International Journal of Computing and Corporate Research
ISSN (Online) : 2249-054X
Volume 3 Issue 4 July 2013
Manuscript ID : 2249054XV3I4072013-04

AN EMPIRICAL SURVEY ON USEFULNESS OF COMMENTS IN SOFTWARE PROGRAMMING LANGUAGES

Manjot Singh Ahuja, Surender Dhaiya, Neha Sadana
{manjot.ahuja, surendahiya, sadananeha25}@gmail.com
Computer Science and Engineering Department
Shivalik Institute of Engineering and Technology, Aliyaspur (Ambala)

ABSTRACT

Each programming language and script has its own coding conventions. There are numerous general points that can be followed to ensure that the code is well organised so that it can be easily understood for maintenance and reengineering. These guidelines are helpful in formalising code so that interpretation, reuse, maintenance, re-engineering and reverse-engineering of code become easier. The source code comment gives high-level advice for software developer. This paper presents an overview of conventions and importance of comments in source code during development as well as in maintenance.

Keywords - Code Quality, Source Code, Source Code Comments, Source Code Comments Quality

1 Introduction

Source code documentation is the text which accompanies computer software and either explains how it operates or how to use it. Code comments are used to embed programmer-readable remarks in the source code of a program. These annotations are ignorable to compilers and interpreters. The source code comments are typically additional artifacts with the rationale of the ease to comprehension of source code. The syntax and rules for comments diverge and are typically defined in a programming language specification. While developing software it is important that you include the clear documentation as well as the code comments to make it further reusable. This documentation will take form of external, internal documentation, comments, and readme files.

External documentation - Explains the methods of software development.

Internal documentation - Explains the way to implement the code.

Comments - Comments must be added to the code to explain the implementation details of the source code. It is mandatory to avoid adding of obvious or lengthy information. Prior to the project development, we should agree on how frequent comments should be and their location, format and length in the file. These conventions may need to be agreed on for block, single-line and end-of-line comments.

Readme file - Every package should have the readme file unfolding the purpose and functionality of the software and information on exterior dependencies.

2 Taxonomy of the Comments

2.1 Prefacing

This is the practice of starting each programming with a block comment that briefly describes it. Ideally, the preface should not be overly long, and it should summarize the purpose of its programming unit. The advantages of prefacing are twofold, it is a useful tool for any maintainers who may need to understand

the code in the future; but it can also be beneficial for the developer writing the code, helping to concretize the purposes in his mind. In addition, best practices have always stipulated that programming units should be single-purpose, and in writing the prefacing comments, the developer will catch himself if he is about to flout this rule.

2.2 Comment-Driven Development

This is a programming methodology that encourages the developers to start out complex projects by building a wireframe of their procedures using little more than comments and basic pseudo code to describe each step of the algorithm. CDD help developers to encounter and work out problems before they write a line of actual code; it also has the advantage of helping clearly delineate the routes between the high-level problem and the many small-picture fractals it is composed of. The comments created using CDD may survive the process of actual coding and development as line comments throughout the programming unit; however, it sometimes make sense to delete them after their purpose has been served.

3 Comments Style and Format

- Style (inline/block)
- Parse rules (ignored/interpolated/stored in memory)
- Recursivety (nestable/non-nestable)
- Uses (docstrings/throwaway comments/other)

Inline Comments - Inline comments are generally those that use a newline character to indicate the end of a comment, and an arbitrary delimiter or sequence of tokens to indicate the beginning of a comment.

Examples: // (C++, C#, D, Go, Java, JavaScript, Object Pascal (Delphi), Objective-C, PHP), # (bash, Cobra, Perl, Python, Ruby, Windows PowerShell, PHP, Maple), % (TeX, Prolog, MATLAB, Erlang, S-Lang, Visual Prolog), etc..

Block comments - Block comments are generally those that use a delimiter to indicate the beginning of a comment, and another delimiter to indicate the end of a comment. In this context, whitespace and newline characters are not counted as delimiters.

Examples: /* */ (C, C++, C#, D, Go, Java, JavaScript, Objective-C, PHP, Visual Prolog, CSS), { } (Object Pascal (Delphi), Pascal), etc.

4 Approaches and Protocols for Comments

Commenting styles should be governed by a company's development guidelines and, even within these guidelines, there is often enough wriggle room for individual style and subjectivity. The following are few rules and practices that should be followed when commenting code.

To include a preface: It need not be a long essay, in fact, it should not be long. However, it is important to include one. Apart from focusing the developer's mind around the task and assisting any maintainer that may come along in the future, prefaces can serve an important, additional function. Some languages, most notably Java, but also PHP include API-generating functionality: Javadoc and PHPdoc, respectively. PL/SQL presents us with no such nicety, but if all programming units contain prefacing comment blocks, then we can write of our own quite easily.

