Origins of poor code readability
(Short Paper, Conjectural)

Arun Saha[1]
Fujitsu Network Communications
1250 E Arques Avenue, Sunnyvale, CA, 94085
arunksaha@gmail.com

Keywords: software readability, psychology of programming

Abstract

The topic of source code readability has paramount importance in software engineering. Literature exists on how to write readable code; how to create analytical models and automatically predict readability; and how readability influences software cost and eventually the economy. In this article we follow a different path; we explore the question of why and how unreadable code gets written.

Motivation

The software community is increasingly concerned about the code readability (Buse et al. 2010):

> The readability of source code is related to its maintainability, and is thus a key factor in overall software quality. Typically, maintenance consumes over 70% of the total lifecycle cost of a software product (Boehm 2001). Aggarwal claims that source code readability and documentation readability are both critical to the maintainability of a project. Other researchers have noted that the act of reading code is the most time-consuming component of all maintenance activities. Readability is so significant, in fact, that Elshoff and Marcotty, after recognizing that many commercial programs were much more difficult to read than necessary, proposed adding a development phase in which the program is made more readable. Knight and Myers suggested that one phase of software inspection should be a check of the source code for readability to ensure maintainability, portability, and reusability of the code.

Readability is a human judgment of how easy it is to understand program source code. It is important to note that readability is unrelated to functionality; the readability of source code can be increased or decreased without affecting program correctness. Unlike some other aspects which have objective definitions readability is always subjective. As the proverb goes, “Beauty is in the eye of the beholder;” readability is very much a human construct. What constitutes readable code may differ from programmer to programmer or from one group of programmers to another, or from programmers of one language to programmers of another language. We recognize this subjectivity on small snippets of code, but we believe that for relatively significant chunks of code (for example, a class definition and implementation), the human judgment generally reaches agreement.

In this article, we first explore the various importance of readability. Later, we explore how and why readability gets compromised.

[1] The opinions expressed in this article are solely the author’s – not his employer’s.

Importance of Readability

The importance of code readability is clearly stated in the book Structure and Interpretation of Computer Programs (Abelson and Sussman 1996):

> Programs must be written for people to read, and only incidentally for machines to execute.

Readability of source code is important for various reasons, as highlighted in this section.

Understanding

A portion of code written by a programmer (author) must be understandable by current stakeholders, e.g. the author's immediate team members (and even the author at a future time). But that is not all; the code must be understandable by future stakeholders, e.g. rest of the programmers in the project or organization, especially programmers who might be hired in future.

A program might be an application program, a library, a framework, or any other software. Understanding code has many perspectives, the two main ones are: application-level understanding and programming language-level understanding. In application-level understanding, a reader follows the business logic of the code: What is the code doing? Which use case or use cases is it necessary for? What are the inputs, outputs, and side effects? Does it fit with business logic of the rest of the code? Is it extensible, at least in the near and foreseeable future? In programming language-level understanding, a reader must understand the detailed mechanics of the code, the use of language constructs, the use of data structures and algorithms, the API, the construction and naming of variables and methods – in summary all the details.

For the current stakeholders, the code must be understandable both from an application perspective and a language perspective. However, as the software development project progresses through different stages of its lifecycle, the stakeholders also change. So, it is necessary for the code to be code understandable by future stakeholders as well.

Understanding encompasses a wide spectrum. At one end is the architectural understanding, useful mainly for newcomers and personnel not involved in day-to-day maintenance of the code.

The other end of the spectrum is detailed understanding. This involves high-level understanding plus answers to various questions: What are the namespaces/packages and classes? What are the object relationships? What is the responsibility of each class? What are the methods in each class?

In the middle of the spectrum is working understanding. It is neither high-level architectural understanding, nor detailed understanding. Rather, it is a mechanical understanding from inputs to outputs. It is almost the same as programming language-level understanding.

Generally, the needs of current stakeholders are towards the "detailed" end, the architects need to develop understanding at the "high-level" end, and the rest of the stakeholders are somewhere in the middle with a varying level of working understanding. That is normal and desirable.

Interfacing

New modules interface with existing modules. At that time, the author(s) of a new module has to understand the interfaces exposed by the existing module. In theory, understanding only the interface is enough. But, in reality, the inner details of the existing modules need to be understood, at least at a high level.

Extending/Enhancing

When a module is extended or enhanced, the existing model and concepts need to be understood so that the extension aligns with the existing code.

Fixing

This is perhaps the most cited reason for readability. Many times, a programmer is responsible for fixing an unfamiliar module. The existing code, the classes, the methods, the variables, their names, and the control flow, must be understandable enough so that such a newcomer to the module can easily identify the place to fix and the nature of the fix. If the previous author(s) have addressed the working understanding required for non-current stakeholders, then it is easier for the maintenance programmers.

Reasons for Readability Deterioration

There can’t be a Standard for Readability

According to the recommendation and common wisdom, every project generally follows a coding standard. If programming is considered as a craft, i.e. mixture of science and art, then readability falls under art. And art cannot be standardized. So, unfortunately, there can’t be a standard for readability. (There are very good guidelines that can be followed, though.)

Lack of Culture

Since readability cannot be standardized as a process, the alternative is to have a culture among the programmers to write readable code. We must recognize here that having a culture does not mean having the ultimate readable code right from the outset. Rather, the culture of readability takes time to develop, and allows for a feedback mechanism among the programmers. The result of this feedback mechanism should be that the author improves the readability of the previously committed code in subsequent commits, or other programmers do that in consultation with the author or author’s team. This culture is a continuous process, without requiring specific instruction or time allotment.

