On the Quality of Examples in Introductory Java Textbooks

JÜRGEN BÖRSTLER and MARIE NORDSTRÖM
Umeå University
and
JAMES H PATERSON
Glasgow Caledonian University

Example programs play an important role in the teaching and learning of programming. Students as well as teachers rank examples as the most important resources for learning to program. Example programs work as role models and must therefore always be consistent with the principles and rules we are teaching.

However, it is difficult to find or develop examples that are fully faithful to all principles and guidelines of the object-oriented paradigm and also follow general pedagogical principles and practices. Unless students are able to engage with good examples, they will not be able to tell desirable from undesirable properties in their own and others' programs.

In this paper we report on a study in which experienced educators evaluated the quality of object-oriented example programs for novices from popular Java textbooks. The evaluation was accomplished using an on-line checklist that elicited responses on the technical, object-oriented, and didactic quality of examples.

In total 25 reviewers contributed 215 reviews to our data set, based on 38 example programs from 13 common introductory programming textbooks. Results show that the evaluation instrument is reliable in terms of inter-rater agreement. Overall, example quality was not as good as one might expect from common textbooks, in particular regarding certain object-oriented properties.

We conclude that educators should be careful when taking examples straight out of a textbook.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Principles, Guidelines, Examples

Additional Key Words and Phrases: Example programs, check list, courseware, textbooks, assessment

1. INTRODUCTION

Although example programs are perceived as one of the most important tools for the teaching and learning of programming [Lahtinen et al. 2005], there is very little research regarding their properties and usage. There is a large body of knowledge on

Author's address: Jürgen Börstler, Department of Computing Science, Umeå University, SE-90187 Umeå, Sweden; email: jubo@acm.org.

program comprehension (e.g., [Brooks 1983; Burkhardt et al. 2002]) and software quality and measurement (e.g., [Purao and Vaishnavi 2003]), but this is rarely applied in an educational setting [Börstler et al. 2007; Magel 1982].

In this paper, we describe a checklist-based approach for assessing object-oriented example programs for novices. The checklist covers technical, object-oriented and didactic qualities of example programs. The checklist has been used to evaluate common textbook examples. Our results show that the checklist is reliable and helps to indicate strengths and weaknesses of example programs.

Although checklist-based assessment is a common approach for quality assurance in industry [Brykczynski 1999], it has rarely been used in educational contexts. Sanders and Thomas [2007], for example, have reviewed typical novice misconceptions to develop checklists that educators can use when designing or grading student programs.

The work presented in the present paper is based on the author's earlier work on checklist-based evaluation of example programs [Börstler et al. 2009; Börstler et al. 2008]. In the present work we have extended our detailed analysis to further reviewers and examples. Furthermore, we also also provide analyses which are different from those in our previous work. We have identified a set of characteristics and examined the examples and reviews in detail with reference to these to determine which characteristics may be influential in defining "good" examples, and we relate our results to traditional software measures.

2. IMPORTANCE OF EXAMPLES

Examples play an important role in teaching and learning programming. Students and teachers alike cite examples as the most helpful resource for learning to program [Lahtinen et al. 2005]. With respect to LISP programming, for example, Anderson et al. [1984] showed that all novices needed example programs to complete their first recursive example program. When given a choice, students generally prefer examples over written instructions for solving problems [Reed and Bolstad 1991]. Even when the example conflicts with written problem solving instructions, students typically use the example information and disregard the written instructions [LeFevre and Dixon 1986].

Examples are powerful role models; novices use examples as templates for their own work.

> What novices often do, however, is employ a knowledge-lean style of analogical reasoning, analogical transfer *within* a domain, not across domains. For instance, and most important, students use worked-out examples provided in textbooks, by the teacher, or by their peers when solving new problems. [Reimann and Schult 1996, p. 123]

Examples must therefore be consistent with the principles and rules being taught and should not exhibit any undesirable properties or behaviour. In other words, all examples should follow exactly the same principles, guidelines, and rules we expect our students to eventually learn. If our examples do not do so consistently, students will have difficulty in recognizing patterns and distinguishing an example's surface properties from those that are structurally or conceptually important. In other words, 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 [Mason and Pimm 1984].

Trafton and Reiser [1993] 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 to distinguish between good and bad designs.

> Simply learning to perform procedures, and learning in only a single context, does not promote flexible transfer. The transfer literature suggests that the most effective transfer may come from a balance of specific examples and general principles, not from either one alone. [Bransford et al. 2004, p.77]

With carefully developed examples, we can reduce misinterpretations, premature generalizations or otherwise unintended conclusions. This helps to prevent misconceptions, which might hinder students in their further learning [Clancy 2004; Guzdial 1995; Malan and Halland 2004].

3. RELATED WORK

Textbooks are an important component of teaching introductory programming. They are a major source for example programs and also function as a reference for how to solve specific problems. Although examples play an important role in the teaching and learning of programming, there are very few systematic evaluations of textbook examples.

De Raadt et al. [2005] compared 40 introductory programming textbooks by measuring their amount of content particularly relevant for the textbook's usefulness as a learning tool, such as the number of pages covering examples, exercises, bibliographies, appendices, language reference, index, glossary, and other chapter content. In the 22 texts covering object-oriented programming (in C++, Java, Eiffel, and Delphi), examples covered from 0% to 43% of the total page counts (about 17% on average). About examples, 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. [1999] analysed the examples of 16 Taiwanese high school computer textbooks (using BASIC as a programming language), covering 967 examples. Each example was categorized according to its problem type and coverage of problem solving as a process. Their analysis revealed that only 2% of the examples displayed a thorough analysis of the problem statement. Testing and debugging was discussed for only 0.2% of the examples. Only 25% of the examples dealt with real-life problems, the remaining examples were categorized as Math (27%), Graphics

(25%), Syntax (21%) and Miscellaneous (2%). The authors conclude that the material should be made "potentially meaningful to the students in order for meaningful learning to occur". Furthermore they advise a more thorough coverage of the problem solving process.

A subsequent analysis of 32 Taiwanese high school computer textbooks found that most of these problem still persisted [Lin and Wu 2007]. In particular the authors also criticize "inadequate analogies" and "dry examples".

Malan and Halland [2004] 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 which are either too abstract or too complex and examples which apply concepts inconsistently or even undermine the concept(s) being introduced.

An important aspect of examples is *misconceptions* and how to avoid them. Holland et al. [1997] outlined a number of student problems and how they might relate to properties in example programs, such as 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 [2000], who discussed how students constructed their own rules by misapplying correct rules. She described certain cases in which 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 [Westfall 2001] which created a series of follow-ups on the object-orientedness of common introductory programming examples [CACM Forum 2002; Ourosoff 2002; Dodani 2003; CACM Forum 2005]. Surprisingly, the main discussion has been centered on how to adjust the '`HelloWorld`'-example to better fulfil the characteristics of object-orientation, not on whether this is a good example at all.

Hu [2005] discusses the related problem of data-less objects (of which '`HelloWorld`' is an instance). 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.

