

Neither a perpetuum mobile nor a perfect software: Sincerity in the relationship between the manufacturer and the client with respect to software defects

Ricardo Salim Koussa

Cautus Networks Corporation, USA

and

Carlos Ferrán-Urdaneta

College of Business, Rochester Institute of Technology

Rochester, NY 14623, USA

ABSTRACT

Certain mechanical shortcomings in engines cannot be ascribed to errors in design or defects in construction. They are due to the fact that it is impossible to design and construct engines in a way that completely eliminates friction between its parts. Friction causes wear and tear that is proportional to the amount of use. Likewise, when it comes to software one should not expect it to behave any differently. One must expect a set of errors in the software that cannot be attributed to faulty design or construction. A set of errors will materialize as a function of the equally unavoidable logical “friction” between, on the one hand the necessary but hardly specifiable complexity of software, and on the other hand the following two requirements: 1) The necessity to observe strict limits of the cost and time of development, and 2) The necessity to offer a maximum simplicity of operation. In other words, just as mechanical deficiencies will be proportional to the amount of use to which an engine endures, the user of a software application will experience informational flaws (bugs) that are proportional in number to the use that s/he makes of the product.

Once this is shown, we argue that the proper way to approach these software defects is neither via innumerable rounds of trial and error in laboratories, nor with mathematical algorithms that try to demonstrate correctness to 100%. Instead, we believe that the right way would be –analogous to mechanical systems– to deliver the product to the user with a minimal standard of quality but with a certifiable plan of preventive and corrective maintenance. Such a plan would assure the user that any errors that emerge are looked upon in due time, and that the manufacturer will try to keep the downtime as short as possible.

All this leads us 1) to discuss the concept of complexity up to the point where we can show that it is possible to contrast it quantitatively with the limitation of cost and time of development, as well as with operational simplicity; the purpose being to establish a clear distinction between defects which can be attributed to complexity, and those which cannot; 2) to show how the

occurrence of defects due to complexity is a function of use of the software in the same way as defects due to wear and tear are a consequence of the use of the engine; 3) to show the viability of the concept of a minimum standard quality in software; 4) to sketch the model of preventive and corrective maintenance of defects due to complexity and use of software, analogous or at least as effective as the preventive and corrective model of maintenance of mechanical wear and tear; 5) identify successful experiences in the software industry which resemble the above mentioned model; or elements that may contribute to its implementation; and 6) to propose and promote a culture of information system maintenance which would make it foreseeable for the user, acceptable, and finally budgetable, in a similar manner as the maintenance of an automobile. All of these should bring about a sincere relationship between producer and client, and reduce the level of discord between them.

Keywords: Software Maintenance, Software Development, Ethics, Error prevention, Error Detection,

INTRODUCTION

There are certain flaws in mechanical systems that cannot be attributed to errors in design or defects in construction but they are the unavoidable effects of friction. Friction is always present when different surfaces come into contact at different speeds. Friction damages these surfaces (wear and tear) and dissipates kinetic energy from the system. Therefore there is a need to replace the energy as well as the worn-out components, otherwise the system will eventually stop working. Thus friction has the effect of a force opposed to velocity. Hence it impedes perpetual movement.

Mechanical systems, like engines, consist of moving parts subject to friction. When those parts wear out beyond a certain point, the system fails to continue functioning. In order to minimize such failures, engines are designed in a way that surfaces are exposed to as little friction as possible. Additionally, lubrication is used for the same purpose, to reduce the wear and tear caused by friction. Finally, one resorts to preventive or corrective replacement

of substantially worn-out pieces. Eventually, this is the only way to maintain a functioning system; protecting the system against wearing out by replacing the worn-out part. This is what we call maintenance. Mechanical wear and tear increases progressively with use, as do the damages caused by defects due to lack of maintenance.

Obviously, the need for maintenance is a function of a friction coefficient multiplied by a number that represents the amount of usage of the system. Engine parts with the same friction coefficient that are not used much will suffer proportionally less wear and tear than those that are used more. If usage increases, wear and tear will also increase and so will the need for maintenance. The defects caused by lack of maintenance are not the fault of errors in design or construction and should not be attributed to them.

Something analogous, or at least similar to the above, happens to certain non-mechanical systems, that is, systems made out of non-material but informational, logical, or software components. Examples of these are information-intensive systems (excluding their hardware components). There is what we could call a *logical friction* between the complexity of such systems and the expected simplicity of their operation. There is also an additional *logical friction* between the uncertainties inherent in their complexity and the strict limits of cost and time of delivery usually imposed on their development. The only way that we could eliminate this *logical friction* would be by having an unlimited amount of time and other resources to freely spend in the development of an information system. But as long as resources are limited, especially in respect to system complexity, we will have *logical friction*.

