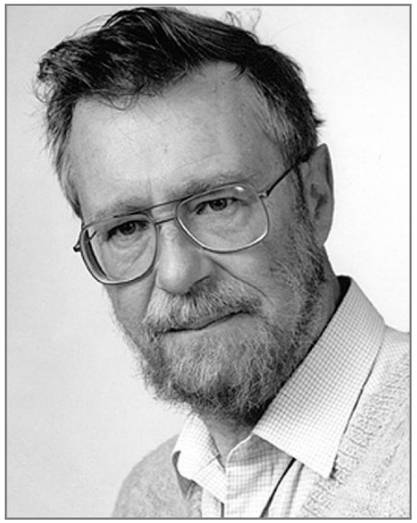
Lines of Code (LoC)

Yegor Bugayenko

Lecture #1 out of 24
80 minutes

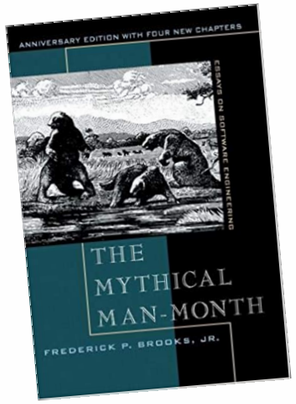
The slidedeck was presented by the author in this YouTube Video

1. Quality of code means maintainability: how quickly other programmers can understand your code.



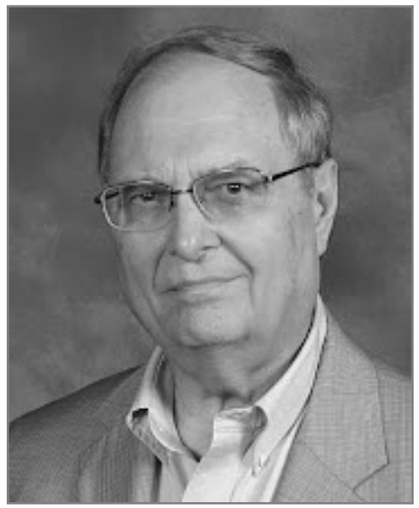
“It has been suggested that there is some law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true.”

— Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. doi:10.1145/355604.361591



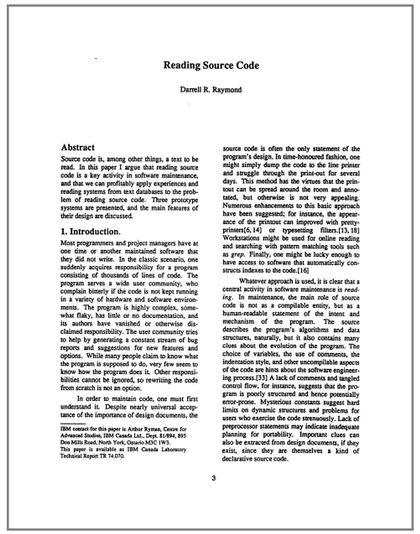
“A programmer depends upon other people’s programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable.”

— Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1978. doi:10.5555/540031



"One of the most overlooked programming skills is the ability to read a program, an activity the programmer is called upon to do with surprising frequency."

— Lionel E. Deimel Jr. The Uses of Program Reading. *ACM SIGCSE Bulletin*, 17 (2):5–14, 1985. doi:10.1145/382204.382524



Reading Source Code

Darrell R. Raymond

Abstract

Source code is, among other things, a text to be read. In this paper I argue that reading source code is a key activity in software maintenance, and that we can profitably apply experiences and reading systems from text databases to the problem of reading source code. Three prototype systems are presented, and the main features of their design are discussed.

1. Introduction.

Most programmers and project managers have at one time or another maintained software that they did not write. In the classic scenario, one suddenly acquires responsibility for a program consisting of thousands of lines of code. The program serves a wide user community, who complain bitterly if the code is not kept running in a variety of hardware and software environments. The program is highly complex, somewhat flaky, has little or no documentation, and its authors have vanished or otherwise disclaimed responsibility. The user community tries to help by generating a constant stream of bug reports and suggestions for new features and options. While many people claim to know what the program is supposed to do, very few seem to know how the program does it. Other responsibilities cannot be ignored, so rewriting the code from scratch is not an option.

