

Logical Friction: Some Lessons from the Automotive Industry to the Software Industry

¹ Carlos Ferran, ² Ricardo Salim

¹ Governors State University, IL, USA

² IESA, Caracas, Venezuela

¹ cferran@govst.edu, ² rsalim@softwaredevenezuela.com

ABSTRACT

Even a perfectly manufactured vehicle requires periodic maintenance in order to function properly throughout its life cycle. Software also requires periodic maintenance to function as expected. The absence of maintenance against wear-and-tear (mechanical or logical) eventually breaks down any system. Logical and mechanical frictions are not identical; yet their effects are the same. Logical maintenance is as important to software as mechanical maintenance is to machines. The availability, quality, and cost of software maintenance should be considered during the purchase decision and included as part of the total cost of acquisition.

Keywords: *Software development, bugs, errors, maintenance, TCA*

1. INTRODUCTION

Certain flaws in software cannot be attributed to errors in design or to defects in manufacturing. They are not subject to warranties. They are analogous to the unavoidable wear-and-tear due to friction in mechanical systems. Friction damages the surface of parts that come into contact at different speeds and creates the need to replace not only the dissipated energy but also the worn-out components; otherwise, the system eventually stops working. If a salesperson argues that a car will not require any maintenance, he will surely lose the sale because the customer would not believe him. However, if she were selling software, many would believe her and purchase it.

That is because mechanical maintenance is expected to replace worn-out components while software maintenance is not really thought as true maintenance but as continuous improvements towards perfection.

This paper introduces the concept of logical friction as a phenomenon similar to physical friction since it causes in software something comparable to the wear-and-tear of machines. Understanding this concept is important because it explains the need for software maintenance and disproves the myth of attaining perfect software. Just as there is no perpetual motion, there is no perfect software. There are neither service free cars nor service free software.

2. COMPLEXITY AS THE SOURCE OF LOGICAL FRICTION

There are several sources of logical friction but the most important one is that a high degree of complexity is inherent in software, which in turn is an inevitable cause for errors. There is a direct relationship between complexity and the number of errors present in software [1-3]. Thus, we argue that bugs due to complexity are as inevitable as the wear-and-tear due to friction. This is because, based on the concept of entropy of statistical mechanics, the probability of their occurrence is many

orders of magnitude higher than the probability of not occurring [4].

Similar to the way that mechanical friction can be calculated by the product of the friction coefficient times the amount of usage, one form of logical friction can be derived from the product of complexity times the amount of usage. As the effects of mechanical friction are only solved by replacing parts through preventive and corrective maintenance, the effects of logical friction cannot be solved during development but through post-implementation preventive and corrective maintenance.

Maintenance should not be considered a contingency but a structural part of the software life cycle [see for example 5, 6].

We cannot assert the impossibility of never-failing software with the same force that we can assert the impossibility of a frictionless engine. It could be argued that it is possible to write a program so simple that it is guaranteed to be infallible. However, that program would execute on top of (and interact with) more complex software which in turn creates the possibility of errors.

Furthermore, very simple software can hardly fulfill what we call a function. It would represent something equivalent to an engine without moving pieces or with a movement almost equal to zero. Thus, for our purposes, this essay refers only to complex software like operating systems (i.e. Windows, UNIX, Android, iOS), application software (i.e. office suites, ERPs, browsers, mobile applications –APPS–, etc.), utilities (i.e. antivirus, file compressors, communication tools, recovery software, etc.), and mission critical software (i.e. embedded software for devices used in surgery, intensive care units, military defense, air or space navigation, satellites, etc.).

The measure of complexity is an important part of software metrics [7-13]. However, there are many more software metrics that do not relate to complexity [14, 15].

Moreover, some are based on software cost estimations [16], risk [8], or information flow [17] just to name a few.

Complexity measures are based or focused on different aspects of software. Some are based on the number of lines of code or on the number of points in the flowchart, for example, cyclomatic complexity [18-20] and complexity based on function points [19]. Other complexity measures are based on database points [7], on software design [21], on project requirements [22] and even on psychological criteria [23]. And not surprisingly, the unnecessary and the induced complexity –obfuscation– is also measured [24]. However, more recently the usefulness of more complexity measures has been questioned [25].