Logical friction causes a phenomenon comparable to the erosion of the components subjected to friction. It is a form of “logical wear and tear” which consists on the probability that unavoidable faults (bugs) will occur as the system is used. Faults, when given the *logical friction*, cannot be considered avoidable mistakes of design or construction. It should be clear that we are not saying that the information or the software wears out with use. We only use the term “wear out” as an analogy to the mechanical concept of wearing out, as it also implies a probability that faults may occur in the system in direct relationship to its use. Within these faults we also include the unanticipated faults that -with limited developmental resources -can only be detected by using the system. Friction and the ensuing logical wear and tear, clashes with the perfect functioning of the system, and poses the question of the need for a revision of the system’s complexity, costs, delivery time table, and ease of operation.

Information-intensive systems, such as those applied to management and information systems in general are well known examples of this kind.

In order to minimize the probability of faults occurring, the design of an information system usually tries to limit or reduce the number of choices or options that the user needs to make to operate the system. In addition, they are complemented by subsystems of automatic maintenance. Finally, we resort to additional post-delivery manpower (analysts and programmers) for the correction of faults and the prevention of such incidents. Sooner or later, this is the only way to maintain the system functioning against *logical deterioration*. This is what we would call logical maintenance or more precisely, software maintenance. *Logical deterioration* (in the same manner as mechanical deterioration) and damages due to the lack of maintenance increase with use.

Thus, the necessity for software maintenance, like in the mechanical case, is a function of a *logical friction* coefficient multiplied by the intensity of system usage. With the same friction coefficient, the components of a certain application that is not used often will involve fewer opportunities to present failures. If it is used more, failures will be more frequent and correspondingly the need for maintenance.

By now, the analogy between the complexity of information systems and motion in mechanical systems should be clear. Both are inherent to the respective systems and at the same time both are necessarily limited or restricted by their environment. We no longer continue the futile pursuit of trying to design mechanical or information intensive systems without movement or complexity restrictions that at the same time would be practical and useful. Restrictions on complexity generate a *logical friction* the same way as restrictions on movement create mechanical friction. And both produce an irreducible probability of occurrence of failures once the system starts operating. Failures that cannot be attributed to design or construction errors. Failures that can only be controlled by means of preventive or corrective maintenance. And the amount of preventive or corrective maintenance needed, is a function of the friction coefficient and the intensity of system usage. There is neither perpetual motion nor perfect software.

COMPLEXITY

One cannot assert the impossibility of never-failing software with the same force as we assert the impossibility of a frictionless engine. Sure enough, it is possible to write a program so simple as to guarantee its infallibility, although more than one experienced programmer would have serious doubts and the general public would agree while remembering, for example, the millennium bug. On the other hand, a very simple piece of software could hardly fulfill what we might call a function. It would represent something equivalent to an engine without moving pieces or with a movement almost equal to zero. So, in what follows, we shall refer only to software that can be called complex.

This last requisite (of being complex) would be tautological, unless we are able to provide some measurement of the complexity in question. This measurement relates to the “degrees of freedom” or, what is the same thing for our purpose, the possibility of a programmer to make mistakes while writing a program. If the programmer could select one symbol from a choice of two, he would have only one degree of freedom, in other words, could commit only one error. If s/he could choose one out of four possible symbols, s/he would have two degrees of freedom, and in general, if s/he could choose once one out of N, s/he would have $\log_2(N)$ degrees of freedom.

The programmer chooses symbols, not only once, but many times, while writing a program. In fact, s/he can make one mistake every time s/he writes a bit (a bit is the smallest unit of information, it has two states, that is, one degree of freedom). Furthermore, s/he can make eight mistakes on each occasion that s/he writes a byte (e.g. a letter). Of course, computers, operating systems, and programming languages offer many facilities to avoid errors at the bit level, at the byte level, and even at the word level. Each time the programmer writes a command (a predefined symbol at a specific level) the system generally checks it and makes sure that there are no errors at lower or dependent levels. We call “high level languages” those whose symbols can be decomposed into lower level symbols. Those lower level symbols may in turn be decomposed into symbols of an even lower level. The lowest level possible is that of machine instructions. Today programmers rarely write machine instructions but they use high-level languages to develop those complex applications to which we are referring in this article.

