Software is different

Boris Beizer

ANALYSIS, 1232 Glenbrook Road, Huntingdon Valley, PA 19006, USA
E-mail: bbeizer@sprintmail.com, bbeizer@acm.org, bbeizer@ieee.org

The reliability notions that have worked so well for hardware do not work for software. It is not just reliability issues that makes software engineering different than most traditional engineering disciplines, but fundamental, usually unrecognized paradigms. Twelve assumptions that are rarely questioned in traditional engineering fields are explored and each is shown not to hold in software engineering. These differences between software engineering and traditional engineering are often at the core of misunderstandings between their practitioners.

1. The problem

Who will argue that software quality is not a problem? As software users, we do not get the quality we need and, as software developers, we do not deliver the quality users want – and if we do, it is at a higher cost and takes longer than our managers and marketeers would have it. Who, if anybody, is to blame? The usual practice has been to put the blame on software designers, implying that if only they worked more carefully, then these perennial software quality problems would be solved. That may be a popular thing to do, but it will not lead to better software. Blaming programmers has been the prevailing approach for a half century of software development: It has not solved the problem yet, so it is time to look in different directions.

Quality assurance, as a discipline, is deeply rooted in the manufacturing process. The design of a widget, say, is rarely questioned because most design defects have been wrung out before manufacturing begins. If there is a problem, it is usually a production problem. The core of the differences between software quality and widget (traditional) quality is that traditional quality people often have preconceived notions about the world. Although these assumptions are rooted in common sense, they do not apply to software. Users also make the same common-sense assumptions about software. And, unfortunately, many software developers also accept these misconceptions. The consequence is that users, managers, and we who develop or assure software's quality, have inappropriate expectations for it.

2. The fundamentally flawed assumptions

2.1. General

In his classic and, at the time, revolutionary book, *Games People Play* [Berne 1964], Eric Berne talks about what he calls “crossed transactions.” In each of us, Berne says, there are three persons: a parent, an adult, and a child. Therefore any interaction between two individuals has 81 possible variations if we consider all the hidden agendas. Figure 1(a) shows the best kind of open transaction, of adult talking to adult while figure 1(b) shows the worst: what Berne calls a “crossed transaction.”

In useful transaction, both parties are talking to each other, adult-to-adult, without the interfering hidden agendas of either’s internal parent or child. In the crossed transaction they are talking *at* each other. The traditional QA type, who I call Mr. Q.A. Widget, says:

“Why can’t you software types adopt proper methodology and quality assurance methods, such as statistical quality control, in your work?”

An adult-to-child transaction. The hidden assumption in Widget’s criticism is that software types are ignorant, that they do not know that methodology and quality are important, or that there is such a thing as statistical quality control. The software engineer is no better, because she counters:

“You boob! You don’t even begin to understand the issues. Your notion of software is something a high-school kid knocks off in Basic in a few hours. And your notions of ‘proper’ engineering methodologies don’t apply to software!”

They are crossed transactions. Conversation must be adult-to-adult in both directions before communication is possible. We must see where each party is coming from as a step in achieving that objective. Widget QA is rooted in traditional engineering disciplines: old, and for civil engineering, ancient traditions. Ancient traditions that work very well, thank you – for traditional engineering. The underlying assumptions of traditional engineering are part of the fundamental paradigms of Western thought. Software engineers should not blame traditional engineers for unconsciously invoking these paradigms because it takes much effort and brain twisting, as I am sure most

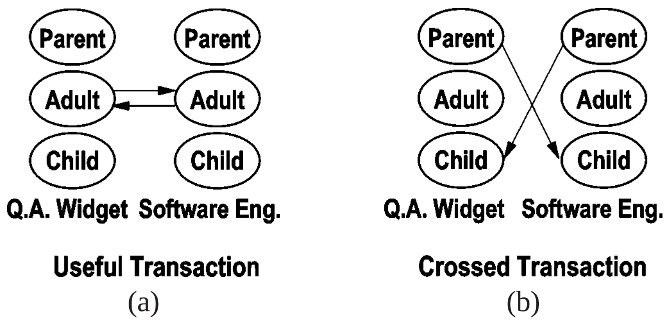

Figure 1. Useful and crossed transactions.

software engineers forget, to get those restrictive ideas out of their head so that they can do good software.

