An Evaluation Instrument for Object-Oriented Example Programs for Novices

Jürgen Börstler, Marie Nordström, Lena Kallin Westin,
Jan Erik Moström, Henrik B. Christensen, Jens Bennedsen

UMINF-08.09
ISSN-0348-0542

UMEÅ UNIVERSITY
Department of Computing Science
SE-901 87 UMEÅ, SWEDEN

An Evaluation Instrument for Object-Oriented Example Programs for Novices

Jürgen Börstler, Marie Nordström,
Lena Kallin Westin, Jan Erik Moström
Dept. of Computing Science
Umeå University, Sweden
{jubo,marie,kallin,jem@cs.umu.se}

Henrik B. Christensen
Dept. of Computer Science
University of Aarhus, Denmark
{hbc@daimi.au.dk}

Jens Bennedsen
IT University West
Aarhus, Denmark
{jbb@it-vest.dk}

UMINF 08.09

Abstract

Research shows that examples play an important role for cognitive skill acquisition. Students as well as teachers rank examples as important resources for learning to program.

Students use examples as templates for their work. Therefore examples must be consistent with the principles and rules of the topics we are teaching; free of any undesirable properties or behaviour. By repeatedly exposing students to "exemplary" examples, desirable properties are reinforced many times. Students will eventually recognize patterns of "good" design, gaining experience in telling desirable from undesirable properties.

We therefore need to be able to tell apart "good" and "bad" examples. To do so objectively and consistently requires some kind of measurement instrument.

In this paper, we describe the development and initial validation of an evaluation instrument for example programs, based on three aspects of quality; technical quality, object-oriented quality, and didactical quality. Validation was performed by six experienced educators using examples from popular introductory programming textbooks.

Results show that the instrument helps indicating particular strengths and weaknesses of example programs. In all but one example program, we identified aspects that need to be improved. However, inter-rater agreement is low. The instrument can therefore not be considered reliable enough for evaluations on a larger scale. Further work is necessary to fine-tune the instrument and provide guidance for its usage.

Contents

1 Introduction

Examples play an important role in teaching and learning programming, students as well as teachers cite examples as the most helpful resource for learning to program [19]. This is also supported by research in cognitive science which confirms that "examples appear to play a central role in the early phases of cognitive skill acquisition" [30]. Research in cognitive load theory furthermore shows that an alternation between worked examples and problems increase learning outcome [28].

Examples work as role models; students use examples as templates for their own work. Examples must therefore be consistent with the principles and rules being taught and not exhibit any undesirable properties or behavior. In other words, all examples should follow the very same principles, guidelines, and rules we expect our students to eventually learn. If our examples do not do so consistently, students will have a difficult time recognizing patterns and telling an example's superficial surface properties from those that are structurally or conceptually important. I.e., it is important to present examples in a way that conveys their "message", but at the same time be aware of what learners might actually see in an example [23].

Trafton and Reiser [29] note that in complex problem spaces (like programming), "[l]earners may learn more by solving problems with the guidance of some examples than solving more problems without the guidance of examples". By continuously exposing students to "exemplary" examples important properties are reinforced. Students will eventually gain enough experience to recognize general patterns which helps them telling apart "good" and "bad" designs.

With carefully developed examples, we can reduce misinterpretations and prevent students from drawing inappropriate or undesired conclusions, e.g., by overgeneralization. This helps preventing misconceptions, which might hinder students in their further learning [8, 16].

Although examples are perceived as one of the most important tools for the teaching and learning of programming [19], there is very little research regarding their properties and usage. There is a large body of knowledge on program comprehension (e.g., [4, 5]), software quality and measurement (e.g., [25]), but this is rarely applied in an educational setting [2, 20].

In this report, we describe the development of an evaluation instrument that tries to capture technical, object-oriented and didactical qualities of an example. To validate the instrument, a group of CS teachers has reviewed a number of carefully chosen examples. Results show that the instrument helps indicating particular strengths and weaknesses of example programs. Results show furthermore that most example programs had weak spots that need to be improved. However, inter-rater agreement is low. The instrument can therefore not be considered reliable enough for evaluations on a larger scale. Further work is necessary to fine-tune the instrument and provide guidance for its usage.

2 Related Work

Textbooks are an important component in teaching introductory programming. They are a major source for example programs and also work as a reference for how to solve specific problems. Although examples play an important role in

the teaching and learning of programming, there are no systematic evaluations of textbook examples.

De Raadt et al. [10] compared 40 introductory programming textbooks by measuring their amount of content particularly relevant for the textbooks usefulness as a learning tool, like the number of pages covering examples, exercises, bibliographies, appendices, language reference, index, glossary, and other chapter content. Of the 14 books covering object-oriented programming (in Java), examples covered from 0% to 43% of the content. The authors conclude that

> Examples should concisely illustrate a technique. They should include line numbers for reference, though should preferably be as self-contained as possible, not requiring the reader to keep referring back to the accompanying text discussion. Better examples will often include the author's comments maybe accompanied with some lines and arrows like the typical classroom blackboard example.

Wu et al. [32] address the question of how examples are used and presented in high school textbooks (using BASIC as a programming language). They categorize *Presentation style* based on the presence or absence of four major problem-solving steps; problem analysis, solution planning, coding and testing/debugging. An analysis of 16 textbooks, in total 967 examples, shows that only 2% of the examples display a thorough analysis of the problem statement (only 3 of 16 books). And, even worse, only 0.2% of the examples discuss how the programs can be tested and debugged.

Malan and Halland [21] discussed the potential harm "bad" example programs might do when learning object-oriented programming. They identified four main problem areas, based on their own experiences as teachers; examples are either too abstract or too complex and examples are applying concepts inconsistently or even undermining the concept(s) being introduced.

An important aspect of examples is *misconceptions* and how to avoid them. Holland et al. [17] outlined a number of student problems and how they might relate to properties in example programs, like object/class conflation, objects as simple records, and reference vs. object. Along with each problem they provide a pedagogical suggestion for avoiding potential misconceptions by choosing suitable examples.

A similar problem is noted by Fleury [12], who discussed how students constructed their own rules by misapplying correct rules. She described certain cases how these *student-constructed rules* can systematically lead students to incorrect solutions.

In 2001, a discussion on '`HelloWorld`'-type examples was initiated in Communications of the ACM [31] which created a series of follow-ups on the object-orientedness of common introductory programming examples [6, 7, 11, 24]. Surprisingly, the main discussion has been on how to adjust the '`HelloWorld`'-example to better fulfill the characteristics of object-orientation, not on whether this is a good example at all.

Negative implications of oversimplified examples are discussed from other aspects as well. Hu [18], for example, discusses the general problem of data-less objects (which '`HelloWorld`' is an instance of). Using data-less objects is contradictory to the basic notion of objects when introducing them to novices. He calls them merely containers holding (static) methods and simulating a procedural way of programming.

3 Desirable Properties of Example Programs

In the literature the terms 'example' and 'example program' are often used interchangeably but can mean quite different things. In the context of the present work, we define *example program* and *good example* as follows.

Example Program. A complete description of a program, i.e., the actual source code plus all supporting explanations related to this particular program. This is opposed to code snippets. For brevity, we usually omit 'program', when this is implicit from the context.

Good Example. An example that relates to the needs of novices in programming and object-orientation.

Novices have no or only a limited frame of reference. They rarely have any prerequisite knowledge about (object-oriented) programming languages or the programming process. It is therefore important to carefully control cognitive load [9] and strip examples from unnecessary complexity.

- Example context should be as familiar as possible.
- Examples must be small and/or focus on a single concept.
- Object-orientated principles must be upheld and enforced.

This leads to somewhat contradictory demands on examples. Upholding object-oriented principles does not necessary lead to easy, straightforward solutions to small problems. In fact, the strength of object-orientation is in managing complexity. Kristen Nygaard, one of the originators of object-orientation, often stressed that object-orientation is a better problem solving tool for complex problems than for simple ones. During the work presented here it also became obvious that there were aspects of examples that might be rated good from one perspective, but bad from another.

In our opinion, an object-oriented example program must fulfill certain basic properties to be effective as an educational tool. It must

- be technically correct,
- be a valid role model for an object-oriented program,
- be easy to understand (readable),
- promote "object-oriented thinking", and
- emphasize programming as a problem solving process.

However, how these characteristics should be implemented is not easily agreed upon. The educational quality of an example is not a definite, measurable quantity. Although some quality aspects, like code complexity, are quantifiable, it is very difficult to capture didactical and contextual properties [2]. This becomes apparent from the debates around the common '`HelloWorld`' example [6, 7, 11, 18, 31], where object-orientedness is often argued about in technical terms, i.e. concrete syntax. From a didactical point of view, it might be argued that none of the proposed versions of '`HelloWorld`' could serve as a good example, since the example as such seems artificial and not serving any "real" purpose. Which kind of application would need objects of this kind? Does it exemplify any important or general concept, besides concrete syntax? Even if the example fulfilled the criteria for good examples in a technical sense, its didactical qualities are still questionable.

On a more general basis there are guidelines for object-oriented software. Code-smells [13], heuristics [26] and principles [22] all focus on large-scale software systems and can not easily be adopted for the educational situation.

4 Method

The project was carried by two Scandinavian research groups, one from the University of Umeå, Sweden, and the other from the University of Aarhus, Denmark. The groups are both actively involved in teaching introductory programming as well as research in computer science education.