4. METHOD

Data collection and initial analysis started out as an ITiCSE working group [Börstler et al. 2009] and was based on earlier work by some of the co-authors [Börstler et al. 2008]. In the present work we have extended the pool of analysed reviews by 57% and also provide different analyses from the previous work.

An electronic checklist (described in Section 4.4) was administered for evaluating object-oriented example programs from common CS1 textbooks.

In total, we selected 38 examples from 13 introductory programming textbooks (mainly Java). Examples were nominated by working group participants, who were given the guidance that examples were preferred from textbooks which were published in 2007 or later and in the second (or later) edition, the latter being taken as an indicator of common usage. The examples were grouped into two sets:

mandatory and optional. All working group participants were required to review mandatory examples (16) and to review optional examples (22) if time permitted. A number of additional reviewers were invited to review as many examples as they were able to do. In the present paper, we further discuss only those 21 examples that received ≥3 reviews (191 reviews in total) covering 11 of the 13 textbooks. For detailed information about the full set of examples and textbooks the interested reader is referred to [Börstler et al. 2009].

4.1 Textbooks

When selecting source for examples, we aimed at a broad and representative coverage with respect to popularity, coverage, presentation style and pedagogic approach.

In order to get comparable examples, we looked for the first example in a source that exemplified a certain topic or concept. Examples were furthermore required to be *complete and clearly identifiable*, where complete means that the full code is shown together with a discussion or explanation.

Following these requirements, we had to exclude a wide range of sources. For example we excluded all sources working mainly with code snippets (incomplete code) as well as sources introducing examples gradually through successive versions (no clearly identifiable example). We also had to exclude all online tutorials we looked at, since they either had no clearly identifiable first example or lacked a discussion or explanation. Our selection of sources is therefore not fully representative.

This left us with textbooks as the only sources for examples, We classified these into two main categories: those with a clear and early focus on object-orientation (category *OO*) and those with a more traditional imperative first approach (category *Trad*). A first classification was made based on the texts' titles, back cover texts, prefaces or web pages. For a second classification, we followed VanDrunen's approach [VanDrunen 2006] and looked at each text's detailed table of contents to determine when various key concepts were introduced. In addition to that, we also looked into each text to get a feeling for actual focus and depth of a section and the style of concept presentation.

The 11 textbooks, from which the examples in the present paper are taken, are listed in Table I. One can observe a clear discrepancy between the "self-declared" approach of a text and our classification. Although most texts claim to focus on object-orientation and follow some kind of object-early or -centric approach, a closer inspection reveals that the texts in the OO category are actually in a minority. More information about the textbooks can be found in [Börstler et al. 2009].

4.2 Example Programs

To get down to a manageable but still representative set, we focused on examples with comparable properties. We considered only *complete examples* in the sense that the full source code should be present together with an explanation. Each example should furthermore be the first one in a text exemplifying certain high level concepts or ideas. To reach a good balance of varying types of examples, we categorized all examples according to the concepts they illustrate. Table II gives an overview of the number of examples per category.

FUDC: First user defined class. These examples reflect the first occurrence of a

Table I. Summary of textbooks. Column `Ex. pp.` refers to the page numbers of the examples reviewed. Entries in column `Self declared as` are quotes from the texts' back cover texts, prefaces or web pages.

					Category	
Text	*Pages*	*Edition*	Ex. pp.	Self declared as	*Self*	*Our*
[Barnes and Kölling 2009]	516	4*th*	18–22, 56–71	Objects first	*OO*	*OO*
[Bravaco and Simonson 2010]	1210	1*st*	404–408	Fundamentals first	*Trad*	*Trad*
[Deitel and Deitel 2007]	1596	7*th*	86–103	Early classes and objects pedagogy	*OO*	*Trad*
[Farrell 2010]	870	5*th*	370–375	—	—	*Trad*
[Horstmann 2008]	1204	3*rd*	85–90, 236–241, 438–442	Objects gradual	?	*OO*
[Lewis and Loftus 2009]	832	6*th*	190–194, 241–243, 515–527	True object-orientation	*OO*	*Trad*
[Malik and Burton 2009]	1018	1*st*	185–192	Objects early but gently	*OO*	*Trad*
[Niño and Hosch 2008]	1040	3*rd*	85–92, 123–135	Objects first	*OO*	*OO*
[Riley 2006]	769	2*nd*	263–265, 386–395	Object centric	*OO*	*OO*
[Roberts 2008]	587	2*nd*	190–198, 332–338	Modern objects first approach; class hierarchies from the beginning	*OO*	*Trad*
[Wu 2008]	987	5*th*	156–162, 309–310	Objects first	*OO*	*Trad*

user defined class in a text. We consider these examples particularly important, since they "set the stage" for how students are expected to think about object-oriented class design.

OOD: Multiple user defined classes. Examples in this group exemplify some kind of design decision/strategy. They show how existing classes can be "used" for defining new classes (inheritance, composition) or how designs can be made flexible (interfaces, polymorphism). Examples in this group can be considered role models for determining relationships between classes.

CS: Control structures. The main purpose of the examples in this group is to exemplify the usage of control structure (selection and repetition). One could argue that object-orientedness does not matter in this category, since this is not the purpose of the example. However, these examples are interesting as they demonstrate the authors' approach to teaching language syntax within the context of OOP.

Note that for the OOD and CS categories examples generally reflected the first occurrence of a particular topic, for example loops, and not necessarily the first example within the category, to avoid skewing the selection towards particular topics within each category.

4.3 Reviewer Demographics

The reviews were performed by experienced educators from a diverse range of institutions in five countries (Denmark, Germany, Sweden, UK, and USA). On average reviewers have more than 10 years of experience with teaching object-orientation specifically. In addition to that, several reviewers also have considerable profes-

Table II. Number of Examples chosen per book category.

	Text category		
Example Category	*OO*	*Trad*	$\sum$
First user defined class (FUDC)	4	5	9
Multiple user defined classes (OOD)	4	4	8
Control structures (CS)	2	2	4
$\sum$	10	11	21

Table III. Characterization of reviewers. `Years of Practical OO experience` refers to work experience such as "professional trainer", "researcher", "professional programmer" or "systems/software engineer". `Novice courses` are the number of OOP classes for novices taught in the last 10 years.

	Years of OO experience		*Novice*	
Reviewer	*Practical*	*Teaching*	*courses*	Comments
*R*1	0	5	$1-3$	Works in teacher education
*R*2	3	9	$7-9$	
*R*23	11	10	$7-9$	
*R*4	4	15	$4-6$	Dealt with OOD in PhD
*R*41	17	15	≥ 10	
*R*5	0	8	≥ 10	Textbook author (OOP/Java)
*R*6	5	20	$7-9$	Dealt with OO notation in PhD
*R*7	0	11	≥ 10	
*R*71	0	2	$1-3$	
*R*72	4	12	$7-9$	Background as professional programmer
*R*8	> 0	9	≥ 10	20+ years of professional programming

sional experience with object-orientation, for example as a researcher or a professional programmer. Most reviewers teach or have taught object-orientation to novices many times, some of them are doing so more than once a year. This work is focussed on the quality of examples for teaching object-oriented programming to novices and among other things reviewers were asked specifically to rate examples for object-oriented quality. A summary of the background data of the 11 reviewers contributing ≥ 4 reviews can be found in Table III.

