Evaluating OO Example Programs for CS1

Jürgen Börstler
Dept. of Computing Science
University of Umeå, Sweden
jubo@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

Marie Nordström
Dept. of Computing Science
University of Umeå, Sweden
marie@cs.umu.se

Lena Kallin Westin
Dept. of Computing Science
University of Umeå, Sweden
kallin@cs.umu.se

Jan Erik Moström
Dept. of Computing Science
University of Umeå, Sweden
jem@cs.umu.se

Michael E. Caspersen
Dept. of Computer Science
University of Aarhus, Denmark
mec@daimi.au.dk

ABSTRACT

Example programs play an important role in learning to program. They work as templates, guidelines, and inspiration for learners when developing their own programs. It is therefore important to provide learners with high quality examples. In this paper, we discuss properties of example programs that might affect the teaching and learning of object-oriented programming. Furthermore, we present an evaluation instrument for example programs and report on initial experiences of its application to a selection of examples from popular introductory programming textbooks.

Categories and Subject Descriptors

K3.2 [**Computers & Education**]: Computer and Information Science Education—*computer science education*

General Terms

Experimentation, Human Factors, Measurement

Keywords

CS1, example programs, object-orientation, quality

1. INTRODUCTION

Examples are important tools for teaching and learning. Both students and teachers cite example programs as the most helpful materials for learning to program [10]. Also research in cognitive science confirms that "examples appear to play a central role in the early phases of cognitive skill acquisition" [18, p 515]. Moreover, research in cognitive load theory has shown that carefully worked-out examples (so called worked examples) play an important role in order to increase learning outcome [5].

In mathematics education exemplification is a well researched topic [11] and "the choice of examples that learners are exposed to plays a crucial role in developing their ability to generalize" [21, p 131]. Examples must therefore always be consistent with all learning goals and follow the principles, guidelines, and rules we want to convey. Otherwise, students will have a difficult time recognizing patterns and telling an example's non-essential properties (noise) from those that are structurally or conceptually important.

Learner's will learn from examples, but we cannot guarantee that they abstract the properties and rules we want them to learn. They might not see the generality and just mimic irrelevant example properties and features [13]. The rules they construct might be erroneous, since there are many ways to interpret and generalize example features [7, 9, 18]. It is therefore also 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 [13].

Carefully developed and presented examples, can help preventing misconceptions [4, 8].

In this paper, we discuss quality properties of example programs and formulate criteria which are used to develop an evaluation instrument. We then present the results of using this instrument on a selection of program examples from popular textbooks. Finally, we discuss the issues concerning the design and usage of the evaluation instrument and outline how our work can be taken further.

2. RELATED WORK

Although examples are perceived as one of the most important tools for the teaching and learning of programming, there is very little research in this area. Most often example issues are only discussed in the narrow context of a single simple and concrete example, like the recurring "Hello World"-type discussions [6, 19], or they are regarded as a language issue [2, 15]. Only few authors have taken a broader view by investigating features of example programs and their (potential) effects on learning.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'08, June 30–July 2, 2008, Madrid, Spain.
Copyright 2008 ACM 978-1-60558-115-6/08/06 ...$5.00.

Wu et al. studied programming examples in 16 high school computer textbooks and concluded that most of them “lacked detailed explanation of some of the problem-solving steps, especially problem analysis and testing/debugging” [20, p 225]. Almost half of the examples fell into either the math-problem (27%) or syntax-problem (21%) category.

Holland et al. [9] provide guidelines for designing example programs to prevent object-oriented misconceptions, which are successfully used by Sanders and Thomas [16] for assessing student programs.

Malan and Halland [12] describe four common pitfalls that should be avoided when developing example programs. They argue that examples that are too abstract or too concrete, that do not apply the taught concepts consistently, or that undermine the concept they are introducing, might hinder learning.

Furthermore, there are many studies of software development in general showing that adherence to common software design principles, guidelines, and rules [3], as well as certain coding, commenting, naming guidelines, and rules [14, 17] support program understanding.

There is also a large body of research on worked examples providing general guidelines regarding the form and presentation of examples [5].