While being extremely conservative, any experienced programmer would still agree that a piece of software that is reasonably functional would have at least ten lines of high-level code. Each one of these lines would have at least three slots to write one among three possible independent symbols, such as variables, commands, data columns, relationships, etc. This means the programmer has 10^9 (a ten followed by nine zeros) opportunities to make a mistake. This is clearly an immense figure, but even the simplest accounting software, for example contains many more lines of code, and therefore the opportunities to make a mistake are much larger. All this while assuming that there are no mistakes 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 symbols [4].

No doubt, 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 way the probability of remaining mistakes is reduced at each step exponentially. But this systematic methodology of testing

becomes in its turn a program of a certain complexity, and as such, it is also subject to mistakes, not less difficult to detect, and so on. We are then struggling with a somewhat measurable complexity, but by no means less problematic.

All this implies that to the number of programmer-hours required to write a certain amount of symbols, we must add an additional number of programmer-hours to prove 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). This second number is obviously a function of the complexity. So that once measured, the complexity of the system already designed and even in an advanced stage of construction, the number of programmer-hours that are required to test it can be calculated. It is known that this number usually surpasses by far the first one [3].

Logical friction occurs when the total number of programmer-hours required by the complexity exceeds the stipulated or budgeted amount. *Logical friction* also occurs when the development time required exceeds the deadline stipulated by the project.

Logical friction also occurs when the operational complexity of the system handed over to the final user is higher than the operational simplicity expected. The operational complexity is the minimum number of decisions (binary, independent) that the user has to make in order to operate it satisfactorily. It is usually expressed as “how many keys or clicks the user has to press in order to bring about certain operation”. On the other hand, the expected simplicity is the maximum number of decisions that the user is willing to make to properly operate the system. Maximum simplicity is expressed as “you press any key and that’s it!”. *Logical friction* occurs when we try to reconcile the fact that the minimum required might be greater than the maximum expected.

If the complexity is higher than expected, we face these possibilities: 1) return the system to the programmer in order to improve (correct) it accordingly or 2) accept the system as it is. Unfortunately, making a more simple system for the user, means making it more complex for the programmer. So that 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*.

The sum of the *logical frictions* is the basis for calculating -given certain proportionality constants- the probability of failures (bugs) occurring in the system, once it enters into operation. These proportionality constants correspond to the average number of failures per *logical friction* unit that can be observed in statistical samples of systems in operation. We are not saying that it will be easy to obtain these proportionality constants, but that it is possible.

Failures due to total *logical friction* cannot -by definition- be attributed to errors of design or construction of the system.

USE AND PROBABILITY OF FAILURES

No failures will be detected in a system that is never used. As usage increases (in both intensity and length) so will the probability of the user going into new, not previously used options or combinations. This is generally due to either some operational mistake or by newly arising operational requirements. And as the probability of working with options that were not used previously increases, so do the probability of failures; failures that were not previously detected, or much less corrected.

To compound this problem, logical failures have a similar effect as mechanical failures. Software failures tend to produce additional damages if the use continues after the appearance of such failure. This is due to the fact that a failure by definition is something unforeseen. And we can presume that the consequences ensuing from something unforeseen will not be constructive, or even innocuous, but destructive (the Second Principle of Thermodynamics establishes that ways leading to create or arrange something are far less frequent as those leading to its destruction or disorder). To expect that some unforeseen circumstance might result with a certain probability in something desirable is akin to expecting that a deck of cards without any order, falling accidentally from a table, might arrange themselves into perfect order.

We also have to consider intermittent failures. These occur after some rather complex sequence of events, which cannot be reproduced easily. Thus, these failures do not show up during the normally short period of time when the person in charge of maintenance is present and hence increases the probability of causing additional damage.

Finally we have to take into account that correcting failures, once they have been discovered, is a task of a complexity similar to that of the original programming of the system that creates *logical friction* and possibly new failures.

This confirms the assumption that, like in mechanics, the probability of occurrence of failures in information-intensive systems is proportional to their use. Still, this results from an exploratory study using qualitative data collected through over 35 years of added professional experience of the authors in the field. Confirmatory research using more quantitative data is still required for a more categorical confirmation of this assumption.

Let us also compare this to the generally accepted curves of systems stabilization. According to these curves, 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. We see the period of intense occurrence and corrections of failures not as part of normal use of the system, but as the process of

transferring it to the final user. This is analogous to the *break-in period* of a machine during which initial maladjustments are detected and corrected and final adjustments in accordance with the requirements of the user are carried out. We see the stable period as corresponding to normal use. But this period is stable not because failures do not occur, but due to a balance between the occurrence of failures and the normal maintenance of the system. It 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 additional damage.

