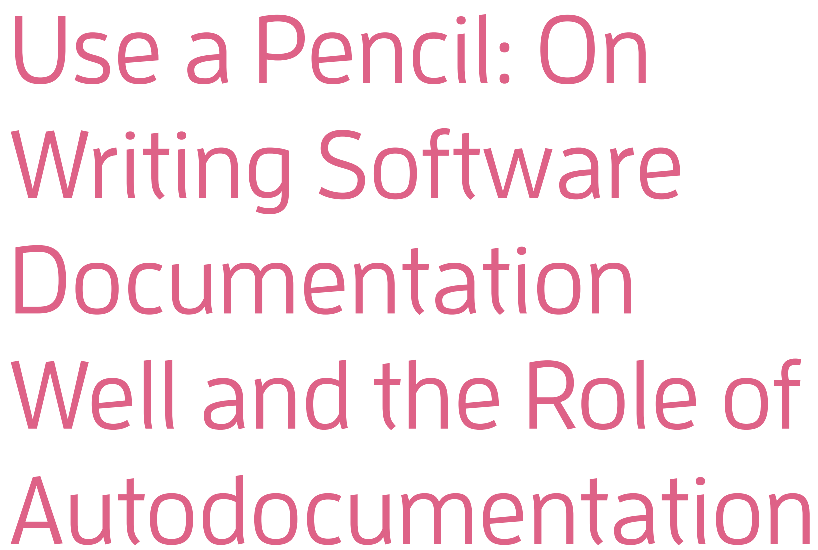

COPYRIGHT ISTOCKPHOTO, CREDIT:LVCANDY

Use a Pencil: On Writing Software Documentation Well and the Role of Autodocumentation

Phil Laplante, IEEE Fellow

Proficiency in technical writing is an important skill that all software engineers should develop, but there is a trend toward automation of documentation. This article discusses the past, present, and future of software documentation.

Imagine a time where software teams can avoid all documentation activities because the development framework and other tools can capture enough artifacts along the way to generate and keep current any documentation artifact, such as requirements specifications, design documents, or user manuals. Even reverse engineering requirements specifications from the end-product could be possible. Now picture a forward engineering path where a requirements specification could automatically be generated from a few keywords, or content captured via brainstorming sessions, and extrapolated to a specification using generative artificial intelligence (AI). These scenarios may be in our future.

While the state of the practice in software documentation is lacking, I am conflicted about using automatic generation of software documentation (autodocumentation), such as requirements specifications, design documents, user manuals, and so on. Is autodocumentation of these artifacts good or bad? Will the use of generative AI be a positive or negative driver of autodocumentation?

Digital Object Identifier 10.1109/MC.2024.3374008
Date of current version: 6 May 2024

U.S. Government work not protected by U.S. Copyright.

EDITOR PHIL LAPLANTE
IEEE Fellow;
plaplante@psu.edu

GOOD SOFTWARE DOCUMENTATION IS GOOD TECHNICAL WRITING

All software documentation are forms of technical writing. There is no universally accepted definition of "technical writing" to differentiate it from other forms of writing, but there are two main differences: precision and intent. Precision is crucial in technical writing. When you express an idea in technical writing, it may be realized in some device or process. If the idea is wrong, the device or process will also be wrong: syntax is destiny. Technical writing should also not provoke an emotional response from the reader. The technical writer should convey information as concisely and correctly as possible. This is characteristically different than poetry, prose, news reporting, and even business writing, where the reader's emotional persuasion and possible reaction are desired.[1]

There are no official standards for technical writing, but there are many writing style guides, for example, Chicago, American Psychological Association, Modern Language Association, and IEEE's own. These largely dictate the correct use of pronouns, punctuation, etc., and are of marginal importance in technical writing of software documentation; really, the style guide of the entity underwriting the software will prevail. But for all technical writing there are some principles to be obeyed. I called my favored set principles *The 5 Cs of Technical Writing* (*5 Cs*), and I use these in my own writing and to judge the quality of the writings of technical documentation, articles, and so on. Briefly, the 5 Cs represent *correctness*, *clarity*, *completeness*, *consistency*, and *changeability*. The 5 Cs are also closely related to the IEEE 29148 qualities for good requirements specifications. The 5 Cs are self-explanatory, but a good discussion of them can be found in Laplante.[1]

Achieving the 5 Cs in any significant writing is not easy and takes constant practice and refinement. One of my favorite books on basic writing principles is *On Writing Well*.[2] This book has profoundly influenced and improved my technical (and nontechnical) writing (and the title of this article) and it should be consulted by everyone.