4.4 The Checklist

Our checklist comprises 10 quality factors that we grouped into three quality categories; technical quality, object-oriented quality, and didactic quality.

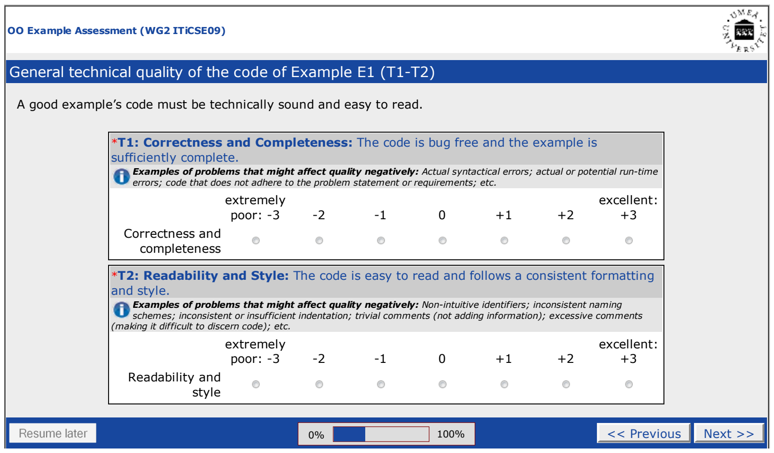
Technical qualities (T1–T2) capture properties that are independent of a particular programming paradigm, such as correctness and readability:

—Correctness and Completeness (T1): The code is bug free and the example is sufficiently complete.

—Readability and Style (T2): The code is easy to read and follows a consistent formatting and style.

Object-oriented qualities (O1–O5) capture commonly accepted principles, guidelines and heuristics of object-oriented design and programming:

—Reasonable Abstractions (O1): Abstractions are plausible from an OO modelling perspective as well as from a novice perspective.


OO Example Assessment (WG2 ITiCSE09)

General technical quality of the code of Example E1 (T1-T2)

A good example's code must be technically sound and easy to read.

***T1: Correctness and Completeness:** The code is bug free and the example is sufficiently complete.

Examples of problems that might affect quality negatively: *Actual syntactical errors; actual or potential run-time errors; code that does not adhere to the problem statement or requirements; etc.*

	extremely poor: -3	-2	-1	0	+1	+2	excellent: +3
Correctness and completeness	○	○	○	○	○	○	○

***T2: Readability and Style:** The code is easy to read and follows a consistent formatting and style.

Examples of problems that might affect quality negatively: *Non-intuitive identifiers; inconsistent naming schemes; inconsistent or insufficient indentation; trivial comments (not adding information); excessive comments (making it difficult to discern code); etc.*

	extremely poor: -3	-2	-1	0	+1	+2	excellent: +3
Readability and style	○	○	○	○	○	○	○

Fig. 1. Example survey screen (Technical quality).

—Reasonable State and Behaviour (O2): State and behaviour make sense in the presented software world context.

—Reasonable Class Relationships (O3): Class relationships are modelled properly (the "right" class relationships are applied for the "right" reasons).

—Exemplary OO code (O4): The example is free of "code smells".

—Promotes "Object Thinking" (O5): The example supports the notion of an OO program as a collection of collaborating objects.

Didactic qualities (D1–D3) capture properties related to the understandability of the example, its discussion and development:

—Sense of Purpose (D1): Students can relate to the example's domain and computer programming seems a reasonable way to solve the problem.

—Process (D2): An appropriate programming process is followed/described.

—Well Balanced Cognitive Load (D3): Explanations and supporting materials promote comprehension; they are neither simplistic, nor do they impose extraneous cognitive load.

For each quality factor we also provided a list of typical problems, which were distilled from the literature on student problems or misconceptions and object-oriented design principles, guidelines and rules (see for example [Fowler 1999; Nordström 2009; Riel 1996]).

Each quality factor was assessed on a 7-point Likert-type scale from -3 (extremely poor) to $+3$ (excellent) using the electronic survey instrument LimeSurvey [LimeSurvey]. Figure 1 shows an example screen for the questions in the category Technical Quality. The interested reader is invited to test the instrument at `http://www.cs.umu.se/proj/limesurvey/`.

In addition to the 10 quality factors described above, we also asked reviewers for their overall impression of example quality before and after the actual assessment. In two final open questions, reviewers could provide additional comments regarding

Table IV. Number of reviews per reviewer.

Reviewer	*Reviews*	*in%*
$R1$	16	7.44
$R2$	38	17.67
$R23$	5	2.33
$R4$	22	10.23
$R41$	4	1.86
$R5$	18	8.37
$R6$	18	8.37
$R7$	21	9.77
$R71$	11	5.12
$R72$	10	4.65
$R8$	16	7.44
All other 14 *reviewers*	36	16.74
$\sum$	215	100

Table V. Number of reviews per example.

Reviews	*Examples*
12	$E1, E2, E13$
11	$E11, E12, E16$
10	$E3, E4, E6, E9, E10, E14, E15$
9	$E5, E7, E8$
8	$E21, E26$
3	$E19, E20, E25$
1–2	$E17, E18, E22–E24, E27–E38$

example quality and the questionnaire itself. The instrument is discussed in more detail in [Börstler et al. 2009].

5. RESULTS

In total, we received 215 valid reviews by 25 individuals performing between 1 and 38 reviews each. Of the 25 reviewers, 9 reviewers submitted $\geq$10 reviews and 11 reviewers submitted $\geq$4 reviews. Of the 38 examples, 21 received $\geq$3 reviews each (191 reviews in total). Of those 191 reviews, 47 (roughly 25%) were contributed by people involved in checklist design. Details of reviews per reviewer and reviews per example can be found in Table IV and Table V, respectively.

Table VI summarizes the main results for all examples with $\geq$8 reviews (examples in parentheses have 3 reviews). Considering our Likert-type scale from -3 (extremely poor) to $+3$ (excellent), we get total average scores in the range $[-30, 30]$. Since we only considered examples from popular textbooks, we would expect most of the scores in the upper positive range. However, as many as 10 out of the 21 examples in Table VI scored below 10 and received an overall final impression (I2) $\leq$0.

The average ratings for overall impression before and after the actual review (I1 and I2, respectively) further corroborate this impression. Only 8 examples received an overall final impression $\geq$1 and as many as 10 examples were rated $\leq$0. Interestingly, the overall impression seems to degrade during the review, in particular for the examples that already have a low overall first impression (I1) (see also Table VII). This indicates that the checklist might help to spot problems that might be easily overlooked.

