Software Qual J
DOI 10.1007/s11219-015-9267-5

Beauty and the Beast: on the readability of object-oriented example programs

Jürgen Börstler · Michael E. Caspersen · Marie Nordström

Abstract Some solutions to a programming problem are more elegant or more simple than others and thus more understandable for students. We review desirable properties of example programs from a cognitive and a measurement point of view. Certain cognitive aspects of example programs are captured by common software measures, but they are not sufficient to capture a key aspect of understandability: readability. We propose and discuss a simple readability measure for software, SRES, and apply it to object-oriented textbook examples. Our results show that readability measures correlate well with human perceptions of quality. Compared with other readability measures, SRES is less sensitive to commenting and whitespace. These results also have implications for software maintainability measures.

Keywords Object-oriented programming · Quality · Measurement · Software readability · Programming education

1 Introduction

Simplicity and understandability are essential properties of example programs. In simple and understandable programs, it is easier to identify the essential elements (Watson and

J. Börstler (✉)
Blekinge Institute of Technology, Karlskrona, Sweden
e-mail: jubo@acm.org

M. E. Caspersen
Aarhus University, Aarhus, Denmark
e-mail: mec@cse.au.dk

M. Nordström
Umeå University, Umeå, Sweden
e-mail: marie@cs.umu.se

Published online: 14 February 2015

Mason 2002). Such programs are also easier to comprehend, since extraneous cognitive load is reduced (VanLehn 1996). Simplicity and understandability are also important properties for production code. About 50 % of software maintenance costs are spent on code comprehension (Foster 1993; Nguyen 2010; Yip and Lam 1994) and more than 40% of the comprehension time is spent on plain code reading alone (LaToza et al. 2006). Readability is therefore an important quality aspect for learning to program as well as for software maintenance.

In this paper, we look at simplicity and understandability from a cognitive and from a measurement point of view.

We start by looking at the role of examples in education (Sect. 2) and at properties of good example programs (Sect. 3). In Sect. 4, we exemplify various aspects of these properties using two example implementations of a class `Date`: “The Beauty” and “The Beast”. In Sect. 5, we then develop a formula to quantify readability, which is a necessary condition for understandability. This formula is inspired by formulas for measuring readability of ordinary text (see Sect. 5.1). In Sect. 6, we apply the formula to 21 examples from popular introductory Java textbooks. The results are then compared with other readability measures, traditional software measures and human assessments of quality. In Sect. 7, we wrap up with conclusions and directions for future work.

Parts of the work presented here are based on a technical report by the authors (Börstler et al. 2007) that has been updated significantly and extended with a case study.

2 Examples in education

Examples are important teaching tools. Research in cognitive science confirms that “examples appear to play a central role in the early phases of cognitive skill acquisition” (VanLehn 1996). An alternation of worked examples and problems increases the learning outcome compared with just solving more problems (Sweller and Cooper 1985; Trafton and Reiser 1993).

Students generalize from examples and use them as templates for their own work (Liz et al. 2006; Mason and Pimm 1984; Reimann and Schult 1996). Examples must therefore be easy to generalize and consistent with current learning goals. Furthermore, they should also be consistent with the principles and rules of the subject and free of any undesirable properties or behavior. The “[c]hoice of examples is important in helping students develop generalisations of structures rather than surface features” (Watson and Mason 2002).

By perpetually exposing students to “exemplary” examples, desirable properties are reinforced many times. Students will eventually recognize patterns of “good” design and gain experience in telling desirable from undesirable properties. Trafton and Reiser (1993) note that in complex problem spaces, “[l]earners may learn more by solving problems with the guidance of some examples than solving more problems without the guidance of examples”.

With carefully developed examples, teachers can minimize the risk of misinterpretations and erroneous conclusions, which otherwise can lead to misconceptions. Once established, misconceptions can be difficult to resolve and hinder students in their further learning (Clancy 2004; Ragonis and Ben-Ari 2005).

An important question is therefore how to tell “good” examples from “bad” ones and whether example quality can be measured objectively.

3 Properties of good examples

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand. [M. Fowler]

Programming is a human activity, often done in teams. About 40–70 % of the total software lifecycle costs can be attributed to maintenance, and the single most important cost factor of maintenance is program understanding (Glass 2003; Tryggeseth 1997). Understandability is therefore a significant property even for production code. In an educational context, understandability is even more important. When novice students are still learning how to write simple programs that a computer can understand, there is little room to deal with program understanding by humans.

A good example must obviously be *understandable by a computer*. Otherwise, it cannot be used on a computer and would therefore not be a real programming example.

A good example must also be *understandable by students*. Otherwise, they cannot construct an effective mental model of the program, and it becomes even more difficult to learn how to write a program that a computer can understand. Without "understanding", knowledge retrieval works only on an example's surface properties, instead of on its underlying structural and conceptual properties (Deimel and Naveda 1990; Trafton and Reiser 1993; VanLehn 1996).

A good example must furthermore *effectively communicate the concept(s) to be taught*. There should be no doubt about what exactly is exemplified. The structural form of information affects the form of the knowledge encoded in human memory (Tennyson and Cocchiarella 1986). Conceptual knowledge is improved by *best examples* and by expository examples, where the *best example* represents an average, central, or prototypical form of a concept. To minimize cognitive load (Paas et al. 2003), an example should furthermore exemplify only one new concept (or very few) at a time.