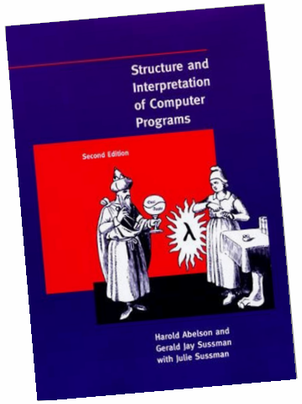
In order to maintain code, one must first understand it. Despite nearly universal acceptance of the importance of design documents, the source code is often the only statement of the program's design. In time-honoured fashion, one might simply dump the code to the line printer and struggle through the print-out for several days. This method has the virtues that the printout can be spread around the room and annotated, but otherwise is not very appealing. Numerous enhancements to this basic approach have been suggested; for instance, the appearance of the printout can improved with pretty-printers[6,14] or typesetting filters.[13, 18] Workstations might be used for online reading and searching with pattern matching tools such as *grep*. Finally, one might be lucky enough to have access to software that automatically constructs indexes to the code.[16]

Whatever approach is used, it is clear that a central activity in software maintenance is *reading*. In maintenance, the main role of source code is not as a compilable entity, but as a human-readable statement of the intent and mechanism of the program. The source describes the program's algorithms and data structures, naturally, but it also contains many clues about the evolution of the program. The choice of variables, the use of comments, the indentation style, and other uncompilable aspects of the code are hints about the software engineering process.[33] A lack of comments and tangled control flow, for instance, suggests that the program is poorly structured and hence potentially error-prone. Mysterious constants suggest hard limits on dynamic structures and problems for users who exercise the code strenuously. Lack of preprocessor statements may indicate inadequate planning for portability. Important clues can also be extracted from design documents, if they exist, since they are themselves a kind of declarative source code.

IBM contact for this paper is Arthur Ryman, Centre for Advanced Studies, IBM Canada Ltd., Dept. 81/894, 895 Don Mills Road, North York, Ontario M3C 1W3.
This paper is available as IBM Canada Laboratory Technical Report TR 74.070.

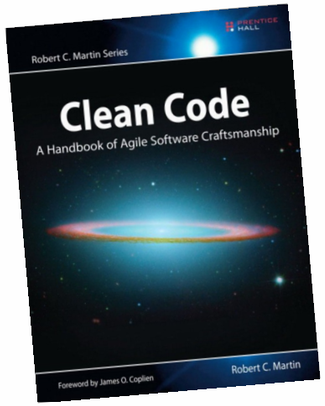
“Whatever approach is used, it is clear that a central activity in software maintenance is reading. In maintenance, the main role of source code is not as a compilable entity, but as a human-readable statement of the intent and mechanism of the program.”

— Darrell R. Raymond. Reading Source Code. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16, 1991. doi:10.5555/962111.962113



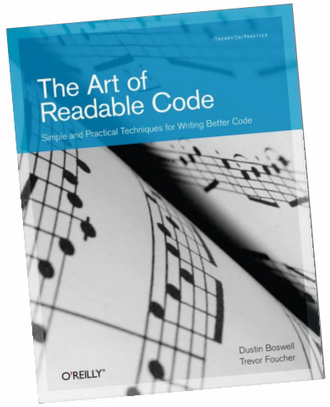
"We want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."

— Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996. doi:10.5555/26777



"Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are constantly reading old code as part of the effort to write new code. Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder."

— Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2008. doi:10.5555/1388398

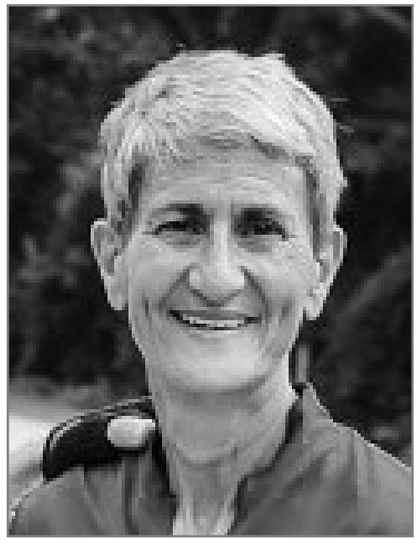


Dustin Boswell

"Code should be written to minimize the time it would take for someone else to understand it. It's so important that we call it The Fundamental Theorem of Readability."

— Dustin Boswell and Trevor Foucher. The Art of Readable Code, 2011

2. Everybody wants higher quality of code, but nobody knows how to measure it.



“It’s not enough to make claims about your software; you must support your claims with measurable evidence.”

— Shari Lawrence Pfleeger. Software Metrics: Progress After 25 Years? *IEEE Software*, 25(6):32–34, 2008. doi:10.1109/MS.2008.160

Mariza A. S. Bigonha

"The application of thresholds could lower the cost of software quality evaluation since they can reduce the amount of software code that should be inspected. Therefore, the thresholds provide a way for quantitative and qualitative evaluations to complement each other, leading to a more efficient quality assessment of object-oriented software systems."