Table VI. Summary of the main results for all examples with ≥ 3 reviews (191 reviews in total). Ranking is top down according to average total score for the 10 quality factors T1–D3 (column `Score` $\in [-30, 30]$). Columns `EC` and `BC` list example (as defined in Section 4.2) and book category (as defined in Table I), respectively. Columns `I1` and `I2` list the average score for the overall impression ($\in [-3, 3]$) of example quality before review start and after finishing the review, respectively.

	EC	*BC*	*Score*	*I1*	*I2*	*I2 − I1*
*E*26	*OOD*	*OO*	23.88	2.00	2.13	0.13
*E*9	*OOD*	*OO*	21.50	1.90	1.80	−0.10
*E*21	*FUDC*	*Trad*	19.63	1.75	1.75	0.00
*E*7	*OOD*	*OO*	18.00	1.11	0.78	−0.33
*E*3	*FUDC*	*OO*	17.00	1.00	1.10	0.10
(*E*20)	*FUDC*	*Trad*	16.67	1.00	1.33	0.33
*E*2	*FUDC*	*OO*	16.42	0.92	0.83	−0.09
*E*12	*OOD*	*Trad*	16.00	1.55	1.36	−0.19
*E*16	*CS*	*OO*	15.00	1.46	1.09	−0.37
*E*1	*FUDC*	*OO*	14.17	1.00	1.00	0.00
*E*5	*FUDC*	*Trad*	13.33	0.56	0.56	0.00
*E*6	*FUDC*	*Trad*	9.90	0.20	−0.20	−0.40
*E*4	*FUDC*	*Trad*	9.90	0.10	−0.60	−0.70
(*E*19)	*FUDC*	*Trad*	7.00	0.00	−0.67	−0.67
*E*10	*OOD*	*Trad*	6.80	0.40	0.00	−0.40
*E*8	*OOD*	*Trad*	6.22	0.11	−0.33	−0.44
*E*11	*OOD*	*OO*	4.55	0.27	−0.18	−0.45
*E*14	*CS*	*Trad*	0.70	0.20	−0.60	−0.80
*E*13	*CS*	*OO*	−1.08	−1.17	−1.17	0.00
(*E*25)	*OOD*	*Trad*	−2.33	−1.33	−1.67	−0.34
*E*15	*CS*	*Trad*	−2.60	−1.50	−1.80	−0.30

Table VII. Changes in overall impression of quality from initial impression `I1` (before starting the review) to final impression `I2` (after finishing the review).

Steps	*Count*	%
≤ -3	4	1.87
−2	12	5.58
−1	48	22.33
0	117	54.42
1	29	13.49
2	5	2.33
≥ 3	0	0.00
$\sum$	215	100
Total negative	64	29.77
Total unchanged	117	54.42
Total positive	34	15.81

The examples in Table VI can be grouped into 4 groups, where the differences in final overall impression (I2) are smaller between the items in a group than between groups (the groups are set apart by horizontal lines in the table).

Note that these groups did not change compared to our previous analysis[1] [Börstler et al. 2009], although we now have 191 data points compared to 122 before. This indicates that our evaluation instrument is quite reliable. This is also corroborated by the high inter-rater agreement (see Table VIII in Section 6).

[1]Except for E10, which moved from the second group (where it was last) to the third group.

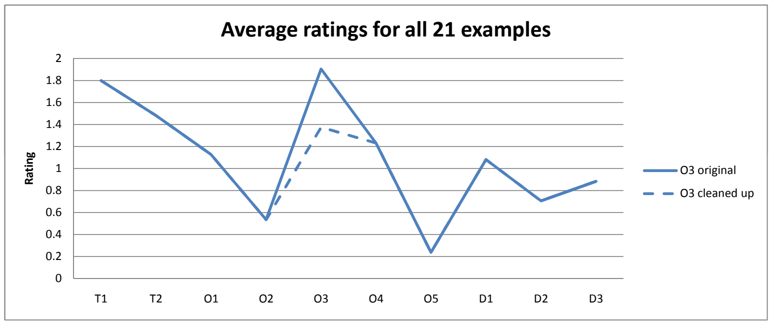


Fig. 2. Average ratings of all quality factors for the 191 reviews.

Figure 2 shows the average ratings for all quality factors. The peak for O3 (Reasonable Class Relationships) might be attributed to our rating instructions to consider this factor as excellent (i.e. +3) "when no relationships are present and the example doesn't call for any". When deleting all examples consisting of a single class only, O3 still has the third largest average (1.38) after the technical quality factors T1 (1.80) and T2 (1.48).

6. DISCUSSION

6.1 Agreement between raters

Reviewers ranked examples in very similar ways, although their absolute ratings could be quite different. The majority of reviewers show a very strong and highly significant correlation with the total average ranking of all reviews (see Table VIII). This strongly indicates that our evaluation instrument is reliable.

The only outliers are reviewers R1 and R23. For R1 the deviation could be a result of this reviewer's different background (see Table III). R1 is the only reviewer with a background in teacher education, whereas all others have a computer science background. Furthermore, R1 is less experienced than most other reviewers. For reviewers R23 and R41, the data includes only 5 and 4 data points, respectively. Their rho-values can therefore be at best be interpreted as indications.

6.2 Differences between Textbook Categories

In our previous work ([Börstler et al. 2009]), we noticed that the examples from textbooks categorized as OO on average scored somewhat higher than the ones from textbooks categorised as Trad. With the extended data in the present analysis this is still true, and the overall difference is still at the same level. Based on their average total score (see Table VI), OO-type book examples have an average rank of 8.2, examples from Trad-type books an average rank of 13. However, one should note that there are examples from both book categories among the examples in the top group as well as among the examples in the bottom group.

Since OO-type books have a clear and early focus on object-orientation (see our definition in Section 4.1) it is not surprising that their examples score higher on a scale that is partly based on object-oriented properties. For Trad-type texts it could be argued that object-oriented qualities are not equally important for all types of

Table VIII. Spearman rank correlation (`rho`) for reviewer's scores and the total average score for all reviews. R23 and R41 had too few reviews to compute meaningful p-values.

Reviewer	*Rho*	*P-value*
$R1$	0.298	0.263
$R2$	0.922	$2.80E$-8
$R23$	−0.290	–
$R4$	0.895	$9.68E$-8
$R41$	0.892	–
$R5$	0.633	0.005
$R6$	0.582	0.011
$R7$	0.899	$4.04E$-7
$R71$	0.904	$1.31E$-4
$R72$	0.876	< 0.01
$R8$	0.883	$5.85E$-6

examples. The three different types of examples (*FUDC: First User-Defined Class*, *OOD: Multiple User Defined Classes*, *CS: Control Structures*) are included in the books to illustrate significantly different concepts. FUDC examples may illustrate classes, objects, attributes, methods and encapsulation, while OOD examples may exemplify any of association, collaboration, message passing, inheritance and polymorphism. The programming concepts in CS examples, on the other hand, are not explicitly object-oriented. We will discuss this issue in more detail in the following sections.

