: Grounded theory procedures and techniques*. Sage Publications, Inc., 1990.

[19] Allison Elliott Tew and Mark Guzdial. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 97–101. ACM, 2010.

[20] Dave Thomas and Andy Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

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

[22] Dylan Wiliam. What is assessment for learning? *Studies in Educational Evaluation*, 37(1):3 – 14, March 2011.