The project was formally initiated by a two day workshop in Umeå in August 2007, but initial discussions and work regarding the need for evaluation techniques had commenced earlier, see [2]. The workshop started with a brainstorming session on desirable properties of object-oriented example programs for novices (see Appendix A). The results from this session were discussed and eventually lead to the definition of an initial checklist, as outlined in Section 4.2. Six representative examples were selected from a number of popular textbooks. Examples were of varying levels of complexity and chosen to address different object-oriented concepts, as described in Section 4.1.

During September 2007 the groups worked on testing the evaluation instrument. Each example was graded by six reviewers using the evaluation instrument. During October the two groups held local meetings as well as corresponded by e-mail, discussing experiences with the evaluation instrument. The following general observations were made:

1. Several quality factors could be interpreted differently by different reviewers. Effort was invested to clarify the statements describing each quality factor to define them more precisely.
2. Several quality factors were found to be highly dependent on each other. This was considered problematic, as this would lead to double penalties or rewards for a low or high score, respectively. While this cannot be avoided completely, effort was invested to minimize this problem.
3. Differences in reviewer background could also affect the grading. For instance, one of the reviewers taught primarily advanced programming courses in recent years. This lead to a higher attention to aspects less relevant for examples target towards novices. These issues were clarified and focus on novice examples were emphasized to lessen the impact of reviewer background on the evaluations.
4. One of the books was removed from the list as it appeared to be more aimed at non-novice programmers, which meant the the set of examples was down to five examples in four books.

These discussions lead to several changes to the evaluation instrument giving it a clearer structure and more concise formulations leading to a better consensus about evaluation criteria among the evaluators. During November 2007, the six reviewers performed a second evaluation of the remaining five examples once again, using the revised instrument described in Section 4.2.

4.1 Reviewer Demographics and Examples Considered

The instrument, described Section 4.2, has been used by six reviewers each reviewing five examples. Two of the reviewers are women and four men, the range of age is between 37 and 48. All reviewers are experienced computer science lecturers. Furthermore, all have been working with courses in object-oriented programming at an introductory level. However, three of them have been primarily teaching more advanced courses in recent years. Four of the reviewers work at the Department of Computing Science at Umeå University and two work at the Department of Computer Science at the University of Aarhus.

We searched a large range of currently popular textbooks with different focus and didactical approaches (OO-first vs. OO-late; process-focus vs. programming-mechanics-focus, concrete-syntax-focus vs. general-concept-focus, etc.) for examples to test the instrument. To cover a wide range of styles, complexity, and concepts, we selected five representative examples covering the following aspects; the very first example of a textbook, the first exemplification of developing a user-defined class, the first application involving at least two interacting classes and a non-trivial (but still simple) example of using inheritance.

E1: First example with more than one "user defined" classes. An example consisting of a class representing a display for a digital clock. Uses non-primitive attributes of the class NumberDisplay in a ClockDisplay class, Pages 52-70, Code 3.3-3.4 in Barnes, D.J. and Kölling, M.: *Objects First with Java*, Prentice Hall, 3rd edition, 511 pages, 2006.

E2: The very first textbook example. An example adding two numbers and displaying the result using console I/O. Implemented by extending the ACM library's `ConsoleProgram`. Pages 30-34, figure 2.2 in Roberts, E.S.: *The Art & Science of Java—An Introduction to Computer Science*, Addison-Wesley, 1st edition, 587 pages, 2008. The book is based on the library developed by the ACM Java Task Force [27].

E3: First example of developing a "user defined" class. An example with a single class representing a student. No other classes using it. Pages 190-198, figure 6.5 in Roberts, E.S.: *The Art & Science of Java—An Introduction to Computer Science*, Addison-Wesley, 1st edition, 587 pages, 2008.

E4: First example of developing an "user defined" class. An example consisting of a class encapsulating die objects and text-based output for demo purposes in a `main`-method (implemented in a separate application class). Pages 189-196, listning 4.1-4.2 in Lewis, J. and Loftus, W.: *Java Software Solutions*, Addison-Wesley, 5th edition, 780 pages, 2007.

E5: First example concerning inheritance and polymorphism. An example that creates a very simple cartoon-like animation. Two kinds of smileys (with different behavior) are displayed in a frame. Pages 113-121, Listing 3.1-3.5 in Sanders, K.E. and van Dam, A.: *Object-Oriented Programming in Java—A Graphical Approach*, Addison-Wesley, Preliminary edition, 638 pages, 2006

Two of the books described above (examples E1 and E4) have been used as course literature by some of the reviewers. The main features of our five examples are summarized in Table 1.

Table 1: Main features of evaluated example programs.

	First example	First user-defined class	Several classes	Attributes of class-type	Inheritance	Polymorphism
E1	—	—	X	X	—	—
E2	X	—	—	—	X	—
E3	—	X	—	(X)[a]	—	—
E4	—	X	X	(X)[b]	—	—
E5	—	—	X	X	X	X

[a] `String` only
[b] In "test"-class only

4.2 The Instrument

Our initial workshop contained a brainstorming phase that resulted in an initial checklist of example characteristics that should be examined (see Appendix A for the full list). This checklist contained both concrete characteristics (like "Are provided libraries (re)used consistently") as well as suggestions for relevant metrics that could be collected (like "Classify objects used as immutable/mutable, stateless/statefull, all attributes are primitive, etc."). Like on other occasions (see [3]) it was quite easy to agree on desirable example characteristics, but much harder to reach consensus about their interpretation and utilization.

4.2.1 Quality Categories

The onset of the instrument was clearly to formulate our intuitive feelings for "god" and "bad" into something more precise and measurable. However, the first test of the checklist on an actual textbook example demonstrated that more structure and a clearer definition of evaluation criteria was needed to support more consistent evaluations. Inspiration was taken from the checklist-based evaluation by the Benchmarks for Science literacy[1], and it was decided to structure the checklist acoording to three categories evolved; *technical quality*, *object-oriented quality* and *didactical quality*.

Technical quality. A good example must be technically correct. Code must be written in a consistent and exemplary fashion, i.e., appropriate naming, indenting, commenting and so on. This is not primarily connected to the object-oriented paradigm, but programming in general.

Object-oriented quality. A good example must be a valid role model for an object-oriented program. Object-oriented concepts and principles should be emphasized and reinforced. It is therefore important to show objects with mutable state, meaningful behavior, and communication with other objects.

[1] `http://www.project2061.org/publications/bsl/default.htm`

Didactical quality. From a didactical point of view, a good example must be easy to understand, promote object-oriented thinking and emphasize programming as a problem solving process.

4.2.2 Quality Factors

To develop an operational checklist, we defined a set of quality factors, hereafter denoted QF's, corresponding to desirable example properties for each of the categories above. To reach well-defined and unambiguous QFs, we strived for the following properties:

1. Each QF is based on accepted principles, guidelines, and rules.
2. Each QF is easy to understand.
3. Each QF is easy to evaluate on a Likert-type scale.
4. There are as few as possible QFs.
5. There is as little redundancy as possible.
6. All QFs are at a comparable level of granularity/importance, i.e., they contribute equally well to the overall "quality" of an example.
7. The set of QFs covers all relevant aspects of "quality".
8. The set of QFs must be applicable to any example, regardless of pedagogical approach and order of presentation.

In the following, we present the final list of QFs, as used in our evaluation, grouped by category. An example of a filled-in review form can be found in Appendix B and the results of our evaluation are presented in Section 5. Each QF is formulated as a statement that can be evaluated on a five point Likert-like scale (1=strongly disagree, 2=disagree, 3=neutral, 4=agree, 5=strongly agree).

General technical quality (T1-T3)

T1: Problem versus implementation. The code is appropriate for the purpose/problem (remember: the solution need not be object-oriented, if the purpose/problem does not suggest it).

T2: Content. The code is bug-free and adheres to general coding guidelines and rules. E.g., all semantic information is explicit: for instance if preconditions and/or invariants are used, they must be stated explicitly; if there are dependencies to other classes, they must be stated explicitly, etc., testing preconditions is a client responsibility, objects are constructed in valid states, no code duplication, flexibility, etc.

T3: Style. The code is easy to read and written in a consistent style. E.g., well-defined intuitive identifiers, useful (strategic) comments only, consistent naming, and indenting, etc.

Regarding T1 it is important that students see solutions that actually solve the stated problem. An example will score low in case the program does something different than what is stated as its purpose or does much more. T2 and T3 focus on the actual example code disregarding whether or not it solves the problem it is intended for. T3 is focused on readability and a low score is given for misleading identifier names, inconsistent indenting, etc. To get high scores examples should have useful strategic comments, i.e., comments that add information instead of just repeating existing information. Excessive commenting

works against readability and will therefore lead to lower scores. T2 focuses on methodological rather than syntactic consistency. An examples of an inconsistency would for instance be a class with four primitive attributes where the constructor initializes only two of them which would make students wonder what happens to the other two. Another example for devaluating T2's scores would be, if the same code fragment appears several times without being extracted into a method.

Object-oriented quality (O1-O2)

O1: Modeling. The example emphasizes object-oriented modeling. E.g., emphasizes the notion of object-oriented programs as collections of communicating objects.

