2016 IEEE 29th International Conference on Software Engineering Education and Training

Code Readability Testing, an Empirical Study

Todd Sedano
Carnegie Mellon Unveristy
Silicon Valley Campus
Moffett Field, CA 94035, USA
Email: todd.sedano@sv.cmu.edu

Abstract—**Context: One of the factors that leads to improved code maintainability is its readability. When code is difficult to read, it is difficult for subsequent developers to understand its flow and its side effects. They are likely to introduce new bugs while trying to fix old bugs or adding new features. But how do software developers know they have written readable code?**

Objective: This paper presents a new technique, Code Readability Testing, to determine whether code is readable and evaluates whether the technique increases programmers' ability to write readable code.

Method: The researcher conducted a field study using 21 software engineering master students and followed the Code Readability Testing with each student in four separate sessions evaluating different "production ready" software. After the observations, a questionnaire evaluated the programmer's perspective.

Results: By following Code Readability Testing, half of the programmers writing "unreadable" code started writing "readable" code after four sessions. Programmers writing "readable" code also improved their ability to write readable code. The study reveals that the most frequent suggestions for increasing code readability are improving variable names, improving method names, creating new methods in order to reduce code duplication, simplifying if conditions and structures, and simplifying loop conditions. The programmers report that readability testing is worth their time. They observe increases in their ability to write readable code. When programmers experience a reader struggling to understand their code, they become motivated to write readable code.

Conclusion: This paper defines code readability, demonstrates that Code Readability Testing improves programmers' ability to write readable code, and identifies frequent fixes needed to improve code readability.

I. Introduction

Writing readable code reduces the costs of development and maintenance of software systems. A considerable portion of the software development cost is ongoing maintenance to add new features and fix defects [1]. Even in the early stages of the software's evolution, the ability to read and quickly understand existing code is a key factor that affects the code's ability to change.

While creating programmers who write readable code is not a new problem for the software industry, the previous work focuses around what code should look like, not how to train programmers to write readable code. Developers realize the importance of writing code that is readable by their peers, but they often do not receive feedback on whether their code is readable. Programming constructs that are clear to the author can confuse the next developer. Some programmers bemoan that they can't read their own code six months later. If the code works, clearly the computer can understand it, but can anyone else on the team?

Teaching this skill is not a top priority in computer science and software engineering curricula. The Computer Science Curriculum promotes understanding the programming paradigms of a particular language (e.g. functional vs. non-functional), not how to write readable code [2]. The Software Engineering Body of Knowledge (SWEBOK) does make one reference to writing "understandable code" in the Coding Practical Considerations for the Software Construction knowledge area [3]. This is just one out of 229 subtopics of SWEBOK. The Graduate Software Engineering reference curriculum (GSwE) does not prescribe any further recommendations beyond SWEBOK for this topic [4]. Some undergraduate courses briefly cover the issues of programming style. A few courses will penalize students for producing unreadable code. In rare courses, students swap assignments simulating the experience of inheriting someone else's code. While this sensitizes students to the needs of writing readable code, the experience lacks concrete steps to increase their skill. The emphasis of a computer science curriculum or a software engineering curriculum is on the substantial topics in the reference curriculum.

Companies tend to assume programmers arrive with this skill or will learn it through on the job training. Project teams may have code style guidelines, or best practices around writing code e.g. when a programmer opens a database connection, immediately write the close statement.

There is strong empirical evidence that supports the effectiveness of software inspections and code reviews for uncovering bugs. While theses techniques can identify readability issues, they are not designed to teach developers how to write readable code. When an author receives a list of defects, the author looses the opportunity to learn how the code confuses the reader.

Code Readability Testing reveals areas where the code is not readable, and enables a dialogue between coder and reader. Feedback is instantaneous, as the author sees exactly how reader interprets the code.