Conversely, the software engineers, who have internalized a new set of paradigms, falsely assume that *any* rational person would understand them. They do not. Most people just do not think digitally. And shouting louder does not get the point across any more than shouting louder helps a non-English speaker understand English (or any other language).

To communicate, we must expose the fundamental assumptions on each side and contrast them: by understanding the differences we break the crossed transactions and engage in real adult-to-adult communications. That is what this essay is about.

2.2. Physical constraints

Whether or not we are physicists, we have an intuitive understanding of the physical world. Apples, when released fall down, not up. Actions are causally related to consequences. We expect things to behave sensibly. Our intuitive notion of what is "sensible" is based on common-sense experiences, learned from earliest childhood, and rooted in the physical world. The software world is neither physical nor constrained by physical laws. It is closer to the psychic world of human behavior than to the physical world. Think of how long it takes to get insights into how people behave: most of us take a lifetime at the attempt and usually do not succeed.

The corollary to assuming that software should behave like physical things is the contradictory notion that software is easy to build and easier to change. Here is the syllogistic logic of this unproductive thinking:

1. If software were decent, it would behave like physical systems and obey physical laws.
2. Software does not behave like physical things. It is not even physical. Therefore, it is indecent.
3. Poured concrete and rolled steel (physical things) are hard to change.
4. Software is easy to build. Any high school kid who knows Basic can do it.
5. Therefore, software is easy to change.

The hardest things to change in this world are people's heads – the way they think. Fifty centuries of religious warfare has taught us that. When such thoughts are encapsulated into text, say in the form of programs and documentation, they become even harder to change. Which is tougher to change: a poured concrete foundation that is wrongly sized or the head of the engineer who sized it? You know the answer: you will not change his head until he sees the wrecking ball taking his mistake apart – and even then you might not succeed. Changing software does not merely mean changing a few lines of code. Before that can happen you must change things in a programmer's head. Contemporary software is not the work of one programmer but of dozens and hundreds: all of *their* heads must be changed before that code can change. The physical world is easy to change by comparison: an enthusiastic guy with a bulldozer can do that.

I hope that in the future we will understand the software world as well as we understand the physical world. Surely there are "laws" that are as binding to software as physical laws are to the physical world. There must be laws that will tell us how immutable our software is at various stages of development just like we can predict the mutability of concrete for any given hour after it is been poured. But we have yet to discover such laws. Most of the differences between software systems and physical systems discussed below devolve from a faulty expectation that software follows the laws of physics. That is the first faulty notion to drop.

2.3. The principle of locality

2.3.1. General

In the physical world, the consequences of an action or a bug are closely related in time, space, and severity to the cause. I call this "The Principle of Locality." In the software world, *the consequences of a bug are arbitrarily related in time, space, and severity to the cause*. Simple bugs in simple software often have localized impact; but as the size and complexity of software increases, the locality principle is increasingly violated.

2.3.2. Space locality

Simple bugs obey the principle of locality and are usually found and eliminated by programmers before they start formal testing. The difficult bugs that survive testing and eventually plague users with misbehavior are not like that. If you have a problem with the windshield wipers of your car, you do not expect it to affect the radio. If you cannot tune your radio to a station, you do not expect your right-rear tire to blow out. Physical systems have space locality, software does not. The use of constructs such as global data (to meet performance demands) exposes software to the possibility of bugs that do not have space locality. The main reason that programmers encapsulate their programs into subroutines is an attempt to promote locality. Contemporary ideas such as object-oriented programming are also attempts to improve locality[1]. Good programmers are, of course, aware of this problem and good programming and testing practices consisting of systematic unit, integration, and then system testing, are in part dictated by the desire to catch bugs in the most local scope possible. To understand software and how it can misbehave, we must change our expectations and accept the fact that *THE SYMPTOMS OF A BUG CAN BE MANIFESTED ARBITRARILY FAR AWAY (in terms of functionality, location, etc.) FROM THE CAUSE*. To have others (e.g., Q.A. Widget, users) understand software quality, they must understand, and more important, accept this fundamental difference.

[1] But part of that package, dynamic binding and inheritance, works the other way and can destroy locality.