The "goodness" of an example also depends on "external" factors, like the pedagogical approach taken. E.g., when the main learning goal is proficiency in object-oriented programming (in terms of concepts, not specific syntax), examples should always be truthfully object-oriented and "exemplary", i.e., they should adhere to accepted design principles and rules and not show any signs of "code smells"[1] (Fowler 1999; Martin 1993; Riel 1996). If examples are not consistently truthfully object-oriented, students will have difficulties picking up the underlying concepts, principles, and rules.

The three example properties (1) understandable by a computer, (2) understandable by students, and (3) effectively communicate the concept(s) to be taught might seem obvious. However, the recurring discussions about the harmfulness or not of certain common examples show that there is quite some disagreement in the teaching community about the meaning of these properties (Ben-Ari 2010; CACM 2002; Dodani 2003; Westfall 2001).

4 One problem, two solutions

Assume the following two implementations of a class `Date` capable of creating date objects and advancing dates one day at a time. The Beauty and the Beast exemplify design choices for object-oriented example programs targeted at novices. The examples could be

[1] According to Fowler (1999), a code smell is an indication on the code's surface level of a potential problem at a deeper level. It is subtle and easy to spot, but not necessarily an actual problem or defect.

easily criticized for exhibiting undesirable properties, like, for example, the possibility of creating invalid dates, insufficient comments, or their lack of functionality. Solving all these “shortcomings” would arguably lead to better programs, but not necessarily to ones that are more easily understood by novices. Adding more concepts, properties or information increases cognitive load and will therefore decrease understandability (VanLehn 1996). On the other hand, example programs should be sufficiently complete and consistent to make sense (Mayer 2004).

The main difference between the Beauty and the Beast is in the way the problem is decomposed into “components”. Software decomposition is an important factor in managing complexity and affects comprehension and maintenance positively (Woodfield et al. 1981). High degrees of decomposition furthermore decrease cognitive load and support chunking (Cant et al. 1995; Clark et al. 2006; Shaft and Vessey 2006). This supports independent and incremental comprehension, development, and test, which is particularly important from a teaching and learning point of view.

The Beauty is beautiful because there is an explicit representation of key concepts in the problem domain. These can work as cues (so-called beacons) aiding in code comprehension (Gellenbeck and Cook 1991). The components of the solution are also kept simple with a recognizable distribution of responsibilities.

From the process point of view, the Beauty gives cues of how one could compose a complex program from simple components, focusing on one component at a time. The Beauty presented below only considers functional components. We could also use explicit data components, like a static array for the number of days per month or separate (private or public) classes for encapsulating day, month, and year, respectively. We have not done so, though, to keep the number of language concepts in both examples on a comparable level.

```
public class Date_Beauty {
  private int day;
  private int month;
  private int year;

  public Date_Beauty(int y, int m, int d) {
    day = d;
    month = m;
    year = y;
  }

  public void setToNextDay() {
    int daysThisMonth;

    day = this.day + 1;
    daysThisMonth = daysInMonth();

    if ( day > daysThisMonth ) {
      day = 1;
      month = month + 1;
    }
    if ( month > 12 ) {
      month = 1;
      year = year + 1;
    }
  } // setToNextDay

  private int daysInMonth() {
    if ( month == 2 ) {
      if ( isLeapYear() )
        return 29;
      else
        return 28;
    }

    if ( month == 4 || month == 6 ||
         month == 9 || month == 11 )
      return 30;
    else
      return 31;
  } // daysInMonth

  private boolean isLeapYear() {
    return ( isMultipleOf(4) && !isMultipleOf(100) )
           || isMultipleOf(400);
  } // isLeapYear

  private boolean isMultipleOf(int a) {
    return ( year % a ) == 0;
  } // isMultipleOf

} // Date_Beauty
```

The Beast is structured as one monolithic method. All necessary information is contained in a single statement sequence. Although this solution is smaller than the Beauty, it is more difficult to get the full picture. The Beast shows no signs of “work units” or “chunks” with single responsibilities that can be understood independently. That makes it difficult to deconstruct the program and find appropriate starting points for a code comprehension effort. The Beast is also highly nested. Students have to keep track of many conditions at the same time, which significantly increases cognitive load (Paas et al. 2003).

```
class Date_Beast {
  private int day;     // 1 <= day <= days in month
  private int month;   // 1 <= month <= 12
  private int year;

  public Date_Beast(int y, int m, int d) {
    day = d;
    month = m;
    year = y;
  }

  public void setToNextDay() {
    int daysInMonth;
    if ( month == 1 || month == 3 ||
         month == 5 || month == 7 ||
         month == 8 || month == 10 ||
         month == 12 ) {
      daysInMonth = 31;

    } else
      if ( month == 4 || month == 6 ||
           month == 9 || month == 11 ) {
        daysInMonth = 30;
    } else {
      if ( (year%4 == 0 && year%100 != 0)
           || (year%400 == 0) ) {
        daysInMonth = 29;

      } else {
        daysInMonth = 28;
      }
    }
    day = day + 1;

    if ( day > daysInMonth ) {
      day = 1;
      month = month + 1;

      if ( month > 12 ) {
        month = 1;
        year = year + 1;
      }
    }
  } // setToNextDay()

} // Date_Beast
```

Large, monolithic units of code are difficult to understand. To support understanding, software should be decomposed into suitable components with limited responsibility for the overall task. Such decomposition will lead to a more complex design. One could say that in the Beauty the complexity (and thinking) went into the design. As a result, the units of code became simple. In the Beast the design is trivial, but the code is more complex.

