Eliminating Software Caused Mission Failures

Michael Dorin
Universität Würzburg
Würzburg, Germany
mike.dorin@stthomas.edu

Dr. Sergio Montenegro
Universität Würzburg
Würzburg, Germany
sergio.montenegro@uni-wuerzburg.de

Abstract—**The availability of common off the shelf components, as well as the ability for multiple satellites to share a launch vehicle, has dramatically reduced the cost of putting a satellite into space. Many satellites are built, but many of these satellites fail to meet their objectives. Often, a satellite's failure is caused by a problem outside the control of a design team. However, in some cases, the failure is caused by human error, and software failures are a likely candidate for human error. Software problems in a satellite application can mean an enormous loss of invested time and money. In this research, programmers of all skill levels were surveyed to identify the characteristics and implications of complicated software. After analysis of the surveys, an effort was made to identify whether programs with characteristics of being complicated contained more faults or required more effort to maintain. Software from multiple open source repositories was reviewed including an estimation of effort based on bug reports and bug discussions. Results from this analysis show that complicated programs do require more effort and are more problematic. This study finds that complicated software causes more errors, and that such software errors can be avoided by conscientiously developing uncomplicated code.**

TABLE OF CONTENTS

1. INTRODUCTION

The popularity of nano- and pico-satellites has increased dramatically in recent years. Consequences of software failure can be severe in any mission-critical application but are particularly problematic in space applications. Even when software faults are minor, it is often impossible to quickly restart or update software. Flawed software has no place in devices that are not easily accessible, such as in aerospace applications. Software failures are especially unfortunate as they are mostly avoidable. One ingredient of software faults is overly complicated source code. Source code humans view as complicated is more difficult to review and more challenging to debug.

This topic of uncomplicated, understandable code has been analyzed various times, and several trends have emerged. The most prominent of recommendations for addressing complicated software comes in the form of coding standards. Coding standards define what an organization expects from its programmers, such as providing style guidelines as well as naming conventions [1]. Most organizations recognize the importance of coding standards as demonstrated by the number of suggested standards for all languages. Following a coding standard makes code easier to review and therefore easier to debug [2]. More than 25 years ago, David Straker published a book on coding standards which included psychological factors associated with programming and code quality[3]. In the area of aerospace and satellites, the Jet Propulsion Laboratory (JPL) published a coding standard for 'C' programming and escalating requirements based on the defined criticality of a mission [4]. The JPL recommendations were further distilled by Holzmann who describes rules for developing safety-critical code [5]. Holzmann's paper addressed the topic of complicated coding standards not being followed and made suggestions for improving compliance. The existence of research on coding standards is helpful in that it provides a path for avoiding development of overly complicated software. However, in general, coding standards research does not address the ramifications when rules are not followed.

Another area of research concerning complicated software is the topic of technical debt. An abundance of literature characterizes technical debt as "the long-term consequences of shortcuts taken during different phases of [the] software development life cycle [6]." Another way of looking at technical debt is through the metaphor of borrowing against future software quality to have an immediate gain, such as a quick software release. A vital aspect of this avenue of literature is that a deliberate decision is made which results in technical debt. For this investigation, an overly complicated code is always considered technical debt. In many cases creating complicated code is not a deliberate decision to shortcut, but is instead made under the belief that the solution presented is proper and optimal. However, there is a strong likelihood the complicated solution is not proper and optimal.

This paper first clarifies what makes software complicated for humans and then demonstrates the impact that complicated software has on software development.

2. METHODOLOGY

The primary research was conducted through surveying software developers with the objective of classifying the characteristics of complicated code and measuring the impact complicated code has on software development. In this process, two surveys were performed at different times. In conjunction with the survey analysis, a code review was undertaken using the information found in the surveys to associate the knowledge learned in the survey with the actual technical debt caused by complicated code.

Software Files Survey

In this survey, programmers were shown a source file and asked to immediately and intuitively identify whether they found it unpleasant to review. The objective of this survey

978-1-5386-6854-2/19/$31.00 ©2019 IEEE

was to determine the significant characteristics of source code developers feel is complicated. Based on the assumption that humans consider complicated code unpleasant to review, it may also be assumed that code which is unpleasant to review is likely complicated. This survey was conducted in early 2018, and detailed results were published in the XP Conference Companion [7]. Another way of looking at it is software is complicated if the person looking at that code thinks it is complicated. Programming style is a significant contributor to complicated code and technical debt. The most negative indications of unpleasant to review files identified by the survey are listed in Table 1 [7].