The precision of these measures depends on the type of software being developed, and none is universally accepted. Furthermore, they have all have been questioned in respect to their potential for predicting quality, reliability, cost or need for maintenance [9].

In any case, for our purposes, it is sufficient to establish a direct relationship between the complexity of a program and the probability that it may contain errors, without making precision a critical condition. To establish this relationship, we can use the most basic notions of complexity and possibility to err, showing the relationship between software complexity and the possibility that a programmer can make a mistake while writing a program.

If we were to write a single instruction program using a language that had only 20 potential instructions and the instruction was selected randomly, the possibility of the program being incorrect was 19 in 20 or 95%.

However, instructions are carefully selected by programmers, not randomly. Therefore, the probability of selecting the wrong instruction is far lower. Nonetheless, programmers make mistakes. For the purpose of this example, let us assume a very conservative probability that the programmer will only make a mistake once in every 1,000 cases (or 0.1%). As the program increases in complexity (has more instructions) we can express the probability of one mistake as $1-(99.9\%)^n$ where n is the number of instructions. Therefore, the probability of at least one error in a program with 100 such instructions is $1-(99.9\%)^{10}$ or 9.6%, but in a program with 500 instructions it is 39%.

The probability of at least one error increases exponentially as the program grows larger. Moreover, any fully functional operating system or business information system written with simple instructions (where the programmer can only select from a small set of 20 options) will certainly have more than 500 instructions. In fact, most of these applications will have several thousand lines of a code that is not made up of simple instructions but of more complex ones that include parameters (another

place where an error may occur, which therefore increases the initial probability of making an error in the line).

Additionally, this assumes that there are no errors in the lower level languages that support the language in which our programmer is writing; an assumption that is very difficult to accept because a language involves, by itself, a great number of independent instructions.

The first versions of accounting software written for the Apple II were classic examples of the above. These programs showed sporadic, insignificant (at the level of decimals), yet unacceptable arithmetic errors. These errors were proved not to be originated by the application software but by the unsuspectedly low numeric precision of its operative system. Thus, it was necessary to complement the application software with low-level code in order to overcome the imprecision of the underlying system software that added an unexpected level of complexity.

Most mistakes made by a programmer are detected and corrected in partial tests that look at a few symbols at a time. These tests are followed by other partial tests on the already tested parts, and so on. This systematic methodology of testing reduces the probability of errors, but the process in itself is also a program of a certain complexity, and as such, it is also subject to mistakes, not less being difficult to detect. We are then struggling with a somewhat measurable complexity, but by no means less problematic [21, 26].

This implies that to the number of programmer-hours required to write a certain number of symbols, we must add an additional number of programmer-hours to test that each independent symbol is correct (note that if the code is found to be incorrect and has to be corrected, we have to test it again). It is known that the number of testing hours usually surpasses by far the number of coding hours [27].

Furthermore, until this point we have assumed that the analysis and the specifications are error-free. However, the probability of errors in the analysis or in the specifications is also very large, making this an added layer of complexity that increases the total probability of errors occurring.

3. OCURRENCE OF LOGICAL FRICTION

Usage is where logical friction really occurs. Just as there is no wear-and-tear in a motionless machine, no bugs are detected in a system that is never used. A system that is never used will never fail. As usage increases (in both intensity and length), so does the probability of the user going into new, not previously used options or combinations, and thus, finding a bug. As the probability of working with options that were not used previously increases, the probability of failures rises proportionally.

The importance of usage in software quality metrics is such that there is a construct called operational profile that formalizes it [28]. The operational profile forms part of many methods of reliability estimation [see for example 29, 30].

To compound this problem, logical failures have a similar effect as mechanical failures. Failures tend to produce additional damages if the use continues after their occurrence. This is because a failure by definition is something unforeseen. We can presume that the consequences ensuing from something unforeseen will not be constructive, or even innocuous, but destructive. To expect that some unforeseen circumstance might result with a certain probability in something desirable is akin to expecting that a shuffled deck of cards will accidentally fall from a table and arrange itself into a pre-established or perfect order. The second principle of thermodynamics establishes that ways leading to creating or arranging something are far less frequent compared to those leading to its destruction or disorder.

