

Code Style

YEGOR BUGAYENKO

Lecture #22 out of 24
80 minutes

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as web sites. Copyright belongs to their respected authors.

Which One Looks Better for You?

C:

```
1 int f(int n)
2 {
3     if (n == 1 || n < 2)
4         return 1;
5     int r = f (n-1);
6     int r2 = f(n - 2);
7     return r +r2;
8 }
```

Java:

```
1 int fibonacci(int n) {
2     if (n <= 2) {
3         return 1;
4     }
5     return fibonacci(n - 1)
6         + fibonacci(n - 2);
7 }
```

Ruby:

```
1 def fibonacci(int n)
2     return 1 if n <= 2
3     fibonacci(n - 1)
4     + fibonacci(n - 2)
5 end
```



BRIAN KERNIGHAN

“The harder it is for people to grasp the intent of any given section, the longer it will be before the program becomes operational. Trying to outsmart a compiler defeats much of the purpose of using one. Write clearly — don’t sacrifice clarity for ‘efficiency.’”

— Brian W. Kernighan and Phillip James Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., 1974



DAVID MARCA

“The effective utilization of extra spaces, blank lines, or special characters can illuminate the logical structure of a program. So we should not be afraid to: indent, indent consistently (3 is a readable minimum), start each statement on a new line, put only one word on a line, use extra pages to visually collect code, put blank lines between code, align keywords, separate code from comments with white space.”

— David Marca. Some Pascal Style Guidelines. *ACM SIGPLAN Notices*, 16(4): 70–80, 1981

```

PROCEDURE GETLINE(
    PROMPT : STRING
    VAR LINE : STRING
    VAR MORELINES : BOOLEAN
)
:
BEGIN
    WRITE(PROMPT);
    READLN(LINE);

    IF EMPTY(LINE)
    THEN
        MORELINES := FALSE
    ELSE
        BEGIN
            LINE := CONCAT(LINE, BLANK);
            MORELINES := TRUE
        END
    END
    (*FI*)
END (*GETLINE*)

```

“The best style enforcer is the computer ... but only if we can easily cope with its ever-present restrictions.”

Source: David Marca. Some Pascal Style Guidelines.
ACM SIGPLAN Notices, 16(4):70–80, 1981



MICHAEL J. REES

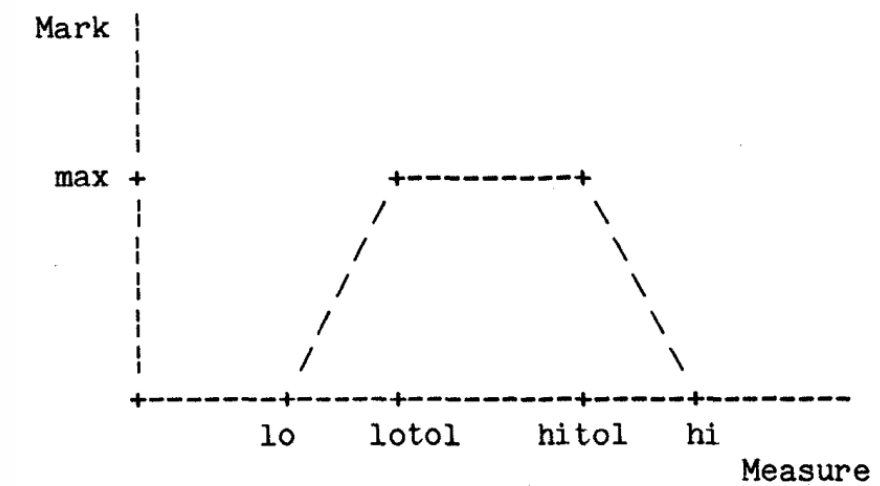
“STYLE was designed to input the source of a syntactically correct Pascal program, make simple measurements on a one-pass line-by-line basis, and yield a style mark out of 100%.”