This period of normal use of the systems, in mechanics as well as in informatics, could continue indefinitely. But in general this does not happen due to system obsolescence where it becomes difficult to find spare parts in the market or technicians who still “remember” the now obsolete 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. Obsolescence only makes maintenance more difficult and expensive.

A STANDARD OF QUALITY (WARRANTY)

Making sure that complex software does not contain errors is normally a task that involves so many hours of testing and corrections that it's per se cost (costs of the manpower involved plus equipments and management) plus the opportunity costs, convert it into something unviable. Hence it is usual to opt for compromised solutions abandoning any pretension of perfection and instead perform tests and corrections in the laboratory up to a point where the software concerned, even though with errors, is worth its price (paid either by a single client or by many who receive the same copy, paying each a lower price).

At one extreme, the user receives, generally at low cost, software for which s/he is resigned to the fact that errors and deficiencies will appear. But such a user also has the expectation that the manufacturer will eventually take care of them by delivering corrected and improved versions of the same software. In these cases the manufacturer does not accept responsibility for later damages ensuing from such deficiencies and does not specify terms for the publication of new versions that correct the deficiencies after the user has reported them.

At the other extreme, we have the case of the so-called critical mission applications, which are those that require a high degree of reliability. This is the case for systems whose imperfections put at risk human lives (like the automatic pilot in airplanes), or the economic or physical safety of a country, institution, or individual. These situations are usually handled, as far as possible, by mathematical verifications. If such verifications are not possible then statistical tests under laboratory conditions

are the next best option. Furthermore, to complement such statistical tests, ongoing automatic or semi-automatic subsystems of diagnosis and recuperation of contingencies as well as safety provisions are undertaken. But each additional measure to insure continual bug-free operation negatively impacts costs and terms of delivery. The higher the cost, the more impractical a system becomes.

It would seem that the equilibrium will be where these coordinates meet: 1) the client is satisfied with the general tests of functionality and the price of the system, 2) the client has been informed about the probability of failures as estimated by the manufacturer (based on a measurement of the *logical friction*) and the operational implications of these failures, 3) the client understands and accepts the costs of the necessary maintenance to face possible failures, 4) the client understands and accepts the date of obsolescence estimated by the manufacturer, and 5) the manufacturer warrants that the system will work in accordance with the functionality listed in part 1 up to the date of obsolescence and at a price not higher than that of the system plus its maintenance.

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

A MODEL AND A CULTURE OF SOFTWARE MAINTENANCE

The analogy between logical and mechanical friction suggests how we can face their consequences. This in turn suggests a software maintenance analogous to the mechanical maintenance that we know, as for example that of a car.

Mechanical maintenance is a service whose main purpose is to counteract the deterioration of parts caused by friction, as well as to repair the damages caused by it. This service is provided by the manufacturer of the machine, by certified service providers, and also by providers who are not certified but on whom customers rely due to market factors. Furnishing maintenance manuals and supplies like spare parts, tools, lubricants, etc complement the service. The supplier may be the machine manufacturer or some others who prove the quality of their service either in a formal way (some kind of certificate) or informally through the selective process of the market. The service can be protective, corrective, or mixed and the coverage either periodic or contingent. The availability, quality, and cost of the service influence the decision of the buyer of the machine and s/he will usually verify them and sometimes even require a warranty before purchasing the product.

Following this analogy, software maintenance should then, a) be 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) be supplied by the manufacturer of the system or by certified providers or even non-certified providers, trusted by users in accordance with market conditions; c) should be complemented by the supply of maintenance manuals and software (known as utility software) provided by the manufacturer of the system as well as any other who proves the quality of his/her service either formally (by certification) or informally (through the normal selective processes of the market); d) be corrective, preventive or mixed with a periodic or contingent coverage. Its availability, quality, and cost will affect the informational system buyer's decision and s/he should verify and where convenient, ask for a formal warranty before concluding the purchase.

To what degree does all this in fact happen in the market?

The answer is quite clear: there can be no doubt that it does happen in the market, although not in a frank and conscious way, but through market forces in disguise, and with a lack of sincerity by the actors involved. Let us see why.

There is no general awareness of what we have called *logical friction* even though the market transactions engender it consciously or implicitly. As a matter of fact, the suppliers of software usually offer new versions of their systems to their clients. This offer may be on top of services covering their guarantee of quality. Maintenance contracts or contracts of technical assistance that include new improved or revised versions are quite common. And looking at it closely, this is nothing else but a service designed to check what we have called here as the logical wear and tear.