O2: Style. The code adheres to accepted object-oriented design principles. E.g., proper encapsulation/information hiding, Law of Demeter (no inappropriate intimacy), no subclassing for parameterization, etc.

The difference between O1 and O2 is that O1 focuses on design while O2 is concerned with the implementation. An example may very well score low on O1, but still be implemented according to object-oriented principles. For example, classes being mere containers with primitive attributes and set/get-methods only can still be graded high on O2, if attributes are private and manipulation is done through appropriate methods. Another example may be rated good from a design point of view, but score low on implementation if basic principles are violated. The infamous '`HelloWorld`'-example can do well on O2 but still be a bad abstraction.

Didactic quality (D1-D6)

D1: Sense of purpose. Students can relate to the example's domain and computer programming seems a relevant approach to solve the problem.

D2: Process. An appropriate programming process is followed/described, i.e., the problem is stated and analyzed, a solution is designed, implemented and tested/debugged.

D3: Breadth. The example is focused on a small coherent set of new issues/-topics. It is not overloaded with new material or details introduced "on the fly".

D4: Detail. The example is at a suitable level of abstraction for a student at the expected level and likely understandable by such a student.

D5: Visuals. The explanation is supported by relevant and meaningful visuals.

D6: Prevent misconceptions. The example illustrates (reinforces) fundamental object-oriented concepts/issues. Precautions are taken to prevent students from drawing inappropriate conclusions.

The didactical qualities are somewhat difficult to evaluate without taking into consideration the larger context of the example, like the textbook's intended audience, pedagogical approach (objects-first/late etc.), usage of projects or running examples and so on. However, in an evaluation like the one intended here, one cannot take care of all these aspects. To some extent, examples must have a quality of there own.

Most textbooks explicitly state their target audience. Highly technical or mathematical examples will get low scores in textbooks for a general audience. Reliance on local concepts like time/date formats or measurement systems will also score low as will concepts or terms that are difficult to understand for non-native speakers of the language.

When evaluating process (D2), consistency of the pedagogical approach (top-down, objects-first, etc.) can also be weighed in. Consistency is crucial and the evaluation should have this in mind. However, evaluators must keep in mind that the back-cover texts of textbooks are advertisements and not necessarily correct descriptions of the pedagogical approach actually followed by a textbook.

D3 is concerned with the focus of the example. As a general rule, only one new concept should be introduced at a time. This makes it easier for students to concentrate on the relevant aspects regarding this particular concept. Making things more complex than necessary will likely introduce unnecessary cognitive load. Showing three different ways to print a value, while introducing the different ways to communicate with an object, would for example result in a low score.

D4 is to some extent dependent on the order of which conceptual elements are introduced. Using API-classes that implement interfaces (e.g., Collections) might be to abstract if used too early.

Visuals are not always necessary, but often useful for explanations. However bad visuals can be as bad as no visuals at all, so D5 is somewhat difficult to evaluate. It is important to be aware of the relative interpretation of this quality factor. Reviewers might very well judge the use of visuals differently. An example may score high without visuals, if there is no real need for them. On the other hand, overuse of visuals or inappropriate visuals might make it more difficult to understand a concept and can therefore result in a lower score. It is therefore important to agree on an evaluation policy beforehand.

Preventing misconceptions is extremely important, but limited time and space tends to invite teachers/authors to do potentially dangerous shortcuts. A well designed example class can be of great instructional value, but by adding a main-method that self-instantiates the class we run the risk of seriously confusing the novice programmer. Another example is the use of printing for debugging which often leads novices to believe that printing is the only means for "getting things" out of a method. D6 is responsible for capturing these kinds of problems, but it is still the reviewers responsibility to consider implications of different elements of the example.

5 Results

In this section, we summarize the results from six reviewers, R1–R6, using the evaluation instrument described in the previous section on the examples E1–E5 described in Section 4.1.

Figure 1 shows the overall average grades and the average standard deviations for each example. For each of the QFs an average and a standard deviation among the reviewers are calculated. Then the overall average grades and the average the standard deviations are calculated. Another way to calculate the overall average grade would be to first calculate an average for each category and then an overall average. Doing this do, however, not change the relations

between the five examples.

As can be seen, E1 receives the highest average grade and also has the lowest average standard deviation. E2 and E5 receive the lowest average grades but have the largest average standard deviations. It seems that reviewers are more likely to agree on high rankings than on low rankings. This pattern seems to hold also when looking at quality categories (see Figure 2) and individual QFs (see Figure 12 in Appendix C).

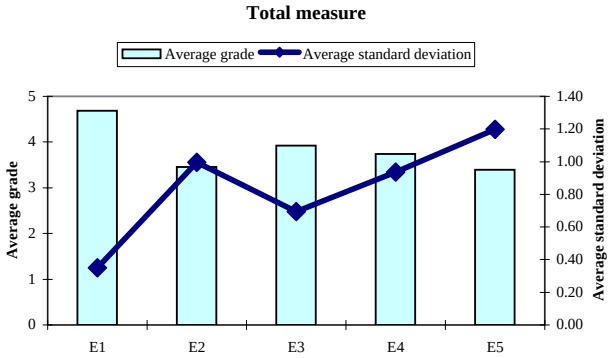

Figure 1: Overall average grade (bars) and average standard deviation (line) for example E1–E5.

Figure 2 shows that the overall gradings of our test examples was quite high for technical quality, consistently better than neutral for didactical quality and quite inconsistent for object-oriented quality. Figure 3 shows that technical quality (T1–T3) also had the smallest average standard deviation, i.e., reviewers were more likely to agree on example grades in this category. Regarding QF D3 (Breadth), reviewers seemed to agree the least.

Inconsistent gradings might be caused by reviewers who consistently give higher or lower grades than others. Figure 4 shows the distribution of grades for all reviewers. It can be noted that all reviewers prefer gradings 4 and 5; 5s are actually used for 40% or more of the gradings for all but one reviewer (R5). However, only one reviewer, R6, does not give a single 1 in the reviews. Detailed data about all gradings can be found in Appendix C.

When comparing the reviewers' implicit internal ranking of the examples (see Figure 5), it can be seen that all six reviewers rank E1 as the "best" example and most reviewers rank E3 second. Figure 5 also shows that the rankings for an example can differ considerably in the different categories, i.e., it seems that the categories capture different aspects of quality.

6 Discussion

In the following subsections, we will discuss several questions regarding the usefulness of the evaluation instrument and the validity of our results.

- How well do the results presented in Section 5 correspond to the reviewers intuitive perception of example quality? Is the example that attains the highest grade (E1) also the best one according to our "gut feeling" (see Section 6.1)?

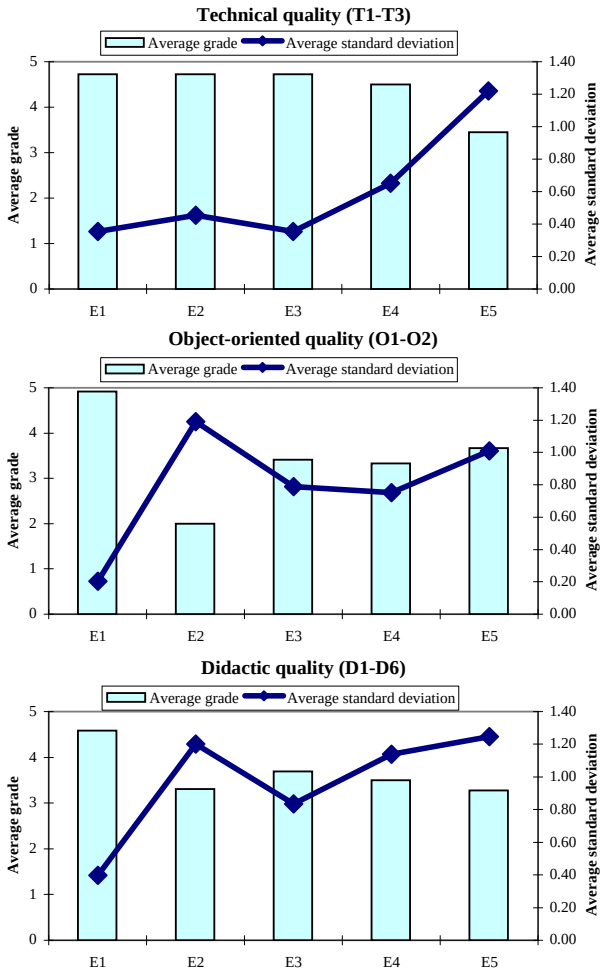

Figure 2: Average grade (bars) and average standard deviation (line) for example E1–E5 for each quality category.

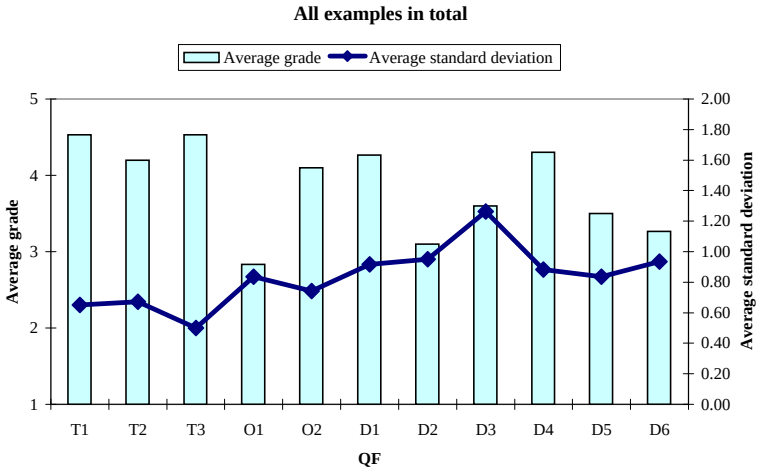

Figure 3: Average grade and average standard deviation for all QFs.

