On the Normalization of Software Bugs

A Research Paper submitted to the Department of Engineering and Society

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

James Leo Yuan Huang

Spring 2023

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Advisor

Kent A. Wayland, Department of Engineering and Society

Introduction

We have a lot of computers now, and they do lots of things. You can find computers everywhere: in refrigerators (Tapellini, 2015), in airline crew scheduling (Wallace, 2023), in guided missile systems (Acuff & Hughes, 1992). In 2018, 92% of all U.S. households owned at least one type of computer (Martin, 2021).

Computers do things by running software. Software is unfortunately a bit of a weak link in computer systems. Software errors (also called *bugs*) have led to plane crashes (Gates & Baker, 2019), radiation overdose (Leveson & Turner, 1993), warship shutdowns (Stutz, 1998), and more.

There are plenty of bugs in non-critical systems as well. For example, the top 5 starred software projects hosted on GitHub currently have some 2773 open bug reports in total[1], the Chromium project (part of the widely-used Google Chrome web browser) currently has 52833 open bug reports[2], and Sakai (the software behind one of University of Virginia's current learning management systems, UVACollab) currently has more than 1000 "major" bug reports[3]. There are lots of bugs that do not kill people, but this does not mean they are exactly innocuous. Tassey

[1] The five projects were tensorflow/tensorflow, electron/electron, microsoft/terminal, bitcoin/bitcoin, and opencv/opencv. Star data was taken from EvanLi/Github-Ranking on February 12, 2023. "Open bug reports" were counted by navigating to each project's GitHub issue tracker and referring to the number of open issues labeled with tags involving bugs.

[2] Taken from Chromium's issue tracker at https://bugs.chromium.org/p/chromium/issues/ on February 12, 2023 based on all issues with type Bug or Bug-Regression.

[3] Taken from Sakai's issue tracker at https://sakaiproject.atlassian.net/jira/software/c/projects/SAK/issues/ on February 12, 2023. Sakai's issue tracker is unable to present more than 1000 results for a single query, hence the vague figure "more than 1000".

(2002) estimated that poor infrastructure for detecting and removing bugs cost the U.S. economy $59.5 billion per year. In addition, bugs often provide security vulnerabilities exploitable by malicious parties (Bilge & Dumitraş).

It seems that bugs are bad. But if they are, why do we still have them? Software is not something like a bridge that struggles against its physical environment; software operates in an abstract environment atop hardware specified entirely by humans. Why have humans failed to control their own creation? Or was complete control ever an option? By attempting to answer these questions, maybe we can gain a deeper understanding of the nature of bugs and perhaps how to squash them from existence. In this paper I will explore how bugs have crawled their way into software and attempt to answer the question: have we normalized software bugs?

Literature review

I was unable to find sufficient literature that directly discussed the normalization of bugs. The closest I found was an article by Don Gotterbarn (1999) in *Journal of Business Ethics* that criticized software developers for their "tendency to minimize descriptions of software errors" and suggested that software bugs were treated differently from "the disasters of patients dying or dams crumbling" because software development was "more of a craft than science". However, Gotterbarn concluded that the (then-recent) creation of a software-engineering code of ethics signaled the transformation of software development into a "profession" that "has now said that a caviler [sic] approach to software errors is unacceptable" (p. 81). Though the article provided food for thought, it did not provide substantial explanation of central concepts like "craft" and "profession", and also essentially claimed that the problem had been solved by the creation of the software-engineering code of ethics. I aim to analyze whether the problem has really been solved 24 years later, potentially using different reasoning.