The client accepts thus implicitly, that a probability of failure in the system exists. A failure that can be taken care of only by way of maintenance, even though s/he does not understand its real cause or ascribes it to negligence or profiteering by the supplier. This inconvenience for the client can be abated by divulging the concept of *logical friction* and comparing it to the concept of mechanical friction.

Maintenance of information-intensive systems still relies mostly on the manufacturer or at most on authorized agents. Still, there are examples of independent providers who have gained market acceptance, as is the case of Peter Norton (Norton Utilities). Norton is still another sign of the market trend to accept the inevitable probability of failures in information-intensive systems. Users of Norton Utilities behave in respect to their software in a similar manner as users of cars do in case of mechanical failures.

There are clear signs that manufacturers accept the presence of independent providers of services as adding value to their own products without additional costs to them. This knowledge encourages them to open the access to information and the necessary utility software,

promoting the emergence of independent providers. This is the main reason for programs of assistance by manufacturers to independent agents who add value to their products.

However, we are still far off from the point in which manufacturers tolerate and much less assist independent providers, at a low cost or no cost at all, without any bureaucratic red tape or exclusivity clauses. There is always a fear or mistrust of the lack of capacity or irresponsibility of a totally independent provider of services, without taking into account that it is precisely the proliferation of such providers that will allow the customer, and hence the manufacturer, to discard the less competent and irresponsible as s/he can resort to many others better prepared and more dependable. Antimonopoly lawsuits against certain manufacturers should possibly be resolved not by dismembering, but by obliging them up to a point, to release their source code or to develop more comprehensive maintenance manuals.

But while there are market signs of moving towards an informational maintenance model similar to that of the mechanical maintenance model, we cannot pretend today that informational products are already sufficiently open. Most products are still dependent on their manufacturer. Most users do not have a menu of choices similar to that available to users of mechanical machinery. Manufacturers, intermediaries, and users do not yet have a common culture of realistic acceptance of the inevitable probability of the occurrence of failures (*logical friction*) in informational systems. Neither do they yet understand the necessity to face this problem in a similar way, as does the market in the case of mechanical systems. The purpose of the present article is to foster this line of thinking.

Let us examine the following example. The promoters of the Code of Ethics "Software Engineering Ethics and Professional Practice" [2] include in their motivation the case of a "smart ship" of the American Navy left stranded in 1997 [1]. The problem that the ship faced was due to a programming error. They state that if something similar had happened to a physician (to leave the forceps in the heart of a patient after an operation) or to a civil engineer (to underestimate the quantity of concrete in a dam), s/he would have been imprisoned.

We wonder whether this last analogy would not suit better our purpose than the one we have proposed with mechanical systems. One does not have to be an expert in medicine or engineering to understand that forgetting a forceps in the body of a patient or erroneously calculating the mixture of concrete is not an unavoidable event, mechanically or logically. No physical law leads to it as it does in the case of mechanical wear and tear and no overwhelming complexity justifies the error as it does in the case of *logical friction*. Moreover, both errors can be verified almost immediately, let us say, by X rays in the first example and by an analysis of a sample in the second

case. We raise the question whether, in addition to suggesting a code of ethics that would enable a judge to condemn a programmer for negligence it would be convenient, helpful, necessary, as well as realistic, to coordinate efforts to promote a culture of *logical friction*.

CONCLUSION

We encourage a culture of software maintenance. A culture that espouses the concept of *logical friction* in the same way as mechanics accept mechanical friction. A culture that would reduce the possibilities that a ship on a critical mission would depend on supposedly perfect software. A culture that would require on such a case, the presence of a software maintenance engineer well prepared to handle contingencies the same was as we would certainly require the presence of a mechanical maintenance engineer.

REFERENCES

- [1] Gotterbarn, D. Not all codes are created equal: The software engineering code of ethics, a success story. *Journal of Business Ethics*, 22 (1; 2). 81-89.
- [2] Gotterbarn, D., Miller, K. and Rogerson, S. Software engineering code of ethics is approved. *Communications of the ACM*, 42 (10). 102-107.
- [3] Humphrey, W.S. Bugs or Defects? 1999, 2 (1) http://interactive.sei.cmu.edu/Columns/Watts_New/1999/March/Watts.mar99.pdf.
- [4] Pressman, R.S. *Software engineering : a practitioner's approach*. McGraw-Hill, New York, 1991.

ACKNOWLEDGMENT

We are grateful for the help and suggestions provided by Humberto Bello of Cautus Networks Corporation.

Paper presented at The 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2001) in Orlando, Florida on July 24th, 2001 and included on the Conference Proceedings.