Include a revision history: Anyone maintaining your code in the future will thank you for including a revision history. The identity of the developer who has made a particular change is not so important, however, it may one day be

vital to know when a change was made and why.

Use line comments: A good rule is to use a single line comment at the start of each logical block describing what it does. If, in the future, someone needs to maintain your code, they should not need to read every line of code to pinpoint the section they intend to edit; in fact, they should be able to navigate your whole procedure or package without reading a single line of code outside the actual section they are looking for. A good developer, unlike a good novelist, is one whose best work is never read in full.

Never depend on your memory: Writing something complicated? Comment it. Code, algorithms and assumptions that are perfectly clear at the time of writing, can quickly fester in a morass of spaghetti code with time.

Comment while maintaining code: If you are forced to wade through old poorly-commented or uncommented code that you or another developer has written, do not leave it as you found it. Comment it. While you may not be responsible for the way you find code, you are responsible for the way you leave it.

Keep it simple: Comments should be easily readable. Keep them as short as possible, and as plain as possible. Unless absolutely necessary, do not include any code in your comments.

Good and meaningful comments make code more maintainable. However,

- Do not write comments for every line of code and every variable declared.
- Use // or /// for comments. Avoid using /* … */
- Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.
- Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
- Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
- If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.
- If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.

5 Literature Survey

In early 1980s Donald Knuth suggested literate programming [5], in order to combine the process of software documentation with software programming. Comments are ignored by compiler, so, in order to differentiate source code and documentation, a specific documentation or programming syntax has to be used. Programmers typically lack the appropriate tools and processes to create and maintain documentation, it has been widely considered as an unfavorable and labor-intensive task within software projects [6].

For generating comments in JAVA, Javadoc [7] is an automated tool that generates API documentation in HTML using Java source code and source code comments. Javadoc comments added to source code are distinguishable from normal comments by a special comment syntax (/**).

In [8], Khamis et al. provided an analysis tool called JavadocMiner for analyzing the quality of inline documentation. Thereby, the authors mainly focused on inline-documentation in the form of Javadoc comments. By using a set of heuristics,

they aimed to evaluate both the quality of the language and the consistency between source code and its comments. The quality of the language was measured with heuristics such as counting the number of tokens, nouns, and verbs, calculating the average number of words, or counting the number of abbreviations. With heuristics including the Fog index or the Flesch reading ease level, the authors targeted the readability of comments. For detecting inconsistencies between code and comments, this approach computes the ratio of identifiers with Javadoc comments to the total number of identifiers and checked whether all aspects of a method such as parameter or return type are documented. The heuristics are grouped into two categories, Internal (Natural Language quality only) and code/comment consistency.

Lehman and Belady in their paper [9] surveyed that with millions of lines of code written every day, the importance of good documentation cannot be overstated. Well-documented software components are easily comprehensible and therefore, maintainable and reusable. This becomes especially important in large software systems.

Storey et al [10] focused on study to explore the role embedded task annotations play in source code. They gathered data from a survey among software developers, from code analysis of open source projects, and from personal interviews. They found that task comments are frequently used in software development with the majority of comments containing TODO tags. TODO comments mainly serve the purpose of documenting small tasks when opening a new bug report. However, there is the potential risk that developers never revisit task comments. Based on this analysis, the authors suggested several implications for tool designer, such as providing altering mechanism for task views, supporting meta data within task comments, or introducing ad-hoc task clean-up wizards.

In addition to [10], Ying et al [11] also investigated the role of task comments. The authors interpreted the intention of task comments, resulting in a detailed categorization of task comments such as bookmarks on past tasks, current tasks, future tasks, pointers to change requests, or tasks used for communication. However, there was no automatic or semi-automatic assessment of task comment quality.

Fluri et al. [12] concluded that code and comments are not necessarily get updated and evolve at the same time. Therefore, the authors investigated how code and comments evolve. They used a mapping between code and comments to observe their co-evolution over multiple versions of the system. The authors conducted a case study on three different Java projects. The case study, in contrast, revealed that comment change is triggered by source code change; about 97% are done in the same revision as the source code change. Furthermore, approximately 70% of comments are mapped to one of the following seven types of source code entities: attributes, class and method declarations, control structures, loops, method calls, and variable declarations. The authors also evaluated the ratio between comments and source code over time to give a trend analysis whether developers increase or decrease their effort on code commenting. However, the results differed greatly among the three test cases such that no unique answer can be given. In particular, the authors did not differentiated between different types of comments, e. g., lines of commented out code were also counted in the ratio between comments and source code.