Methods

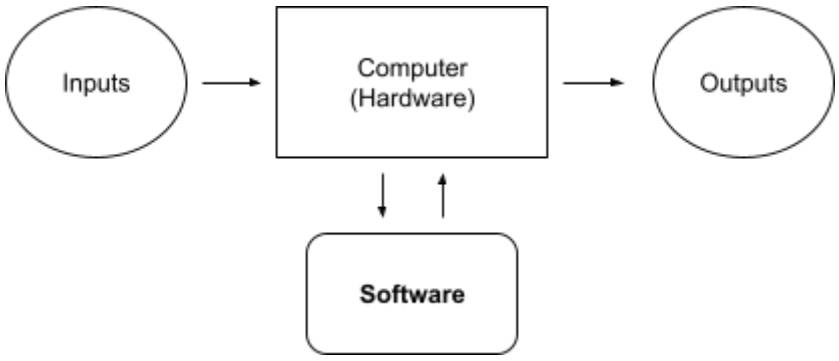
To answer whether we have normalized software bugs, I consulted bug-related papers and articles as well as commentaries and histories of software engineering published by reputable sources (e.g. ACM and IEEE), trying to interpret the "true" nature of bugs by synthesizing how all parties characterized them. And from this interpretation I constructed my argument, drawing from materials I had come across earlier.

What is software?

I have introduced bugs as "software errors". But what is software, and how can it have errors? This is a bit of a philosophical question, but in its most concrete form, software is a long list of instructions that a *computer* executes. A *computer* (in its most common, concrete form) is an electronic instruction-processing device that can receive signals from "input devices" (e.g. a keyboard, mouse, gyroscope, thermometer) and send signals to "output devices" (e.g. a screen, printer, speaker, motor). Thus software is the list of instructions that computers execute to compute outputs based on inputs; this is illustrated in figure 1.

Figure 1

Software in Relation to Computers



What about software *errors*? When software causes the computer to compute something unexpected[4] (which usually causes unexpected outputs), the software in question has an *error*, or alternatively, a *bug*. For example, a software bug caused a U.S. warship computer to (unexpectedly) divide a value by zero, producing a value that caused other parts of the software to do unexpected things, eventually causing the computer to reach an invalid state and fail, ultimately causing the ship's entire computer network to fail and leaving the ship dead in the water (Stutz).

The most easily identifiable *bug* in this case was the fact that the software made the computer divide by zero, producing a value that caused the rest of the system to fail. But one could argue that the network's inability to cope with one computer's failure was the deeper bug that caused the actual failure of the entire system. Or perhaps the "unexpected" state produced by dividing by zero was not the problem; maybe it was the rest of the software's problem for failing to expect this state. There are many different ways to pin the bug in this situation, but it is clear that there is *some* bug, because the system produced unexpected outputs[5].

Where do bugs come from?

We now have a rough understanding of what a *bug* is, but one might ask: why do we have bugs in the first place? Unlike the hardware that it runs on, software does not particularly fight against the limits of the physical world like other feats of human engineering do. Yet software seems to be in an endless struggle with bugs: Fred Brooks (1995) in their essays on software engineering suggested that half of a software project's entire timeline be dedicated to testing for

[4] e.g. that a coffee machine should eject 500ml of water instead of an expected 250ml.

[5] Or in this case perhaps none at all, since the system just failed and left the ship sitting in the water.

and fixing bugs (p. 20); Bill Gates (co-founder of Microsoft[6]) once described their company as more of a "testing" organization than a "software" organization, stating that they had as many full-time software testers as they did developers (who also functioned as part-time testers) (Foley & Murphy, 2002).

A manifestation of the essential nature of software engineering

Brooks (1995) suggests four fundamental problems that all software systems face: complexity, conformity, changeability, and invisibility (p. 182). Software is *complex* because of the sheer number of states that it manages[7], that it must *conform* to arbitrary human design[8], that it must *change* more often than other systems, and that it is ultimately *invisible* in that it lacks an adequate visible abstraction. Brooks argues that these four properties are the essential struggles

[6] A successful tech company

[7] Imagine for example a piece of software that has 300 instructions, each of which is capable of changing the computer's state in 2 ways. The software is then capable of putting the computer into 2^{300} different states (assuming that each unique combination of changes produces a different state), which is a large number of states. This is a shallow example, but the main idea is that a large combination of even non-complex things can grow very complex very quickly.

