Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1308-1313
DOI: 10.23940/ijpe.18.06.p22.13081313

Coding Standards and Human Nature

Michael Dorin* and Sergio Montenegro

Universität Würzburg, Würzburg, 97070, Germany

Abstract

Intuition tells us that code that is difficult to review is likely complicated and faulty. Many organizations will create a coding standard to encourage higher quality software development. Coding standards are not always followed, and even when they are, complicated code continues to be written. Human nature demonstrates that people do not put effort into activities that they believe to be unproductive. It is also true that people have limited capacity for remembering and following directions, so extra requirements from a coding standard may even inhibit creativity. Because of human behavior, this paper recommends that organizations have two layers of a coding standard. The first layer should be easy to remember items. The second layer should be the long-established coding standard an organization wishes to comply with.

Keywords: coding standard; human behavior; multi-layer coding standard; creativity

(Submitted on March 13, 2018; Revised on April 26, 2018; Accepted on May 12, 2018)

1. Introduction

Coding standards are thought to be an excellent way to encourage consistent software quality. As stated by Jean-Pierre Rosen, the goal of a coding standard is to improve the quality of code, not just for the sake of having one [6]. An unnecessarily complicated program is not generally thought of as quality code. Because not all software engineers have mastery or even respect for established coding rules, positive influence from guidelines is not always found. A significant and essential part of coding standards is a defined coding style, and while modern programming editors can enforce many stylistic rules, some specific tasks are still up to the programmer. If a programmer is required to remember too many things, it is possible that some coding standard requirements will be missed. A discussion of literature investigating other reasons why essential directions are not always followed is also presented. Additional research was also conducted to clarify what is considered complicated software. A resulting recommendation is for organizations to create two-layer coding standards. Layer-1 should be comprised of rules that are easy to remember, and Layer-2 should enforce more formal rules. It is not suggested that detailed coding standards be eliminated; instead, they should be supplemented.

2. Human Nature and Following Rules

In 1994, Lawrence Zeitlin conducted an experiment involving activities with chainsaws. Safety orientation was provided, and upon completion of training, understanding of the rules was verified. When the investigation was complete, it was determined that only 55.6% of the group followed the rules. Perhaps even more surprising, more experienced users were less in compliance [20]. A paper by Reason et al. describes the steps organizations with safety-critical applications take to assure compliance with production rules [13]. Consider how aircraft maintenance workers are required to follow workplace standards for their safety as well as the safety of those using the aircrafts they have maintained. If we couple the lack of rule compliance with the immediate danger presented by chainsaw operation, it is not too surprising when software engineers disregard some of the rules of a coding standard as there is no physical danger present. However, it is possible that much like the aircraft

* Corresponding author.
E-mail address: michael_andrew.dorin@stud-mail.uni-wuerzburg.de

maintenance workers, the work of the software engineer may put someone's life in peril. Unlike the aircraft maintenance worker, it is possible that faulty software is not just on one plane, but many planes.
Individual differences in working memory also affect following directions. An observation made by George Miller is that the number of objects an average person can hold in working memory is about seven [8]. If you consider a mind maintaining a collection of coding standard rules while an engineer is creatively writing software, it seems probable that some of the rules are likely to be dropped. Some participants in an online conversation felt that coding standards are a creativity and productivity destroyer [4]. It has even been suggested that disregarding the rules may be a rebellious act directed towards company policy or management [14].

The governmental Occupational Safety and Health Administration (OSHA) provides one more reason behind failure to follow directions. OSHA feels a lack of compliance is because many do not understand why compliance is essential [12]. For example, it may be concluded that after safety training and more personal practice with chainsaws, the users did not see the connection of the rules to safety and disregarded them. The chainsaw experiment showed that even with immediate physical consequences of failure to comply, guidelines are still not followed. It seems that if a coding standard is made, it needs to have the most critical rules concisely explained and therefore can be easy to remember. The software engineer will need to have a clear understanding that a mistake in the software they write may have dire consequences on the lives they touch. Dire consequences can exist for all types of software, not merely software written for the control apparatus. For example, Lewis Morgan of IT Governance Blog states his January 2018 report of data breaches is one of the most extensive lists he ever put together [11]. This is undoubtedly a software failing with no physical consequences.

