Beauty and the Beast

Toward a Measurement Framework for Quality of Example Programs

Jürgen Börstler
Department of Computing Science
Umeå University
SE-90187 Umeå, Sweden
jubo@cs.umu.se

Michael E. Caspersen
Department of Computer Science
University of Aarhus
DK-8200 Aarhus N, Denmark
mec@daimi.au.dk

Marie Nordström
Department of Computing Science
Umeå University
SE-90187 Umeå, Sweden
marie@cs.umu.se

ABSTRACT

Examples are important tools for programming education. In this paper, we investigate desirable properties of programming examples from a cognitive and a measurement point of view. We argue that some cognitive aspects of example programs are "caught" by common software measures, but they are not sufficient to measure understandability of examples. We conclude that a framework for measuring understandability of examples should also consider factors related to the usage of the example.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education - *Computer Science Education*.

General Terms

Design, Measurement.

Keywords

CS1, Programming Examples, Measurement, Understandability.

1. INTRODUCTION

Example isn't another way to teach. It is the only way to teach. [A. Einstein]

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" [34]. More specifically, research in cognitive load theory has shown that alternation of worked examples and problems increase learning outcome [31].

Students use examples as templates for their own work. Examples must therefore be consistent with the principles and rules of the topics we are teaching and free of any undesirable properties or behaviour. If not, students will have a difficult time recognizing patterns and telling an example's superficial surface properties from those that are structurally important.

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 [32] 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, we can minimize the risk of misinterpretations and erroneous conclusions, which otherwise can lead to misconceptions. Once established, misconceptions can hinder students in their learning and be difficult to resolve [8, 27].

But how can we tell "good" from "bad" examples? Can we measure the quality of an example?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE 2007, June 25–27, 2007, Dundee, Scotland.
Copyright 2007 ACM X-XXXXX-XX-X/XX/X.

2. 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 [33]. That said, Fowler makes an important point in the quote above. In an educational context, this statement is even more important. In the beginning of their first programming course, students can't even write a simple program that a computer can understand.

A good example must obviously be *understandable by a computer*. Otherwise it cannot be used on a computer and would therefore be no real programming examples.

A good example must also be *understandable by students*. Otherwise they cannot construct an effective mental model of the programs. Without "understanding", knowledge retrieval works on an example's surface properties only, instead of on its more general underlying structural properties [10, 32, 34].

A good example must also *effectively communicate the concept(s) to be taught*. There should be no doubt about what exactly is exemplified. To minimize cognitive load [25], an example should furthermore only exemplify one (or very few) new concept at a time.

The "goodness" of an example also depends on "external" factors, like the pedagogical approach taken. E.g., when our main learning goal is proficiency in object-oriented programming (in terms of concepts, not specific syntax), our examples should always be truthfully object-oriented and "exemplary", i.e. adhere to accepted design principles and rules and not show any signs of "code smells" [12, 22, 28]. If examples are not always truthfully object-oriented, students will have difficulties picking up the underlying concepts, principles, and rules.

These three properties might seem obvious. However, the recurring discussions about the harmfulness or not of certain common examples show that there is quite some disagreement about the meaning of these properties [37, 1].

3. SOFTWARE MEASUREMENT

When you can measure what you are speaking about, and express it in numbers, you know something about it; but

when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind. [Lord Kelvin]

From our discussion in the previous sections, it would be useful to find some way of determining the understandability of a programming example. A suitable measure could help us choose between examples and guide the shaping of examples.

According to SEI's quality measures taxonomy, understandability is composed of *complexity*, *simplicity*, *structuredness*, and *readability* [29]. Bansiya and Davis [3] describe understandability as "[t]he properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure".

There are large bodies of literature on software measurement [14, 26, 4] and program comprehension [6, 21, 5, 13]. The work on software measurement focuses mainly on the structural complexity of software. There is only little work on measuring the cognitive aspects of complexity [7, 30]. The work on program comprehension focuses on the cognitive aspects, but is mainly concerned with the comprehension process and not with software measurement.

4. ONE PROBLEM, TWO SOLUTIONS

Technical skill is mastery of complexity, while creativity is mastery of simplicity. [C. Zeeman]

Let us forget for a moment about actual software measures and look at two example programs for implementing a class *Date*: the Beauty and the Beast.

4.1 The Beauty

The Beauty (Figure 4-2) is developed according to sound principles of decomposition; we could call it *extreme decomposition*. The Beauty consists of four classes: *Date* with components *Day*, *Month*, and *Year*. A *Date* object knows its *Day*, *Month*, and *Year*. The three classes *Day*, *Month*, and *Year* are encapsulated as inner classes of the *Date* class, since they are not relevant to the surroundings. Their existence is a result of our choice of representation for class *Date* (see Figure 4-1).