[8] For example, if I fail to buy something with U.S. dollars from an Australian vendor, I am suffering from the "arbitrary complexity" of our financial systems. I have something of value to offer (U.S. dollars), but the arbitrary nature of how humans have decided to exchange things with different currencies prevents me from making the exchange. In software, "arbitrary complexity" would likely be things like what languages and "ecosystems" something is built around, all the random details of different software that you have to read documentation to know, etc.

of software engineering and that "there is inherently no silver bullet" – that software engineering is, in essence, a difficult process that does not seem to be radically improvable[9].

Following this line of thought, it would seem that bugs are simply a manifestation of the essential nature of software engineering: creating software is difficult, and bugs are simply part of this difficulty. Further supporting this idea, Brooks writes that "finding nitty little bugs is just work", and that "with any creative activity come dreary hours of tedious, painstaking labor, and programming[10] is no exception" (pp. 8-9). And looking at how much time software developers spend on finding and removing bugs adds to this reasoning: one study found that developers estimated they spent 20-60% but in reality spent 13-95% of their working time on debugging (Alaboudi & LaToza, 2021).

Humans' failure to write correct programs

Edsger Dijkstra offers a less tolerant view of bugs, writing years earlier: "The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it disguises that the error is the programmer's own creation" (1988). Though he does not deny the difficulty of writing software, he argues that humans struggle with the "radical novelty" of computers: that they are digital devices[11], and that the computing power they provide is unprecedented. He suggests that programming has its roots in formal mathematics and should be tackled with similar techniques like formal systems and proofs.

---

[9] In Brooks's words: "We cannot expect ever to see twofold gains [in software productivity, reliability, and simplicity] every two years." (p. 192)

[10] In this context, "programming" can be taken to mean "writing software" (with "program" meaning "software). The difference between "programming" and "software engineering" is a complicated topic that deserves its own paper.

[11] i.e. that they operate on discrete values, as opposed to "analog" devices.

In this line of thought, humans take a lot more of the blame: even if writing well-behaving software is difficult, it is the human's responsibility to use the proper techniques (e.g. formal mathematics) that let them build bug-free systems. Instead of inherently buggy systems that must be expelled of bugs, Dijkstra suggests that bugs are simply a manifestation of an incorrect program: that a program with a single bug be considered not "almost correct" but simply "wrong".

…or maybe not?

However, others think otherwise. Lieberman and Fry (2001) criticize what they call the "macho programming culture" in which programmers are "psychologically reluctant to admit the prevalence of bugs in their programs" and thus "unwilling to devote time and money to improving the process of dealing with them." They further write:

> John Guttag, a professor of software engineering at MIT, said, “Finding a bug in your program is like finding a cockroach in your kitchen; if you have one you probably have more and is not something one should be pleased about,” a distasteful metaphor suggesting the presumed cause of a bug is negligence on the programmer’s part. We think it is this denial of the normalcy of bugs and debugging that has led directly to the unreliability of software. (p. 123)

This seems to fit in with Brook's ideas discussed earlier: If bugs were indeed a manifestation of the essential difficulties of software engineering, denying that bugs are normal would be to deny that software engineering is inherently difficult – an unproductive and perhaps "macho" attitude towards the matter.

Denning (1992) suggests that software quality would be improved by focusing not on the more formal and mathematical side of software development (that was endorsed by Dijkstra) but

on customer satisfaction. They argue that "software quality" is not the mathematical correctness of a system but is how well it pleases the customer, which may be related but is ultimately removed from program correctness. They write: "Program correctness is essential but is not sufficient to earn the assessment that the software is of quality and is dependable" (p. 15).

Are bugs a fact of life?

And so we see that it is not exactly clear where bugs come from and who exactly to blame for them. Some blame it more on the inherent difficulties of software engineering itself, and some blame it more on the humans that fail to use what they deem adequate software-engineering techniques. It is hard to answer whether we have "normalized" bugs, because if bugs are simply a fundamental part of software engineering, it is unclear if bugs can really be "normalized" at all, just as it is unclear if we have "normalized" breathing and drinking water. Is it humans' incompetence that allows bugs to prosper, or are they always here to stay, no matter what techniques we develop?