Logical friction is found in the fact that the software is developed to solve a real life problem and real life evolves. Thus, a program that solves the old problem and not the new and evolved one is also considered have bugs but there was no way for the developing team to foresee that the problem would change and consider that into the system. Only (maintenance) updates can remove this problem that was not the fault of the developer.

Most applications not only work on top of others, but also co-exist and interact with other applications. This creates yet another level of complexity.

Another form of logical friction occurs when the total number of hours required by the complexity exceeds the amount budgeted. Because time is a critical factor in software development [31], logical friction occurs when the development time exceeds the required deadline.

Yet another form of logical friction occurs when the operational complexity of the system handed over to the final user is higher than the expected operational simplicity. Operational complexity is the minimum number of decisions (binary, independent) that a user has to make to operate it satisfactorily. It is usually expressed as the number of keystrokes or clicks needed to execute an operation. In contrast, expected simplicity is the maximum number of decisions the user is willing to make to operate the system. Maximum simplicity is expressed as: "You press any key and that's it!" Logical friction occurs when the minimum required is greater than the maximum expected.

There are two options when the complexity is higher than expected: 1) return the system to the programmer in order to improve (correct) it accordingly, or 2) accept the added complexity. Unfortunately, simplifying the system for the user makes it more complex

for the programmer. In either case, we have a total complexity greater than expected and consequently a greater probability of errors surfacing, or in our terms: logical friction.

Once a system enters operations, the sum of the logical frictions is the basis for calculating –given certain proportionality constants– the probability of failures (bugs). These proportionality constants correspond to the average number of failures per unit of logical friction that can be observed in statistical samples of systems in operation. However, while viable, we have not yet calculated these values and it would not be easy to do.

Finally, correcting failures, once they have been discovered, is a task of a complexity similar to that of the original programming; thus, it creates additional logical friction and possibly new failures.

4. CONSEQUENTIAL SIMILARITY OF FRICTIONS

The consequential similarity between physical machines and software can be observed in the curve of system stabilization. This curve shows that after a period of intense occurrence of failures and their correction, systems enter into a state of stable operation, in which failures occur infrequently or not at all. The initial period of frequent failures and corrections is not part of normal system use but of the process of transferring it to the final user. This is similar to the break-in period of a machine during which initial maladjustments are detected and corrected, and then final adjustments are made. The stable period is what corresponds to normal system use.

Nevertheless, this period is stable not because failures do not occur, but due to a balance between the occurrence of failures and maintenance of the system. And this is comparable to the usually long period during which machines work without any serious problems as long as they receive adequate maintenance and worn out parts are replaced before causing any damage.

This period of normal use, in mechanics as well as in informatics, could continue indefinitely. However, in general, this does not happen. Systems ultimately become obsolete when we can no longer find spare parts or technicians who still know how to work with the old technology on which the system is based. Hence, it is the absence of maintenance against wear-and-tear (mechanical or logical) that puts the systems out of use. Age only makes maintenance more difficult and expensive.

Logical and mechanical frictions are not identical; yet their effects are the same. Logical maintenance is as important to software as mechanical maintenance is to machines.

Software maintenance should be: (a) understood as a service whose main objective is to act against the failures of information-intensive systems caused by logical

friction and to correct the damages due to them; (b) supplied by the manufacturer of the system or by trusted third-party providers in accordance with market conditions; (c) complemented by the supply of maintenance manuals and software (known as utility software); d) corrective, preventive, or mixed with a periodic or contingent coverage. The availability, quality, and cost of software maintenance should be considered during the purchase decision and be included as part of the total cost of acquisition.

The automotive industry has learned to co-exist with the inevitable failures. Even though they strive for continuous improvements in design and manufacturing, they also recognize the theoretical limits and the increasing cost-benefit relationship. The acceptance of the impossibility of a perfect or perpetual motor (perpetuum mobile) has resulted in the continuous growth and improvement of a worldwide service network that provides preventative and corrective measures. This network is so widespread that we expect to find an auto repair shop even in the middle of the Sahara desert.

On the other hand, the software industry has not accepted the impossibility of a perfect large system.