Sundbakken [13] assessed the comment density of maintenance phase code contributions to components of four open source projects. He observes that consistent commenting correlates highly with maintainability of components. The measured comment density ranged from 0.09% for poorly maintainable

components to 1.22% for highly maintainable components. In contrast to [13], in a study on the comment density of a closed-source compiler project in its maintenance phase, Siy and Votta found a consistent comment density of around 50% [14]. In another study of 100 Java open source classes, Elish and Offutt found an average comment density of 15.2% with a standard deviation of 12.2% [15]. Fluri et al. presented an approach for assessing the comment density of software projects and demonstrated the approach using three selected open source projects [12]. They also observed that new code is barely commented, implying that comment density decreases over time.

Oliver and Dirk in their paper [16] concluded that commenting source code is a consistent practice of active open source projects. It has led to an average comment density of about 19%. This density is maintained by dedicated commenting activities as well as in regular on-going programming activities. Also, they have found that the average comment density is independent of team size and project size, suggesting that as teams and projects get larger, successful open source projects maintain their commenting discipline. However, the average comment density is not independent of a project's age but rather declines with an aging project. That decline is statistically significant; however, it is rather small and thus has limited practical implications.

6 Limitation

Source code comments are essential for understandability of code and less work has been done on the qualitative analysis of comments. If we put aside quality of source code comments, nearly no work has been done related to understandability of source code with respect to comments. And works done by authors till now are limited to any specific language, no model has been proposed that will get fit to many programming languages.

Conclusion

Technical experts documented the varying viewpoints on whether and when comments are appropriate in source code. Some commentators avow that source code should be written with few comments, on the basis that the source code should be self-explanatory. Others suggested that code should be extensively commented. These views assert that excessive comments are neither beneficial nor harmful, what matters is that they should be correct and kept in sync with the source code, and omitted if they are superfluous, excessive, difficult to maintain or otherwise unhelpful. It is very important to specify the comments in the software source code very carefully so that it will be understandable, readable as well as modifiable. It also reduces maintenance cost, helps in maintainability, understandability, software reusability, re-engineering and in reverse-engineering. This becomes especially important in large software systems. As we have seen in the earlier section that there are many categories of documentation and comments. Some suggested ideas for in-line comments, some for block-comments, some for specific language, and some suggested where comments should be added and where not to add, some talked about comment density.

Since in-line comments come in contact with various stakeholders of a software project, it needs to effectively communicate the purpose of a given implementation to the reader. So research demands quality of in-line comments and we are trying to achieve the same by taking other parameters in consideration. And our work will not be limited to any specific programming language; we will try to implement it on variety of programming languages and will study how quality of source code comments effect understandability.

References

[1] Penny Grubb, Armstrong Takang (2003). Software Maintenance: Concepts and Practice. World Scientific. pp. 7, 120–121. ISBN 981-238-426-X.

[2] Vermeulen, Al (2000). The Elements of Java Style. Cambridge University Press. ISBN 0-521-77768-2.

[3] "Using the right comment in Java". Retrieved 2007-07-24.

[4] W. R., Dietrich (2003). Applied Pattern Recognition: Algorithms and Implementation in C++. Springer. ISBN 3-528-35558-1. offers viewpoints on proper use of comments in source code. p. 66.

[5] Knuth, D.E.: Literate Programming. The Computer Journal 27(2) (1984) 97{111

[6] Brooks, R.E.: Towards a Theory of the Comprehension of Computer Programs. International Journal of Man-Machine Studies 18(6) (1983) 543{554

[7] Kramer, D.: API documentation from source code comments: a case study of Javadoc. In: SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation, New York, NY, USA, ACM (1999) 147{153

[8] Ninus Khamis, Rene Witte, and Jurgen Rilling. Automatic Quality Assessment of Source Code Comments: the JavadocMiner. In Proceedings of the Natural Language Processing and Information Systems, and 15th International Conference on Applications of Natural Language to Information Systems, NLDB '10, pages 68-79. Springer-Verlag, 2010.

[9] Lehman, M.M., Belady, L.A., eds.: Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA (1985)

[10] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 251{260. ACM, 2008.

[11] Annie T. T. Ying, James L. Wright, and Steven Abrams. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. In Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05, pages 1-5. ACM, 2005.

[12] Beat Fluri, Michael Wursch, and Harall Gall. “Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes.” In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE2007). Page 70-79.

[13] Marius Sundbakken. Assessing the Maintainability of C++ Source Code. M.S. Thesis,Washington State University, 2001.

[14] Harvey Siy, Lawrence Votta. “Does the Modern Code Inspection have Value?” In Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM 01). IEEE Press, 2001. Page 281-290.

[15] Mahmoud Elish, Jeff Offutt. “The Adherence of Open Source Java Programmers to Standard Coding Practices.” In Proceedings of the 6th IASTED International Conference Software Engineering and Applications. Page 193-198.

[16] http://www.cs.uoregon.edu/events/icse2009/images/postPosters/The%20Comment%20Density%20of%20Open%20Source%20Software%20Code.pdf “The Comment Density of Open Source Software Code”