2.3.3. Time locality

We have not yet seen bugs that travel in time – although it sometimes seems that way. In the physical world, we expect immediate consequences to bugs. That is in keeping with our common-sense notion of causality. The tire blows out causing the car to lurch to the right immediately. Typically, cause and effect are closely related in time. The physical world *does* surprise us now and then when the consequences of an action are not felt until hours, weeks, or years after the fact. However, this is exceptional, and when it does occur, we usually make a special note of it. In software, the consequences (i.e., the symptoms of a bug) are manifested arbitrarily and unpredictably close or far in time from the cause. In software, this is not the exception, but the rule. While the symptoms of simple bugs *are* closely located in time to the moment at which the faulty code is executed such bugs usually do not survive to a working product. When the symptoms show up mostly results from the way the user employs the software. Not just the general way, but the specific (and unpredictable) sequence of actions executed by the user. No programmer can anticipate all such scenarios: yet bug-free software demands that they do. Good software development practices require that programmers execute specific tests, such as stress and synchronization tests designed to catch such bugs. They also use automated audit tools to detect if any data or program items have been corrupted as a consequence of running the program – because it is most often corrupted data and programs that lie in wait like time bombs with uncertain fuses. So we must again change our expectations and accept the idea that *THE SYMPTOMS OF A BUG CAN BE MANIFESTED ARBITRARILY LONG AFTER THE EXECUTION OF THE FAULTY CODE*.

2.3.4. Consequence locality and proportionality

We expect justice in the world. “Let the punishment fit the crime,” we say. We expect justice in our legal systems when we limit liability to be proportional to damage. Things wear out in the physical world. Knives get gradually dull. It is not as if the knife cuts perfectly today and tomorrow it cannot cut at all. Our tires wear out gradually, so we gradually lose road-handling; not catastrophically and suddenly when the tread depth falls below 4.3 mm. The intuitive notion of wear-and-tear applies to the physical world, at least in its simplest aspects. Many physical situations *do* exhibit disproportionate severity: a threshold is reached and the part no longer functions, resulting in catastrophic failure: but again, these are exceptional. Our notion of the world is analog rather than digital, and in analog systems reactions are generally proportional to action. “To every action there is an equal and opposite reaction.” Ever since Newton said it, we have believed it: not just theoretical physicists, but music teachers, taxi drivers, users, children, and even politicians. It is not true for software.

Justice is not proportional in the software world. Individual bugs do not exhibit consequences in proportion to the error: an error in two bytes does not produce symptoms that are twice as severe or that occur twice as often as those produced by an error in one byte. Furthermore, the same bug, depending on circumstances and the specific history of usage, can have consequences ranging from trivial to catastrophic.[2] There is also no relation between the severity of the symptom and the effort or lines of code that must be changed to correct the bug. All such proportionalities between bugs and symptoms do not exist for software and, indeed, it is likely that all such proportionalities are theoretically impossible.

Adding to the problem is the fact that a given bug may exhibit many different symptoms and a given symptom may be caused by many different bugs. There has never been (except for trivial software) and it is likely that there never will be, any consistent way to map causes onto effect or effect onto causes. *THE CONSEQUENCES OF A BUG (i.e., the symptoms) ARE ARBITRARILY RELATED TO THE CAUSE.*

2.4. Synergistic bugs

When catastrophes occur in the physical world, we want to find a cause – that is consistent with our common-sense notions about the world. If there is a design defect in a product, then someone can be found who made an error. If there is a manufacturing defect in a product, then a tool is maladjusted or a person needs retraining. The idea of personal responsibility is one of the first things we learn as infants and learning that is a basic step in our socialization. And if we are personally responsible, it follows that it is possible to fix responsibility (i.e., blame) for all catastrophes.

Not so with software. While the simplest unit bugs can be localized to an error in a single routine, good unit tests find them long before the product is fielded. Integration bugs are more insidious and difficult: the bugs that result from unpredictable interactions between otherwise correct components. For them, it is not possible to point to a specific line of code and say "That is where the bug is!" and, therefore "The person who wrote that line is to blame." Most integration bugs are distributed. Not one program location is wrong, but several. Furthermore, it is not a set of distributed locations that is wrong because the individual components *are* correct. The bug is synergistic. It is manifested only when two or more components are used together. There are no sure ways to test all such combinations, or even to conceive of them. Good testing does require a systematic approach to finding such bugs and most *can* be eliminated before the software is used: but it is inevitable that some will remain and elude the most careful testing because it is both theoretically and practically impossible to test all possible combinations.