From an instructional design point of view, the Beauty has a big advantage over the Beast. In the Beauty, the teacher can provide the difficult part (i.e., the design) and leave the simpler part (i.e., the actual coding) to the students. In the Beast, on the other hand, the design is trivial, which leaves the difficult part to the students. If teachers do not consistently provide students with good role models of design, they will never be able to recognize patterns of "good" design. With bad (or no) designs, the students will always be left with unnecessarily difficult tasks. The Beauty is therefore not only superior in structure, it is also superior from a learning theoretic point of view.

Small units reduce cognitive load (Clark et al. 2006; Paas et al. 2003), structural similarities support the recognition of programming plans or patterns (Burkhardt et al. 2002; Trafton and Reiser 1993; VanLehn 1996), and the frequent appearance of mnemonic names helps to give meaning to program elements (Deimel and Naveda 1990; Gellenbeck and Cook 1991).

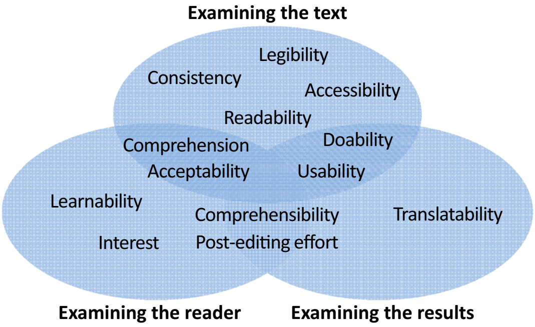

Fig. 1 A characterization of aspects of readability measures for text, according to Cadwell (2008)

5 The importance of readability

Text understanding depends on many factors, like, for example, familiarity with the vocabulary, sentence structure, or reader interest in the text's subject (DuBay 2004). There are therefore many ways to characterize or capture readability, see Fig. 1.

A basic prerequisite for understandability is readability. The basic syntactical elements must be easy to spot and easy to recognize. Only then, one can establish relationships between the elements. And only when meaningful relationships can be established, one can make sense of a text.

Like for ordinary text, understandability is an important quality factor for software. Readability is therefore an important cost factor in software development and maintenance, as described earlier. A good overview over code readability issues is presented in (Deimel and Naveda 1990).

5.1 The Flesch Reading Ease Score

The Flesch Reading Ease Score (FRES) is a measure of readability of ordinary text (DuBay 2004; Flesch 1948). Based on the average sentence length (ASL—words/sentences) and the average word length (AWL—syllables/words), FRES (see Eq. 1) computes a score for the readability of a text on a scale 0..100. Texts with $\text{FRES} < 30$ are considered very difficult and $\text{FRES} > 90$ are considered very easy.

$$\text{FRES} = 206.835 - 1.015\text{ASL} - 84.6\text{AWL} \tag{1}$$

Lower values of the ratios ASL and AWL indicate more easy-to-read text, and higher values indicate more difficult to read text. I.e., the shorter the sentences and the shorter the words in a text, the easier the text is to read. One should note that the FRES does not say anything per se about understandability. The FRES is just concerned with the "parsing" of a text. Flesch's work was quite influential and has been applied successfully to many types of texts. There are also measures for other languages than English.

Flesch's and similar readability formulas have been criticized heavily for not taking into account important factors, such as properties of the vocabulary, text cohesion and coherence, or sentence structure. Recent research, though, shows that including such factors does not necessarily lead to better predictors for readability (Benjamin 2012). "The [readability] formulas have survived 80 years of intensive application, investigation, and controversy, with both their credentials and limitations remaining intact" (DuBay 2004).

5.2 A Reading Ease Score for software

Following the idea of Flesch, a Software Readability Ease Score (SRES) could be defined by interpreting the lexemes of a programming language as syllables, its statements as words, and its units of abstraction as sentences. One could then argue that the smaller the average word length and the average sentence length, the easier it is to recognize relevant units of understanding—so-called "chunks" (Clark et al. 2006; Gobet et al. 2001; Miller 1956; Paas et al. 2003).

A chunk is a grouping or organization of information, a unit of understanding. Chunking is the process of reorganizing information from many low level "bits" of information into fewer chunks with many "bits" of information (Miller 1956). Chunking is a cognitive abstraction process that helps humans to manage complexity. Since abstraction is a key computing/programming concept (Armstrong 2006; Kramer 2007), proper chunking is highly relevant for the understanding of programming examples.

To measure SRES, the average word length (AWL) and average sentence length (ASL) was computed as follows (see (Abbas 2010) for the detailed counting rules): AWL corresponds to the average length of lexemes (identifiers, keywords, and symbols) in number of characters (note that the Flesch reading ease score uses average number of syllables) and ASL corresponds to the average number of words per statement or block (delimited by curly brackets and semicolons).