Software Statements Survey

Higher functionality does not necessarily require higher complexity. In the second survey, volunteers where shown complicated code, as well as the same functionality,implemented using an easier to understand approach. The volunteers were asked to evaluate the code and describe the anticipated output if the code were run. The survey intended to reflect how difficult it is to review complicated code properly and also to identify a complicated code penalty. Put another way; the second survey attempted to determine if there is an actual cost associated with the characteristics of complicated software. The survey measured how long statements took to review as well as the accuracy of the volunteer's evaluation.

Open Source Evaluation

Finally, a cursory evaluation of actual projects was performed to see if the results of the surveys have merit. Source code from three relatively popular open source repositories was analyzed. The three projects were randomly selected from the published Bugzilla users [8]. The three projects selected were GCC [9], LyX [10], and Sudo [11]. All three projects have public bug tracking spanning many years, and all three projects are in active use. The tool, nsiqcppstyle [12] was configured to only scan for non-conformance with those rules listed in Table 1. Each project had a collection of milestones selected for evaluation and then was scanned. As each project is different in size and scope, development effort was estimated based on the individual projects' characteristics.

Table 1. Unpleasant to Review Styles

Style Name
There should be space around operators
Do not write over 120 columns per line
Have Short functions
Indent blocks inside of a function
Put matching braces in same column
Use less than 5 parameters in function
Do not use the question keyword
Avoid deeply nested blocks
Use braces for even one statement

Table 2. Results of Software Statements Survey

Condition	Count	Time	Success
Embedded Comment	85	52.45	10.59%
Badly Indented 'if'	86	27.01	34.88%
Yes space around operators	86	73.43	45.35%
No space around operators	79	62.58	51.90%
#ifdef block	82	66.91	59.76%
Max 120 column lines	83	75.0	60.24%
Yes ternary operator	86	38.08	94.19%
no #ifdef block	88	31.93	95.45%
Properly Indented 'if'	89	28.98	95.51%
No Indentation 'if'	89	33.94	96.63%
K and R (Java Style) Braces	87	28.06	97.7%
No ternary operator	84	24.01	98.1%

3. Results

Software File Survey

The first survey had more thn 400 participants who reviewed 10 randomly selected C and C++ source files. Volunteers provided the source files or the files were found in open source repositories. When the survey was completed, the source files were analyzed using nsiqcppstyle to determine the characteristics of the programs which were considered unpleasant to review [12]. The most prominent characteristics of source code considered unpleasant to review are displayed in Table 1. This survey was first mentioned in the ACM Companion of the XP Conference [7].

Software Statements Survey

In total 140 people participated in the code statements survey, with 75 participants completing the entire survey. The survey allowed volunteers to evaluate as much code as they desired, with the complete survey lasting 10 questions. As mentioned previously, this survey displayed both complicated and uncomplicated constructs, and volunteers were asked to evaluate the results of the code snippet. The correctness of the volunteer's code evaluation, as well as the time which was taken to do the evaluation, were recorded. The results of this survey are shown in Table 2.

Code Reviews

Results Code Review—GCC is the largest of the three evaluated projects, with more than 4 millions of lines of code and many participating developers. The effort was measured based on the number of comments per reported bug and the bugs themselves. Four releases of GCC were scanned and conformance measurements were taken. A Bugzilla report was also generated showing the number of bugs and how many replies each bug had. In the analysis, it was assumed more replies meant more effort spent on that particular release. Results of the evaluation of GCC are shown in Table 4. It is important to note that 8.x is still an active release. At the time of this writing, the current release is 8.2.