However, to our knowledge, neither of the above principles, guidelines, and rules have been used to evaluate example programs from programming textbooks.

3. RESEARCH APPROACH

This project is carried out by two research groups from two countries. During an initial two-day workshop, a large number of example programs from different textbooks were discussed to identify common strengths and weaknesses. The goal was to define a set of criteria to effectively discriminate between different levels of “quality”, based on accepted principles, guidelines, and rules from the literature (see Section 2) and our own teaching experience. The outcome of this workshop was an initial evaluation instrument and a test set of textbook examples.

The instrument was tested on two examples by four reviewers, which lead to several revisions of the instrument. After testing further examples, the instrument was finally refined to the one described in Section 4. The instrument was then used by six reviewers (two female, four male; age 37–48) to evaluate five example programs. All reviewers are experienced computer science lecturers in object-oriented programming, most of them at the introductory level. A summary of the evaluation results are presented in Section 5. Finally, validity and reliability of detailed results were discussed between reviewers and researchers (the groups overlapped considerably) in small groups and by e-mail. A summary of these discussions is presented in Section 6.

Table 1: Categorization of example programs.

	First example	First user-defined class	Several classes	Inheritance
E1	—	—	X	—
E2	X	—	—	partly
E3	—	X	—	—
E4	—	X	partly	—
E5	—	—	X	X

To cover a wide range of aspects, we chose representative examples of different levels of quality and complexity covering the following aspects; the very first example of a textbook, the first exemplification of developing/writing a (user-defined) class, the first application involving at least two interacting classes and a non-trivial (but still simple) example of using inheritance. Table 1 summarizes the features of our five examples E1–E5.

4. EVALUATION INSTRUMENT

Inspiration for the evaluation instrument was drawn from the checklist-based evaluation by the Benchmarks for Science Literacy project [1] by defining a set of specific, well-defined criteria that can be evaluated on a uniform scale. All criteria should be based on accepted programming principles, guidelines, and rules; educational research; and the groups’ collective teaching experience. The resulting set of 11 criteria was grouped into three independent categories of quality; *technical quality* (three items), *object-oriented quality* (two items) and *didactic quality* (six items).

Technical quality (T1–T3). The criteria in this category focus on technical aspects of example programs that are independent of the programming paradigm. Examples should be syntactically and semantically correct, written in a consistent style, and follow accepted programming principles, guidelines, and rules (see Table 2).

Table 2: Checklist items for technical quality.

T1	**Problem versus implementation.** The code is appropriate for the purpose/problem (note that the solution need not be OO, if the purpose/problem does not suggest it).
T2	**Content.** The code is bug-free and follows general coding guidelines and rules. All semantic information is explicit. E.g., if preconditions and/or invariants are used, they must be stated explicitly; dependencies to other classes must be stated explicitly; objects are constructed in valid states; the code is flexible and without duplication.
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 indentation.

Object-oriented quality (O1–O2). The criteria in this category address technical aspects that are specific for the object-oriented paradigm, i.e., how well an example can be considered a role model of an object-oriented program. In contrast to technical quality, the principles, guidelines, and rules covered here are specific for the object-oriented paradigm (see Table 3).

Table 3: Checklist items for object-oriented quality.

O1	**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); models suitable units of abstraction/decomposition with well-defined responsibilities on all levels (package, class, method).
O2	**Style.** The code adheres to accepted OO design principles. E.g., applies proper encapsulation and information hiding; adheres to the Law of Demeter (no inappropriate intimacy); avoids subclassing for parameterization; etc.

Didactical quality (D1–D6). The criteria in this category deal with instructional design, i.e., comprehensibility and alignment with learning goals for introductory (object-oriented) programming (see Table 4).

In order to evaluate an example the expected previous knowledge of a student and supporting explanations must be taken into account. In textbooks the placement of an example naturally defines the expected previous knowledge of

Table 4: Checklist items for didactic quality.