3. Metrics and Measuring Success

There should also be a plan on how to measure success of a coding standard. “A Metric is a quantitative measure of the degree to which a system, system component, or process possess a given attribute” [15]. Accumulating metrics has been practiced in software development for many years, and the oldest metric likely is lines of code (LOC). Two important metrics worthy of consideration are cyclomatic complexity and Halstead measurements of difficulty. In 1976, Thomas McCabe developed the cyclomatic complexity metric, which is a measure of the number of linearly independent paths through a program [10]. In 1977, Halstead published complexity metrics based on distinct operators, distinct operands, the total number of operators, and the total number of operands [5]. With these attributes, a value for Halstead difficulty and Halstead effort can be made. Newer metrics than lines of code (loc), Halstead, and McCabe indeed exist; however, these established metrics are easy to understand, and tools for them are readily found and applied to many programming languages.

4. Complicated Software

As previously mentioned, a desired outcome of coding standards is to produce higher quality software. McCabe Software is a professional organization in the business of software quality. A whitepaper from McCabe describes how complex programs potentially have security problems and are harder to understand and test [9]. Coding standards should encourage the development of less complicated code.

For the sake of argument, let us accept that software that is unpleasant to review is likely complicated. In the paper “Coding for Inspections and Reviews,” a survey was conducted where volunteers examined source code and immediately indicated if they felt a program would be pleasant or unpleasant to review. This survey paper presented source code from a collection of C++ programs. The paper analyzed each source file to calculate software complexity metrics as well as stylistic characteristics. An interesting outcome of this paper is that traditional metrics of complexity seemed to correspond nicely with the data obtained by human interpretation of “reviewability.” The paper found that files that were considered pleasant to review generally had characteristics of lower cyclomatic complexity as well as lower Halstead difficulty. It also found that average lengths of functions were higher in unpleasant to review code [1]. Programming style was also very important, and aspects of these results are covered in the Layer 1 recommendations.

5. Software Design

The importance of software design should also not be overlooked, and designing an architecture should not be an afterthought. Schach describes the code-and-fix life cycle model as a software product implemented without requirements, specifications, or any attempt at design [17]. Schach points out that maintenance costs of the code-and-fix model are higher than for formally designed software products [17]. The traditional view of maintenance is that it begins after product delivery to correct faults. Many organizations have a large, existing code base, and many projects are products based on code reuse. We might consider

maintenance beginning as soon as the source code needs to be looked at again. A software engineering mentor once said that reading great code is like reading a book [2]. The intent of the author is so clear that the code “reads” like a story [3]. Successful literary works are rarely written without some plan or outline. Even the best coding standard will not make up for no design.

6. Two Layer Coding Standard

The Jet Propulsion Laboratory provides a thorough coding standard for applications including requirements for flight-related software [7]. Linus Torvalds has produced a very real-world coding standard as well, and many of his ideas are incorporated as part of the Layer-1 recommendations described below [18]. As stated previously, there is no proposal for the elimination of comprehensive coding standards. What is proposed here is to put a small layer or shim on top of current and accepted standards. This top layer is to provide a fixed number of "must-haves" that are not impossible for a coding person to remember. The must-haves need to be completely compatible with more detailed standards, such as the JPL standard, as the written code will ultimately have to comply.

In the 1960 movie "The Magnificent Seven," a group of seven was hired to protect a small village in Mexico from a group of plundering bandits [16]. With the obvious consideration to Miller's law, it seems for the first layer coding standard that it is appropriate to have seven rules protecting software from marauding bugs.

6.1. Consistent Indentation

It has been shown that source code with proper indentation was generally more pleasant to review and indentation is also prominently covered in the Linus Torvalds standard. Proper indentation gives a programmer a picture of the control flow of a function or method with a glance. Programmers should not have to struggle or rely on comments to find the beginning and end of a block [3]. Indentation should be consistent across all constructs such as structures and switch statements. The Torvalds standard suggests indentations match at eight characters, and Mr. Torvalds argues that if you need to indent more than three times, you should rewrite your code [18]. Intuitively, it seems that multiple indentations lead to more complicated code.