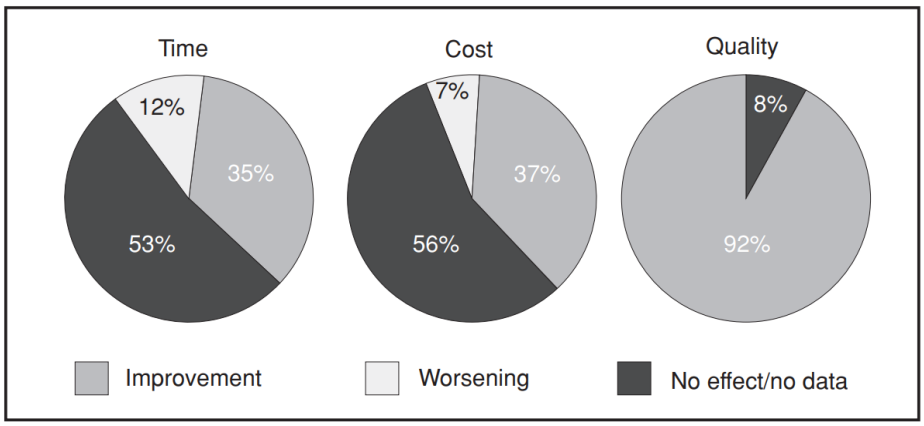
Formal methods

What about the mathematical techniques endorsed by Dijkstra? Does using mathematical, "formal" methods actually produce bug-free software? This is a tricky question, because formal methods can be applied to different parts of the software development process, and even when applied, they can be applied incorrectly (Reid et al., 2020). So an unsatisfying answer would be: yes when they are applied right, and no when they are applied wrong. However, a survey of individuals involved with projects that extensively used formal methods found that 92% of respondents thought that using formal methods increased the quality of the software, not only helping to detect bugs but also improving the design and understanding of the system (Woodcock

et al., 2009, p. 9). Formal methods have also been widely applied to safety-critical systems such as railway signaling and avionic systems (Woodcock et al.).

Figure 2

What Effect Did Using Formal Methods Have on the Time, Cost, and Quality of the Project?



Note. from Woodcock et al.

If formal methods apparently work, why do we not use them?

So it seems that formal methods do produce software that has some degree less of bugs, depending on how rigorously they are applied. Yet even in 2020, applying formal methods is "a bit of a niche activity" (Reid et al.). Why formal methods have not been widely adopted is a tricky question. A survey of the U.S. aerospace industry reported that education (people simply did not know how to apply formal methods) and tooling (relevant tools were unsatisfactory) were the two biggest barriers to adoption (Davis et al., 2013, pp. 66-67).

Another common theme is that many are unsure of whether the benefits of formal methods outweigh its costs: although the survey by Woodcock et al. clearly showed the benefits of using formal methods, its effect on the time and cost of projects is ambiguous (see figure 2: only 35% and 37% of respondents reported that formal methods positively affected the timeliness and cost of their project, respectively), with many reporting that formally specifying

the system took significantly longer but likely sped up the rest of development. Hailpern and Santhanam (2002) explain that ever-changing software requirements and the "belief in the malleability of software—that whatever is produced can be repaired or enhanced later" lead to informal requirements that make formal specification difficult (p. 7). In other words, what software is supposed to do changes so much during (and after[12]) development that it is unfeasible to specify what it is actually supposed to do. But regardless of *why* they are hard, Reid et al. agree that because applying formal methods is "hard and expensive", formal methods have only been picked up by organizations that "have a reason to value" them, e.g. those developing safety-critical systems.

The normalization of bugs

So have we normalized bugs in software? I assert that yes, we have. Even though we have powerful methods for developing less-bug-prone software, we choose to reserve them for only safety-critical systems, where bugs can cost human lives. Other systems, though?