A. Research Objectives

Using the goal template from Goal Question Metric (GQM), the goal is to... **Analyze** Code Readability Testing **for the purpose of** determining its effectiveness in improving programmers' ability to write readable code **with respect to** their effectiveness **from the point of view of the** researcher **in the context of** the "craft of software development" course at Carnegie Mellon University.

This paper decomposes this goal into four questions:

Research Question 1: Would programmers who repeatedly follow Code Readability Testing increase the readability of their code?

Research Question 2: What kinds of issues does Code Readability Testing detect?

Research Question 3: How time-consuming is readability testing?

Research Question 4: How did programmers perceive Code Readability Testing?

II. Background and Related Work

Improving code readability and programming style is not a new topic for the software industry.

Kernighan and Plauger, in their 1974 seminal book The Elements of Programming Style, document heuristics for improving coding practices and code readability by rewriting code used in computer science textbooks [5]. In the 1982 book, Understanding the Professional Programmer, Gerald Weinberg emphasizes that the programmer is a more important reader of the code than the computer's compiler or interpreter. He suggests that just like the writing process for English text, code needs to be rewritten several times before it becomes exemplary code. He encourages programmers to spend time reworking code that will be frequently read in the future [6].

In recent books aimed at professional programmers, Andrew Hunt, David Thomas, Kent Beck, and Robert Martin tackle coding style in a variety of ways. In Pragmatic Programmers, Hunt and Thomas examine the tools, processes, and tricks that help programmers master their craft [7]. In Clean Code, Robert Martin addresses techniques to help a developer become a better programmer [8]. In Implementation Patterns, Kent Beck addresses good software development design patterns [9]. In short, they distill their life long experiences into best practices, some of which address code readability.

In recent studies, researchers examine code readability from different approaches: automated improvement techniques, naming of identifiers, syntax, and automated metrics. Several studies attempt to automate techniques to improve code readability. Wang examines the automatic insertion of blank lines in code to improve readability [10] whereas Sasaki reorders programming statements to improve readability by declaring variables immediately before their utilization [11]. Several researchers examine the naming of identifiers [12], [13], [14], [15]. Relf's tool encourages the developer to improve variable and method names [15]. Binkley observes that camel case is easier to read than underscore variables [16]. Jones looks at the issues with operator precedence in code readability [17].

While human assessment remains the gold standard of code readability, automated metrics often serve as a proxy. Several studies strive to create code readability metrics so that a computer program determines the readability [18], [19], [20].

Incorporating these metrics into static analysis tools, development environments, and IDEs provides an inexpensive assessment. Substituting the computer for a human produces problems. Metrics using character counts or dictionary words might score a variable named "something_confusing" or "something_vague" as equally readable as a variable that is "exactly_what_i_mean." While a statistical approach to readability metrics is helpful, these measures do not reveal the programmer's intention.

A. Comparison to other techniques

Fagan Inspections, Code Reviews, and Pair Programming are other techniques that improve code quality as summarized in Table I. Fagan Inspections are a proven, time intensive process for finding defects where a committee of developers reviews code [21]. Inspections often include programming style guides and coding standards. While the author is present, the emphasis is on defect identification, not revealing why reviewers might be confused by the code. Code Reviews are a popular, light-weight process where one developer reviews code before it is committed to the master branch or trunk of a source code management system [22]. The author receives a list of suggested changes or issues to fix. Since the author is not present, the author does not see the process the reviewer goes through to understand the code. Developers primarily use code reviews for bug detection [22], [23], not for training developers how to write readable code. Pair programming occurs when two developers write the code at the same time. Pair programming enables continuous reviewing of code, but doesn't provide a fresh perspective to reveal issues for which the authors are blind to observe. [24] Resistance to adoption comes either from management who sees it as more expensive than solo programming or from programmers who do not like the social implications of the process.

Note: Bacchelli and Bird report that programmers thought the purpose of code reviews is to find defects, when in reality the programmers are increasing their understanding of the code [23]. If this is the main benefit of code reviews, it is possible to design other mechanisms to increase code understandability more efficiently than the code review technique.