6.2. Limit Preprocessor Directives

Depending on the programming language used, preprocessor directives can get in the way of understanding. In ‘C' language, many conditional preprocessor directives, #if, #ifdef, #else, #ifndef, etc., can make code entirely undesirable for review and ultimately unmaintainable. When nesting #ifdefs and #ifs, it can be very difficult to determine whether statements are included in a compilation without painful examination of preceding lines or simulation of the execution of the preprocessor [3].

6.3. No "dead code."

There should never be "dead code.” Dead code should not be found in comments or anywhere else in a source file. It is extremely frustrating to spend time looking at and trying to understand a function only to find out later this function is no longer called. Do not use "dead code" for a source control system [18].

6.4. Limit Line length

An unsurprising result found in the Dorin paper was that programs which had lines longer than 120 characters were not pleasant to review. Limiting line length to what can be seen without effort is an essential consideration for any programming language [17]. The Torvalds coding standard suggests that lines should not be longer than 80 characters [18].

6.5. Be mindful of working towards efficiency

“More computing sins have been committed in the name of efficiency (without necessarily achieving it) than any other reason" [19]. The Dorin paper indicated that programs undesirable to review also generally had larger Halstead Difficulties and higher Cyclomatic Complexities. Code having high cyclomatic complexities or Halstead difficulties should be considered technical debt and rewritten [17].

6.6. Show Intent

When reviewing code, it is crucial that code reviewers understand the intent of your work. Comments should tell what a

function does, not how a function works. It is not okay to use comments as a substitute for weak program structure or poorly named functions. The code should be self-documenting, and its operation should be apparent to a reviewer [17]. A program written in self-documenting code has variable and function names chosen carefully, with the code crafted exquisitely and almost wholly removing the need for comments [3].

The ternary operator was shown to be something "unpleasant to review" [1]. This is not difficult to believe, as the ternary operator is not commonly in use. While expanding the ternary operator takes more lines of code, the intent of the code author is evident.

Braces for even one statement also make the programmer's intention clear. For example, consider nested and compound if-statements. See Example 2 and Example 3 from the Drew Technology coding standard below [3]. Indicating a begin and end for even one statement makes the intent abundantly clear.

```
if ( a == ONE_THING )
    do_something();
else if ( a == ANOTHER_THING )
    do_something_else();
else if ( a == YET_ANOTHER )
    do_yet_another();
else
    do_last_thing();
```

Figure 1. An example of unclear statements

```
if ( a == ONE_THING )
{
   do_something();
}
else
{
   if ( a == ANOTHER_THING )
   {
      do_something_else
   }
   else
   {
      if ( a == YET_ANOTHER )
      {
         do_yet_another()
      }
      else
      {
         do_last_thing();
      }
   }
}
```

Figure 2. Clear statements with compound-ifs expanded

The example in Figure 1 is not ambiguous to the compiler, but it can be unclear to a human reviewer. Figure 2 shows how the compound-if statement is expanded and is also a good demonstration of clear programmer intent.

Use of parentheses is also an effective way to make intent clear, even for those who have learned the many precedence rules. Not everybody has learned or remembers all the precedence rules.

```
while((a != b) && ((c == d) || (e == f)))
{
  do_something();
}
```

Figure 3. Using parentheses to avoid misunderstanding

The placement of the parenthesis in Figure 3 telegraphs the intent of the programmer, and there is no doubt as to how the

logical expression will be evaluated [3].

6.7. Have a consistent and straightforward naming convention

Names of variables and functions should be descriptive and consistent. In the 1970s, variable name length was a concern. This is no longer the case, and auto-complete in coding editors eliminates the need to memorize complicated identifiers, so there is no excuse for not having descriptive names. Abbreviations should not be used, as people tend to lose consistency in making abbreviations over time [17]. When reviewing code, it is sometimes required to guess what a variable name is during a debugging exercise. When named consistently, it can be easy to deduce. If abbreviations are used, especially if they are used inconsistently, it can be challenging to guess a name to find the code that needs evaluation. For example, consider Schach's example of how the following four variables are declared: averageFreq, frequencyMaximum, minFr, and frqncyTotl. A programmer reviewing the code must know if freq, frequency, fr, and frqncy all refer to the same thing [17]. Finally, choose a style such as camelCase or under_bars, but do not mix both in the same project.