A 2019 report from Facebook[13] reports how engineers fixed 0% of bugs found by automatic bug-finding tools, until the bugs were given to them in the context of what they were currently working on (at which point 70% of bugs were fixed). It explains that this was due to the "mental effort of context switch", essentially that developers were too mentally burdened to switch from the problem they were working to fixing a potentially unrelated bug. But something else to take away from this is that developers do not really make a great deal of non-catastrophic

[12] A related statistic is that a 1978 survey of software-development organizations found that the average organization dedicated more resources to maintaining and enhancing existing systems than to developing new systems (Lientz et al., 1978).

[13] A successful tech company

bugs. Furthermore, a 2014 study of open-source[14] software projects reported that 5-9% of reported bugs ended up taking more than a year to be fixed, despite 90% of these bugs affecting users' normal working experiences and 40% having simple fixes (a few changes in a single file) (Saha et al.). Evidently safety-uncritical systems are not worth it!

Why, though? How did making faulty software become the norm? Perhaps it is because software engineering is, ironically, too easy. Armour (2013) argues that the sheer power of modern computing has maybe made software development simply too fast, encouraging developers to be lazy and write software via "experimentation": to simply try whatever first comes to mind, see where it fails, try something that might fix the problem, see where it fails, try something else that might fix the problem, and so on, essentially developing the system by trial and error rather than with an understanding of how the system should actually be designed. Software constructed like this may end up as "a sequence of ad hoc amendments to an initially weak and ill-considered design" (p. 31).

This echoes how Boehm (2006) describes the history of software engineering in the 1960s: once people found out how easy it was to modify software, they developed a "code-and-fix" (essentially "experimentation" as described above) approach to development and began producing "heavily patched spaghetti[15] code". This also echoes the aforementioned "belief in the malleability of software" that prevents software projects from successfully applying formal methods. When software is so easy to write and change, it is hard to imagine how software is *not* malleable, that the rough draft hastily written today can be revised into tomorrow's final draft.

[14] Open-source software is software whose code is publicly available and usually accepts public contributions.

[15] "Spaghetti code" is a negative term that describes logically complex, unstructured code that is difficult to understand and thus modify properly.

But I ultimately think that this is why bugs are the norm today: systems are defined *as they are built* and ultimately change after they are built. As a result developers lack a consistent understanding of what they are exactly building and so cannot formally specify or verify that they are doing the correct thing[16], and thus they inevitably do incorrect things that only surface once the final software does strange things.

Conclusion

The humble software bug has plagued us for decades, at worst crashing our planes, at best going unnoticed, and most of the time mildly annoying us or making us turn it on and off again. It is unclear where *exactly* they come from, but a good guess is that they arise from the interaction of our fluid, informal specifications that evolve alongside their equally fluid, patched-together implementations. Formal, mathematical methods show promise to produce bug-free software but are incompatible with the informal methods that we have grown accustomed to. I believe that our choice to stay with our fast and informal methods and fix the bugs they inevitably produce instead of designing systems that are (relatively) bug-free in the first place shows that we have indeed normalized software bugs. The only real exception to this is perhaps safety-critical systems where bugs could cost lives.

The implications of this are not strong enough to suggest something as direct as "bugs are a choice", but it does suggest that various social factors (to be identified by further research) have led us to believe that informal methods that produce bugs to be driven out is the normal way to engineer software. It also raises the ethical question of whether treating such methods as the norm is ethical.

[16] A classic software-engineering adage(?) is: "It's not a bug, it's a feature."

We have a lot of computers now, all running lots of software. Because of its complexity and abstract nature, it is difficult to discern how to best construct software. By critically analyzing our perceptions of software and its engineering, hopefully we can learn how to make better software, whatever "better" may be.