Lack of Awareness

Sometimes it may happen that programmers are unaware of the importance of readability. This mainly happens with programmers who are at the beginning of their career. They falsely assume that it is enough for code to be functionally correct. In addition, experienced programmers at times sacrifice readability in the name of efficiency.

Readability has no conflict with efficiency. Readability comes from properly following good programming guidelines, such as (but not limited to): abstraction, encapsulation on the conceptual front and avoiding repetition, following consistent naming, spacing, indenting, and bracing style on the pragmatic front.

Schedule Pressure

When a programmer is pressurized to commit a piece of code in an unreasonably short time, she[2] has two main options: (1) Refuse to succumb to that pressure and either (a) negotiate a more viable deadline or (b) consume the amount of time allotted; and (2) Do whatever is possible in the allotted time. It is a separate discussion on what the ideal course of action is, but in reality most programmers choose the second option. This may happen during any phase of development. In the early phase of a project, the requested piece of code can be a feature or an infrastructure with a timescale of the order of months/weeks; in later phases it can be a bug fix with a timescale of the order of days. As a result of this phenomenon, code gets written only with functionality in mind, and there is no time to improve on the readability.

Lack of Programmer Recognition

It is not uncommon for some programmers to be aware and concerned about readability and really write highly readable code. Furthermore, it is not uncommon for such programmers to be improving readability by spending extra time outside usual hours because their “official” schedule does not budget for such activities.

[2] To avoid repetitive use of “he or she”, this article uses “she” as a pronoun for all programmers.

But, has anyone ever heard of a programmer being recognized (via promotion, pay raise, or reward) for writing readable code? Probably not!

Other programmers who read/modify/interface a good programmer's code know about it, and secretly express thankfulness, but the effort expended to write readable code never gets institutional recognition.

Often a programmer's status in the team hierarchy depends on how many different features/modules she has authored. It does not matter if those modules are well written from the software engineering perspective. In other words, an author of two poorly written modules enjoys higher status than an author of a single well-written module. It is a different matter that over time, the project expends much more time and resources to maintain those poorly written modules.

Economic Factors

Code has two distinctly different kinds of costs: (i) one-time cost of initial construction, and (ii) recurring cost of maintenance. Well-written code has relatively high cost of initial construction, but relatively low cost for maintenance. For poorly written code, the opposite applies.

If a project team disregards that factor and spends less in initial construction, the outcome is less-readable code, and eventually higher maintenance costs.

Knowledge Captivity

Sometimes programmers are judged and valued based on the amount of code they know about. In particular, if there is some (important) piece of code which only a single programmer knows about, then that programmer enjoys more importance than his or her peers. Sometimes programmers treat that knowledge as job security. In that environment, the natural tendency of programmers is to know more about others' modules but reveal less about one's own modules. From the game theory perspective, every programmer might think the same which collectively leads to unreadable, and sometimes obfuscated, code.

Broken Window

The Broken Window Theory (Hunt et al. 1999) says:

> Don't leave "broken windows" (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered. If there is insufficient time to fix it properly, then board it up.

Though this is very good advice, in practice it is not followed as often as it should. Fixing broken windows in code is mostly a thankless job. There is much more glory in designing or authoring a new module, or writing (context dependent) special areas of code, or fixing a critical bug (bonus points if it is a customer-raised bug) than making a functionally correct code readable.

Even if there is a will, there is no time allotted for readability improvements. One common response is, "let's ship this release and then clean up." But, according to the theory, delaying to fix the windows worsens the situation, and the fixes which are trivial when discovered become complex when the time comes for cleanup.

Once unreadable code gets committed it is hard to improve it. One cannot open a bug for lack of readability. Even if it is possible to do so, no one wants to open such a bug and become a "bad guy" to the project manager. If the next programmer touching the file is passionate or enthusiastic, then she can perform some enhancements. But if the next programmer is not interested, then the readability improvements won't happen.

Lack of Ownership

Humans maintain and polish things that they own. If pieces of code do not have ownership, then they rot and become unreadable. This can happen when the original author is no longer available to maintain the code, and no other programmer or team is assigned for maintenance.

Long Edit-Compile-Test Cycles

Readability improvements need at least a complete cycle of edit, build, and regression test. If the cycle is long, then programmers, even if they agree in principle, avoid readability improvement edits.

Lack of Strong Peer Review

Most software projects have established processes for code review. But, code reviewers seldom flag code based on readability. If they do, their comments are tactfully ignored. Gradually, they are socially engineered to stop raising such comments.

Conclusion

Every piece of unreadable code has some origin and in this article we explore the possible origins. Of course, not all the reasons apply in every instance, but unreadability is often the result of more than one cause.

This is a small step in exploring the important topic of code readability. We hope that if the origins are well understood, then the problem can be stopped at its origins. Such prevention might be more useful and productive than the after the fact cure; since in this case prevention is not better than the cure, it *is* the cure.

References

B. Boehm and V. R. Basili, “Software defect reduction top 10 list,” Computer, vol. 34, no. 1, pp. 135–137, 2001.

Raymond P. L. Buse, Westley R. Weimer, “Learning a Metric for Code Readability,” Transactions on Software Engineering, vol. 36, no. 4, pp. 546-558, July/August 2010.

Andrew Hunt, David Thomas, “The Pragmatic Programmer; From Journeyman to Master,” Addison Wesley, 1999.

Harold Abelson, Gerald Jay Sussman, “Structure and Interpretation of Computer Programs” The MIT Press, 1996