6.3 Dependencies between Quality Factors

Our previous results ([Börstler et al. 2009]) indicated that the different quality factors seem to capture different aspects of quality. There were examples that consistently scored high or low on all quality factors and also examples without consistent scoring patterns.

Our current data shows only weak correlations between quality factors over all examples. The Spearman rank correlation for all quality factors lies in the range $[-0.28, 0.35]$, except for two slightly higher values of 0.48 (O1 vs. O2) and 0.76 (O4 vs. O5) (p-values in the range $[0.01, 0.025]$).

When looking at the quality factors O1–O5, in particular, we can see interesting differences between the example categories (Figure 3). For CS and OOD examples, example rankings are almost the same for all five quality factors. For the category CS, the only deviation is for O3 which is concerning class relationships. Those small examples of control structures rarely include any relationships and that is most likely to considered reasonable. Furthermore, all ratings are close to the average rating for O1–O5[2].

The variation for FUDC examples is much greater than the variation for CS and OOD examples. We can find a similar pattern of variation between ratings for the different example categories for technical quality (T1–T2) and didactic quality (D1–D3), but much less pronounced.

[2]The differing behaviour on O3 is an artefact of our rating instructions as explained at the end of Section 5.

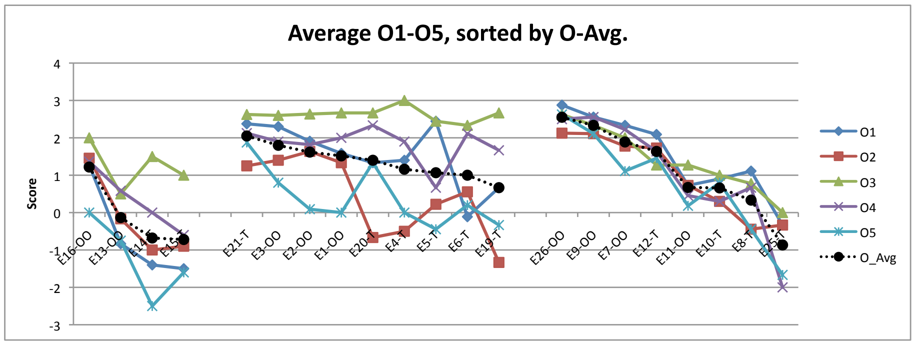


Fig. 3. Average ratings on O1–O5 for all examples, grouped by example category; CS, FUDC, and OOD (from left to right).

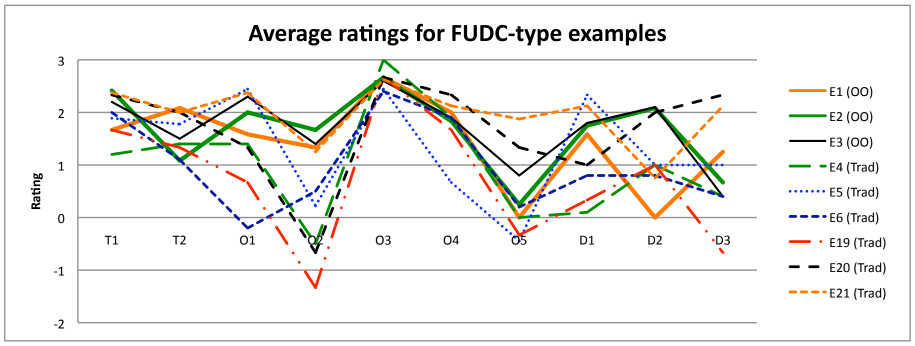


Fig. 4. Average ratings of all examples exemplifying the first user-defined class (FUDC).

We conclude from this analysis that good FUDC examples are more difficult to find or develop than the other types of examples we have looked at. This is hardly surprising, since the demands on a FUDC example seem more difficult to satisfy. It should be a role model for a "good" object-oriented class, i.e. score well on the characteristics captured by O1–O5. However, it should also be small and use as few concepts as possible, since most concepts have not been introduced yet.

Quality factors O1–O5 have been more closely examined by example category in Section 6.4–6.6.

6.4 Object-oriented quality of FUDC-examples

An example of a user-defined class should make it clear why it is necessary to create that class and why it needs to have particular state and behavior. Figure 4 shows the average ratings for O1–O5 for FUDC examples. Here we consider which quality factors indicate how successfully the examples achieve this and what characteristics of the examples may influence ratings for these quality factors. O1 (Reasonable Abstractions) and O2 (Reasonable State and Behaviour) relate closely to these requirements. O1 is likely to be influenced as much by the "cover story" (the text which introduces the context of the example) as by the code and is not considered in

Table IX. Book category , number of fields, default constructor (Def.), number of constructors (Cnstr.), number of methods (Meth.), way(s) in which class is exercised (Ex.) and number of objects instantiated (Obj.), average scores for quality factors O2 and O5, and average total score (TOD) for FUDC examples. Possible values for Ex. are IDE (interactive instantiation in IDE), TC (test class) and TM (test method).

	Book	*Fields*	*Def.*	*Cnstr.*	*Meth.*	*Ex./Obj.*	*O*2	*O*5	*TOD*
*E*2	*OO*	1	*y*	1	1	*IDE*/1	1.64	0.09	16.42
*E*3	*OO*	1	*n*	1	2	*IDE*&*TC*/1	1.40	0.80	17.00
*E*1	*OO*	3	*n*	1	2	*IDE*/2	1.33	0.00	14.17
*E*21	*Trad*	1	*y*	1	1	*TC*/2	1.25	1.88	19.63
*E*6	*Trad*	2	*y*	2	1	*TC*/2	0.56	0.20	9.90
*E*5	*Trad*	4	*n*	1	0	*TM*/1	0.22	−0.44	13.33
*E*4	*Trad*	1	*y*	2	0	*TC*/2	−0.50	0.00	9.90
*E*20	*Trad*	1	*y*	1	0	*TC*/2	−0.67	1.33	16.67
*E*19	*Trad*	1	*n*	1	0	*TC*/2	−1.33	−0.33	7.00

this discussion. O5 (Promotes “Object Thinking”) presents an interesting problem for authors—how can single-class examples promote object thinking? The way in which instances of the class are created and exercised is important here. The data for O3 (Reasonable Class Relationships) has clearly been influenced by the instruction that examples with no relationships should be given score of 3, and this data is not considered here.

Neither is O4 (Exemplary OO code)—this quality factor does not appear to discriminate strongly between the examples and suggests that these are generally free from “code smells”.

The following characteristics of each example were identified by examining the code listings:

—number of fields/attributes – in all examples the fields are appropriately implemented as private with accessors and mutators as required depending on whether the field is read-only or read-write

—provision of a default constructor

—total number of constructors, including the default constructor if provided

—number of methods – this number excludes accessors/mutators and `toString` implementations, and also excludes methods which play one of these roles (for example, a method which simply prints out a message containing the value of a field)

—the way in which the class is exercised – all the examples provide an example of creating and using instances, either in the form of a separate test class or by interactive instantiation in an IDE such as BlueJ or Dr Java. The only exception provides a test method which acts upon an instance of the UDC, but does not provide a full working example to show how the method could be implemented.