— Tarcísio G. S. Filó, Mariza A. S. Bigonha, and Kecia A. M. Ferreira. Evaluating Thresholds for Object-Oriented Software Metrics. *Journal of the Brazilian Computer Society*, 30(1):313–346, 2024. doi:10.5753/jbcs.2024.3373

Table 3. Catalog of Thresholds (extracted from Filó *et al.* [2015])

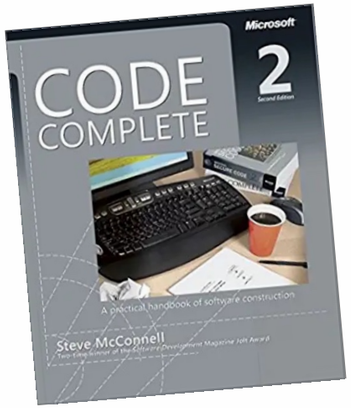
Metrics	*Good/Common*	*Regular/Casual*	*Bad/Uncommon*
AC	$m \leq 7$	$7 < m \leq 39$	$m > 39$
EC	$m \leq 6$	$6 < m \leq 16$	$m > 16$
DIT	$m \leq 2$	$2 < m \leq 4$	$m > 4$
LCOM	$m \leq 0.167$	$0.167 < m \leq 0.725$	$m > 0.725$
MLOC	$m \leq 10$	$10 < m \leq 30$	$m > 30$
NBD	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NOC	$m \leq 11$	$11 < m \leq 28$	$m > 28$
NOF	$m \leq 3$	$3 < m \leq 8$	$m > 8$
NOM	$m \leq 6$	$6 < m \leq 14$	$m > 14$
NORM	$m \leq 2$	$2 < m \leq 4$	$m > 4$
NSC	$m \leq 1$	$1 < m \leq 3$	$m > 3$
NSF	$m \leq 1$	$1 < m \leq 5$	$m > 5$
NSM	$m \leq 1$	$1 < m \leq 3$	$m > 3$
PAR	$m \leq 2$	$2 < m \leq 4$	$m > 4$
RMD	$m \leq 0.467$	$0.467 < m \leq 0.750$	$m > 0.750$
SIX	$m \leq 0.019$	$0.019 < m \leq 1.333$	$m > 1.333$
VG	$m \leq 2$	$2 < m \leq 4$	$m > 4$
WMC	$m \leq 11$	$11 < m \leq 34$	$m > 34$

To evaluate our catalog, in that work, we handled a case study to assess proprietary software from a public organization with a bad internal quality to verify the proposed thresholds' ability to indicate it.

Source: Tarcísio G. S. Filó, Mariza A. S. Bigonha, and Kecia A. M. Ferreira. Evaluating Thresholds for Object-Oriented Software Metrics. *Journal of the Brazilian Computer Society*, 30(1):313–346, 2024. doi:10.5753/jbcs.2024.3373

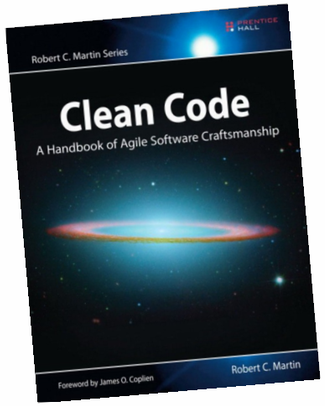
Source: Tarcísio G. S. Filó, Mariza Bigonha, and Kecia Ferreira. A Catalogue of Thresholds for Object-Oriented Software Metrics, 2015

> 3. Size of code is the key contributor to its quality, while Lines of Code (LoC) is the basic measurement of size.



“Studies have found that larger routines are more error-prone than smaller ones. Keeping routines short helps reduce errors and makes the code easier to maintain.”

— Steve McConnell. *Code Complete*. Pearson Education, 2004. doi:10.5555/1096143



"The first rule of functions is that they should be small. The second rule of functions is that they should be even smaller than that."

— Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2008. doi:10.5555/1388398

4. It may be wrong to measure productivity of a programmer by counting lines of code, but for the quality of code the LoC metric is a perfect indicator.