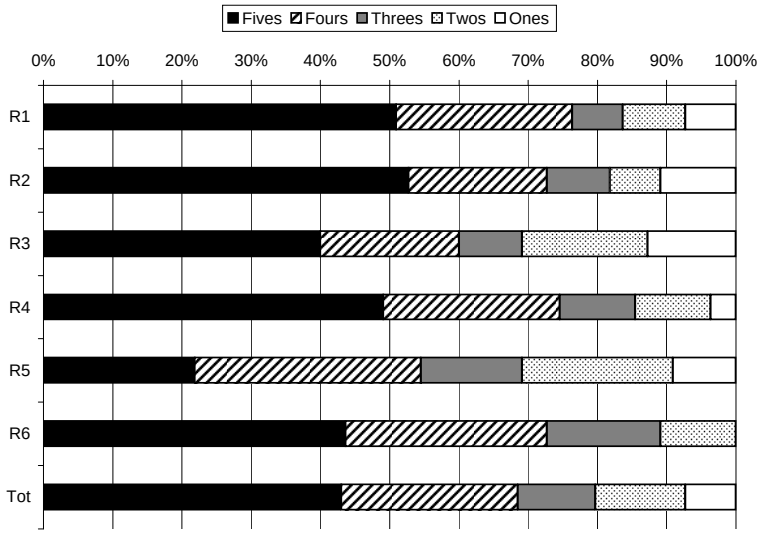

Figure 4: The distribution of grades given by the six reviewers.

	Ranking only T-qualities					Ranking only OO-qualities					Ranking only D-qualities				
	E1	E2	E3	E4	E5	E1	E2	E3	E4	E5	E1	E2	E3	E4	E5
R1	4	1	1	1	5	1	5	4	3	2	2	5	4	1	3
R2	1	1	4	1	5	1	5	3	2	3	1	4	2	3	5
R3	1	1	3	4	5	1	5	2	3	3	1	2	3	5	4
R4	1	1	4	1	5	1	4	4	3	2	1	5	2	4	3
R5	1	3	1	3	5	1	5	2	3	3	1	3	4	2	4
R6	3	2	1	3	5	1	4	2	5	2	1	3	5	3	2
Overall ranking	2	1	4	3	5	1	5	3	4	2	1	5	3	2	4

	Total ranking of examples				
	E1	E2	E3	E4	E5
R1	1	5	3	1	4
R2	1	4	2	2	5
R3	1	2	2	4	5
R4	1	5	2	2	4
R5	1	4	2	2	5
R6	1	4	3	5	2
Overall ranking	1	4	2	3	5

Figure 5: Implicit ranking of the examples based on the reviewers' individual scores. The number 1 denotes the example which the reviewer has given the highest average grade and 5 is given to the example with the lowest average grade.

- Can the grading be considered valid despite the sometimes rather drastic disagreements among the reviewers? Are there specific explanations for the disagreement? For example, are some QFs[2] more difficult to interpret and to evaluate (see Section 6.2)?
- How can the grades for the various QFs be weighed together? Should all QFs have equal weight? How do different weighing schemes affect the total grade (see Section 6.3)?
- Does a textbook's approach to examples affect the evaluation somehow? Does the order or kind of presentation affect example quality as reflected in our instrument (see Section 6.4)?
- How well-defined are the QFs? Do they fulfill the properties we defined in Section 4.2.2 (see Section 6.5)?
- Are there any limitations to the applicability of our instrument (see Section 6.6)?

6.1 Agreement with Gut Feeling

Example quality is difficult to capture. Experienced teachers develop a sense for "good" and "bad" and "know it when they see it". They intuitively recognize examples that exhibit desirable properties and to the point convey "the message" of the example in a clear and unambiguous way. It is therefore important to know, to which extent the instrument captures this intuitive sense of quality.

Among the examples evaluated, example E1 receives the highest average grade (4.68) together with the lowest average standard deviation (0.35). E1 receives the highest average grades of all example programs in all QFs, except T2, T3, and D3 (see Figure 11 in Appendix C). It is furthermore ranked first

[2]Quality Factors T1–T3, O1–O2, D1–D6

in total by all reviewers (see Figure 5). Our discussions after the evaluation revealed that this is consistent with the "gut feeling" of all the reviewers. The example is considered of high technical quality, demonstrates many desirable object-oriented qualities (like mutable state, utilization of multiple instances of a class, attributes of class-type, etc.), and is generally presented and explained in a clear and understandable fashion.

The lowest grade is given to example E5 (avg grade: 3.39, avg std.dev: 1.20) closely followed by example E2 (avg grade: 3.45, avg std.dev: 1.00). The reason for their low grading is quite different though. Example E5 is ranked last in technical quality by all reviewers, while example E2 is ranked last in object-oriented quality by all but one reviewer. On average, example E2 is also ranked quite low in didactical quality (actually last in total, see Figure 5). Example E5 contains a defect and the resulting program does not fulfill the requirements stated in the textbook (the example actually works, but not as described in the text). Since textbook examples are carefully developed and tested, one would expect very high technical grades for technical quality on average. This is also clearly reflected by our results (see Figure 3). The low overall ranking of example E5 is also consistent with the reviewers' "gut feeling". Defects in textbook examples are not acceptable. Example E2 is the very first example in an "objects-late" textbook and not really focused on object-orientation. Its low object-oriented quality could therefore be "excused". However, its low overall ranking in didactical quality was more of a concern, even considering the reviewers "gut feeling".

In general, we can say that the implicit rankings, calculated from the reviewers' grades (see Figure 5), correspond quite well to our "gut feeling" which puts example E1 first, examples E2 and E5 last, and examples E3 and E4 somewhere in between.

6.2 Reviewer Disagreement

Although all criteria were discussed thoroughly and the instrument was tested twice, we underestimated the semantical issues concerning the criteria. It was implicitly assumed that all reviewers shared the same interpretation of each single QF. However, careful inspection of all evaluations revealed that grading still was difficult in some cases. Several of the QFs received rather different grades. Already when developing the instrument, a recurring topic of discussion was when to give a 1 or a 5, i.e., to get a common understanding of the extremes for each criterion. During these discussions, examples were often used to illustrate these extremes. Despite these discussions, reviewers utilized the rating scale quite differently (see Figure 4). However, utilization of the rating scale might be attributed to differences in personality, since all reviewers gave grades deviating negatively and positively from the majority (see Table 2). Furthermore, not all reviewers were actively involved in all discussions. Therefore, some differences should be expected.

It should also be noted that different people will very likely assess things differently, even if they are trained assessors. Expecting reviewers to fully agree is unrealistic. A study on essay assessment in Danish primary school, for example, "found remarkable interscorer variation on both the categories and the frequency of criteria used, as well as on the marks assigned to the essays" Gregersen [15, p. 30].

Table 2: Number of occasions where a reviewer's grade was the opposite of the majority's grade (see Figure 11 in Appendix C for details).

Reviewer	R1	R2	R4	R3	R5	R6
Better than majority	4	1	2	4	1	2
Worse than majority	3	4	9	4	7	2

We therefore strongly recommend supplying a written instruction, containing prototypical examples, with the instrument. Nevertheless, there are only four occurrences in two of the 5 examples, where reviewers used the full scale (1–5 inclusive); example E2 (QFs O2 and D3) and example E5 (QFs O1 and D1). In D1 (Sense of purpose) there are two explanations of what the QF embodies (Students can relate to the example's domain and computer programming seems a relevant approach to solve the problem), but some reviewers only considered the first part.

Some reviewers found QF O2 difficult to assess meaningfully in cases where O1 was low, i.e., when an example was not considered object-oriented. Although QF O1 and O2 were intended to be independent (see Section 4.2.2), this was not found particularly intuitive; lack of object-orientedness should result in low object-oriented quality.

Some reviewers gave high "grades" to non-object-oriented examples. This means that examples that were actually not considered object-oriented could increase their average rating for object-oriented quality. Discussions after the assessment showed that some reviewers actually followed their intuition and others strictly followed the (non-intuitive) definition. This explains the large standard deviation for QF O2 in example E2 (see Figure 12 in Appendix C).

Conditional QFs might be a solution to this problem. First the general object-orientedness of an example is evaluated. Only if the example exceeds a certain threshold or fulfills certain criteria, further QFs are evaluated.

Teaching experience can also influence the grading. Two reviewers have exclusively (R5) or mainly (R3) taught advanced programming courses over the last years, which might explain their high numbers of opposing grades (see Table 2) and slightly lower grades on average (see Figure 4).

Overall, our actual grades as well as our "gut feeling" tell us that agreement of what is "good" is better than of what is "bad". To some extent this can be explained by the reviewers spotting different problems (see for example Figure 9 in Appendix B. But this might also happen to the same reviewer making reviews at different times. Since the same examples were re-evaluated with two versions of the instrument, some reviewers found more problems and/or had more or more specific critique the second time. I.e., more careful evaluation might have lead to worse grades for the examples that have been evaluated most carefully.