Perspective-Based-Reading reviews requirements documents from prescribed roles such as user, developer, and tester [25]. A developer will convert requirements into a design and a tester converts requirements into a test plan in order to determine if there are omissions and defects in the requirements.

Yet the question remains, "Can the relevant community understand and maintain the code?" Thus we can ask ourselves, "how do we know if our code is readable?"

III. Code Readability Testing

The technique proposed here uses an experienced programmer to read code samples by thinking out loud and expressing the reader's thought process in understanding the code. During

TABLE I
COMPARISON TO OTHER TECHNIQUES

Technique:	Code Readability Testing	Code Review	Fagan Inspection	Pair Programming
Purpose:	Understand code	Find defects, Understand code	Find defects	High quality code
Roles:	Author Reader	Author Reviewer	Author Moderator Inspector (2+) Recorder Reader / Timekeeper	Author Author
Feedback to the author is:	Synchronous	Asynchronous	Asynchronous	NA

the session, the author of the code observes if and where difficulties emerge. At the end of the session, the two programmers discuss approaches to improve code readability. This process reveals to the code author how another programmer parses and understands the author's code [26].

1) The author tells the reader the main use case, story card, or functionality produced. The author does not explain the design or the code.
2) The author indicates which files were added or modified. Starting with test cases helps the reader understand how the code is used by client code.
3) The reader reads the code aloud and explains the reader's mental thought process. If the code is unclear, the reader speculates on the intention of the code. The reader verbally describes how he or she thinks the code works and explains his or her thought process. Voicing questions helps focus the reader and author. If the reader does not understand a line of code due to unfamiliar programming syntax, the reader asks the author what the operation does.
4) The author does not respond to what the reader is thinking or asking. The author can take notes about what makes particular sections confusing.
5) At the end, the reader confirms with the author that the reader properly understands the code. The author then asks the reader any clarifying questions about the experience.
6) The author and the reader discuss how to improve the code.

The Usability Testing technique [27] from Human Computer Interaction serves as a model for this process. In usability testing, user experience designers watch representative users attempt tasks on a prototype or the actual interface of a product. The researcher observes the user to determine what is obvious and what confuses the user. In particular, the user's natural interaction with the system informs natural affordances for the user experience design. The system deviating from user expectations indicates opportunities for improved design. In Code Readability Testing, the product is the source code, and the user is another developer.

The ideal reader represents future developers and those who will maintain the system. For the typical team, developers on the same team serve as ideal readers. For an open source project, core developers and contributors serve as ideal readers. The ideal reader possesses experiences similar to those of the author, and is proficient with the programming language, framework, and libraries used. If programmers expect their code to be routinely read by less experienced programmers, then novices would be ideal readers.

IV. FIELD STUDY

The researcher followed the Code Readability Testing with each programmer in four separate one-on-one sessions to assess effectiveness and observe improvements over time. The programmers were 21 software engineering graduate students enrolled in the "Craft of Software Development" course at Carnegie Mellon University in Silicon Valley during the Spring 2013 semester.

The researcher scheduled each session for thirty minutes, spaced three weeks apart, thus producing 84 data points. For each session, the researcher asked the students to bring "production ready" code, software that was ready to be released on a real project. The students selected their own projects to work on. At the end of each session, the researcher recorded the review's duration, the number and type of issues detected, and assessment of the overall readability score.

The student's professional development experience ranged from zero to eight years. The average number of years of experience was three years.

A. *Readability Score*

This paper defines code readability as the amount of mental effort required to understand the code. After examining the code, the researcher assigned a readability score following this scale:

4) Easy to read
3) Pretty easy to read
2) Medium difficulty
1) Very challenging

In existing studies [10], [11], [20], [28], [29], there is no standard readability definition or score. In both the Buse and Dorn studies, participants rate code on a Likert scale from "very unreadable" 1 to "very readable" 5, from "unreadable" to "readable" [18], [19]. The participants define their own meaning for readable.