In common text readability formulas, AWL is given a higher weight (by about a factor of 10), since word length is considered more important than sentence length for the readability of regular text. For program text, we argue that the situation is different. AWL should be reasonably high, since longer identifiers can carry more meaning (Liblit et al. 2006). Research also shows that longer identifiers support program comprehension (Lawrie et al. 2006). ASL should be short though, since it is the factor that is most important for chunking. The authors would furthermore argue that ASL is more important than AWL. Since most words are familiar, but the grammar unfamiliar, reading the "sentences" of a program is more difficult that reading its "words".

This argumentation leads to the following candidate formula for SRES, where smaller values indicate higher readability:

$$\mathrm{SRES} = \mathrm{ASL} - 0.1\mathrm{AWL} \tag{2}$$

There is more to software readability than AWL and ASL, like, for example, indentation or commenting. In SRES, like in traditional measures for text readability, we focus on inherent properties that cannot be easily changed without restructuring the code. Like in text readability, which distinguishes readability and legibility,[2] we want to capture readability independently from aspects or code features that are purely representational. Hargis (2000) points out that "[r]eadability depends on things that affect the readers' eyes and minds. Type size, type style, and leading affect the eye. Sentence structure and length, vocabulary, and organization affect the mind." The properties that "affect the eye" can be changed in most modern editors. SRES focuses on the properties that cannot be easily changed without restructuring the code.

5.3 Other measures of software readability

Buse and Weimer (2010) (B&W) propose a readability measure based on human ratings of the perceived readability of small code snippets (7.7 lines of code on average) extracted from production code. Their measure is based on 25 code features like number and length of identifiers, counts of various syntactical elements and structures, line length, and indentation. Their readability measure correlates well with code changes, automated defect reports, and defect log messages.

Posnett et al. (2011) reevaluated the B&W-measure and identified several weaknesses in its statistical modeling. They also propose a simpler readability model that outperforms the B&W-model on the original dataset:

[2] DuBay (2004) defines readability as "what makes some texts easier to read than others. It is often confused with **legibility**, which concerns typeface and layout."

$$\mathrm{PHD} = 8.87 - 0.033V + 0.40\mathrm{Lines} - 1.5\mathrm{Entropy}, \tag{3}$$

where V is Halstead's Volume, Lines the number of snippet lines (incl. comments and whitespace), and Entropy the relative distribution of the tokens/characters in the code.

5.4 Other related measures

Names account for up to 70% of the source code (Deißenböck and Pizka 2006; Haiduc and Marcus 2008) and are an important element for the understandability of code (Butler et al. 2010; Liblit et al. 2006; Salviulo and Scanniello 2014). Research also shows that tools can help developers to define better and more consistent names (De Lucia et al. 2011; Relf 2005). However, there is little research on measures for software readability that incorporates linguistic information beyond Halstead's software science (Halstead 1975).

Marcus et al. (2008) propose a measure for conceptual cohesion of classes (C3) that takes into account the information encoded in source code comments and identifiers. C3 complements traditional structural cohesion metrics by capturing readability in terms of "the textual coherence of a class within the context of the entire system". They show that C3 can improve the prediction of faulty classes when used together with traditional structural cohesion measures.

Høst and Østvold (2009) propose a method to identify "naming bugs", which they use to measure the percentage of potentially unsuitable names in software. Liblit et al. (2006) give a thorough overview over the cognitive perspectives on naming issues in software that affect understanding, but none of this work has resulted in actual readability measures.

6 Applying SRES

In the following subsections, we first compare SRES with common software quality measures using our example programs from Sect. 4. In Sect. 6.2, we compare SRES with some of the readability measures discussed in Sect. 5.3 using object-oriented textbook

Table 1 Candidate software measures

Acronym	Description
SRES	The software reading ease score as described in Sect. 5.2
B&W	The software readability measure defined by Buse and Weimer (2010), see Sect. 5.3
PHD	The software readability measure defined by Posnett et al. (2011), see Sect. 5.3
CC	The average McCabe cyclomatic complexity per method; the control flow complexity in terms of the number of (statically) distinct paths through a method (McCabe 1976)
E	Halstead's Effort; the (mental) effort required to construct a program, based on its number of operators and operands (Halstead 1975)
MI	Maintainability Index; a measure for predicting software maintainability (Welker et al. 1997)
CC/NOS	Average control flow complexity per Java statement
NOS	Number of Java statements
LOC	Lines of code, including comment lines and empty lines
CD	Comment density. Lines of comments per statement

examples that have been studied in the literature (Börstler et al. 2009, 2011). The results from these comparisons are discussed in Sect. 6.3.

6.1 The Beauty and the Beast

Table 2 summarizes the measurement results for the Beauty and the Beast presented in Sect. 4, which we complemented with three other versions of the Beauty: Beauty-array, using an array for storing the days per month; Beauty-switch, using a switch-statement instead of a nested if in method `daysInMonth`; and Beauty-private, using a private class each to model day, month, and year.

In addition to the SRES measure and the readability measures described in Sect. 5.3, the example programs are compared along a range of other measures that have been selected for their reported importance for software quality in general or readability and understandability in particular (see Table 1).

CC, E, and NOS have been chosen for their high predictive value in program comprehension tests and software maintenance tasks, in particular for small programs (Curtis et al. 1979; Fitzsimmons and Love 1978). A systematic review by Jabangwe (2014) shows that CC and size (LOC) have been widely and successfully used for measuring maintainability and reliability. MI is a composite measure to quantify the maintainability of software systems (Welker et al. 1997) and has also been used widely (Welker 2001). CC/NOS has been suggested by Lanza et al. (2005) as a simple composite measure for characterizing the quality of object-oriented software. CD has been added to capture the potential effect of the amount of comments on software readability.