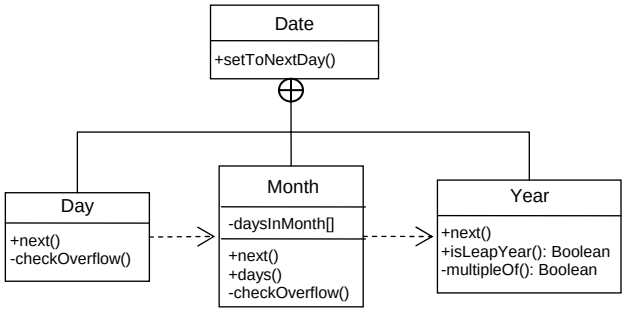


Figure 4-1: *UML diagram for the Beauty*

The Beauty is beautiful for several reasons. First, there is an explicit representation of each of the key concepts in the problem domain. These can work as clues (beacons) aiding in code comprehension [13]. Second, the interfaces and implementations of all classes are very simple and represent an easily recognizable distribution of responsibilities. Third, carefully chosen identifiers, matching problem domain concepts, enhance the readability of the code. Fourth, extreme decomposition supports independent and incremental comprehension, development, and test of each of the four component classes, as well as each of the methods in the classes.

A drawback of The Beauty is that one has to look into several classes to get the full picture of the solution. This problem can, however, be easily solved by providing a class diagram, like the one in Figure 4-1.

```
public class Date_Beauty {
  private Day day;
  private Month month;
  private Year year;

  public Date_Beauty(int y, int m, int d) {
    this.year = new Year(y);
    this.month = new Month(m);
    this.day = new Day(d);
  }
  public void setToNextDay() {
    day.next();
  }

  private class Day {
    private int d;   // 1 <= d <= month.days()

    public Day(int d) {
      this.d = d;
    }
    public void next() {
      d = d + 1;
      checkOverflow();
    }
    private void checkOverflow() {
      if ( d > month.days() ) {
        d = 1;
        month.next();
      }
    }
  } // Day

  private class Month {
    private int m;   // 1 <= m <= 12
    private final int[] daysInMonth=
      {0,31,28,31,30,31,30,31,31,30,31,30,31};
      /*  1  2  3  4  5  6  7  8  9 10 11 12 */

    public Month(int m) {
      this.m = m;
    }
    public int days() {
      int result= daysInMonth[m];
      if ( m == 2 && year.isLeapYear() ) {
        result = result + 1;
      }
      return result;
    }
    public void next() {
      m = m + 1;
      checkOverflow();
    }
    private void checkOverflow() {
      if ( m > 12 ) {
        m = 1;
        year.next();
      }
    }
  } // Month

  private class Year {
    private int y;

    public Year(int y) {
      this.y = y;
    }
    public void next() {
      y = y + 1;
    }
    public boolean isLeapYear() {
      return (isMultipleOf(4) && !isMultipleOf(100))
             || isMultipleOf(400);
    }
    private boolean isMultipleOf(int a) {
      return ( y % a ) == 0;
    }
  } // Year
} // Date_Beauty
```

Figure 4-2: *The Beauty*

4.2 The Beast

The Beast (Figure 4-3) is structured as one monolithic method. We could say it was developed according the principle of *no decomposition*.

The Beast has the advantage of collecting everything in one place. This leads to much less code in total. All necessary information is contained in a single statement sequence. The drawbacks are however numerous.

First, there is no explicit representation of the key concepts in the problem domain. Although this solution is much smaller than The Beauty, it is nevertheless difficult to get the full picture. It is not even possible to provide a high-level diagram to resolve that problem, since all processing is contained in a single method. Second, there is mainly one long statement sequence where everything is happening. Such an approach makes it impossible to introduce meaningful identifiers as clues (beacons) aiding in code comprehension. Third, the Beast shows no signs of "work units" or "chunks" of information. That makes it difficult to deconstruct the program and find appropriate starting points for a code comprehension effort. Students might furthermore conclude that such a program is constructed as a large monolithic unit. The Beast does not lend itself as a pattern for incremental testing and development. Fourth, The Beast is highly nested. Students have to keep track of many conditions at the same time, which increases cognitive load [25].

```
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
```

Figure 4-3: *The Beast*

4.3 Conclusion

Large, monolithic units of code are difficult to understand. Program decomposition into suitable units[1] is important to understanding.