In using this scale, the researcher noticed that the duration of the review correlated with the amount of effort required. For example, reviewing "easy to read" code didn't take much time to review. The average length was 8 minutes with 4 minutes variance. Reviewing "very challenging" to read code often consumed the whole session. The correlation between readability score and the time to review was 0.77

Typically "easy to read" code presents the reader with a simple to follow narrative, keeping a few items in short term memory. "Very challenging" code obscures the programmer's intention. When the reader grabs a sheet of paper and manually executes the computer program by writing down variable values in order to understand the program logic, then the code is "very challenging" to read.

There are common solutions to many programming problems. "Very challenging" code might avoid typical solutions or typical constructs for a solution. When the code's solution is different from the reader's expectation for the solution, the reader finds the code "very challenging."

After reviewing the data, it became clear that the data could be collapsed into two distinct groups, "readable" and "unreadable" code. The researcher grouped "Pretty easy to read" and "Easy to read" code samples as "readable" code and groups "Very challenging" and "medium difficulty" code samples as "unreadable" code. For unreadable code, the code clearly required rework before submission on a project. When comparing these two groups, the code samples were indeed, night and day.

V. Results

Research Question 1: Would programmers who repeatedly follow Code Readability Testing increase the readability of their code?

After graphing trends in the data, the researcher lumped the data into four groups: programmers who initially wrote readable code and made small improvements, programmers who initially wrote unreadable code and made large improvements, programmers whom initially wrote unreadable code and continued to do so, and programmers whose results are not clear.

Result	**Count**
Readable to readable (with small improvements)	11
Unreadable to readable (with large improvements)	5
Unreadable to unreadable	1
Results are not clear	4
Total	21

Starting from the first session, 11 of the programmers wrote readable code consistently. While small improvements can be made to the code, the reader easily understood the code. Of these 11, five progressed from "pretty easy to read" to "easy to read" as represented by Figure 1. The process did not hurt the programmer's ability to write code.

Five programmers initially produced "unreadable code" but over time started improving and finished by writing "readable code" as illustrated by Figure 2. For some, immediate changes occurred, whereas for one programmer, the change required a few sessions.

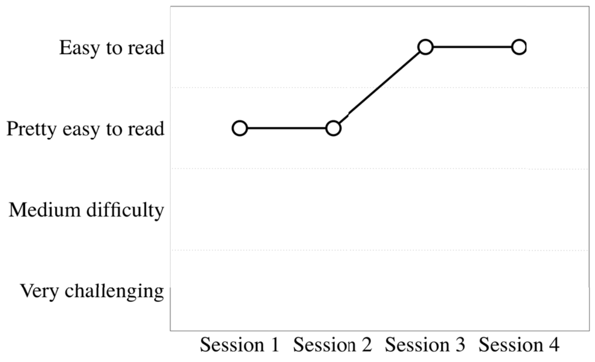

Fig. 1. Programmer #16 consistently wrote "readable" code with small improvements

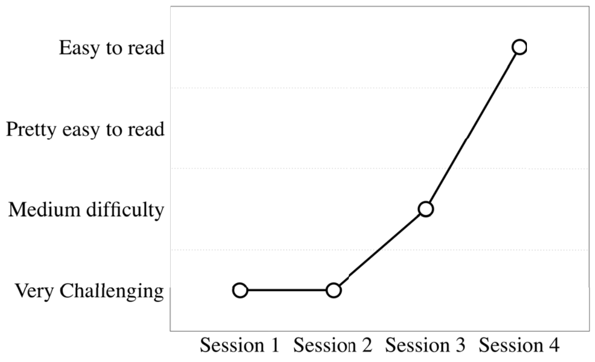

Fig. 2. Programmer #11 started by writing "unreadable" code and progressed to "readable" code

One programmer consistently wrote "unreadable code" during each session as shown in Figure 3. While the programmer improved variable and method naming, the programmer ignored feedback such as breaking multiple nested for loops and if statements. Instead of taking the time to increase readability, the participant reasoned, "I want my code to be as efficient as possible." (Ironically, by only making readability improvements, the readable code was more efficient than the original code.)