D1	**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., flat 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).
D2	**Process.** An appropriate programming process is followed/described. I.e., the problem is stated explicitly, analyzed, a solution is designed, implemented and tested.
D3	**Breadth.** The example is focused on a small coherent set of new concepts/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 the actual topic is introducing classes.
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 (avoid irrelevant detail). In contrast to, e.g., when an example sets out to describe the concept of state of objects, but winds up detailing memory layout in the JVM).
D5	**Visuals.** The explanation is clear and supported by meaningful visuals. E.g., uses visuals to explain the differences between variables of primitive (built-in) types and object types. In contrast to, e.g., showing a generic UML diagram as an after-thought without relating to the actual example.
D6	**Prevent misconceptions.** The example illustrates (reinforces) fundamental OO concepts/issues. Precautions are taken to prevent students from overgeneralizing or 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.

a student. In the context of this work an example is considered as a complete application or applet plus all supporting explanations related to this particular program.

To summarize, one could say that T1–T3 and O1–O2 assess the actual code of an example program and D1–D6 assess how it is presented to and conceived by a student. The categories complement each other; an example of high technical and object-oriented quality will not be very effective, if it cannot be understood by the average student. However, such an example might still be a very valuable teaching resource, in case the educator using it finds better ways to explain it.

All ratings in the resulting checklist are on a Likert-type scale from 1 (strongly disagree) to 5 (strongly agree). Since all checklist items are formulated positively, 5 is always best. Beyond the criteria described in Tables 2–4, the actual checklist also contains additional fields for commenting each rating and a field for overall comments that might not fit any of the available criteria. An example of a filled-in checklist can be found at `http://www.cs.umu.se/research/education/checklist_iticse08.pdf`.

5. RESULTS

The results presented here are based on the evaluation that was made in order to answer two questions:

- Can the instrument distinguish between "good" and "bad" examples?
- Do reviewers interpret the items of the instrument in the same way?

Figure 1 summarizes the results of six reviewers' evaluation of the five examples, E1–E5 (see also Table 1). As can be seen, only one example (E1) is consistently rated very high across all three quality categories. Low average ratings have almost always a relatively high standard deviation (i.e., disagreement between reviewers).

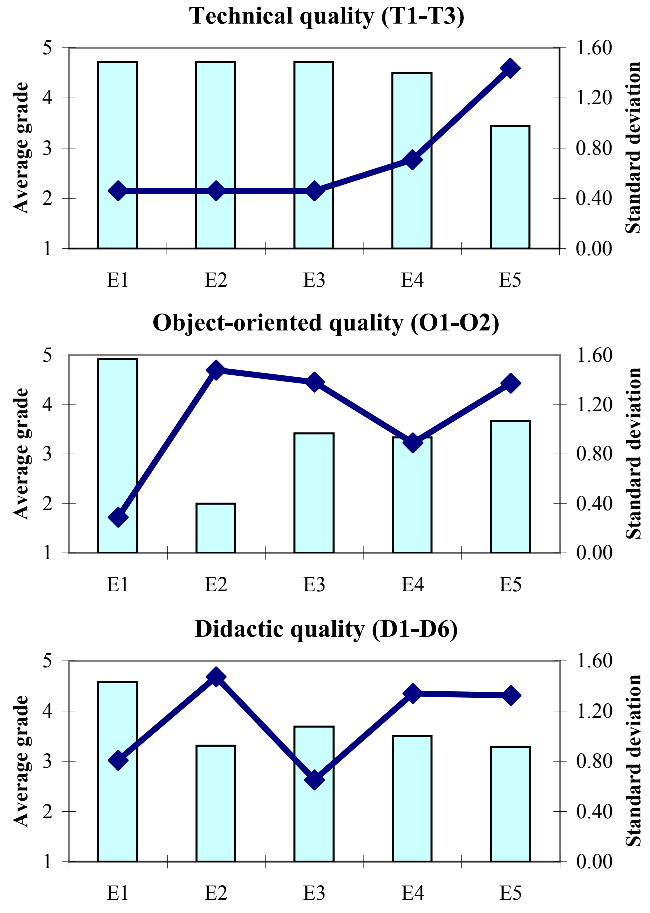

Figure 1: Average grade (bars) and standard deviation (line) for evaluation of five examples. Results are shown by item category (technical, object-oriented, and didactic quality).