All measures, except SRES, B&W, PHD, and LOC have been computed using the measurement tool JHawk v5.1.[3] SRES is computed with Equation 2, using measurements for word data and sentence data taken with Pogje (Abbas 2010).[4] The measurement values for B&W and PHD are computed using software provided by the authors of (Buse and Weimer 2010)[5] and (Posnett et al. 2011)[6], respectively. LOC was counted manually.

Our measurements show that the Beast ranks as the worst version in all measures, except for PHD and measures of size (NOS and LOC). Our data also show that readability measures like SRES, B&W, and PHD might be useful in assessing example program quality. This will be investigated in a case study in the next section.

6.2 A case study

In (Börstler et al. 2009, 2011), the authors report on a study on the quality of object-oriented example programs. Example programs were taken from corresponding chapters of 12 popular introductory Java programming textbooks and were 24–194 LOC in length (86.4 LOC on average). The examples were evaluated by 11 experienced programming educators from 5 countries with over 10 years of subject-specific teaching experience on average. More than half of the reviewers also had professional experience as software developers. They evaluated example quality from a technical, object-oriented and didactic

[3] http://www.virtualmachinery.com/jhawkprod.htm.

[4] Pogje as well (Abbas 2010) can be downloaded from http://www.bth.se/com/jub.nsf.

[5] Available from http://www.arrestedcomputing.com/readability/.

[6] Available from https://github.com/darylposnett/readability.

Table 2 Measurement results for the Beauty and the Beast for the candidate measures in Table 1

Example	SRES <	B&W >	PHD >	CC <	E <	MI >	CC/NOS <	NOS <	LOC <	CD >
Beauty	4.22	0.0175	−3.63	2.0	4,640.51	121.40	0.077	26	52	0.154
Beast	**5.04**	**0.0006**	−6.11	**3.5**	**12,655.31**	**99.23**	**0.135**	26	47	**0.115**
Beauty-array	4.53	0.0026	−8.34	1.6	5,572.42	120.82	0.059	27	47	0.185
Beauty-switch	2.99	0.0247	−4.52	2.2	5,770.17	117.68	0.069	32	58	0.156
Beauty-private	4.81	0.0013	**−10.52**	1.27	6,879.51	126.37	0.027	**47**	**81**	0.128

The "worst" value for each measure is in boldface. A < (>) indicates that lower (higher) values are "better"

Table 3 Additional measures used in case study

Acronym	Description
TOD	Human perception of quality of object-oriented example programs (Börstler et al. 2011)

The remaining measures are listed in Table 1

point of view. The overall quality score was captured by measure TOD (see Table 3). The TOD ratings for the example programs showed high inter-rater agreement and can therefore be considered reliable. In the following, we compare those ratings with the measures for software readability discussed in Sects. 5.2 and 5.3.

For this case study, we took measurements for SRES (Sect. 5.2), B&W (Buse and Weimer 2010), PHD (Posnett et al. 2011), MI, CC, TOD (Börstler et al. 2011), NOS, and CD for the following programs: the Beauty, the Beast, and the example programs E1..E26 from (Börstler et al. 2011).[7] The values for TOD are taken from (Börstler et al. 2011). The other measurement values have been obtained using the same tools as described in Sect. 6.1. CC and LOC have not been included in this case study. Since commenting and usage of whitespace varies significantly between example programs, NOS gives a more accurate value for size. Control flow complexity is captured by CC/NOS.

Table 4 summarizes the Spearman rank correlations for all measures. A summary of the actual measurement values, including TOD, can be found in Table 6 in the appendix. Table 4 shows that the readability measures SRES and PHD correlate strongly and significantly with E, MI, TOD, and each other. PHD also correlates strongly and significantly with CD. It can be noted that none of the traditional measures correlates significantly with TOD. B&W shows no significant correlation with any of the other measures. The differences and similarities between the readability measures are discussed in more detail in the next section.

6.3 Discussion

Our measures for the Beauty and the Beast examples (see Table 2) show a large variation. Although the measures focus on different quality aspects of a program, they show that programs with higher degrees of decomposition (i.e., the different versions of the Beauty) consistently have better values, except for PHD and measures of size (NOS and LOC). The

[7] A list of references to the source code of E1..E26 can be downloaded from http://www.bth.se/com/jub.nsf.

Table 4 Spearman's rank correlations for measurement data in Table 6 (see "Appendix")

	B&W	PHD	E	MI	CC/NOS	TOD	NOS	CD
SRES	−0.018	−0.633**	0.788**	−0.860**	−0.246	−0.558*	0.450	−0.345
B&W		−0.029	−0.128	−0.123	0.189	0.212	−0.328	−0.008
PHD			−0.640**	0.762**	0.002	0.711**	−0.324	0.808**
E				−0.840**	−0.572*	−0.372	0.824**	−0.382
MI					0.129	0.471	−0.443	0.545*
CC/NOS						0.024	−0.855**	−0.195
TOD							−0.120	0.683**
NOS								−0.100

* Significant at $\alpha < 0.01$

** Significant at $\alpha < 0.001$

measures also show that higher degrees of decomposition not necessarily lead to larger programs. In fact, some versions of the Beauty are actually as small as the Beast, which has no decomposition at all.