There is no doubt that solutions like the Beauty should be preferred. The Beauty is not only superior in structure, it is also superior from a learning theoretic point of view. Small units reduce cognitive load [9, 25], structural similarities support the recognition of programming plans or patterns [6, 32, 34], and the frequent appearance of mnemonic names help to give meaning to program elements [10, 13].

The essence of developing programming examples is finding an appropriate structure that supports understanding, and hence learning. But when is one structure better than another? And how much better is it? Can we provide a yardstick for measuring the potential understandability of programs?

5. READABILITY AND UNDERSTANDABILITY

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 program. Although readability is a component of understandability in SEI's quality measures taxonomy [29] and there is a large body of literature on software measurement, we couldn't find a single publication on measures for software readability.

5.1 The Flesch Reading Ease Score

The Flesch Reading Ease Score (FRES) is a measure of readability of ordinary text [11, 35]. Based on the average sentence length (*words*/*sentences*) and the average word length (*syllables*/*words*) a formula is constructed to indicate the grade level of a text. Lower values of the ratios indicate easy to read text and higher values indicate more difficult to read text. I.e. the shorter the sentences and words in a text, the easier it is to read.

Please note that FRES does not say anything about understandability. The FRES is just concerned with "parsing" a text. Its understanding depends on further factors, like for example familiarity of the actual words and sentence structure, or reader interest in the text's subject.

Flesch's work was quite influential and has been applied successfully to many kinds of texts. There are also measures for other languages than English.

5.2 A Reading Ease Score for Software

Following the idea of Flesch, we introduce a Software Readability Ease Score (SRES) by interpreting the lexemes of a programming language as syllables, its statements as words, and its units of abstraction as sentences. We 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" [9, 15, 24, 25]).

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 [24]. Chunking is an abstraction process that helps us to manage complexity. Since abstraction is a key computing/programming concept [2, 16, 19], proper chunking is highly relevant for the understanding of programming examples.

Clearly, there are other factors influencing program readability, like for example control flow, naming, and how much the students have learned already. We will come back to these factors in our discussion section. For a good overview over code readability issues, see [10].

5.3 Measurement Data

As mentioned above SRES only measure readability (ease of parsing) of a program. Readability is necessary but not suffi-

[1] These units can be declarative, functional, or object-oriented. The Beast could for example be improved significantly without introducing further classes.

cient for understanding a program. Other factors such as the structural and cognitive complexity also influence understanding. If we use cyclomatic complexity (*CC*) [23] as a measure of structural complexity and difficulty (*D*) [17] as a measure of cognitive complexity, and calculate these measures for the Beauty and the Beast, we get the figures as shown in the embedded table.

	Program	
Measure	*Beauty*	*Beast*
SRES	10.3	16.2
CC	3.0	17.0
D	7.9	43.2
Total (Σ)	21.2	76.4

As indicated by the figures, the SRES measure clearly is in favour of the Beauty. Even more so are the standard measures of cyclomatic complexity and difficulty. According to these, the Beast is in total 3.6 times more difficult to understand than the Beauty.

Of course, this is just an example; the programs we measure as well as the measures we apply are more or less randomly chosen among countless options. To expand a bit on the empirical investigations, we have investigated a number of other standard measures, and we have extended the suite of program examples.

The measures we have investigated have been selected for their reported significance in the literature; the selected measures are presented in Table 1.

Selected measures	
Acronym	**Description**
LoC	Total lines of code
SRES	The software reading ease score as described in section 5.2.
$CC_{max}(m)$	Cyclomatic complexity; the number of (statically) distinct paths through a method; should be <10 [23].
D	The difficulty of the program [17]
avgV(*c*)-*avgLOC*(*c*)	Factors of the *Maintainability Index*, a measure with high predictive value for software maintainability [36]. The measures report average values for Volume, *V* (size in terms of the numbers operators and operands [17]), *CC*, and *LoC* per class (*c*).
CC/LoC	Average *CC* per *LoC*; should be ≤0.16 [20].
LoC/m	Average *LoC* per method; should be ≤7 [20].
m/c	Average number of methods per class; should be ≤4 [20].
WMC	Weighted Method Count, a product of the three previous measures; should be ≤5 [20].

Table 1: *Selected measures*

The suite of program examples is extended from two to five representing a continuum of programs solving the Date problem: *Beauty* (E_1), *Good* (E_2), *Bad* (E_3), *Ugly* (E_4), *Beast* (E_5). E_2 is the same as E_1 except that *daysInMonth* is handled by nested if's. E_3 is the same as E_2 except that the classes are not nested. E_4 is the same as E_5 except that *setToNextDay* is decomposed into helper methods.

The result of our investigations is captured in Table 2. For all measures, lower values are considered better. Threshold values suggested in the literature are given in column *T*.