Table IX shows the above information for each of the examples together with: book category; average score for O2; average score for O5; average total score. The results are shown in order of average O2 score.

It is strikingly clear from Table IX that the books which we have categorized as OO attain higher scores for O2 than books in the Trad category. This is not the case for O5, however. Interestingly, the books in our OO category all choose to exercise classes interactively in an IDE while the Trad books simply provide test classes.

The 'size' of the class in terms of the number of fields does not appear to influence "Reasonable state and behavior"—the examples with the highest and lowest O2 scores each have a single field. Constructors also have little influence—some reviewers have made negative comments where examples do not include a default constructor, but this is not particularly reflected in the scores. The most significant code feature for O2 appears to be the methods provided. Examples with only accessors/mutators and print operations score poorly, not surprisingly as this issue is clearly identified in the examples of problems given in the checklist. What is perhaps surprising is that four out of the nine examples presented a first user defined class with no meaningful behavior. This may promote misconceptions about the purpose of classes. It is interesting to note that the ranking by O2 score is quite different from the ranking by total score. E20 in particular has a good total score but a very poor O2 score. The comments for that example explicitly identify the limited behavior but excuse this—for example *"perhaps this example does not lend itself to behaviour which can be modeled"*.

The ranking by O5 score is quite different from the ranking by O2, and again class size and constructors do not have any clear influence. The only common factor which can be observed is that examples with only accessors/mutators and print operations score poorly. Scores for "Promoting object thinking" do not appear to be strongly influenced by the way in which classes are exercised. Test classes and interactive instantion in an IDE appear to be equally acceptable, although the lowest score is attained by the example which shows an incomplete test class. There is, though, a wide range in the scores for this quality factor. This suggests that factors other than the code itself are influential here.

It is difficult to draw firm conclusions on what makes an "exemplary" FUDC example. It does appear, perhaps not surprisingly, that examples which illustrate meaningful behavior display greater object-oriented quality than examples which include no or trivial behavior.

6.5 Object-oriented quality of OOD examples

It might be expected that examples in the OOD category should score highly for object-oriented quality. An object-oriented program is in essence a group of collaborating classes, and a well designed example should illustrate collaborations and the programming structures which allow these collaborations to occur. The examples reviewed were generally among the first multiple class examples in each book.

These examples have been examined further to ascertain any clear influence on scores for object-oriented quality of following factors:

—number of classes in the example
—type(s) of relationship represented in the example

The number of classes in these examples ranges from 2 to 6. Some examples exercise the classes using a client class, while others either do not provide a way to exercise the classes, or do so using the capability of the IDE to instantiate and interact with objects. Client classes are not included here in the number of classes listed here.

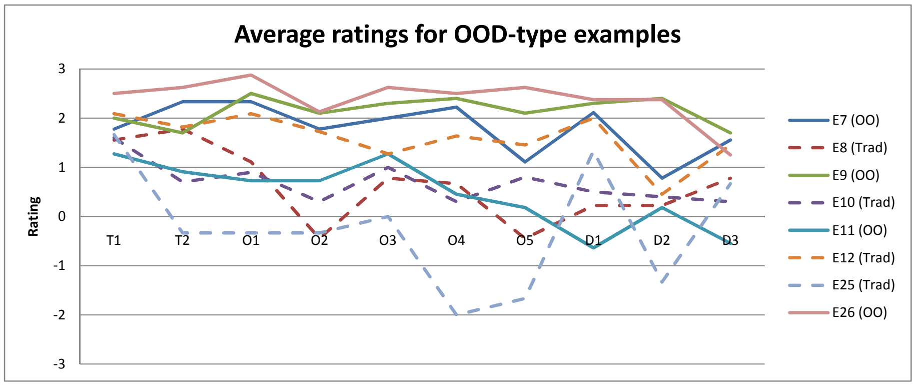


Fig. 5. Average ratings of all examples exemplifying object-oriented design involving several classes (OOD).

A class relationship is a binary association between classes in the example. Each example may include one or more binary associations. The types of relationships represented have been grouped into three categories which we define as:

IN: Inheritance. This relationship indicates specialization, and is realised by the implementation of one class as a subclass of another.

CO: Composition. This relationship indicates containment, and is realized by the presence of a field in one class having a field which holds a reference to one or more instances of another class. This includes relationships commonly described as composition or aggregation, or as a "has-a" relationship.

AS: Association. This indicates a transient relationship, where one class has a reference to one or more instances of another class as a method parameter, method return value or local variable. This includes relationships commonly described as association or dependency, or as a "uses-a" relationship.

Note that under these definitions, a composition and association relationships provides means for objects to collaborate, while inheritance represents a static structural relationship.

Table X shows the following information for each of the examples: number of classes; average score for O3 (Reasonable Class Relationships); average score for O5 (Promotes "Object Thinking"); average total score. Of all the quality factors, O3 and O5 are likely to have been most strongly influenced by the representation of class relationships. O3 is likely to have been considered in the way it was intended to be for these particular examples by reviewers as there clearly are relationships present. The results are shown in order of average O3 score. It is apparent that the examples which score well for object-oriented quality factors also score well overall—it is not only their representation of relationships which makes them good examples.

The number of classes does not appear to have any influence on the results. The highest and lowest rated examples both contain only two classes. It is clearly possible to create examples which provide a useful illustration of the relationship between a pair of classes. Examples with more classes can illustrate a more complex

Table X. Book category, Number of classes (Cl.), class relationship types represented (Rel.), average scores for quality factors related to relationships and average total score (TOD) for OOD examples. **In E10 two additional subclasses are discussed but the code for these is not listed.*

	Book	*Cl.*	*Rel.*	*O3*	*O5*	*TOD*
*E*26	*OO*	2	*CO*	2.63	2.63	23.88
*E*9	*OO*	2	*AS*	2.44	2.22	22.44
*E*7	*OO*	2	*IN*	1.88	1.25	18.38
*E*12	*Trad*	6	*IN, CO*	1.50	1.50	16.50
*E*11	*OO*	3	*IN*	1.30	0.20	4.90
*E*8	*Trad*	2	*IN*	1.00	−0.38	6.63
*E*10	*Trad*	3∗	*IN, CO*	1.00	0.67	6.33
*E*25	*OO*	2	*IN*	0.00	−1.67	−2.33

system of collaborations, but need to be designed carefully so that the details and significance of each collaboration are not lost.