It should be noted, though, that SRES attempts to measure readability, not size. Longer programs take longer to read, but that does not necessarily mean that they are more difficult to read. Our data shows neither a significant correlation between size (NOS) and any of the readability measures (SRES, B&W, and PHD), nor does it show a significant correlation between NOS and TOD.

This contradicts Posnett et al. (2011)'s observation that "the number of lines in the snippet is positively associated with readability". One should note, though, that the snippets in this dataset were only 7.7 lines long on average and did not contain declarations. The observation might therefore be an artifact of the experimental set-up.

Our case study data corroborate Posnett et al's critique on the B&W-measure, which only shows very weak correlations with the other measures in Table 6. SRES and PHD, on the other hand, show significant and moderate to strong correlations with E, MI, TOD, and each other.

Our data also corroborate earlier results showing that Halstead's Effort (E) correlates well with size and complexity (Curtis et al. 1979; Fitzsimmons and Love 1978).

A major difference between SRES on one hand and B&W and PHD on the other is that the latter two are based on empirical data from the perceived readability of very small code snippets that did not contain declarations. A significant part of object-oriented programs are declarations, though. The examples used in our case study are complete classes containing declarations and are therefore more realistic.

PHD is the only readability measure that shows a strong and significant correlation with comment density (CD). Many examples in the case study are heavily commented. Over 40% of the examples have more comment lines than Java statements (CD > 1, see Table 6). Since PHD considers *Lines* as a factor in its computation (see Eq. 3), it is also sensitive to commenting. When recomputing PHD without factor *Lines* (see PHD2 in Table 5), it neither correlates significantly with CD nor does it with TOD. However, it still correlates with SRES, but not as strongly and significantly as before (−0.633** in Table 4). Its correlation with NOS becomes almost 1. This seems to corroborate that the B&W-measure is very sensitive to size Posnett et al. (2011).

Table 5 Spearman's rank correlations for PHD measure without factor *Lines* (PHD2)

	SRES	B&W	PHD	E	MI	CC/NOS	TOD	NOS	CD
PHD2	−0.577*	0.247	0.446*	−0.858**	0.550*	−0.759**	0.195	−0.939**	0.118

* Significant at $\alpha < 0.01$

** Significant at $\alpha < 0.001$

Besides the *Lines*-factor, SRES and PHD are based on the same underlying ideas as common text readability scores: The properties and the distribution of "tokens" in a program (text) are the key factors that affect its readability. Whereas SRES considers average word and sentence lengths as key factors, PHD is based on token entropy and V, which are both forms of token distribution. Entropy measures have also been used successfully in other areas in the software measurement literature (Arnaoudova et al. 2010; Marcus et al. 2008). The factor *Lines* in the formula for PHD (see Equation 3) makes PHD very sensitive to commenting and whitespace, though. It is therefore more sensitive to legibility issues ("things that affect the readers' eyes" (Hargis 2000), see Sect. 5.1) as SRES, which tries to capture inherent factors of software readability ("things that affect the readers' … minds" (Hargis 2000)).

7 Conclusion and future work

In this paper, we have discussed quality aspects of (object-oriented) example programs. Research from the learning sciences supports that example quality affects learning. To measure a key aspect of example program quality–readability, a simple measure, the Software Readability Ease Score (SRES), has been proposed and compared with other readability measures as well as traditional software measures.

SRES correlates well with Halstead's Effort E, the Maintainability Index MI, and the quality of object-oriented example programs as perceived by human experts (TOD). SRES might therefore be a useful tool for helping educators in the selection and development of suitable example programs.

Since SRES correlates well with the quality of textbook examples, it would be interesting to investigate the relationship between SRES and the quality of programs written by students. Such a measure could for example be integrated into teaching tools to give students immediate feedback on certain qualities of their programs.

Since software readability also is an important factor for software maintenance in general, it would also be interesting to investigate the utility of SRES in predicting various aspects of software maintainability.

Although, the current version of SRES is quite "crude", the results presented here are promising. SRES performs as well or better as other code readability measures (B&W (Buse and Weimer 2010) and PHD (Posnett et al. 2011)) on common Java textbook examples. It is also less sensitive to legibility issues like commenting and whitespace. Complementing SRES with measures that take into account further aspects that might affect software readability and understandability, like, for example, token entropy (as in PHD) or identifier naming issues could improve its utility. More research is, however, necessary.

Appendix: Measurement results

Table 6 Measurement results for the Beauty, the Beast and the example programs E1..E26 from (Börstler et al. 2011)