		Program				
Measure	*T*	E_1	E_2	E_3	E_4	E_5
LoC		50	59	57	31	32
SRES	7±2	10.3	8.9	9.3	11.9	16.2
$CC_{max}(m)$	10	3	7	7	7	17
D		7.92	7.15	9.71	22.4	43.2
avgV(c)		387	412	363	752	798
avgCC(c)	10	4.8	6.25	5.25	14.0	18.0
avgLoC(c)		12.5	14.8	14.3	31.0	32.0
CC/LoC	0.16	0.4	0.42	0.37	0.45	0.56
LoC/m	7	2.9	3.27	4.09	6.75	14.0
m/c	4	3.3	3.75	2.75	4.0	2.0
WMC	5	3.6	5.2	4.1	12.2	15.8
	T	E_1	E_2	E_3	E_4	E_5

Table 2: *Values of selected measures for sample programs*

6. DISCUSSION

Although the measures focus on different aspects of a program, it can be noted that they "favour" programs with high degrees of decomposition (E_1–E_4). This is not surprising, since all research in software design and measurement proposes decomposition as a tool to manage complexity. In relation to education it is important to note that a high degree of decomposition also is an advantage from a cognitive point of view.

However, there are many important aspects of understandability not covered by any measure, like for example the choice of names, commenting rate, etc. Any example must furthermore take into account the educational context, i.e. what the students already (are supposed to) know.

7. TOWARD A MEASUREMENT FRAMEWORK

From the discussion above, we conclude that a framework for measuring programming example understandability should consider properties of the *example itself* as well as the *context of its use*. These properties could be divided into the following orthogonal *intra-example* factors:

- *Readability*: Captures how easy a programming text is to read, based on SRES or similar measures.
- *Structural complexity*: Captures the structural properties of a program, based on measures for control flow complexity (cyclomatic complexity), coupling cohesion, etc.
- *Cognitive complexity*: Captures the information contained in a program, based on Halstead's measures or information theory [18].
- *Commenting*: Captures how well the example is commented (excessive use of comments may be a bad thing).
- *Size*: Captures the size of the example, based on a common size measure like LoC.
- *Consistency*: Captures how well the example follows accepted design principles and rules, based on the amount of "code smells".

and the following orthogonal *inter-example* factors related to usage:

- *Presentation*: Captures the degree of conformance to a style guide or standard or the similarity of style with other examples.
- *Progression*: Captures how well the example "fits" with what the students (are supposed to) know.
- *Vocabulary*: Captures the familiarity of the names occurring in the example (could be a sub-factor of progression).

Such a factorization makes it easier to argue about understandability. A perfectly readable and structured program can very well be difficult to understand, when all identifiers are chosen badly, and perfect values for all example factors are no guarantee for an understandable program, when its usage is badly timed.

8. CONCLUSION AND FUTURE WORK

We have shown that many common software measures respect basic cognitive aspects of example programs, in particular cognitive load; all measures we have investigated say that decomposition is good—the more extreme the decomposition, the better. We also propose and discuss a new measure for software readability (SRES). We conclude that all these measure, although useful, lack in their disregard of factors related to the usage of examples. Based on our discussion, we propose a framework for measuring the understandability of programming examples that aims to take such factors into account.

In future research we aim at developing and empirically validating a simple quality measure of example programs by studying a wide variety of examples from textbooks and course material.

9. REFERENCES

[1] ACM Forum "`Hello, World' Gets Mixed Greetings", *Communications of the ACM*, Vol 45(2), 2002, 11-15.

[2] Armstrong, D. "The Quarks of Object-Oriented Development", *Communications of the ACM*, Vol 49(2), 2006, 123-128.

[3] Bansiya, J., Davis, C. G. "A Hierarchical Model for Object-Oriented Design Quality Assessment", IEEE Transactions on Software Engineering, Vol 28(1), 2002, 4-17.

[4] Briand, L., Wüst, J. "Empirical Studies of Quality Models in Object-Oriented Systems". In M. Zelkovitz (ed.) *Advances in Computers*, Academic Press, Vol 56, 2002, 1-46.

[5] Brooks, R. "Towards a Theory of the Comprehension of Computer Programs", *Intl. J. Man-Machine Studies*, Vol 18(6), 1983, 543-554.

[6] Burkhardt, J.-M., Détienne, F., Wiedenbeck, S. "Object-riented Program Comprehension: Effect of Expertise, Task and Phase", *Empirical Software Engineering*, Vol 7, 2002, 115-156.