Results Code Review— LyX is a favorite Latex formatting tool. Though we found LyX through Bugzilla, the Lyx project

Table 3. Success Rate Evaluation

Success Rate	Count	Average
Success >90%	6	37.15 seconds
Success <90%	6	59.57 seconds

Table 4. **Results of GCC Evaluation**

Major Release	Replies	Bugs/ Release	Conformance
5.x	5339	1042	58.5%
6.x	5994	1206	57.27%
7.x	7822	1510	53.52%
8.x	6149	1418	53.5%

Table 5. **Results of Lyx Evaluation**

Release	Critical Bugs	Conformance
1.1	3	70.2%
1.2	49	78.04%
1.3	50	68.65%
1.4	118	66.62%
1.5	209	50.03%
1.6	263	42.11%

ironically did not use Bugzilla for bug tracking. According to the LyX website, the LyX project was started in 1995 with the most recent release made in September of 2018. LyX has more than four hundred thousand lines of C, C++, and header file source code. For LyX, the number of reported critical bugs on their external bug tracking was used as an indication of effort. The results are shown in Table 5. A linear relationship is shown in Figure 1.

Results Code Review—Sudo is a popular tool on Linux and claims that is: "allows a system administrator to delegate authority to give certain users (or groups of users) the ability to run some (or all) commands as root or another user while providing an audit trail of the commands and their arguments [11]." Sudo happens to be the smallest project with about one hundred thousand lines of code and has been in active use and development since 1994. Since the number of errors reported was relatively small compared to the other projects, this investigation made a simple comparison of the number of reported bugs from their external bug-tracking as a measure of effort. The results are shown in Table 6. Note that Sudo is

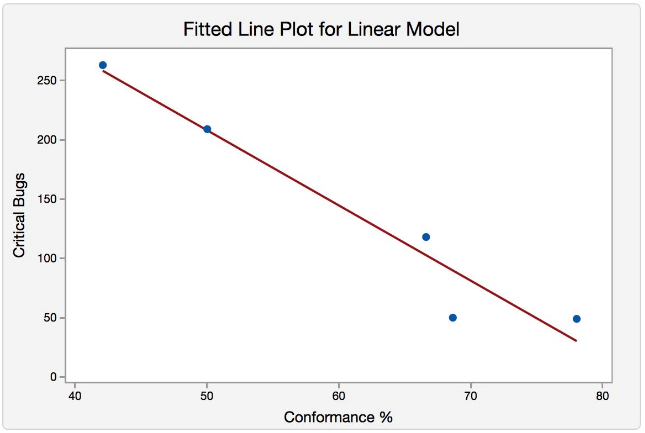

Figure 1. Lyx Critical bugs versus conformance

Table 6. **Results of Sudo Evaluation**

Release Range	Reported Bugs	Years	Average Conformance
1.6.3-1.7.6	21	2000-2012	50.68%
1.8.0-1.8.25	44	2011-2018	23.08%

in active production and the current stable release at time of writing is Sudo 1.8.25p1 [11].

4. Discussion

It was expected that complicated lines of code took longer to review, and the Software Statements Survey results shown in Table 2 confirms that. In Table 3, we can see that complicated statements took about 40% more time on average to review than uncomplicated statements. A dramatic and somewhat unexpected result was how often the result of the complicated code review was incorrect. If we consider a successful evaluation to occur when results are higher than 90% , on average, less than 50% of the complicated code was evaluated correctly. According to Schach, code reviews and inspections are the least expensive way to find faults in software [13]. If complicated constructs are used, one can conclude at the very least the benefit of code reviews is lost or not taken advantage of to its maximum potential. Considering the low success rates of evaluations of complicated code, one can imagine faults slipping through into the final applications. We also observe that lousy indentation is far more problematic than no indentation at all. Fortunately, indentation is something development tools can manage. Space around operators did not seem to help evaluations of complicated formulas as they were poorly done with and without spaces. The ternary operator success rate was not all that different from alternative coding in the successful evaluation. However, it seems it takes more time to digest a ternary statement than code built with "if" statements.

In the GNU GCC code review, it was interesting to see that the GCC maintainers must be enforcing a coding standard. The conformance of GCC concerning items unpleasant to review only marginally varied over many years. Further examination of GCC found that two popular issues of conformance were braces for even one statement and space around operators, which some may argue are not significant issues. Looking at the GCC results, we see the number of bugs per release is relatively stable. An interesting part of the review is that when the conformance dropped marginally, the number of bugs and amount of effort changed as well. (Version 8.x of GCC is still under active development.)