But there are ways to improve one's technical writing. Spinellis recommends that new software engineering students should read (study) well-written code before even attempting to write code.[3] I wholeheartedly agree and suggest that this principle should be extended to software documentation. Perhaps before writing any software documentation artifacts new software engineers should study exemplars of that artifact. For example, if you want to write high-quality design documents, review high-quality ones first.

While every organization should have a set of these, not all do, and I think there should be public libraries of good software documents for study (but not copying). But this raises intellectual and proprietary issues (the best design documents probably won't be shared). Deciding which software documentation is of high quality is a problem without standards, review panels, etc., however, and the subject of evaluating software documentation quality is left for future discussion.

DISCLAIMER

The author is completely responsible for the content in this message. The opinions expressed here are his own. Autogenerated text is so noted.

AUTODOCUMENTATION PAST, PRESENT, AND FUTURE

Automatic documentation tools are not new. Early programming languages were cast as automatic program generators [they could translate a "specification" automatically into (assembly) code]. The "specification" (for example, the Fortran program) was the code documentation.[4] Over the years, other tools, such automatic flowchart generators, emerged to extract and format code comments into a kind of design specification. Since the emergence of the first high-level programming languages, there have been myths of "self-documenting code" generators, and I have witnessed these come and go. All of these attempts remind me of the 1888 Paige Compositor, an automated typesetting machine, which one might consider to be the first autodocumentation machine. Alas, it was a financial failure, costing investor Mark Twain most of his fortune.[5]

If the idea is wrong, the device or process will also be wrong: syntax is destiny.

There is no question that high-quality software documentation is important, yet it is also well known that it is not always given priority due to market pressures, which has perpetuated this quest for practical autodocumentation generators. For example, Aghajani et al.[6] surveyed 146 practitioners and found that

"Increasing the budget dedicated to documentation was a recurring solution often mentioned by participants suggesting that software documentation does not receive the attention it deserves." Behutiye et al. interviewed 15 practitioners and conducted workshops in companies using agile software development, and concluded that rather than traditional documentation, tools, backlogs, iterative prototypes, and artifacts, such as epic and stories, were used to document requirements if they were documented at all.[7] These realities imply the need for automatic generation of documentation useful for different tasks and context-aware documentation recommender systems.[6]

These collaborative lifecycle models also emphasize the need for high-quality communications, particularly written communications, and documentation throughout the process.

DevOps and DevSecOps are focused on increasing the speed and quality of software development and delivery by integrating developers and operations teams, and in the latter case, addressing security early and throughout the software development lifecycle. These collaborative lifecycle models also emphasize the need for high-quality communications, particularly written communications, and documentation throughout the process. DevOps and DevSecOps are also tool-driven, so it is no surprise that they help drive research in autodocumentation. For example, one researcher built an autodocumentation system compatible with DevOps style development. Using Pandoc, LaTeX, and Docker for deployment on either GitLab or Bitbucket, the system can automatically generate documentation with new releases. This approach resulted in 38% faster generation of plain text documents and are claimed to be more esthetically pleasing than typical formatted documents.[8]

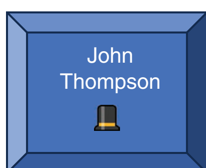

FIGURE 1. John Thompson, a colonial hatmaker, developed this concise sign for his store after several revisions of the original text: "John Thompson, Hatter, Makes and Sells Hats for Ready Money."

THE DANGERS OF CUT-AND-PASTE AND GENERATIVE AI

I know of no autodocumentation systems for software that use generative AI (yet), but I am sure they are in the experimental phase in various settings and they will be emerging soon for commercial use. To test their potential, I tried a little experiment. In Laplante[1] I recount the apocryphal story of John Thompson, a colonial hatmaker who wished to create a sign for his shop. The process of refinement from a verbose (and expensive) sign to a short and powerful pictogram (Figure 1) emphasizes the power of conciseness and the importance of the human editorial process in achieving all of the 5 Cs of good technical writing.

I wanted to see if ChatGPT could in any way match the Thompson's result. In my experiment, I gave ChatGPT the following prompt: Create a sign for John Thompson, a maker of hats. ChatGPT created the following "sign":

🎩 Welcome to John Thompson's Hat Emporium! 🎩
Step into a World of Elegance and Style!
Discover the Perfect Hat for Every Occasion!
Unmatched Quality | Timeless Designs | Superior Comfort | Attention to Detail
Visit Us Today and Elevate Your Style!
[Address] [Contact Information]
Open Hours: Monday – Saturday: 9:00 a.m. – 7:00 p.m. Sunday: 10:00 a.m. – 5:00 p.m.
Experience the Art of Hat-Making at John Thompson's!