6.3 Weights and Classification of Quality Factors (QFs)

It might be tempting to compute a single value for the grading of an example. We have tried to balance criteria within a category. However, an overall total (over all categories) does not seem very meaningful. Granularity and impact of criteria can never be perfectly balanced—is for example the importance of visuals (D5) as important as to prevent misconceptions (D6)? Probably not—

and even within a category, criteria will be valued differently by different people. Moreover, categories with many criteria might be overemphasized.

Another question is classification of QFs. QF D6 (Prevent misconceptions) implicitly emphasizes object-orientation, since this is the application domain for the evaluation instrument. It might therefore fit better in the category "object-oriented quality" rather than "didactic quality". In Figure 6 and Figure 7, we show the effects of re-categorization of QF D6 as belonging to the object-oriented quality category instead of the didactical quality category.

Including D6 in the object-oriented quality category has some effect of the average grade, it makes E3-E5 more equal in average grade. However, the average standard deviation for E3 and E4 increases. Excluding D6 from the didactical quality category has only minor effects, probably due to the fact that this category had the most QFs associated with it and still have five QFs.

We can see that the ranking of the examples changes both when looking at the total ranking of examples as well as looking at the ranking of the object-oriented quality and didactical quality respectively. As an example, E2 and E4 goes from being the lowest ranked examples to be ranked in the middle of the examples when concerning didactical quality when D6 is removed from that category.

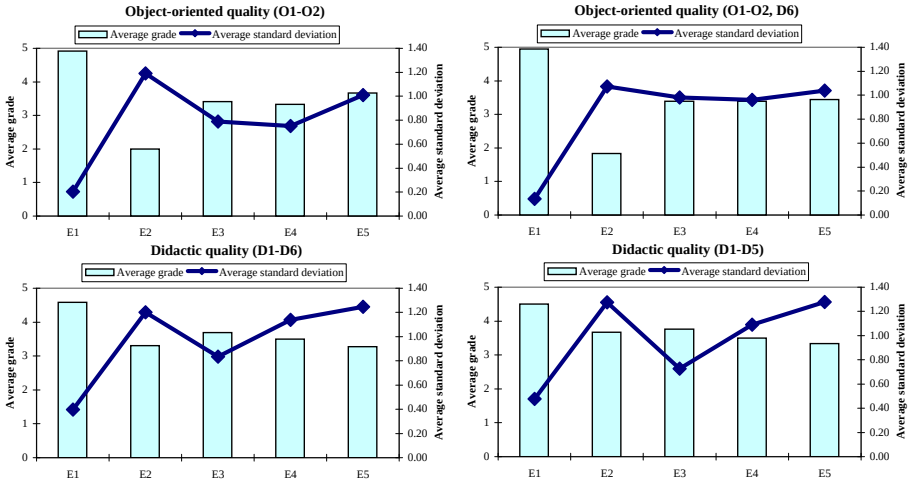

Figure 6: Effects of categorization of QF D6 as belonging to the object-oriented quality category (to the right) as compared to categorization of D6 as belonging to the didactical quality category (to the left) on average grades and average standard deviations for the two quality categories.

6.4 Order of Presentation

Textbooks often follow certain patterns for introducing new material, like starting with finished program code that is "dissected" for explanation or explaining the basic elements before composing them into a program.

Furthermore, complex examples can be presented in two major ways; starting with either the "whole" or the "parts". This may (perhaps wrongly) induce a sense of process: a presentation modeled over a part-whole order brings a sense

	Total ranking of examples					Ranking only OO-qualities					Ranking only D-qualities				
	E1	E2	E3	E4	E5	E1	E2	E3	E4	E5	E1	E2	E3	E4	E5
R1	1	5	3	1	4	1	5	4	3	2	2	5	4	1	3
R2	1	4	2	2	5	1	5	3	2	3	1	4	2	3	5
R3	1	2	2	4	5	1	5	2	3	3	1	2	3	5	4
R4	1	5	2	2	4	1	4	4	3	2	1	5	2	4	3
R5	1	4	2	2	5	1	5	2	3	3	1	3	4	2	4
R6	1	4	3	5	2	1	4	2	5	2	1	3	5	3	2
Overall ranking	1	4	2	3	5	1	5	3	4	2	1	5	3	2	4

	Total ranking of examples, moving D6 to OO-qualities					Ranking only OO-qualities, including D6					Ranking only D-qualities, excluding D6				
	E1	E2	E3	E4	E5	E1	E2	E3	E4	E5	E1	E2	E3	E4	E5
R1	1	5	4	2	3	1	5	4	2	2	2	5	4	1	2
R2	1	4	3	2	5	1	5	3	2	4	1	2	3	4	5
R3	1	3	2	4	5	1	5	2	3	3	1	2	3	5	4
R4	1	5	3	2	4	1	5	4	3	2	1	5	2	4	3
R5	1	4	2	3	5	1	5	2	2	4	1	2	4	2	4
R6	1	4	2	5	3	1	4	3	5	2	1	4	5	3	2
Overall ranking	1	4	2	3	4	1	5	4	2	2	1	3	5	2	3

Figure 7: The resulting ranking when including D6 in the object-oriented quality category instead of the didactical quality category.

of bottom-up programming while a whole-part order points towards top-down programming.

Describing and showing the process is important and can be done in different ways. Our instrument captures process in a single general QF (D2). It does not take side for one type of process or presentation or the other.

6.5 Properties of QFs

In Section 4.2, a number of desired properties are stated for the quality factors. In this section, we discuss to what extent they have achieved in the current version of the instrument.

1. **Each QF is based on accepted principles, guidelines, and rules.** Agreement on desirable characteristics was high among the developers.
2. **Each QF is easy to understand.** As discussed in Subsections 5–6.3, definition and interpretation of QFs was sometimes ambiguous.
3. **Each QF is easy to evaluate on a Likert-type scale.** This worked well for most of the QFs. Maybe the most difficult QF is D5 (Visuals). What constitutes "relevant and meaningful visuals" is difficult to grade objectively. Different reviewers have different opinions of when and how visuals can or should be used to improve an explanation. Adding visuals might actually impair an otherwise excellent explanation by causing higher cognitive load. I.e., even examples without any visuals at all could get high grades for visuals (in case there are no relevant and meaningful visuals that could be added without negative side-effects).
4. **There are as few as possible QFs.** At present the instrument consists of eleven QFs, and this seem to be a reasonable number for the practical use of the instrument. Using fewer QFs would probably make the single QFs

too course grained.

5. **There is as little redundancy as possible.** There is some redundancy and this seem to be unavoidable. QF D6 (Prevent misconceptions), for example, is closely related to object-orientedness and it might be argued that this is already covered in category object-oriented quality (see discussion in Subsection 6.3). On the other hand, all QFs have strong didactical implications.
6. **All QFs are at a comparable level of granularity/importance.** This goal could not be achieved and will probably never be. Importance depends on many factors, for example usage, reviewer experience and preference etc. Currently, not even the number of factors are the same within each category. This makes the calculation of an overall grade troublesome. However, each QF could be easily supported by a weight that could be adjusted according to reviewer preferences.
7. **The set of QFs covers all relevant aspects of "quality".** Taking into account the CS1-oriented perspective, we have found that the basic characteristics required from an object-oriented example are captured by our list of QFs. Our evaluation instrument provides an additional field for "things that were not fully covered by the checklist". This field was, however, rarely used by the reviewers. In the few cases it was used, it was used to emphasize a particularly high or low grade.
8. **The set of QFs must be possible to apply to any example.** As discussed in Subsection 6.2, the reviewers had some problems applying QF O2 (Style) to examples that were not considered object-oriented. Despite that all QFs were applicable to all tested examples. Since our examples were chosen from representative textbooks following different approaches, we think there should not be any problems with applicability.

6.6 Validity/Instrument design

In the design of the instrument we did not consider specific pedagogical approaches that textbooks might take regarding object-orientation (objects-first vs. objects-late etc.) or example presentation (running examples vs. projects etc.). The instrument is designed primarily to evaluate examples, not textbooks. It should therefore be possible to evaluate an individual example given a suitable description/context. The focus of the instrument is on object-oriented quality and we claim that an example must exhibit high quality independent of the teaching approach. Textbook examples are often (re)used out of context or adapted to new contexts; it is therefore important that a good example remains good, even when used outside its original context (assuming that this context is removed carefully).

An important design-descision was to keep the instrument small and easy to use. This meant, among other things, to keep the number of QFs small. One consequences of this is that the QFs are rather coarse-grained. Fine-grained checklists are less manageable and it is likely that not all aspects will be applicable to all examples.

It might be argued that we need to lower the bar, since "real-world" applications and programs do not primarily exhibit the kind of properties targeted by this instrument. Degenerated classes and methods are actually very prevalent

in industrial software, see [14] for a thorough study. Are we not supposed to teach the students the real thing? Keeping in mind the CS1-context, i.e., evaluating examples or novices, it is important to focus on the most central issues of object-orientation. It is our firm belief that novices must learn the basics correctly, before being able to understand and use exceptions from the general principles and rules. This point of view is grounded in a large body of work from the learning sciences, as described in the Introduction of this report. The main question this instrument should answer is therefore "Is this example suitable for a novice?".