Besides the overall high rating of E1, there are several other noteworthy observations. The overall technical quality of the reviewed example programs is very high, except for E5 which did not correctly implement its stated requirements. The section on object-oriented quality has the largest variation. It should, however, be noted that E2 is a "Hello World"-type example which cannot be expected to achieve high ratings in this category. Given that we used examples from quite popular textbooks, the overall ratings for didactic quality and the ratings of E3–E5 on object-oriented quality were surprisingly low.

Figure 2 shows the overall distribution of ratings for each of the six reviewers, R1–R6. It can be noted that the reviewers utilize the rating scale differently. Reviewer R5, for example, used the best grade (5) only half as much as the average (21.8% compared to 43% for all reviewers together). Reviewer R6, on the other hand, did not use a single 1. However, except for reviewer R5, the distributions of ratings are quite similar (in total the usage of rating 1 was only 7.3%).

It seems that teaching experience somewhat influences the grading. One reviewer, R5, has exclusively taught advanced

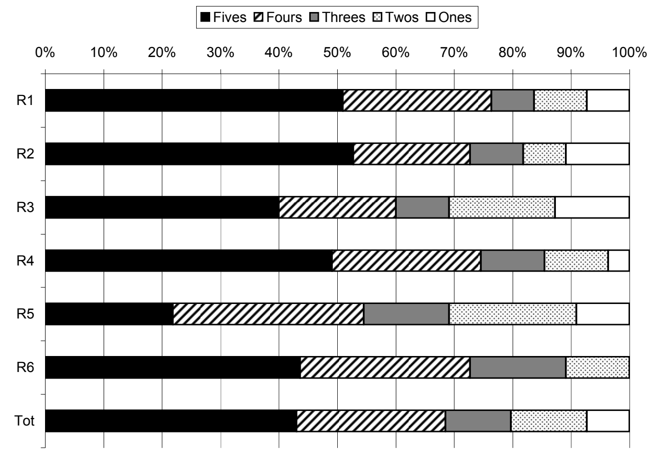

Figure 2: Distribution of ratings between reviewers.

programming courses for the last couple of years, and grades given by R5 tend to be slightly lower on average.

To summarize, it is evident that the instrument distinguishes between examples. Furthermore, the example with the overall highest ratings, E1, is also considered to be a "good" example by the reviewers. However, when looking at Figure 2, it is evident that there are differences in ratings among the reviewers.

6. DISCUSSION

The purpose of the presented instrument has been to replace intuitive "I know it, when I see it" assessments of example programs with a more objective and reliable measurement. So what conclusions can we draw from the results? Are the criteria meaningful? How well do assessment results using the instrument reflect an experienced teacher's overall "gut feeling" of example quality?

This section summarizes the discussions among researchers and reviewers regarding validity and reliability of the evaluation instrument.

6.1 Quality Categories

Overall, we find that the instrument ranked examples as we might have done based on "gut feeling" alone. The instrument was quite useful to point out particular strengths and weaknesses. The reviewers found the three categories natural and covering all important issues of examples.

Technical quality (T1–T3). Assuming that textbook authors have developed and tested their examples carefully, one would generally expect the technical grades to be very high. This is also reflected in consistent and high ratings. The only exception is E5, that contains a defect and the resulting program does not fulfill its stated requirements. This issue was well captured by criteria T2. The standard deviation is generally low, indicating objectivity of the criteria.

Object-oriented quality (O1–O2). In general, the object-oriented characteristics of examples seem to be captured well. E1 received the highest rating and was also agreed to be an exemplary example. E2 is the first example in a textbook (see Table 1) and not really focusing on object-oriented techniques, which is reflected by its low rating in this category. In three of the examples, the standard deviation is high. One reason for this seems to be a lack of common agreement on the importance of explicit object interaction. Another reason seems to be the dependency between O1 and O2 (see Section 6.2).