Curiously, ChatGPT chose to make all kinds of unprovable claims about the hats and even invented store hours. Here, we see a direct violation of both characteristics of technical writing: precision and intent. The sign is imprecise in its use of immeasurable qualities, such as "elegance" and "style." And it is clearly injecting unwanted intent. I didn't ask the autogenerator to persuade anyone to buy hats from John Thompson. The sign may also be wrong. How could it know if the hours of operations listed are correct? Only focused human review could correct any errors introduced by the tool.

We all know about the dangers: cut-and-paste reuse facilitates plagiarism and the shortcomings of failed grammar checking. Overreliance on tools, such as grammar and spelling checkers, also promotes complacency with respect to these skills. Word processing of software documentations also facilitates the use of generative AI. It is so easy to autogenerate draft writing, then tweak it by hand. This is not a good approach to writing software documentation.

Some might argue that autodocumentation tools can help promote the 5 Cs, but I don't think that is true. For example, consistency in a document can't be automatically checked because doing so is a form of the Boolean satisfiability problem, which we know is nondeterministic polynomial-time complete. Consistency

checking is related to the Goedel Incompleteness problem. Clarity (nonambiguity) requires an encoding of all human knowledge– good luck with that. Even conciseness is a difficult quality to automatically capture

The real problem is that we need to instill good technical writing practices in the humans who build these tools.

(viz my experiment). Perhaps changeability—the ease with which changes in the document can be managed—is the only one of the 5 Cs that seems tractable today.

Even if some of the 5 Cs could be embodied in an autodocumentation tool, I worry about other aspects, particularly when using generative AI. For example, for any significant system, in addition to all kinds of unwanted features, autodocumentation of any software artifact could propagate bad design decisions, code vulnerabilities, hard to understand descriptions of functionality, etc.

RECOMMENDATIONS

The real problem is that we need to instill good technical writing practices in the humans who build these tools. We should teach software engineers how to write effectively from the outset, and this applies to technical documentation, professional communications, etc. How can you recognize problems in software documentation generated automatically if your writing stinks? I conclude that we must be careful with autodocumentation in research, education, and practice and make the following recommendations.

- Teach technical writing early in the software engineering curriculum, and in professional settings to all who work in software. Continuing education in technical writing and documentation for all should be conducted throughout their careers.
- Create public libraries and documentation repositories of "great papers" and exemplars for study and review (not copying).
- Create independent documentation quality review panels through professional societies, such as the IEEE.
- Develop standards for software documentation (not just writing style guides) that focus on appropriate implementation of the 5 Cs.
- Carefully monitor the use of tools, particularly autodocumentation and generative AI, and review and question their outputs. Perhaps even create standards of quality for these tools.

Sometimes I wonder if we should start all of our writing, including software, by hand instead of using computers. The handwriting process encourages thoughtfulness and discourages cut-and-paste (plagiarism). It's tempting to think we can push a button and generate all software documentation, but consider the costs of error and of suborning the responsibility to generative AI. If syntax is destiny, do we want to yield our destinies to some program? Maybe we should go back to just using pencils instead of word processors. ■

REFERENCES

1. P. A. Laplante, *Technical Writing: A Practical Guide for Scientists, Engineers and Nontechnical Professionals*, 2nd ed. Boca Raton, FL, USA: CRC Press, 2019.
2. W. K. Zinsser, *On Writing Well: The Classic Guide to Writing Nonfiction*. New York, NY, USA: HarperCollins, 2001.
3. D. Spinellis, *Code Reading: The Open Source Perspective*. Reading, MA, USA: Addison-Wesley, 2003.
4. J. W. Backus et al., "The FORTRAN automatic coding system," *presented at the Western Joint Comput. Conf., Techn. Rel.*, Feb. 26–28, 1957, pp. 188–198, doi: 10.1145/1455567.1455599.
5. R. Powers, *Mark Twain: A Life*. New York, NY, USA: Free Press, 2006.
6. E. Aghajani et al., "Software documentation: The practitioners' perspective," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 590–601, doi: 10.1145/3377811.3380405.
7. W. Behutiye, P. Seppänen, P. Rodríguez, and M. Oivo, "Documentation of quality requirements in agile software development," in *Proc. 24th Int. Conf. Eval. Assessment Softw. Eng.*, Apr. 2020, pp. 250–259, doi: 10.1145/3383219.3383245.
8. D. Söderberg, "Automation of non-code documentation in a DevOps environment," Master's Thesis, Luleå Univ. of Technol., Luleå, Sweden, 2022.

PHIL LAPLANTE, State College, PA 16801 USA, is a computer scientist and software engineer, a Fellow of IEEE, and an associate editor in chief of *Computer*. Contact him at plaplante@psu.edu.