Four of the data plots were "all over the place." While two of them trended towards more "readable code," the researcher classified them as outliers. Considering the entire sample size, this means that 16 of the 21 programmers improved their ability to write readable code. When considering the 10 programmers who could benefit from improving readability testing, five achieved large improvements.

Looking only at the first and last sessions, then an interesting result emerged. During the first session, 13 programmers wrote readable code and all still wrote readable code at the end. During the first session, eight programmers wrote

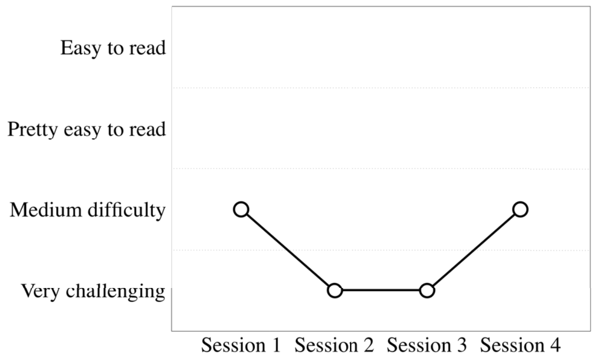

Fig. 3. Programmer #1 continued to write “unreadable” code

unreadable code, and at the end two wrote unreadable code, and six wrote readable code.

Result 1: Most programmers who write “unreadable” code significantly improve and start writing “readable” code after four sessions. Programmers who initially write “readable” code also improve their ability to write readable code.

Research Question 2: What kinds of issues does Code Readability Testing detect?

In reviewing the notes on the 84 sessions, the researcher classified suggestions and feedback based upon feedback type. The researcher relied on unstructured interview notes, not an inspection checklist. The following table prioritizes the feedback by the frequency of each feedback type across all 84 sessions. For example, 45 of the 84 reviews mentioned altering the name of variables as a means improve readability.

Improve code readability by	Number of Reviews
Improving variable names	45 / 84
Improving method names	25/ 84
Extract method to reduce code duplication	26 / 84
Simplifying if conditions	10 / 84
Reducing if nesting	11 / 84
Simplifying loop conditions	11 / 84
Reducing loop structures	5 / 84
Improving class names	3 / 84
Re-sequencing method arguments	1 / 84
Simplifying data structures	1 / 84

Although not a specific goal, readability testing found nine defects in eight of the code samples.

Result 2: Code readability testing detects readability issues that are solved by improvements to variable names, improvements to method names, the creation of new methods to reduce code duplication, simplifying if conditions and nesting of if statements, and simplifying loop conditions.

Research Question 3: How time-consuming is readability testing?

The reader’s subjective experience was that processing “easy to read” code was not time consuming. If a system is composed entirely of “easy to read” code, then the overhead of this process is small. If a system has “very challenging” sections of code, then it is worth reviewing. When the reviewer detects unreadable code, terminating the process allows a discussion of ways to improve code readability.

Readability Score	Median time on review
Very challenging	* 30 minutes
Medium difficulty	20 minutes
Pretty easy to read	11 minutes
Easy to read	8 minutes

Note: the sessions were limited to 30 minutes, the length of the meeting. Often another session was scheduled after any given session. If the reader could not understand the code after 30 minutes, the session was ended.

Result 3: For readable code, readability testing is straightforward. For unreadable code, the process takes significant time. Once unreadable code is detected, the reader and the author can agree that the code needs rework and end the session early.

Research Question 4: How did programmers perceive Code Readability Testing?

At the end of the four sessions, the programmers answered an anonymous survey about their experience with 20 of the 21 participants completing the survey. The self-assessment exposes the programmers’ perception of the technique.

Question: “Was it worth your time or not worth your time?”