It is worth noting that despite several meetings and general consensus on what constitutes a good example, there are considerable differences in the reviews, see see Figure 11 in Appendix C. Although reviewer R3 participated only in our initial meeting, R3 has about the same amount of opposing grades than the other reviewers (see Table 2). When looking at the frequency of opposing grades per QF, we can note that reviewer agreement looks much better when single outliers[3] are excluded, see Table 3.

Table 3: Number of times a QF included conflicting grades (see Figure 11 in Appendix C for the full details).

QF	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
Total	1	1	1	3	1	2	3	4	3	3	3
W/o single outliers	0	1	0	1	1	1	0	2	1	0	3

Generally it can be noted that the agreement of what is "good" is better than of what is "bad". Our data is, however, somewhat contaminated by the fact that the same examples were evaluated twice; before and after the refinement of the instrument and that the same reviewers did both reviews.

7 Conclusions

In this report, we have described the design and test of an instrument for evaluating object-oriented example programs for educational purposes. The design resulted in three categories of qualities, *Technical quality*, *OO-quality* and *Didactical quality*. Each quality containing a varying number of quality factors.

The instrument was tested by six experienced educators on five examples. The examples were chosen to be representative of a wide range of examples from introductory programming textbooks.

Our results show that such an instrument is a useful tool for indicating particular strengths and weaknesses of example programs. Although only five examples from introductory programming textbooks were formally evaluated, the results indicate that there might be large variations regarding object-oriented and didactic quality of textbook examples. Since example programs play an important role in learning to program, it would be valuable to formally evaluate textbook examples at a larger scale.

Note that the same example may score differently depending on the reviewer as well as the intended audience. This means that the important result of an evaluation are not the absolute numbers, but the possibility of comparisons

[3] A QF where only a single grade is opposing the majority.

between examples. Computing a total score for an example is of less interest than examining the scores in the three quality categories separately. They can aid in showing the qualities an example promotes.

The evaluation instrument presented here can not, at this state, be considered reliable enough for evaluations on a larger scale; inter-rater agreement is too low. As discussed in Section 6, this problem can be reduced by revising the quality factors to avoid misunderstandings and, most importantly, developing detailed rating instructions. Several revisions and testing would be necessary before a large scale evaluation can take place.

Finally, we believe that an instrument of the kind presented here, is useful for several reasons. For an individual, it is a way of learning about the qualities example programs must hold to be successful learning and teaching tools. It helps in the construction of examples for educational purposes. For the teaching community, it can be a useful tool for evaluating larger bodies of example programs.

Acknowledgments

The authors would like to thank Michael E Caspersen for fruitful discussions during construction and validation of the evaluation instrument.

References

[1] D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[2] J. Börstler, M. Caspersen, and M. Nordström. Beauty and the beast—toward a measurement framework for example program quality. Technical Report UMINF-07.23, Dept. of Computing Science, Umeå University, Umeå, Sweden, 2007.

[3] J. Börstler and I. Hadar. Eleventh workshop on pedagogies and tools for the teaching and learning of object oriented concepts. In *ECOOP 2007 Workshop Reader*, volume LNCS 4906 of *Lecture Notes in Computer Science*, pages 182–192. Springer, 2008.

[4] R. Brooks. Towards a theory of the comprehension of computer programs. *Intl. Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[5] J. Burkhardt, F. Détienne, and S. Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[6] CACM Forum. 'Hello, World' gets mixed greetings. *Communications of the ACM*, 45(2):11–15, 2002.

[7] CACM Forum. For programmers, objects are not the only tools. *Communications of the ACM*, 48(4):11–12, 2005.

[8] M. Clancey. Misconceptions and attitudes that infere with learning to program. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 85–100. Taylor & Francis, Lisse, The Netherlands, 2004.

[9] R. Clark, F. Nguyen, and J. Sweller. *Efficiency in Learning, Evidence-Based Guidelines to Manage Cognitive Load*. Wiley & Sons, San Francisco, CA, USA, 2006.

[10] M. de Raadt, R. Watson, and M. Toleman. Textbooks: Under inspection. Technical report, University of Southern Queensland, Department of Maths and Computing, Toowoomba, Australia, 2005.

[11] M. H. Dodani. Hello World! goodbye skills! *Journal of Object Technology*, 2(1):23–28, 2003.

[12] A. E. Fleury. Programming in Java: Student-constructed rules. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 197–201, 2000.

[13] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] J. Y. Gil and I. Maman. Micro patterns in java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 97–116, San Diego, CA, USA, October 2005.

[15] J. Gregersen. Marking strategies in denmark with special reference to strategies in the danish comprehensive school. *Studies in Educational Evaluation*, 13(1):21–34, 1987.

[16] M. Guzdial. Centralized mindset: A student problem with object-oriented programming. In *Proceedings of the 26th Technical Symposium on Computer Science Education*, pages 182–185, 1995.

[17] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. In *Proceedings of the 28th Technical Symposium on Computer Science Education*, pages 131–134, 1997.

[18] C. Hu. Dataless objects considered harmful. *Communications of the ACM*, 48(2):99–101, 2005.

[19] E. Lahtinen, K. Ala-Mutka, and H. Järvinen. A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 14–18, 2005.

[20] K. Magel. A Theory of Small Program Complexity. *ACM SIGPLAN Notices*, 17(3):37–45, 1982.

[21] K. Malan and K. Halland. Examples that can do harm in learning programming. In *Companion to the 19th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–87, 2004.

[22] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Addison-Wesley, Reading, MA, 2003.

[23] J. Mason and D. Pimm. Generic Examples: Seeing the General in the Particular. *Educational Studies in Mathematics*, 15(3):277–289, 1984.

[24] N. Ourosoff. Primitive types in Java considered harmful. *Communications of the ACM*, 45(8):105–106, 2002.

[25] S. Purao and V. Vaishnavi. Product metrics for object-oriented systems. *ACM Computing Surveys*, 35(2):191–221, 2003.

[26] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, MA, 1996.

[27] E. Roberts, K. Bruce, R. Cutler, S. Grissom, K. Klee, S. Rodger, F. Trees, I. Utting, and F. Yellin. The ACM Java task force: Final report. *ACM SIGCSE Bulletin*, 38(1):131–132, 2006.

[28] J. Sweller and G. Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2:59–89, 1995.

[29] J. G. Trafton and B. J. Reiser. Studying examples and solving problems: Contributions to skill acquisition. Technical report, Naval HCI Research Lab, Washington, DC, USA, 1993.

[30] K. VanLehn. Cognitive skill acquisition. *Annual Review of Psychology*, 47:513–539, 1996.

[31] R. Westfall. 'Hello, World' considered harmful. *Communications of the ACM*, 44(10):129–130, 2001.

[32] C.-C. Wu, J. M.-C. Lin, and K.-Y. Lin. A content analysis of programming examples in high school computer textbooks in taiwan. *Journal of Computers in Mathematics and Science Teaching*, 18(3):225–244, 1999.

A Appendix: Brainstorming Results

The brainstorming session resulted in a large number of desirable properties for "good" object-oriented examples. Below we present the initial list of suggestions, that finally lead to the definition of three quality categories[4] with eleven quality factors[5] used for the instrument.

Suggested items in checklist

- Delimit the examples to those chapters of the book that deal with the 8 quarks ([1])
  - Inheritance
  - Object
  - Class
  - Encapsulation
  - Method
  - Message passing
  - Polymorphism
  - Abstraction (modeling sense)
- Preamble for each example:
  - Where? (book, page, figure, table, you know!)
  - What? (domain, purpose)
- Is there a sense of purpose to the topic of the example (beyond syntax)?
  - Does it make sense to the target audience?
  - Is the domain well-understood?
  - Does the math put people off?
  - Is the problem artificial / 'constructed'
  - Is example domain consistent with intended audience? (for CS, biologists, physicists, ...)
- Is there a problem description?
- Is there an analysis?
- Is there a design description?
- Is there a explanation of the implementation?
- Is visual notation used to illustrate problem/design/implementation?
- List the 'forward references' to unexplained thingys
- List the new concepts/thingys introduced by the example
- List involved classes, classify as Java library, author lib, part of example.
- Check consistency in:
  - Naming
  - Coding style (indentation, etc.)
  - Design

[4] *Technical quality*, *Object-oriented quality* and *Didactical quality*
[5] QF: T1-T3, O1-O2, D1-D6

  - Refactoring ("before" and "after" code) [anti-example: empty default constructor in one class, not in another (circle example)]

- Are pre/post conditions used?
  - Check that methods obey stated pre/post conditions
  - List failures
- Are 'problems' in the example (perhaps due to concepts/constructs not introduced yet) mentioned in the explanation? (Problems in a technical/design related aspect) List them [Tick if they are indeed explained] Examples:
  - math example: use enum instead of char operation,
  - use exception throwing instead of returning -1
  - account with debt initialized to balance = 0
  - program statements without any purpose (circle class empty constructor)
  - constructor does not fully construct object (ex: invoke 'init' after constructing)
- Are 'problems' in early example used as motivation for introducing a better solution?
- Is naming intuitive and not misleading (example: count in variable for summations, 'rad' in circle != radians)? List non-intuitive examples from the examples
- Are comments sufficient and meaningful? [Is there a difference between style in book and on download src.] Guideline: Strategic commenting in the book, JavaDoc commenting in download src.
- Are objects/classes used that are consistent with problem domain? (anti-example: use circles for the washers that are really cylinders)
  - List inconsistent usage.
- Are provided libraries (re)used consistently (emphasis reuse instead of writing from scratch) Avoid duplicated code throughout examples. (read-Double probably copy-pasted reused multiple times)
- Classify object with respect to property X and see if it is used properly, where X is:
  - Mutable/immutable
  - Stateless/Statefull
  - Attributes are all primitive
  - Struct (only set/get) / functional objects /
  - behavioral objects ('meaningful' behavior wrt. problem domain)
  - Interacts with other objects ('community of agents')
  - Demonstrated usefulness of making multiple instances
  - List exceptions
- Classify/count aspects of inheritance usage:
  - Only add attributes/get/set methods in subclass
  - Polymorphically override 'interesting' behavior

  - Just add behavior (not set/get)
  - Pure subtyping (only override existing methods, no added methods)
  - Superclass is interface/abstract/concrete
  - Inheritance used instead of simple parameterization
  - Downcasting
- Count examples that are revisited later

B Appendix: Example Assessments

OO Program Example checklist

By ***example*** we refer to a complete description of an example program, i.e., the ***actual source code*** plus all ***supporting explanations*** related to this particular example.