Righteous programmers can do their best and bad things will still happen. Recognizing this possibility is a paradigmic shift, especially for users and the legal system – it is often impossible to blame anyone for a bug.[3]

[2] This is exploited by experienced software testers. An initial symptom might be something innocuous such as a momentary hesitation in the processing or an unexpected flicker of the screen. The tester then varies parameters of the scenario, following a path of ever-increasing severity, to create a new scenario that results in a catastrophic failure.

[3] Which is why I use the word "bug" rather than "fault." In today's big, complex, software, most bugs that survive proper unit testing are not anyone's "fault."

2.5. Complexity

2.5.1. General

The issues of software quality are all about complexity and its management. Software is complicated. Software developers do not make it so: the users' legitimate demands and expectations do. Let us compare software with physical systems. Which is more complex:

1. A word processor or a supertanker?

2. A database management package or a 100-floor building?

3. All the monuments, pyramids, and tombs of Egypt combined or an operating system?

I can think of only two things more complicated than software: aircraft (even excluding their software) and a legal system. 650 megabytes of software can be stuffed into a little 11.5 cm CD-ROM and that is a decent part of a law library. We can measure the complexity of an engineered product by one of two reasonable ways: total engineering labor content or total mass of documentation produced. The supertanker probably represents less than 10 work years of engineering labor content – never mind how many work years it takes to build it physically. The word processor is about 200 work years. A 100-story building is about 30 work years. The data base package is also about 200 work years. Each of those monuments probably took at most a year to design by a master builder and an assistant or two. So we might have a few hundred work years of engineering in the combined Egyptian buildings. Operating system labor contents (including testing) is measured in thousands of work years. Similarly, today it is easy to measure documentation size for general engineering products and for software. Software documentation is measured in gigabytes; most other engineered products in megabytes.

Comparing software to a legal code is more appropriate than comparing it to physical products. Humans have had only stumbling success in crafting their legal codes and have been at it for five thousand years that we know of. Overall, I think that what software engineers have accomplished in $1/100$ of that time is remarkable.

2.5.2. Proportional complexity

If you add an increment of functionality to most physical products, it is done at a proportional increment in complexity. Think of a car or an appliance. More features, higher price. Not just because the vendor can get it, but because there is an underlying proportional cost, and therefore, complexity increase. Complexity is generally additive in the physical world. In software, by contrast, complexity tends to follow a product law. That is, if you have two physical components A and B, with complexity C_{A} and C_{B}, respectively, the complexity of the combination (A+B) is proportional to $C_{\mathrm{A}} + C_{\mathrm{B}}$;

but for software the resulting complexity is likelier to be closer to $C_A \cdot C_B$ or worse![4] How often have you heard "We only added a small feature and the whole thing fell apart?"

Here, if there is blame to spread, I must put it onto software developers, managers, and especially marketeers, who despite years of sad experience continue to ignore this fundamental fact of software life. There is a constant, but ever-unfulfilled, expectation that it is always possible to add another bell, another whistle, another feature, without jeopardizing the integrity of the whole. We cannot do that for buildings even though it probably follows proportional complexity growth. How many floors can you add to an existing building before it exceeds its safety factor and collapses? How much more traffic can you allow a bridge to take before it collapses? It is difficult enough to add incremental complexity to physical products and we realize that ultimately, safety margins will be exceeded. The same applies to software, but because the complexity impact tends to a product or exponential law, the collapse seems unpredictable, catastrophic, and "unjust."

2.5.3. Complexity/functionality inversion

In most physical products, more functionality means more complexity. Add features to a product and there is more for the user to master. There is a direct relation between a product's physical complexity and the operational complexity that the product's users see. Software, by contrast, usually has an inverse relation between the operational complexity the user sees and the internal complexity of the product. That is not unique to software: it is an aspect of most complex products. How easy it is to dial an international telephone call: think of the trillions of electronic components distributed throughout the world that it takes to achieve that operational simplicity. While not unique to software – in software, unlike physical products, this inversion is the rule.