Clearly, too many resources are spent on developing such software instead of on developing a more realistic and useful network of software maintenance.

Nevertheless, sooner or later, we always have to resort to additional post-delivery manpower (analysts and programmers) for the correction of faults and the prevention of its incidences. This is the only way to maintain the system functioning against logical deterioration. This is what we call logical maintenance or more precisely, software maintenance.

There is a need to draw the line and determine when it is reasonable to stop testing and bring the system into operations. This is the point when the cost of testing (in terms of time and money) is higher than the cost of suffering the error and correcting it through maintenance.

In the end, both costs are borne by the user and therefore its minimization is necessary.

This minimization is found at the point where: (a) the client is satisfied with the general tests of functionality and the price of the system, (b) the client understands the probability of failures as estimated by the manufacturer (based on a measurement of the logical friction) and the operational implications, (c) the client accepts the costs of the necessary maintenance, (d) the client is aware of the expected life of the system as estimated by the manufacturer, and (e) the manufacturer guarantees that the system will work in accordance with the agreed functionality up to the date of obsolescence and at a cost not higher than the purchase price plus its maintenance.

This set of conditions, with the practical and legal details duly worked out, could represent an appropriate model of quality control for software maintenance similar to that currently used in mechanical systems.

5. CONCLUSIONS

There is much more that needs to be said regarding a culture of software maintenance. Currently, customers understand software maintenance as something that is meant to provide improvements, modifications, additional user training, and end-user support, while existing bugs must be dealt with through warranties.

However, maintenance should also serve to extract the bugs that are already present in the system that cannot be attributed to errors of design or construction but that will certainly appear through usage due to logical friction. This culture of software maintenance must espouse the concept of logical friction in the same way that we accept mechanical friction and therefore reduce the possibilities that a critical mission could depend on supposedly perfect software. This is a culture that requires the presence of a software maintenance engineer to be well prepared to handle critical contingencies, the same way that we currently require the presence of a mechanical maintenance engineer.

There is neither perpetual motion nor perfect software; there are neither service free cars nor service free software.

REFERENCES

- [1] Basili, V.R. and B.T. Perricone, Software errors and complexity: An empirical investigation. *Communications of the ACM*, 1984. 27(1): p. 42-52.
- [2] Cusumano, M.A., Who is Liable for Bugs and Security Flaws in Software? Attempting to determine fault and responsibility based on available evidence. *Communications of the ACM*, 2004. 47(3): p. 25-27.
- [3] Armour, P.G., Real Work, Necessary Friction, Optional Chaos: Challenges in estimating software scope by effort. *Communications of the ACM*, 2004. 47(6): p. 15-18.
- [4] Zurek, W.H., ed. *Complexity Entropy and the Physics of Information*. 1990, Perseus Books.
- [5] April, A. and A. Abran, *Software Maintenance Management: Evaluation and Continuous Improvement*. 2012, New Jersey: John Wiley & Sons.
- [6] Glass, R.L., *Software Maintenance is a Solution, Not a Problem in Software Conflict 2.0: The Art and Science of Software Engineering*, R.L. Glass, Editor. 2006, Developer Books: Atlanta, GA.