20 out of 20 say that following the process was worth their time.

Question: “Why was it worth or why was it not worth your time?”

The free-text responses were grouped according to themes. If participants mentioned multiple reasons, then each reason counts in each theme.

Code Readability Testing...	Count
allows me to see areas of improvement to increase code readability	9
allows me to see a different perspective on my code	7
provides guidance by someone with more experience	4
motivates me to improve the readability of my code	3
allows me to know if my code was understandable	3
allows me to improve my programming speed	1
increases collaboration of software development process	1

Question: “Did you learn how another developer reads and understands your code?”

Out of the 20 participants, 18 participants said yes, and two skipped the question.

Question: “How has this affected the way you write software?”

See Table II for responses.

TABLE II
HOW HAS THIS AFFECTED THE WAY YOU WRITE SOFTWARE?

I now...	Count
choose clearer variable and method names	9
consider the needs of future readers	7
think about the code narrative	5
write shorter methods	2
don't repeat yourself (DRY)	2
avoid deep nested if-else logic	1
re-read code before committing	1
isolate complex logic into a method	1

Questions: "Did you see the reader struggle with understanding your code?"

Out of the 20 participants, 10 participants said yes.

Question: "If so, how did it make you feel?"

I am...	Count
motivated to write more readable code	5
inspired as it was revealing and insightful	4

Result 4: Programmers think following readability testing is worth their time. Their ability to write readable code increases. They articulate concrete improvements to the way they write code. When programmers see a reader struggle to understand their code, the programmers are willing to write readable code and inspired by another developer's point of view..

VI. THREATS TO VALIDITY

A. Construct Validity

1) Code Readability Testing has the reviewer "think aloud" as they read through the code. The "think aloud" activity might not mirror the process a programmer uses when they read code to themselves.

B. Internal Validity

1) The selection of the reviewer: in order to remove the difficulty of inter-reviewer reliability, there is only one reviewer in this study. The reviewer is the researcher, which leads to possible researcher bias. The results might change with a different reviewer. Another reviewer might find more or fewer issues. Another reviewer might be more or less experienced at reading other people's code.
   The reviewer has professional experience in C, C++, Java, and Ruby. The reviewer is able to read and understand the provided C#, Javascript, Objective C, Python, and Dart code. When the reviewer did not understand programming language syntax or idioms, the reviewer asked the author for clarification. While the reviewer is able to understand Javascript code, a more experienced Javascript programmer might find issues not detected.
2) The selection of programming assignments: the programmers selected what to work on. The difficulty level of each session might not be consistent.
3) The selection of programming languages: this study verifies that the approach works within a variety of programming languages and problem domains. For future research, constraining to a particular language may yield stronger insights.
4) Influence from other graduate courses: discussions in the concurrent metrics course and the craft of software development course about code quality might affect the results by sensitizing students to the need to write readable code.

C. External Validity

1) The participants were software engineering students enrolled in a master's program. Their professional development experience ranged from zero to eight years. The average number of years of experience was three years. The correlation between years of industry experience and improvement was 0.31 showing little relationship between improvement and years of industry experience. In fact, the two participants with the most industry experience (seven years and eight years) both dramatically improved their ability to write readable code. Since all the students were still at the beginning of their careers, the drastic improvements in writing readable code might not transfer to more experienced programmers.

VII. FUTURE RESEARCH

Several of the programmers appreciated the value a more experienced developer providing feedback. Future work could reveal the results when the reader and the author possess similar expertise, or if the reader possesses less expertise than the author. If code needs to be readable by less experienced peers, then learning how less experienced programmers read code should contain valuable feedback.

Removing the researcher from the reader role would remove researcher bias. Perhaps students could act as readers for each other if they're given training.

Future work could entail a direct analysis between code reviews and readability testing. Next time, all the programmers could finish the same programming exercise and the researcher could directly compare the results from the two techniques.