— Michael J. Rees. Automatic Assessment Aids for Pascal Programs. *ACM SIGPLAN Notices*, 17(10):33–42, 1982

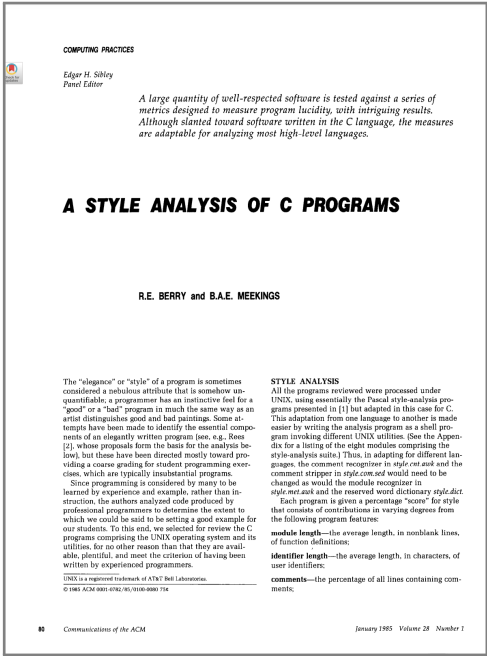
Rees Score, for Pascal

Measure	max	lo	hi	lotol	hitol
chars/line	15	12	30	15	25
% comments	10	15	35	20	25
% indent	12	60	90	70	80
% blank lines	5	8	20	10	15
% spaces	8	8	20	12	18
proc/fn length	20	10	50	20	35
# res. words	10	22	41	26	40
id. length	20	7	16	9	15
# ids	0	0	0	0	0

labels & gotos	20	1	200	3	199



Source: Michael J. Rees. Automatic Assessment Aids for Pascal Programs. *ACM SIGPLAN Notices*, 17(10): 33–42, 1982



“The ‘elegance’ or ‘style’ of a program is sometimes considered a nebulous attribute that is somehow unquantifiable; a programmer has an instinctive feel for a ‘good’ or a ‘bad’ program in much the same way as an artist distinguishes good and bad paintings.”

— R E Berry and B AE Meekings. A Style Analysis of C Programs. *Communications of the ACM*, 28(1):80–88, 1985

Berry-Meekings Score, for C Programs

TABLE I. Metric Boundary Values

Metric	%	L	S	F	H
Module length	15	4	10	25	35
Identifier length	14	4	5	10	14
Comment lines (%)	12	8	15	25	35
Indentation (%)	12	8	24	48	60
Blank lines (%)	11	8	15	30	35
Characters per line	9	8	12	25	30
Spaces per line	8	1	4	10	12
Defines (%)	8	10	15	25	30
Reserved words	6	4	16	30	36
Include files	5	0	3	3	4
Gotos	-20	1	3	99	99

Source: R E Berry and B AE Meekings. A Style Analysis of C Programs. *Communications of the ACM*, 28(1):80–88, 1985

“An individual score for each metric is determined by reference to the value in this table for

1. the point L , below which no score is obtained;
2. the point S , the start of the “ideal” range for the metric;
3. the point F , the end of the ideal range;
4. the point H , above which no score is obtained.”



WARREN HARRISON

“To determine the relationship (if any) between the style metric and error proneness of each module, we performed a simple correlation analysis. The results were discouraging in the sense that a correlation of only -0.052 existed between the observed error frequency and the style metric, suggesting that the style metric bore little relationship to the error frequency encountered in our data.”

— Warren Harrison and Curtis R. Cook. A Note on the Berry-Meekings Style Metric. *Communications of the ACM*, 29(2):123–125, 1986

My Favorite Style Checkers

- ESLint (2013) for JavaScript
- Clang-Tidy (2007?) for C++
- Pylint (2006) for Python
- Rubocop (2012) for Ruby
- PHP_CodeSniffer (2011) for PHP
- rustfmt (2015) for Rust
- Qulice by [Bugayenko, 2014] for Java: Checkstyle (2001) + PMD (2022)

How Many Rules in Style Checkers?

- 690+ in Clang-Tidy (C++)
- 550+ in Rubocop (Ruby)
- 400+ in PMD (Java)
- 130+ in Checkstyle (Java)
- 120+ in Pylint (Python)

Some/most of the rules no only check style, but also find bugs.