The types of relationships represented does appear to be significant. The two best examples illustrate composition and association respectively, while the lowest rated example illustrates a simple two-class inheritance hierarchy. Examples of composition and association are able to exemplify collaboration between objects at runtime, which is helpful in promoting object-thinking. While inheritance is an important concept, it does not by itself reflect object collaboration. Two of the examples combine inheritance with composition to illustrate polymorphism. It might be expected that showing inheritance in a typical collaborative context would lead to better examples than demonstrating it in isolation, but this is not clearly reflected in the reviewers' scores. The best example of inheritance focuses clearly on the issues of additional state and behavior, in contrast to another example which only considers additional data members in a subclass. The lowest rated example also deals with state and behavior, but comments suggest that it is a rather confusing example. Some reviewers commented that examples of inheritance were based on scenarios which would be better modelled as roles and were therefore not exemplary illustrations. This issue would be reflected in the O5 score and may be significant for E8 in particular.

It is clearly possible to create exemplary examples of multiple collaborating classes, and such examples illustrate fundamental concepts in object-oriented programming. The results suggest that early examples in this category should adopt a clear focus on illustrating a specific type of relationship. Examples which develop more complex structures can then build on the simple examples or can be constructed from smaller clearly focused examples of collaborations.

6.6 Object-oriented quality of CS examples

Figure 6 shows that, with one exception, the control structure examples scored poorly on object-oriented quality. As was the case for FUDC examples, the data for O3 (Reasonable class relationships) has probably been artificially influenced by the instructions given in the checklist and is disregarded.

Control structures are fundamental programming concepts and are not a feature of any particular paradigm. The authors of these books follow quite different

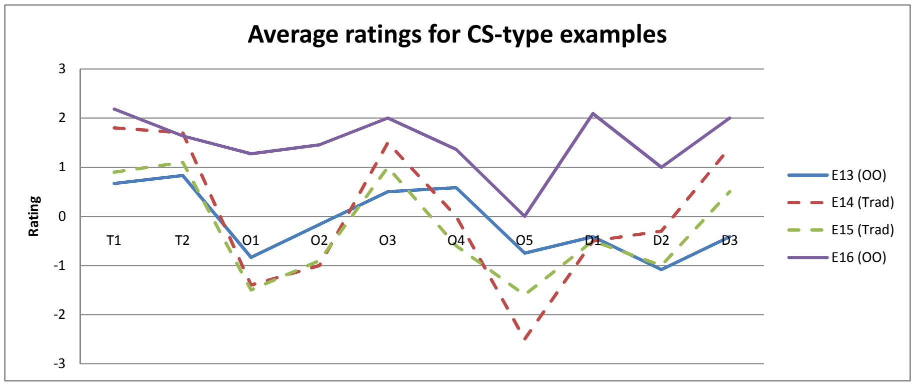


Fig. 6. Average ratings of all examples exemplifying control structures (CS).

approaches to introducing control structures:

—E14 (Trad book category) – shows a single class which consists entirely of a main method. This is an example of the data-less object described by Hu [2005]. It scores poorly, particularly for O5 (Promotes object-thinking).

—E15 (Trad) – a single class which contains a main method which instantiates an instance of the class and calls a “start“ method. One other method is included which is used by the start method. This example also scores poorly—scores are very similar to E14 except for a slightly higher O5 score. The inclusion of the main method in the single class is noted by one reviewer as bad practice, and contrasts with the way in which main methods are used appropriately in test classes in many of the FUDC examples.

—E13 (OO) – presents an if-else structure within a method to handle a mouse button click. It scores poorly, but marginally better for object-oriented quality than E14 and E15. Reviewers comments note the dependence of the example on code presented elsewhere in the book and the complexity of this code.

—E16 (OO) – structured in a similar way to most FUDC examples. It shows a class which represents a clear abstraction and includes relevant state and behavior, and includes a separate test class. The methods provide motivations for the use of control structures. This example received the highest scores among the CS examples for object-oriented quality. The score for O5 is low, although it was noted in section 6.4 that O5 scores may depend strongly on factors other than the example code.

These results suggest that reviewers regard the use of control structures as a way of implementing behavior within meaningful class methods as exemplary in this category.

6.7 Relation to traditional software measures

We also investigated the relationship of total average scores with traditional measures of software quality. In Table XI we have summarized the results from comparing the total average score of our instrument with the measures described below. Most of these measures have been obtained using the measurement tool

Table XI. Spearman rank correlation (Rho) for common software measures and the total average score of our 21 examples.

Measure	*Rho*	*P-value*
Size	−0.080	0.729
Maintainability Index	0.494	0.023
Complexity	−0.138	0.550
Code density	0.557	0.009
Comment density	0.498	0.022

JHawk [JHawk]. Since most of our examples consisted of only a single class, we did not collect any object-oriented measures.

—*Size:* The total lines of code, counting code, comments and empty lines. Although size is a very simple measure it tends to correlate with many software attributes.

—*Maintainability Index (MI):* The Maintainability Index is a measure for predicting the relative effort for software maintainability [Welker et al. 1997]. Since maintainability requires easy to read and to understand code, MI intends to capture these properties.

—*Complexity:* McCabe's cyclomatic complexity measures the number of (statically) distinct paths through a method [McCabe 1976].

—*Code density:* The number of statements divided by Size (see above).

—*Comment density:* The number of comments divided by Size (see above).

Our checklist captures different aspects of quality from traditional measures of software quality. However, since MI, code density and comment density all capture different aspects of readability and understandability, we would expect some kind of relationship to our ratings. Looking at the scatterplots in Figure 7, we can see some indications for such relationships:

—Examples with dense code tend to have worse overall average ratings. The best rated examples tend to have a code density between 0.2 and 0.3. If we look at examples in the two top and two bottom "quartiles" (see Table VI), we can see that 10 of the 11 better examples have a code density below 0.3, whereas 7 of 10 of the worse examples have a code density above 0.3.

—The picture for comment density is somewhat similar. The better examples tend to have more comments, but this relationship is much less pronounced than for code density.

—Although all examples have MI-values above 85, indicating "high maintainability" [Welker et al. 1997], there is a clear difference between the two top and two bottom "quartiles". Nine of the 11 better examples have MI ≥ 130, whereas 7 of 10 of the worse examples are beyond this level.

The actual correlations are, however, only moderate. This indicates that our measure does indeed capture different aspects of quality from traditional software measures.

7. CONCLUSIONS

In this paper we analysed 191 reviews of 21 object-oriented example programs from 11 introductory object-oriented programming texts. The reviews were done

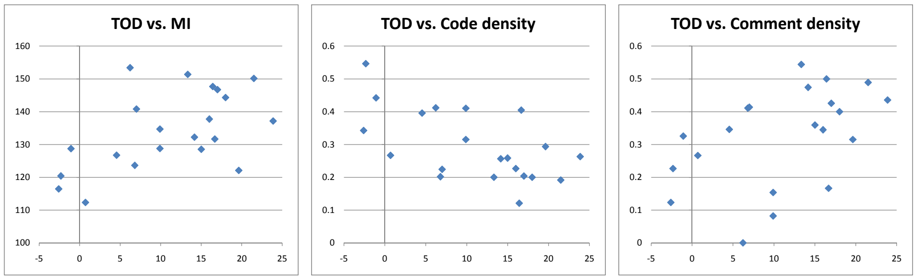


Fig. 7. Scatterplots for the 21 examples' average total scores (TOD) against Maintainability Index (MI), Code density, and Comment density, respectively.

by 11 reviewers from Denmark, Germany, Sweden, the UK, and the USA. The review instrument comprised 10 quality factors, grouped into three quality categories: technical quality (T1–T2), object-oriented quality (O1–O5), and didactic quality (D1–D3).