<http://www.scientific-journals.org>

- [7] Abiad, S., R.A. Haraty, and N. Mansour. Software metrics for small database applications. in The 2000 ACM symposium on Applied Computing. 2000. ACM.
- [8] Fenton, N.E. and M. Neil. Software Metrics and Risk. in FESMA 99, 2nd European Software Measurement Conference. 1999.
- [9] Fenton, N.E. and M. Neil, A Critique of Software Defect Prediction Models. IEEE Transactions on Software Engineering, 1999. 25(5): p. 675.
- [10] Iyengar, S.S., N. Parameswaran, and J. Fuller, A measure of logical complexity of programs. Computer Languages, 1982. 7(3-4): p. 147–160.
- [11] Kafura, D.G. and G.R. Reddy, The use of software complexity metrics in software maintenance. IEEE Transactions on software Engineering, 1987. 13(3): p. 335–343.
- [12] McCabe, T.J., A Complexity Measure. IEEE Transactions on Software Engineering, 1976. 2(4): p. 308–320.
- [13] Stetter, F., A measure of program complexity. Computer Languages, 1984. 9(3-4): p. 203–208.
- [14] Ebert, C. and R. Dumke, Software Measurement 2007, New York: Springer.
- [15] Henry, S.M., D.G. Kafura, and K. Harris. On the relationships among three software metrics. in ACM Workshop on Measurement and Evaluation of Software Quality. 1981.
- [16] Boehm, B. and e. al., Software cost estimation with COCOMO II. 2000, Englewood Cliffs, NJ: Prentice-Hall.
- [17] Henry, S.M. and D.G. Kafura, Software-structure metrics based on information flow. IEEE Transactions on Software Engineering, 1981. 7(5): p. 510–518.
- [18] Gill, G.K. and C.F. Kemerer, Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering, 1991. 17(2): p. 1284-1288.
- [19] Laird, L.M. and M.C. Brennan, Software Measurement and Estimation: A Practical Approach. Quantitative Software Engineering Series. 2006: Wiley-IEEE Computer Society Pr.
- [20] Zhang, H., X. Zhang, and M. Gu. Predicting Defective Software Components from Code Complexity Measures. in 13th IEEE Pacific Rim Dependable Computing conference. 2007. Melbourne, Australia.
- [21] Braha, D. and O. Maimon, The Measurement of a Design Structural and Functional Complexity, in A Mathematical Theory of Design: Foundations, Algorithms and Applications, D. Braha and O. Maimon, Editors. 1998, Springer. p. 241-277.
- [22] Sharma, A. and D.S. Kushwaha. A Complexity measure based on Requirement Engineering Document and its Validation. in International Conference on Computer and Communication Technology (ICCCT). 2010. IEEE.
- [23] Curtis, B., et al., Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979. 5(5): p. 96–104.
- [24] Yongdong, W. and H. Fang, A Framework for Measuring the Security of Obfuscated Software, in 2010 International Conference on Test and Measurement 2010.
- [25] Herraiz, I., and A. and E. Hassan, Beyond lines of code: Do we need more complexity metrics?, in In Making Software: What Really Works and Why We Believe It. . 2011, O'Reilly Media: Sebastopol, CA. p. 125–144.
- [26] Jones, C., Software Measurement Programs and Industry Leadership. CrossTalk, The Journal of Defense Software Engineering, 2001.
- [27] Humphrey, W.S., Bugs or Defects? SEI Interactive, 1999(3): p. 1.
- [28] Musa, J.D., Operational profiles in software reliability engineering. Software, IEEE, 1993. 10(2): p. 14-32.
- [29] Kumar, K.S. and Ravindra B. Misra, Software operational profile based test case allocation using fuzzy logic. International Journal of Automation and Computing, 2007. 4(4): p. 388-395.
- [30] Bai, C.-G., Bayesian network based software reliability prediction with an operational profile. The Journal of Systems and Software, 2005. 77(2005): p. 103–112
- [31] Humphrey, W.S., Why Does Software Work Take So Long? SEI Interactive, 1998(6).

AUTHOR PROFILES

Dr. Carlos Ferran is an Associate Professor of Accounting and Management Information Systems at the College of Business of Governors State University (GSU). He teaches Accounting, MIS, and Strategy. He is the College

Assurance of Learning (AoL) and Online education coordinator, member of the GSU AoL committee, and Peer Reviewer for ABET and the Higher Learning Commission. He started his professional career developing accounting applications for the Apple II and since then he has published many peer-reviewed articles, book chapters, and conference proceedings. He authored a book and co-edited another on Global ERP. His last work outside academia was as a CIO in a major financial institution. Dr. Ferran has taught and given academic lectures in English and Spanish at universities in Argentina, China, Colombia, Costa Rica, Mexico, Spain, United States, and Venezuela.

Dr. Ricardo Salim earned his doctoral degree from Universidad Autónoma de Barcelona. For over 30 years he has worked as an entrepreneur (in Venezuela, Ecuador, Spain, USA and other countries) starting companies and personally designing and developing IT products and services for the retail, service, and banking industries. He has published multiple peer-reviewed journal articles and conference proceedings regarding his research and professional experience. He is a Director of Software de Venezuela and invited professor at IESA, Caracas.