Some Exotic Style Checkers

- Shellcheck for Bash
- markdownlint for Markdown
- Checkmake for Makefile
- xcop for XML



CHRISTIAN COLLBERG

“Code obfuscation means one user runs an application through an obfuscator, a program that transforms the application into one that is functionally identical to the original but which is much more difficult for another user to understand.”

— Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997



HENRY LEDGARD

“An individual’s body language helps clarify the spoken word. In a similar sense, the programmer relies on white space—what is not said directly—in the code to communicate logic, intent, and understanding.”

— Robert Green and Henry Ledgard. Coding Guidelines: Finding the Art in the Science. *Communications of the ACM*, 54(12):57–63, 2011

Figure 9. Examples of K&R, ANSI, and Whitesmiths coding styles.

<pre>if (expression) { statements }</pre>	<pre>if (expression) { statements }</pre>	<pre>if (expression) { statements }</pre>
-------------------------------------------------------	-----------------------------------------------------------	-----------------------------------------------------------

Source: Robert Green and Henry Ledgard. Coding Guidelines: Finding the Art in the Science. *Communications of the ACM*, 54(12):57–63, 2011



PETER C. RIGBY

“We list the reasons why our interviewees rejected a patch or required further modification before accepting it: Poor quality, Violation of style, Gratuitous changes mixed with ‘true’ changes, Code does not do or fix what it claims to or introduces new bugs, Fix conflicts with existing code, Use of incorrect API or library.”

— Peter C. Rigby and Margaret-Anne Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550, 2011



MORITZ BELLER

“Most Automated Static Analysis Tools configurations deviate slightly from the default, but hardly any introduce new custom analyses. ”

— Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016

TABLE V
SUMMARY OF RULE CHANGES FROM DEFAULT CONFIGURATIONS.

Tool	Changed	Reconfigured	No Deviations	Total
ESLINT	80.5%	5.7%	13.8%	4,274
FINDBUGS	93.0%	—	7.0%	2,057
JSHINT	89.6%	0.7%	9.7%	104,914
JSL	94.6%	—	5.4%	848
PYLINT	53.3%	—	46.7%	3,951
RUBOCOP	79.1%	3.2%	17.7%	9,579

Source: Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016



JENNIFER BAUER

“While the perceptual processing of code is required to understand it, higher level processing, such as understanding its semantics and reasoning about its functionality, affect program comprehensibility more strongly. The influence of indentation could have been masked by these side effects, so it might well be that the effect of indentation comes more into play when the code is longer and more complex.”

— Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. Indentation: Simply a Matter of Style or Support for Program Comprehension? In *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*, pages 154–164. IEEE, 2019



WEIQIN ZOU

“A pull request that is consistent with the current code style tends to be merged into the codebase more easily (faster).”

— Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. Indentation: Simply a Matter of Style or Support for Program Comprehension? In *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*, pages 154–164. IEEE, 2019

References

- Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. Indentation: Simply a Matter of Style or Support for Program Comprehension? In *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*, pages 154–164. IEEE, 2019.
- Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.
- R E Berry and B AE Meekings. A Style Analysis of C Programs. *Communications of the ACM*, 28(1): 80–88, 1985.
- Yegor Bugayenko. Strict Control of Java Code Quality. <https://www.yegor256.com/140813.html>, August 2014. [Online; accessed 26-02-2024].
- Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.
- Robert Green and Henry Ledgard. Coding Guidelines: Finding the Art in the Science. *Communications of the ACM*, 54(12):57–63, 2011.
- Warren Harrison and Curtis R. Cook. A Note on the Berry-Meekings Style Metric. *Communications of the ACM*, 29(2):123–125, 1986.
- Brian W. Kernighan and Phillip James Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., 1974.
- David Marca. Some Pascal Style Guidelines. *ACM SIGPLAN Notices*, 16(4):70–80, 1981.
- Michael J. Rees. Automatic Assessment Aids for Pascal Programs. *ACM SIGPLAN Notices*, 17(10): 33–42, 1982.
- Peter C. Rigby and Margaret-Anne Storey. Understanding Broadcast Based Peer Review on

Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software*

Engineering, pages 541–550, 2011.