Our results show that the examples in the analysed sample varied markedly in quality. The object-oriented quality of many examples is not as high as one would expect to find in an introductory programming text. In particular, many examples received low ratings for "object thinking" (O5) and Reasonable state and behaviour (O2). Since examples are the most important tools for learning, these results are alarming. High quality examples are a prerequisite for successfully learning a new skill.

We looked at three different categories of examples (FUDC, OOD, and CS[3]) and two different categories of texts (OO-type and Trad-type[4]).

Our analysis revealed different rating patterns for FUDC examples compared to CS and OOD examples indicating that it is specifically difficult to develop FUDC examples with consistent high ratings for all object-oriented quality factors. Of the 4 CS examples we analysed, 3 are among the bottom 4 examples. This was not surprising, since such examples are said to be impossible to do in an "object-oriented way". However, one of these examples was rated above average, indicating that it actually is possible.

Although examples from OO-type texts receive somewhat higher ratings on average than examples from Trad-type texts, taking an example from an OO-type text is no guarantee for high ratings in object oriented-quality. Examples with high ratings in object-oriented quality can be found in Trad-type textbooks as well as examples with low ratings in OO-type texts.

Our review instrument is highly reliable and measures aspects of quality that are not captured by common size or complexity measures. It can be a useful tool for identifying problems in example programs that might otherwise go unnoticed.

[3]The first example of a user defined class in the text (FUDC), the first example of an object-oriented design featuring multiple classes (OOD), and the first example of control structures (CS).
[4]Texts with a clear and early focus on object-orientation (OO-type) and others (Trad-type).

REFERENCES

Anderson, J., Farrell, R., and Sauers, R. 1984. Learning to program in LISP. *Cognitive Science 8,* 2, 87–129.

Barnes, D. J. and Kölling, M. 2009. *Objects First with Java*, 4th ed. Prentice Hall.

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

Börstler, J., Christensen, H. B., Bennedsen, J., Nordström, M., Westin, L. K., Moström, J. E., and Caspersen, M. E. 2008. Evaluating oo example programs for cs1. In *ITiCSE'08: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education.* 47–52.

Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., and Thomas, L. 2009. An evaluation of object oriented example programs in introductory programming textbooks. *inroads 41,* 4, 126–143.

Bransford, J. D., Brown, A. L., and Cocking, R. R. 2004. *How People Learn, Expanded Edition.* National Academy Press, Washington, D.C., USA.

Bravaco, R. and Simonson, S. 2010. *Java Programming – From the Ground Up*, 1st ed. McGraw-Hill.

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

Brykczynski, B. 1999. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes 24,* 1, 82–89.

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

CACM Forum. 2002. 'Hello, World' gets mixed greetings. *Communications of the ACM 45,* 2, 11–15.

CACM Forum. 2005. For programmers, objects are not the only tools. *Communications of the ACM 48,* 4, 11–12.

Clancy, M. 2004. Misconceptions and attitudes that infere with learning to program. In *Computer Science Education Research*, S. Fincher and M. Petre, Eds. Taylor & Francis, Lisse, The Netherlands, 85–100.

de Raadt, M., Watson, R., and Toleman, M. 2005. Textbooks: Under inspection. Tech. rep., University of Southern Queensland, Department of Maths and Computing, Toowoomba, Australia.

Deitel, H. M. and Deitel, P. J. 2007. *Java – How to Program*, 7th ed. Prentice Hall.

Dodani, M. H. 2003. Hello World! goodbye skills! *Journal of Object Technology 2,* 1, 23–28.

Farrell, J. 2010. *Java Programming*, 5th ed. Thomson.

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

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

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

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

Horstmann, C. S. 2008. *Big Java*, 3rd ed. Wiley.

Hu, C. 2005. Dataless objects considered harmful. *Communications of the ACM 48,* 2, 99–101.

JHawk. Product homepage. `http://www.virtualmachinery.com/jhawkprod.htm`, last visited 2009-11-03.

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

LeFevre, J. and Dixon, P. 1986. Do written instructions need examples? *Cognition and Instruction 3,* 1, 1–30.

Lewis, J. and Loftus, W. 2009. *Java – Software Solutions*, 6th ed. Addison-Wesley.

LimeSurvey. Project homepage. `http://www.limesurvey.org/`, last visited 2009-10-20.

Lin, J. M.-C. and Wu, C.-C. 2007. Suggestions for content selection and presentation in high school computer textbooks. *Computers & Education 48,* 3, 508–521.

Magel, K. 1982. A Theory of Small Program Complexity. *ACM SIGPLAN Notices 17,* 3, 37–45.

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

Malik, D. and Burton, R. P. 2009. *Java Programming – Guided Learning with Early Objects*, 1st ed. Course Technology.

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

McCabe, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering 2,* 4, 308–320.

Niño, J. and Hosch, F. A. 2008. *Introduction to Programming and Object Oriented Design Using Java*, 3rd ed. Wiley.

Nordström, M. 2009. He[d]uristics—heuristics for designing object oriented examples for novices. Ph.D. thesis, Umeå University, Umeå, Sweden.

Ourosoff, N. 2002. Primitive types in Java considered harmful. *Communications of the ACM 45,* 8, 105–106.

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

Reed, S. and Bolstad, C. 1991. Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition 17,* 4, 753–766.

Reimann, P. and Schult, T. J. 1996. Turning examples into cases: Acquiring knowledge structures for analogical problem solving. *Educational Psychologist 31,* 2, 123–132.

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

Riley, D. D. 2006. *The Object of Java*, 2nd ed. Addison-Wesley.

Roberts, E. 2008. *Java – An Introduction to Computer Science*, 2nd ed. Addison-Wesley.

Sanders, K. and Thomas, L. 2007. 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.* 166–170.

Trafton, J. G. and Reiser, B. J. 1993. Studying examples and solving problems: Contributions to skill acquisition. Tech. rep., Naval HCI Research Lab, Washington, DC, USA.

VanDrunen, T. 2006. Java interfaces in CS 1 textbooks. In *OOPSLA Conference Companion – 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications.* 875–880.

Welker, K. D., Oman, P. W., and Atkinson, G. G. 1997. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice 9,* 3, 127–159.

Westfall, R. 2001. 'Hello, World' considered harmful. *Communications of the ACM 44,* 10, 129–130.

Wu, C.-C., Lin, J. M.-C., and Lin, K.-Y. 1999. 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.

Wu, C. T. 2008. *A Comprehensive Introduction to Object-Oriented Programming with Java*, International ed. McGraw-Hill.

Received February 2010; accepted June 2010.