Example	SRES	B&W	PHD	PHD2	E	MI	CC/NOS	TOD	NOS	CD
	<	>	>	>	<	>	<	>	<	>
Beauty	4.22	0.0175	−3.63	−24.43	4,641	121.40	0.077	–	26	0.154
Beast	5.04	0.0006	−6.11	−24.94	12,655	99.23	0.135	–	26	0.115
E1	4.70	0.0002	15.60	−15.60	3,056	126.60	0.043	14.17	23	1.609
E2	3.61	0.4754	16.60	−6.60	1,140	132.89	0.091	16.42	11	2.636
E3	3.02	0.0084	15.90	−5.70	932	141.89	0.071	17.00	14	1.643
E4	4.40	0.0935	8.47	−20.73	6,354	124.72	0.037	9.90	27	0.704
E5	2.90	0.0	21.61	−14.39	1,406	142.09	0.043	13.33	23	2.130
E6	3.70	0.2239	3.76	−11.84	2,269	127.58	0.057	9.90	21	0.286
E7	3.31	0.8499	8.03	−1.97	501	134.31	0.143	18.00	7	1.429
E8	2.40	0.3853	5.03	−8.57	735	143.33	0.059	6.22	17	0.0
E9	2.50	0.4654	14.64	−22.96	1,252	135.50	0.048	21.50	21	2.190
E10	5.50	0.4933	−1.72	−51.32	12,135	110.61	0.031	6.80	39	0.872
E11	6.60	0.2589	−5.32	−78.12	28,869	119.02	0.015	4.55	74	0.811
E12	4.50	0.2786	6.47	−68.33	15,147	125.76	0.013	16.00	84	1.179
E13	5.27	0.0	−0.96	−18.16	2,762	126.13	0.066	−1.08	19	0.579
E14	7.60	0.9960	2.07	−9.93	4,143	106.76	0.222	0.70	9	0.333
E15	5.45	0.0442	−1.09	−30.29	9,599	112.76	0.054	−2.60	28	0.179
E16	4.50	0.3277	5.75	−29.85	5,829	121.40	0.039	15.00	34	0.941
E19	4.30	0.0553	7.25	−15.95	2,100	132.50	0.063	7.00	16	1.375
E20	3.50	0.7231	3.52	−13.28	3,885	121.31	0.050	16.67	20	0.350
E21	5.60	0.4267	6.17	−30.63	11,781	118.99	0.031	19.63	32	0.969
E25	5.60	0.0	−25.20	−102.80	41,031	118.74	0.019	−2.33	103	0.427
E26	3.50	0.0	17.69	−44.71	3,734	133.41	0.030	23.88	43	1.651

TOD-measures for the Beauty and the Beast are not available. A < (>) indicates that lower (higher) values are "better"

References

Abbas, N. (2010). *Properties of "good" java examples*. Master thesis, Umeå University, Umeå, Sweden.

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

Arnaoudova, V., Eshkevari, L., Oliveto, R., Gueheneuc, Y. G., & Antoniol, G. (2010). Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *Proceedings of the 26th IEEE international conference on software maintenance*.

Ben-Ari, M. (2010). Objects never?: Well, hardly ever!. *Communications of the ACM*, *53*(9), 32–35.

Benjamin, R. G. (2012). Reconstructing readability: Recent developments and recommendations in the analysis of text difficulty. *Educational Psychology Review*, *24*(1), 63–88.

Börstler, J., Caspersen, M. E., & 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., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., et al. (2009). An evaluation of object oriented example programs in introductory programming textbooks. *Inroads*, *41*, 126–143.

Börstler, J., Nordström, M., & Paterson, J. H. (2011). On the quality of examples in introductory java textbooks. *ACM Transactions on Computing Education*, *11*, 3:1–3:21.

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

Buse, R., & Weimer, W. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, *36*(4), 546–558.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2010). Exploring the influence of identifier names on code quality: An empirical study. In *14th European conference on software maintenance and reengineering*, pp 156–165.

CACM. (2002). Hello, world gets mixed greetings. *Communications of the ACM*, *45*(2), 11–15.

Cadwell, P. (2008). *Readability and controlled language*. Master's thesis, Dublin City University.

Cant, S., Jeffery, D. R., & Henderson-Sellers, B. (1995). A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, *37*(7), 351–362.

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

Clark, R., Nguyen, F., & Sweller, J. (2006). *Efficiency in learning: Evidence-based guidelines to manage cognitive load*. Pfeiffer: Wiley.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M., & Love, T. (1979). Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on Software Engineering*, *2*, 96–104.

De Lucia, A., Di Penta, M., & Oliveto, R. (2011). Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering*, *37*(2), 205–227.

Deimel, L., & Naveda, J. (1990). *Reading computer programs: Instructor's guide and exercises*. Tech. Rep. CMU/SEI-90-EM-3, Pittsburgh, PA, USA: Software Engineering Institute.

Deißenböck, F., & Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal*, *14*(3), 261–282.

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

DuBay, W. H. (2004). *The principles of readability*. Costa Mesa, CA: Impact Information.

Fitzsimmons, A., & Love, T. (1978). A review and evaluation of software science. *ACM Computing Surveys (CSUR)*, *10*(1), 3–18.

Flesch, R. (1948). A new readability yardstick. *Journal of applied psychology*, *32*(3), 221.

Foster, J. R. (1993). *Cost factors in software maintenance*. PhD thesis, School of Engineering and Computer Science, University of Durham, UK.

Flower, M. (1999). *Refactoring: Improving the design of existing code*. Reading, MA: Addison-Wesley.

Gellenbeck, E., & Cook, C. (1991). An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*, pp 65–81.

Glass, R. (2003). *Facts and fallacies of software engineering*. Reading, MA: Addison-Wesley.

Gobet, F., Lane, P., Croker, S., Cheng, P., Jones, G., Oliver, I., et al. (2001). Chunking mechanisms in human learning. *Trends in Cognitive Sciences*, *5*(6), 236–243.