Reviewer:	R6
Textbook:	…
Page/Figure no:	E4
Brief characterization/description of the example (purpose, revisited/revised version?):	
Write two classes, a class `Die` and a "runnerclass" `RollingDice`.	

General technical quality of code (T1-T3) **A good example's code must be technically sound and easy to read**	**Answer**[1]
1. **Problem versus implementation.** The code is appropriate for the purpose/problem (remember: the solution need not be OO, if the purpose/problem does not suggest it).	**5**
Simulation seems appropriate	
2. **Content**. The code is bug-free and follows general coding guidelines and rules. E.g., all semantic information is explicit (e.g., if preconditions and/or invariants are used, they must be stated explicitly; if there are dependencies to other classes, they must be stated explicitly, etc.), testing preconditions is a client responsibility, objects are constructed in valid states, no code duplication, flexibility, etc.	**4**
A die object has `faceValue` 1 when created – not part of the declaration.	
3. **Style.** The code is easy to read and written in a consistent style. E.g., well-defined intuitive identifiers, useful (strategic) comments only, consistent naming, commenting, and indenting, etc.	**4**
Comments are used before each method – not in JavaDoc. It is argues that the comments make the source code easier to read	
Object-oriented quality of example (O1-O2) **A good example must be a valid role model for an object-oriented program**	
1. **Modeling.** The example emphasizes OO modeling. E.g., emphasizes the notion of OO programs as collections of communicating objects (i.e. objects sending messages to each other), suitable units of abstraction/decomposition with well-defined responsibilities on all levels (package, class, method).	**2**
Object communication is only between the main-method and the `Die` objects. It would have been obvious to include a `DieCup` class.	
2. **Style.** The code adheres to accepted OO design principles. E.g., proper encapsulation/information hiding, Law of Demeter (no inappropriate intimacy), no subclassing for parameterization, etc.	**3**
The roll-method both rolls and returns the value.	
Didactic quality (D1-D6) **A good example must be easy to understand, promote "OO thinking" and emphasize programming as a process**	
1. **Sense of purpose.** Students can relate to the example's domain and computer programming seems a relevant approach to solve the problem. (In contrast to, e.g., washers which are only relevant to engineers, if the concept or word is at all known to students outside the domain (or English-speaking countries))	**4**
GPA is not known outside US (only part of the explanation not the actual example).	
2. **Process.** An appropriate programming process is followed/described. E.g., problem stated and analyzed, design, coding, debugging, testing (constrained by the learning focus)	**2**
No arguments, discussion of the design.	
3. **Breadth.** The example is focused on a small coherent set of new issues/topics. It is not overloaded with new "stuff" or things introduced "by the way". Students' attention must not be distracted by irrelevant details or auxiliary concepts/ideas; they must be able to get the point of the example and not miss "the forest for the trees". (In contrast to, e.g., explaining JavaDoc in detail when actual topic is introducing classes)	**4**
A discussion of several classes in one physical file seems inappropriate.	
4. **Detail.** The example is at a suitable level of abstraction for a student at the expected level and likely understandable by such a student (avoid irrelevant detail). (In contrast to, e.g., when an example wants to describe the concept of state of objects, but winds up detailing memory layout in the JVM)	**5**
5. **Visuals.** The explanation is supported by relevant and meaningful visuals. E.g., use visuals to explain primitive versus class-based object references to explain the code. (In contrast to, e.g., showing a general UML diagram as an after-thought without relating to the actual example)	**5**
Many relevant visuals – both UML diagram (even though it comes as an after thought) and others.	
6. **Prevent misconceptions.** The example illustrates (reinforces) fundamental OO concepts/issues. Precautions are taken to prevent students from drawing inappropriate conclusions. E.g., multiple instances of at least one class (to highlight the difference between classes and objects), not just "dumb" data-objects (with only setters and getters), show both primitive attributes and class-based attributes, methods with non-trivial behavior, dynamic object creation, etc.	**2**
The main methods call the roll method of the `Die`, but the return value is ignored. Does the student know that the `toString`-method is automatically called in `println`?	

Additional comments. Notable things that were not fully covered by the checklist:

[1] On a five point Likert-scale; 1=strongly disagree, 2=disagree, 3=neutral, 4=agree, 5=strongly agree.

Figure 8: Example of a filled-in review form for example E4 and reviewer R6.

Reviewer:	R1–R6
Textbook:	…
Page/Figure no:	E3
Brief characterization/description of the example (purpose, revisited/revised version?):	
A single class `Student`. No other classes using it.	

•••

Object-oriented quality of example (O1-O2) **A good example must be a valid role model for an object-oriented program**		
1. Problem versus implementation. The code is appropriate for the purpose/problem (remember: the solution need not be OO, if the purpose/problem does not suggest it).		
R1:	*It is only one class with no test/run-class. There is no creation of an object in the program…* *Only simple assignments and string concatenation* *There are only getters and setters (and `toString`)*	**1**
R2:		**2**
R3:		**4**
R4:	*A dumb storage class. Just trivial set- and get-methods (one-liners). The class keeps a constant for the number of credits necessary to graduate, but this constant is not used; `Student` objects don't even handle actual credits (i.e., no attributes/methods for that) => not an intuitive design.*	**2**
R5:	*Basically a C struct. No interaction shown. The abstraction is well defined however.*	**2**
R6:	*The name could be a class by itself focusing on the (three) parts of a name.*	**3**

•••

Figure 9: Example of gradings and comments of all reviewers for quality factor O1 (Modeling) on example E3.

C Appendix: Assessment data

The following figures summarize all assessment data.

Figure 10 gives a graphical overview over all individual grades for all reviewers and all examples. Grades correspond to the answers on a five point Likert-style scale[6]. QFs[7] with "conflicting" grades, i.e., containing agreements as well as disagreements, are marked by boxes.

Figure 11 shows the same information as Figure 10 in tabular form with the grades causing the conflict marked green (grades deviating positively from the majority) and pink (grades deviating negatively from the majority). The average grades for each reviewer are shown to the right in the tables. Below the tables, the average grades and standard deviations for each QF are shown as well. Standard deviations above 1 are marked in orange. The maximum and minimum average grades are marked green (maximum) and pink (minimum), respectively.

In Figure 12, the average grade and standard deviation are show for each of the QFs in each example. Included in the figure is also a plot of the average for all examples.

[6] 1 = strongly disagree, 2 = disagree, 3 = neutral, 4 = agree, 5 = strongly agree.
[7] QF = quality factors T1–T3, O1–O2, and D1–D6.

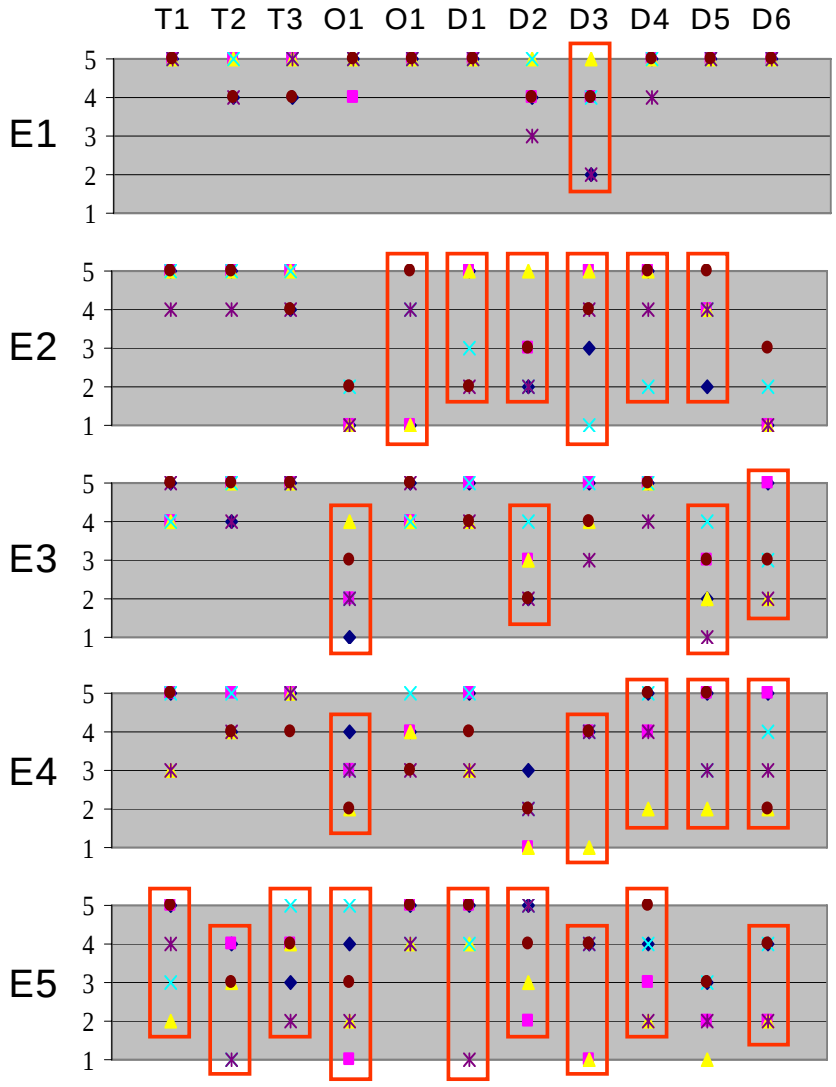

Figure 10: Graphical overview over all individual grades for all examples.