One subject persistently wrote "unreadable" code. The subject defended his strategy because "I want my code to be as efficient as possible." Future work could examine how prevalent is this attitude of writing "efficient" but unreadable code, determine where its origins, and suggest possible mitigation steps. In 1974, Knuth proclaimed that premature optimization is the root of all evil [30], yet the problem remains today.

VIII. CONCLUSIONS

Code readability testing addresses the question, "Is my code readable?" by exposing the thought process of a peer reading the code. In this study, 21 programmers followed

Code Readability Testing in four sessions. Most programmers writing "difficult to read" code became programmers writing "easy to read" code after three sessions. Programmers writing "easy to read" code improved their skill. This study identifies several common fixes to unreadable code including improvements to variable names, improvements to method names, the creation of new methods to reduce code duplication, simplifying if conditions and structures, and simplifying loop conditions. The programmers reported that the technique is worth their time and articulated how readability testing alters their programming habits.

REFERENCES

[1] F. P. Brooks Jr., The mythical man-month. Addison-Wesley, 1975.

[2] Computer Science Curriculum 2008: An Interim Revision of CS 2001. ACM and the IEEE Computer Society, 2008.

[3] P. Bourque and R. Dupuis, Guide to the software engineering body of knowledge. IEEE Computer Society Press, 2004.

[4] A. Pyster and et al, Graduate Software Engineering (GSwE2009) Curriculum Guidelines for Graduate Degree Programs in Software Engineering. Stevens Institute, 2009.

[5] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, 1982.

[6] G. M. Weinberg, Understanding the Professional Programmer. Dorset House, 1982.

[7] A. Hunt and D. Thomas, The pragmatic programmer: from journeyman to master. Addison-Wesley Longman Publishing Co., Inc., 2000.

[8] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, 2008.

[9] K. Beck, Implementation Patterns. Addison-Wesley Professional, 2006.

[10] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in Proceedings of the 2011 18th Working Conference on Reverse Engineering.

[11] Y. Sasaki, Y. Higo, and S. Kusumoto, "Reordering program statements for improving readability," in Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, March 2013.

[12] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," Science of Computer Programming, 2009.

[13] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, Conference Proceedings.

[14] B. Liblit, A. Begel, and E. Sweeser, "Cognitive perspectives on the role of naming in computer programs," in Proceedings of the 18th Annual Psychology of Programming Workshop, Conference Proceedings.

[15] P. A. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," in 2005 International Symposium on Empirical Software Engineering.

[16] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in ICPC '09. IEEE 17th International Conference on Program Comprehension, Conference Proceedings, pp. 158–167.

[17] D. M. Jones, "Operand names influence operator precedence decisions," CVu, 2008.

[18] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," IEEE Computer Society, 2010.

[19] J. Dorn, "A general software readability model," 2012.

[20] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in Proceedings of the 8th Working Conference on Mining Software Repositories.

[21] M. E. Fagan, "Advances in software inspections," IEEE Transactions in Software Engineering, 1986.

[22] J. Cohen, S. Teleki, and E. Brown, Best Kept Secrets of Peer Code Review. Smart Bear, 2006.

[23] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, Conference Proceedings.

[24] J. Cohen. (2013) Does pair programming obviate the need for code review? [Online]. Available: http://blog.smartbear.com/programming/does-pair-programming-obviate-the-need-for-code-review/

[25] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz, "The empirical investigation of perspective-based reading," Empirical Software Engineering, 1996.

[26] T. Sedano. (2011) Code readability testing process. [Online]. Available: http://sedano.org/journal/2011/3/30/code-readability-process.html

[27] J. Nielsen, Usability Engineering. Morgan Kaufmann Publishers Inc., 1993.

[28] M. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?" 2013.

[29] D. Crookes, "Generating readable software," Software Engineering Journal, vol. 2, no. 3, pp. 64–70, 1987.

[30] D. E. Knuth, "Computer programming as an art," Communications of the ACM, vol. 17, no. 12, pp. 667–673, 1974.