Didactic quality (D1–D6). When comparing the average grades in this category, one example is rated high; the others are rated lower but at approximately the same level. This is also in concordance with the comments from the discussion after the evaluation. Three of the examples exhibit a high standard deviation indicating a high degree of disagreement among the reviewers. It seems that the reviewers do not share a common understanding of the meaning of the criteria and how they should be rated.

6.2 Criteria

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 criterion. However, careful inspection of all evaluations revealed that the rating still was difficult in some cases.

O1 vs. O2. Since criteria are supposed to be independent, several reviewers gave high ratings for O2 and low ratings for O1 for the same example. They argued that accepted object-oriented principles and guidelines (like encapsulation and information hiding) could very well be followed even by examples that are not considered object-oriented and we should not "penalize" an example twice for the same reason. However, this is in conflict with the overall intention of the category, namely rating object-oriented quality. Lack of object-orientedness should result in low ratings. Therefore, it is necessary to agree on and carefully describe the intended use of O1 and O2.

A solution could be to replace O1 and O2 by a single item for overall object-orientedness and conditional items for a more detailed assessment of object-oriented characteristics. The detailed assessment would then only be carried for examples above a certain threshold value for overall object-orientedness.

D3 vs. D4. Breadth and detail are two views on extraneous or superfluous material. It can therefore be difficult to decide how to balance ratings on these and there might be cases where an example has been penalized twice. A solution could be to use only one criterion assessing the amount of extraneous or superfluous material.

D5. Some reviewers gave high ratings for visuals, although the example lacked visuals, arguing that the explanation was perfectly clear and understandable even without visuals. Discussing this aspect revealed that there might be a more general problem of rating criteria that are simply not addressed properly by an example, e.g., O1, O2, and D2.

D6. When the instrument was constructed it was debated whether D6 should be regarded as "object-oriented quality" rather than "didactic quality". Moving the ratings for D6 to this category resulted, however, only in minor changes of the results as compared to Figure 1.

Granularity and impact. It might be tempting to compute a single value for the rating of an example. However, granularity and impact of criteria can never be perfectly equal. Even within a category, criteria will be valued differently by different people. Moreover, categories with many criteria might be overemphasized.

We have tried to balance criteria within a category. However, an overall total (over all categories) seems not very meaningful.

6.3 Rating Instructions

Already when developing the instrument, a recurring topic of discussion was when to rate a criterion for an example as 1 and when to rate it as 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 2). If this or a similar instrument is to be used in a community, we strongly recommend supplying a written instruction, containing prototypical examples, with the instrument.

7. SUMMARY AND CONCLUSIONS

In this paper, we have described the design and test of an instrument for evaluating object-oriented examples for educational purposes. The instrument was tested by six experienced educators on five examples, which we consider quite representative for 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 examples. 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 examples play an important role in learning to program, it would be valuable to formally evaluate textbook examples at a larger scale.

However, we consider the evaluation instrument presented here not 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 criteria to avoid misunderstandings and, most importantly, developing detailed rating instructions.

8. REFERENCES

[1] AAAS. Benchmarks for science literacy, a tool for curriculum reform, 1989. `http://www.project2061.org/publications/bsl/default.htm`, last visited 2007-12-07.

[2] L. Böszörményi. Why Java is not my favorite first-course language. *Software-Concepts & Tools*, 19(3):141–145, 1998.

[3] L. Briand, C. Bunse, and J. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering*, 27(6):513–530, 2001.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Liz, Bills, T. Dreyfus, J. Mason, P. Tsamir, A. Watson, and O. Zaslavsky. Exemplification in mathematics education. In *Proceedings of the 30th Conference of the International Group for the Psychology of Mathematics Education, Vol. 1*, pages 126–154, 2006.

[12] 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.

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

[14] P. Oman and C. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, 1990.

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

[16] K. Sanders and L. Thomas. Checklists for grading object-oriented cs1 programs: Concepts and misconceptions. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 166–170, 2007.

[17] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Languages*, 4(143):167, 1996.

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

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

[20] 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.

[21] R. Zazkis, P. Liljedahl, and E. J. Chernoff. The role of examples in forming and refuting generalizations. *ZDM Mathematics Education*, 40:131–141, 2008.