cloc.pl

```
/code/o/eo$ cloc .
    1131 text files.
     951 unique files.
     241 files ignored.

github.com/AlDanial/cloc v 2.02  T=0.69 s (1377.8 files/s, 130550.4 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                           457           3791          21555          31527
YAML                           296            276           6852           7892
XSLT                           105            441           2489           5255
XML                             26             65            304           4058
Maven                           10             40            277           1786
Markdown                        12            184              1            671
ANTLR Grammar                    2            134            187            566
Groovy                          15             43            395            262
Rust                             2             11             46            158
JSON                             3              0              0            150
XSD                              2              8             38            133
Properties                      12             20            252             50
Text                             4              4              0             45
Bourne Shell                     1              1             31             44
Velocity Template Language       1              4              0             37
TOML                             3              3              1             24
-------------------------------------------------------------------------------
SUM:                           951           5025          32428          52658
-------------------------------------------------------------------------------
```

https://github.com/AlDanial/cloc

5. In 2011, Uncle Bob suggested that 200 lines per Java class is a good guideline to stay below.

Table 7
Product size and testing effort

Product	Size	Testing effort (person quarters)
P(C1)	8787	10
P(F1)	154,985	13
P(I1)	87,779	32
P(E1)	181,434	20
P(C2)	121,138	21
P(F3)	237,621	26
P(I3)	43,859	8
P(E10)	161,053	28

Pearson correlation $r = 0.53$, $r_2 = 0.28$, $p = 0.08$.

"We conclude that code size is not a good predictor of testing effort (measured by allocated test engineers) at either product or product line levels. Other metrics, like code coupling and cohesion are more appropriate for such estimation."

Source: Samuel A. Ajila and Razvan T. Dumitrescu. Experimental Use of Code Delta, Code Churn, and Rate of Change to Understand Software Product Line Evolution. *Journal of Systems and Software*, 80(1): 74–91, 2007. doi:10.1016/j.jss.2006.05.034

6. Instead of counting lines, it may be more reasonable to count NCSS (Non Commenting Source Statements), but not always.

LoC vs NCSS

```
1 #!/bin/bash
2 set -e
3
4 # Simple intro:
5 printf "Hello, %s!
6   Your balance is %d." \
7   "${name}" \
8   "$(psql 'SELECT balance
9     FROM user WHERE id = 42')"
```

Lines of Code = ?

NCSS = ?

7. There are 27.8M lines of C code in Linux kernel. What does it tell us?

Largest Open Source Projects

- KDE 57.9M LOC, 19402 years of effort
- NetBSD 50M LOC, 16822 years of effort
- Linux kernel 36.6M LOC, 12,369 years of effort
- Google Chrome 25M LOC, 8131 years of effort
- Mozilla Firefox 20.5M LOC, 6564 years of effort
- FreeBSD 17.2 LOC, 5538 years of effort
- GNOME 15.7 LOC, 4946 years of effort
- LibreOffice 9.5M LOC, 2907 years of effort
- QT 5, 8.2M LOC, 2525 years of effort
- OpenJDK 8.4M LOC, 2583 years of effort
- IntelliJ IDEA community 6.1 LOC, 1850 years of effort
- GNU compiler collections 7.5M LOC, 2,327 years of effort
- Blender 3D 2.1M LOC, 612 years of effort
- PostgreSQL 2.0M LOC, 596 years of effort
- TensorFlow 2.5M LOC, 724 years of effort

Found it on Quora.

How many lines of code are in your repositories? How many lines of code you write every week? How many lines of code you delete every week? How about annually?

8. Java is two times more verbose than Ruby. Does it mean the quality of an average Ruby code is higher?

References

Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996. doi:10.5555/26777.

Samuel A. Ajila and Razvan T. Dumitrescu. Experimental Use of Code Delta, Code Churn, and Rate of Change to Understand Software Product Line Evolution. *Journal of Systems and Software*, 80(1):74–91, 2007. doi:10.1016/j.jss.2006.05.034.

Dustin Boswell and Trevor Foucher. The Art of Readable Code, 2011.

Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1978. doi:10.5555/540031.

Lionel E. Deimel Jr. The Uses of Program Reading. *ACM SIGCSE Bulletin*, 17(2):5–14, 1985. doi:10.1145/382204.382524.

Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. doi:10.1145/355604.361591.

Tarcísio G. S. Filó, Mariza Bigonha, and Kecia Ferreira. A Catalogue of Thresholds for Object-Oriented Software Metrics, 2015.

Tarcísio G. S. Filó, Mariza A. S. Bigonha, and Kecia A. M. Ferreira. Evaluating Thresholds for Object-Oriented Software Metrics. *Journal of the Brazilian Computer Society*, 30(1):313–346, 2024. doi:10.5753/jbcs.2024.3373.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008. doi:10.5555/1388398.

Steve McConnell. *Code Complete*. Pearson Education, 2004. doi:10.5555/1096143.

Shari Lawrence Pfleeger. Software Metrics: Progress After 25 Years? *IEEE Software*, 25(6):32–34, 2008. doi:10.1109/MS.2008.160.

Darrell R. Raymond. Reading Source Code. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16, 1991. doi:10.5555/962111.962113.