Haiduc, S., & Marcus, A. (2008). On the use of domain terms in source code. In *Proceedings of the 16th IEEE international conference on program comprehension*, pp 113–122.

Halstead, M. (1975). Toward a theoretical basis for estimating programming effort. In *Proceedings of the annual ACM/CSC-ER conference*, pp 222–224.

Hargis, G. (2000). Readability and computer documentation. *Journal of Computer Documentation*, *24*(3), 122–131.

Høst, E. W., & Østvold, B. M. (2009). Debugging method names. In *Proceedings of the 23rd European conference object-oriented programming*, pp 294–317.

Jabangwe, R., Börstler, J., Šmite, D., & Wohlin, C. (2014). Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review. *Empirical Software Engineering*. doi:10.1007/s10664-013-9291-7.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, *50*(4), 36–42.

Lanza, M., Marinescu, R., & Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Berlin: Springer.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th international conference on software engineering*, pp 492–501.

Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006). What's in a name? A study of identifiers. In *Proceedings of the 14th IEEE international conference on program comprehension*, pp 3–12.

Liblit, B., Begel, A., & Sweetser, E. (2006). Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th annual psychology of programming workshop*, pp 53–67.

Liz, B., Dreyfus, T., Mason, J., Tsamir, P., Watson, A., & Zaslavsky, O. (2006). Exemplification in mathematics education. In *Proceedings of the 30th conference of the international group for the psychology of mathematics education*, Vol. 1, pp 126–154.

Marcus, A., Poshyvanyk, D., & Ferenc, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, *34*(2), 287–300.

Martin, J. (1993). *Principles of object-oriented analysis and design*. Englewood Cliffs NJ: Prentice-Hall.

Mason, J., & Pimm, D. (1984). Generic examples: Seeing the general in the particular. *Educational Studies in Mathematics*, *15*(3), 277–289.

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? *American Psychologist*, *59*(1), 14.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *2*(4), 308–320. doi:10.1109/TSE.1976.233837.

Miller, G. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*(2), 81.

Nguyen, V. (2010). Improved size and effort estimation models for software maintenance. In *Proceedings of the 26th international conference on software maintenance*, pp 1–2.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments. *Educational Psychologist*, *38*(1), 1–4.

Posnett, D., Hindle, A., & Devanbu, P. (2011). A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pp 73–82.

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, *15*(3), 203–221.

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

Relf, P. A. (2005). Tool assisted identifier naming for improved software readability: An empirical study. In *Proceedings of the 4th international symposium on empirical software engineering (ISESE)*, pp 53–62.

Riel, A. J. (1996). *Object-Oriented design heuristics*. Reading, MA: Addison-Wesley.

Salviulo, F., & Scanniello, G. (2014). Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically informed study with students and professionals. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pp 48:1–48:10.

Shaft, T. M., & Vessey, I. (2006). The role of cognitive fit in the relationship between software comprehension and modification. *MIS Quarterly, 30*(1), 29–55.

Sweller, J., & Cooper, G. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, *2*, 59–89.

Tennyson, R. D., & Cocchiarella, M. J. (1986). An empirically based instructional design theory for teaching concepts. *Review of Educational Research*, *56*(1), 40–71.

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

Tryggeseth, E. (1997). *Support for understanding in software maintenance*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway.

VanLehn, K. (1996). Cognitive skill acquisition. *Annual Review of Psychology*, *47*, 513–539.

Watson, A., & Mason, J. (2002). Extending example spaces as a learning/teaching strategy in mathematics. In *Proceedings 26th Conference of the international group for the psychology of mathematics education*, Vol. 4, pp 377–384.

Welker, K. (2001). The software maintainability index revisited. *CrossTalk*, *14*, 18–21.

Welker, K. D., Oman, P. W., & 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.

Woodfield, S. N., Dunsmore, H. E., & Shen, V. Y. (1981). The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on software engineering*, pp 215–223.

Yip, S. W., & Lam, T. (1994). A software maintenance survey. In *Proceedings of the 1st Asia-Pacific software engineering conference*, pp 70–79.

Jürgen Börstler is a professor of software engineering at Blekinge Institute of Technology (BTH), Sweden. He has a PhD in computer science from Aachen University of Technology (RWTH), Germany. He is a member of SERL-Sweden, the Software Engineering Research Lab at BTH. His main research interests are in requirements engineering, object-oriented methods, software process improvement, software measurement, software comprehension, and computer science education. He is a founding member of the Scandinavian Pedagogy of Programming Network (SPoP) and a senior member of the Swedish Requirements Engineering Network.

Michael E. Caspersen is director of Centre for Science Education and associate professor at Department of Computer Science at Aarhus University, Denmark. He holds a PhD in computer science from Aarhus University. His research interests are in the areas of computing education, programming didactics, programming methodology, and object-oriented programming. He is co-chair of the joint ACM Europe/ Informatics Europe Committee on European Computing Education and a past member of the ACM Education Council. He is a founding member of the Scandinavian Pedagogy of Programming Network and member of the Board of the IEEE Nordic Education Society Chapter. He is member of the editorial board of the international research journal Computer Science Education.

Marie Nordström is a senior lecturer at the Department of Computing Science at Umeå University, Sweden. She holds a PhD degree in computer science from Umeå University. Her main research interest are in computer science education, in particular regarding object-oriented programming.