7. Conclusions

Software development organizations are in constant pursuit of creating a higher quality product. In many aspects of life, established rules and instructions are not always followed, and this is the case when writing software. Implementing a two-layer coding standard will help software engineers write compliant code or write code that is easier to bring into compliance. By understanding a limited set of basic rules, engineers can produce less complicated programs and write code with fewer faults. For many organizations, maintenance costs are as much as three times the original software project costs [17]. Writing less complicated, less faulty code will have the desired effect of making maintenance easier and bring down maintenance costs. A coding standard will not magically make bad programmers into good programmers and cannot make up for a bad design, but it can substantially benefit a development organization [3]. Organizations should create a Layer-1 coding standard from easy-to-remember rules to foster creation of higher quality, less complicated software.

Acknowledgments

Special thanks to Michael Drew for pointing me to the Drew Technologies coding standard.

References

1. Dorin, Michael A. (2018). Coding for Reviews and Inspections. XP 2018. Porto, Portugal.
2. Drew, Michael A. (2018). Personal Communication
3. Drew, Michael L. (1999, October 22). A Coding Standard for the C Programming Language. Retrieved from http://www.drewtech.info/pdf/code_std.pdf
4. Eaton, David (2014). Answer to: Do all programmers follow coding standards. Retrieved from https://www.quora.com/Do-all-programmers-follow-coding-standards
5. Halstead, Maurice H. (1977). Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7.
6. Jean-Pierre Rosen. 2011. Designing and checking coding standards for ada. In Proceedings of the 2011 ACM annual international conference on Special interest group on the ada programming language (SIGAda '11), Michael Feldman, Dan Eilers, Jean-Pierre Rosen, Frank Singhoff, Julien DeLange, Mark Gardinier, and Karl Nyberg (Eds.). ACM, New York, NY, USA, 13-14. DOI=http://dx.doi.org/10.1145/2070337.2070345
7. Jet Propulsion Laboratory California Institue of Technology. (2009). JPL Institutional Coding Standard for the C Programming Language. Retrieved from https://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf
8. Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review.
9. McCabe Software, More Complex = Less Secure, Miss a Test Path and You Could Get Hacked. http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf
10. T. J. McCabe. 1976. A Complexity Measure. IEEE Trans. Softw. Eng. 2, 4 (July 1976), 308-320. DOI=http://dx.doi.org/10.1109/TSE.1976.233837
11. Morgan, Lewis. (2018, January). List of data breaches and cyber attacks in January 2018. Retrieved from https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-january-201-2
12. Oregon OSHA. (Retrieved April 2018). Safety and the Supervisor. Retrieved from http://osha.oregon.gov/OSHAEdu/safety-and-the-supervisor/1-160i.pdf
13. Reason, J., Parker, D., & Lawton, R. (1998). Organizational controls and safety: The varieties of rule related behaviour. Journal of occupational and organizational psychology, 71(4), 289-304.
14. Rodger, Richard. (2012). Why I Have Given Up on Coding Standards. Retrieved from http://www.richardrodger.com/2012/11/03/why-i-have-given-up-on-coding-standards
15. Srikant Sharma. 2015. The Importance of Software Metrics https://www.linkedin.com/pulse/importance-software-metrics-srikant-sharma-tqm/

16. Dir. Sturges, John. Perf. Yul Brynner, Eli Wallach, Steve McQueen, Charles Bronson, Horst Buchholz, Brad Dexter, James Coburn. The Magnificent Seven. (1960). The Magnificent Seven. MGM/UA.
17. Stephen R. Schach. (2001). Object-Oriented and Classical Software Engineering (5th ed.). McGraw-Hill Pub. Co.
18. Torvalds, Linus B. (2001). Linux kernel coding style. Retrieved from https://github.com/torvalds/linux/blob/master/Documentation/process/coding-style.rst
19. Wulf, William. (1972, August). "A Case Against the GOTO," Proceedings of the 25th National ACM Conference
20. Zeitlin, Lawrence R. "Failure to follow safety instructions: Faulty communication or risky decisions?." Human Factors36.1 (1994): 172-181.