A review of the LyX code also showed interesting results. There was no single moment where the conformance suddenly changed, but a gradual change of code conformance happened over time which is shown in Table 5. The one obvious outlier in the table is due to release 1.1 of LyX having just moved from Bugzilla to Trac, so it is likely not all the 1.1 bugs were captured by Trac [14]. The main observation is that as conformance dropped, the number of reported critical bugs increased linearly. See Figure 1.

The final project reviewed was Sudo. As mentioned, Sudo is a smaller project than GCC and LyX with a smaller number of bugs reported on their bug reporting website. Sudo was not analyzed in great detail, but what made their results interesting is how starting with version 1.8, the conformance fell by 50%, but at the same time, the number of reported bugs increased two-fold.

5. Conclusion

The results of these surveys and code reviews show a promising connection between complicated software and software faults and development effort. Creating software is a generally an expensive endeavor. Failure of software not only causes the loss of financial investment but can have adverse effects on people's lives. Understanding what makes software complicated is an essential part of avoiding, faulty, complicated software. The results of the conducted survey indicated that software considered unpleasant to review is likely overly complicated. It is possible for an organization to employ very readily available tools to measure the reviewability of their software, adjust, and improve their software process. A preliminary evaluation shows the paper's findings are reasonable. However further research should be done to determine if indeed, software that is easier to understand and review results in fewer faults.

These findings imply that the probability of success of pico- and nano- satellites improve with the reduction of complexity of the software. Knowing what programming features make a complicated system, it is possible to create practical programming guidelines for programmers to follow. It is hoped that the information provided in this paper will help programmers create less complicated source code, resulting in higher success rates for deployed satellite systems.

Acknowledgments

The authors thank Anders Koskinen and Rebecca Dorin for their assistance in editing this document.

References

[1] M. Dorin and S. Montenegro, "Coding standards and human nature." *International Journal of Performability Engineering*, vol. 14, no. 6, 2018.

[2] M. L. Drew, "A coding standard for the c programming language," 1999.

[3] D. Straker, *C style: standards and guidelines*. Prentice-Hall, Inc., 1992.

[4] B. Kernighan *et al.*, "Jpl institutional coding standard for the c programming language," *California Insititute of Technology*, 2009.

[5] G. J. Holzmann, "The power of 10: rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–99, 2006.

[6] Z. S. Hossein Abad, R. Karimpour, J. Ho, S. M. Didar-Al-Alam, G. Ruhe, E. Tse, K. Barabash, and I. Hargreaves, "Understanding the impact of technical debt in coding and testing: An exploratory case study," in *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice*, ser. SER&#38;IP '16. New York, NY, USA: ACM, 2016, pp. 25–31.

[7] M. Dorin, "Coding for inspections and reviews." New York, NY, USA: ACM, 2018.

[8] Bugzilla. (2018) Bugzilla, installation list. [Online]. Available: https://www.bugzilla.org/installation-list/

[9] G. Team, "Gcc home page," 2018. [Online]. Available: https://gcc.gnu.org/

[10] L. Team, "Lyx home page," 2018. [Online]. Available: https://www.lyx.org/

[11] S. Team, "Sudo main page," 2018. [Online]. Available: https://www.sudo.ws/

[12] J. Yoon and K. Tyagi. (2014) nsiqcppstyle. [Online]. Available: https://github.com/kunaltyagi/nsiqcppstyle

[13] S. R. Schach, *Object-oriented and classical software engineering*. McGraw-Hill New York, 2007, vol. 6.

[14] V. van Ravesteijn, "Is lyx bugzilla closed?" 2010. [Online]. Available: https://www.mail-archive.com/lyx-users@lists.lyx.org/msg80875.html

Biography

Michael Dorin *has nearly 30 years of software development experience and worked in many engineering environments. His background includes engineering work in public safety communications, medical devices, telephony, and aircraft navigation. Michael teaches for the University of St. Thomas and is currently working towards a PhD through the University of Würzburg.*

Sergio Montenegro *received a master degree in computer science from the Technical University of Berlin, followed by a PhD from the same university. He has been programming satellites for the last 20 Years. Currently he is a Professor for aerospace information technology at the University of Würzburg (Germany). His research interests include dependability and simplification for control software.*