References

Alaboudi, A., & LaToza, T. D. (2021). *An Exploratory Study of Debugging Episodes* (arXiv:2105.02162). arXiv. https://doi.org/10.48550/arXiv.2105.02162

Acuff, P. R., & Hughes, W. M. (1992). *Real Time Executive for Missile Systems User's Guide: MC68020 C Interface*. Army Missile Command Redstone Arsenal Al Guidance and Control Directorate. https://apps.dtic.mil/sti/citations/ADA252549

Armour, P. G. (2013). When faster is slower. *Communications of the ACM*, *56*(10), 30–32. https://doi.org/10.1145/2505338

Bilge, L., & Dumitraş, T. (2012). Before we knew it: An empirical study of zero-day attacks in the real world. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 833–844. https://doi.org/10.1145/2382196.2382284

Boehm, B. (2006). A view of 20th and 21st century software engineering. *Proceedings of the 28th International Conference on Software Engineering*, 12–29. https://doi.org/10.1145/1134285.1134288

Brooks, F. P., Jr. (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley.

Davis, J. A., Clark, M., Cofer, D., Fifarek, A., Hinchman, J., Hoffman, J., Hulbert, B., Miller, S. P., & Wagner, L. (2013). Study on the Barriers to the Industrial Adoption of Formal Methods. In C. Pecheur & M. Dierkes (Eds.), *Formal Methods for Industrial Critical Systems* (pp. 63–77). Springer. https://doi.org/10.1007/978-3-642-41010-9_5

Denning, P. J. (1992). What is software quality? *Communications of the ACM*, *35*(1), 13–15.

Dijkstra, E. W. (1988). *On the cruelty of really teaching computing science* [Unpublished manuscript]. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF

Foley, J., & Murphy, C. (2002, May 17). *Q&A: Bill Gates On Trustworthy Computing*. InformationWeek. https://www.informationweek.com/it-life/q-a-bill-gates-on-trustworthy-computing

Gates, D., & Baker, M. (2019, June 22). The inside story of MCAS: How Boeing's 737 MAX system gained power and lost safeguards. *The Seattle Times*. https://www.seattletimes.com/seattle-news/times-watchdog/the-inside-story-of-mcas-how-boeings-737-max-system-gained-power-and-lost-safeguards/

Gotterbarn, D. (1999). Not all Codes are Created Equal: The Software Engineering Code of Ethics, a Success Story. *Journal of Business Ethics*, *22*(1), 81–89. https://doi.org/10.1023/A:1006172527640

Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, *41*(1), 4–12. https://doi.org/10.1147/sj.411.0004

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. *Computer*, *26*(7), 18–41. https://doi.org/10.1109/MC.1993.274940

Lieberman, H., & Fry, C. (2001). Will software ever work? *Communications of the ACM*, *44*(3), 122–124. https://doi.org/10.1145/365181.365236

Lientz, B. P., Swanson, E. B., & Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communications of the ACM*, *21*(6), 466–471. https://doi.org/10.1145/359511.359522

Martin, M. (2021). *Computer and Internet Use in the United States: 2018*. U.S. Department of Commerce. https://www.census.gov/library/publications/2021/acs/acs-49.html

Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., & Laurie, B. (2020). *Towards making formal methods normal: Meeting developers where they are* (arXiv:2010.16345). arXiv. https://doi.org/10.48550/arXiv.2010.16345

Saha, R. K., Khurshid, S., & Perry, D. E. (2014). An empirical study of long lived bugs. *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 144–153.

Stutz, M. (1998, July 24). Sunk by Windows NT. *Wired*. https://www.wired.com/1998/07/sunk-by-windows-nt/

Tapellini, D. (2015, August 27). *Not-So-Smart Samsung Refrigerator Vulnerable to Hacking*. Consumer Reports. https://www.consumerreports.org/topfreezerrefrigerators/smart-refrigerators-privacy--a1663781370/

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, *7007*(11).

Wallace, G. (2023, January 26). *Southwest Airlines is testing a software fix it developed after the Christmas travel meltdown | CNN Business*. CNN. https://www.cnn.com/2023/01/26/business/southwest-software-fix/index.html

Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Computing Surveys*, *41*(4), 19:1-19:36. https://doi.org/10.1145/1592434.1592436