E1	T1	T2	T3	O1	O2	D1	D2	D3	D4	D5	D6	AVG
R1	5	4	4	5	5	5	4	2	5	5	5	4,45
R2	5	5	5	4	5	5	4	4	5	5	5	4,73
R3	5	5	5	5	5	5	5	5	5	5	5	5,00
R4	5	5	5	5	5	5	5	4	5	5	5	4,91
R5	5	4	5	5	5	5	3	2	4	5	5	4,36
R6	5	4	4	5	5	5	4	4	5	5	5	4,64
Avg.	5,00	4,50	4,67	4,83	5,00	5,00	4,17	3,50	4,83	5,00	5,00	4,68
Std. dev.	0,00	0,55	0,52	0,41	0,00	0,00	0,75	1,22	0,41	0,00	0,00	0,35

E2	T1	T2	T3	O1	O2	D1	D2	D3	D4	D5	D6	AVG
R1	5	5	4	1	1	5	2	3	5	2	1	3,09
R2	5	5	5	1	1	5	3	5	5	4	1	3,64
R3	5	5	5	1	1	5	5	5	5	4	1	3,82
R4	5	5	5	2	4	3	2	1	2	4	2	3,18
R5	4	4	4	1	4	2	2	4	4	4	1	3,09
R6	5	5	4	2	5	2	3	4	5	5	3	3,91
Avg.	4,83	4,83	4,50	1,33	2,67	3,67	2,83	3,67	4,33	3,83	1,50	3,45
Std. dev.	0,41	0,41	0,55	0,52	1,86	1,51	1,17	1,51	1,21	0,98	0,84	1,00

E3	T1	T2	T3	O1	O2	D1	D2	D3	D4	D5	D6	AVG
R1	5	4	5	1	5	5	2	5	5	2	5	4,00
R2	4	5	5	2	4	5	3	5	5	3	5	4,18
R3	4	5	5	4	4	4	3	4	5	2	2	3,82
R4	4	5	5	2	4	5	4	5	5	4	3	4,18
R5	5	4	5	2	5	4	2	3	4	1	2	3,36
R6	5	5	5	3	5	4	2	4	5	3	3	4,00
Avg.	4,50	4,67	5,00	2,33	4,50	4,50	2,67	4,33	4,83	2,50	3,33	3,92
Std. dev.	0,55	0,52	0,00	1,03	0,55	0,55	0,82	0,82	0,41	1,05	1,37	0,70

E4	T1	T2	T3	O1	O2	D1	D2	D3	D4	D5	D6	AVG
R1	5	4	5	4	4	5	3	4	5	5	5	4,45
R2	5	5	5	3	4	5	1	4	4	5	5	4,18
R3	3	4	5	2	4	3	1	1	2	2	2	2,64
R4	5	5	5	3	5	5	2	4	5	3	4	4,18
R5	3	4	5	3	3	3	2	4	4	3	3	3,36
R6	5	4	4	2	3	4	2	4	5	5	2	3,64
Avg.	4,33	4,33	4,83	2,83	3,83	4,17	1,83	3,50	4,17	3,83	3,50	3,74
Std. dev.	1,03	0,52	0,41	0,75	0,75	0,98	0,75	1,22	1,17	1,33	1,38	0,94

E5	T1	T2	T3	O1	O2	D1	D2	D3	D4	D5	D6	AVG
R1	5	4	3	4	5	5	5	4	4	3	4	4,18
R2	5	4	4	1	5	5	2	1	3	2	2	3,09
R3	2	3	4	2	4	4	3	1	2	1	2	2,55
R4	3	1	5	5	4	4	5	4	4	3	4	3,82
R5	4	1	2	2	4	1	5	4	2	2	2	2,64
R6	5	3	4	3	5	5	4	4	5	3	4	4,09
Avg.	4,00	2,67	3,67	2,83	4,50	4,00	4,00	3,00	3,33	2,33	3,00	3,39
Std. dev.	1,26	1,37	1,03	1,47	0,55	1,55	1,26	1,55	1,21	0,82	1,10	1,20

Figure 11: All grades given by reviewers R1–R6 for the examples E1–E5.

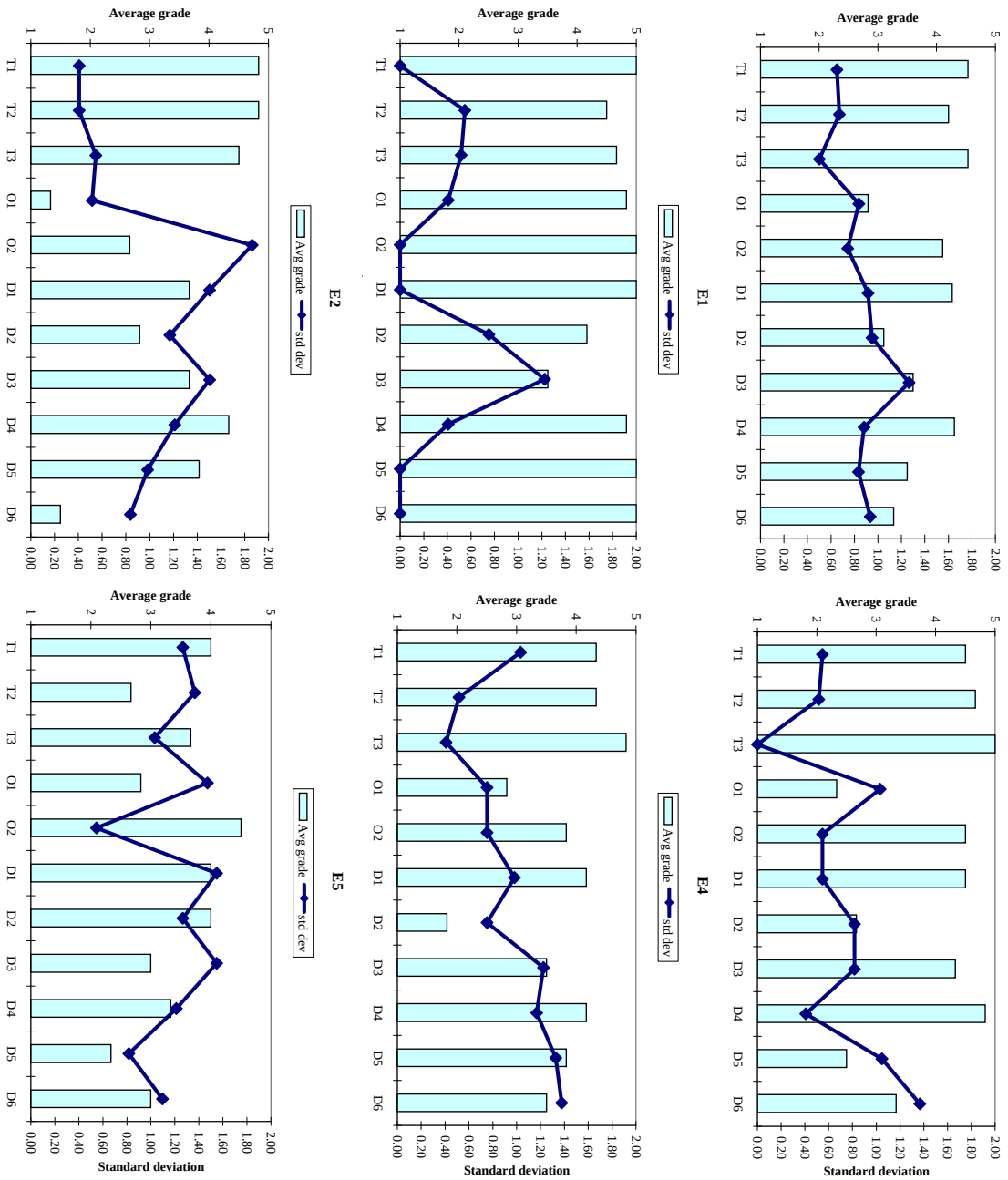

Figure 12: Average grades and standard deviations for each QF for examples E1–E5.