[7] Cant, S. N., Henderson-Sellers, B., Jeffery, D. R. "Application of cognitive complexity metrics to object-oriented programs, *Journal of Object-Oriented Programming*, Vol 7(4), 1994, 52-63.

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

[9] Clarck, R, Nguyen, F and Sweller, J. *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load*, Pfeiffer, John Wiley & Sons, 2006.

[10] Deimel, L. E., Naveda, J. F. "Reading Computer Programs: Instructor's Guide and Exercises", CMU/SEI-90-EM-3, Software Engineering Institute, 1990.

[11] Flesch, R. "A new readability yardstick", *Journal of Applied Psychology*, Vol 32, 1948, pp. 221-233.

[12] Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[13] Gellenbeck, E. M., Cook, C. R. "An Investigation of Procedure and Variable Names as Beacons During Program Comprehension", *Proc. 4th Workshop on Empirical Studies of Programmers*, 1991, pp. 65-81.

[14] Genero, M., Piattini, M., Calero, C. "A Survey of Metrics for UML Class Diagrams", *Journal of Object Technology*, Vol 4(9), 2005, 59-92.

[15] Gobet, F., Lane, P. C. R., Croker, S., Cheng, P. C. H., Jones, G., Oliver, I., & Pine, J.M. "Chunking Mechanisms in Human Learning" *Trends in Cognitive Sciences*, Vol 5, 236-243.

[16] Görz, G.: "Abstraction as a Fundamental Concept in Teaching Computer Science", *Les langages applicatifs dans l'enseignement de l'informatique*, Specif no. special 93, Rennes/Paris, 1993, 168-178.

[17] Halstead, M. H. "Toward a theoretical basis for estimating programming effort", *Proc of the Annual ACM Conference (ACM/CSC-ER)*, 1975, 222-224.

[18] Khoshgoftaar, T. M., Allen, E. B. "Empirical Assessment of a Software Metric: The Information Content of Operators", *Software Quality Journal*, Vol 9, 2001, 99-112.

[19] Kramer, J. "Abstraction—the key to Computing?" *Communications of the ACM*, to appear.

[20] Lanza, M., Marinesu, R. *Object-Oriented Metrics in Practice*, Springer, 2006.

[21] Li, Y., Yang, H. "Simplicity: A Key Engineering Concept for Program Understanding", *Proc. 9th Internat. Workshop on Program Comprehension*, 2001.

[22] Martin, J. *Principles of Object-Oriented Analysis and Design*, Prentice Hall, 1993.

[23] McCabe, T. J. "A complexity measure", *IEEE Transactions on Software Engineering*, Vol 2(4), 1976, 308–320.

[24] Miller, G.A. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *The Psychological Review*, Vol 63, 1956, pp. 81-97.

[25] Paas, F., Renkl, A. and Sweller, J. "Special Issue on Cognitive Load Theory", *Educational Psychologist*, Vol 38 (1), 2003.

[26] Purao, S., Vaishnavi, V. "Product Metrics for Object-Oriented Systems", *Computing Surveys*, Vol 35(2), 2003, 191-221.

[27] Ragonis, N., Ben Ari, M, "A long-term investigation of the comprehension of OOP concepts by novices", *Computer Science Education*, Vol 15(3), 2005, pp. 203-221.

[28] Riel, A. *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[29] SEI (Software Engineering Institute) "Quality Measures Taxonomy", http://www.sei.cmu.edu/str/taxonomies/view_ qm.html, Dec 2006, accessed Jan 11, 2007.

[30] Shao, J., Wang, Y. "A new measure of software complexity based on cognitive weights", *Can. J. Elect. Comput. Eng.*, Vol 28(2), 2003, 1-6.

[31] Sweller, J. and Cooper, G.A. "The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra", *Cognition and Instruction*, Vol. 2, pp. 59-89, 1985.

[32] Trafton, J. G., Reiser, B. J. "Studying Examples and Solving Problems: Contributions to Skill Acquisition", Naval HCI Research Lab, Washington, DC, 1992.

[33] Tryggeseth, E. "Support for Understanding in Software Maintenance", PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 1997.

[34] VanLehn, K. "Cognitive Skill Acquisisition", *Annual Review of Psychology*, Vol 47, 1996, 513-539.

[35] Wikipedia "Flesch-Kincaid Readability Test", http://en.wikipedia.org/wiki/Flesch-Kincaid_Readability_ Test, Jan 8, 2007, accessed Jan 12, 2007.

[36] Welker, K. D. "The Software Maintainability Index Revisited", *CrossTalk*, Aug 2001.

[37] Westfall, R. "`Hello, World´ Considered Harmful", *Communications of the ACM*, Vol 44(10), 2001, 129-130.