Users of software rightfully demand operational simplicity. Menu-driven software based on a Windows motif is easier to use than command-driven software: so they want windows. I would rather move a document by grabbing it with the mouse and dropping into another directory than type "`MOVE document_name TO target_directory_name`." I remember the bad old days when to get a program to run you had to fill out two dozen job control cards in one of the worst languages ever devised, JCL. Double-clicking an icon is much easier. But what is the cost of this convenience?

My latest word processor catches me when I type "hte" instead of "the," or catches my error when I type "sOftware" instead of "software" – and do not think that getting these deliberate errors to stick was easy! A new graphics package learned the pattern of my editing after a few figures and automatically highlighted the right object on the next slide, saving me a dozen key strokes and mouse motions. And the

[4] That follows from the interaction of features. If each feature interacted with every other feature, only one at a time, then we would expect a square-law complexity growth. If they interacted in all possible ways, the growth law would be exponential.

latest voice-writer software eliminates the keyboard for the fumble-fingered typists of the world. All great stuff! But what are the consequences? Internal complexity!

The increased internal complexity can take several forms:

(a) *Increased code size.* This is the typical form it takes.

(b) *Algorithmic and intellectual complexity.* The code mass can actually decrease, but this is deceiving because code complexity has been traded for intellectual complexity. The resulting software is harder to understand, harder to test, and line-for-line, likelier to have a buggy implementation. Furthermore, not only must the implementation of the algorithm be verified, but the algorithm itself must first be proven – adding yet more opportunities for bugs.

(c) *Architectural complexity.* The best example here is object-oriented software. The individual components can be very simple, but the over-all structure, because of such things as inheritance, dynamic binding, and very rich interactions, is very complex.

Operational convenience in software use are usually bought at the cost of great increases in internal complexity.

2.5.4. Safety limits

It is incredible to me that the notion of safety limits and the uncompromised ethical principle of traditional engineering that such safety limits are never to be exceeded are discarded when it comes to software. The traditional engineer when faced with uncertainty over safety limits has always opted to be conservative. It took decades to gradually reduce the safety limits for iron bridges when metal began to replace stone in the Eighteenth century. It is only through experience that safety margins are reduced. Yet, when it comes to software, perhaps because software has no physical reality, not only are software developers urged to throw traditional engineering caution aside and boldly go where none have gone before, but even the very notion that there might be (as yet unknown) safety limits is discarded. And sadly, all too often, it is an engineering executive trained in traditional engineering that urges that safety limits be discarded.

What are the safety limits for software? I do not know – nobody knows. Nevertheless, we agree that it has something to do with our ability to maintain intellectual control. That, in turn, is intimately tied into complexity and how it grows. One of these days (I hope) we will have “Nakamura’s Law.” This (yet to be discovered law by an as yet unborn author) will tell us how to measure complexity and predict reasonable safety margins for software products. But we do not yet have Nakamura’s Law. So what should we, as responsible engineers do, when faced with a situation in which we do not know how to predict safety margins? Do what our traditional engineering forebears did two centuries ago when they did not know how to calculate safety limits for iron bridges – be very conservative. In sailing, we say that the time to reduce your sails is when the thought first occurs to you – because if you do not shorten your sails then, by the time the wind is really strong and you *must* reduce your sails, the very strength of the wind will make it impossible to do so. *The time at which you* ***have*** *lost intellectual control is the time at which it occurs to you that you might be in danger of doing so.* If you think that it might be too complicated, it is.

“We cannot do that,” the marketeer says. “We’ll lose too much market share to our competitor if we do not add this bell and that whistle!” Back to iron bridges. What will your long-term market share be if half your bridges collapse?

2.6. Composition and decomposition

2.6.1. Composition principle

The composition principle of engineering says that if you know the characteristics of a component, then you can, by applying appropriate rules, calculate the equivalent characteristics of a system constructed from those components. This allows us to deduce the strength of a bridge from the strength of its girders and design without building and testing that bridge. Similarly, the behavior of a circuit can be inferred from the behavior of resistors, transistors, etc. and the circuit design. Nowhere is the principle of composition more important than for reliability. There is a well-proven hardware reliability theory that allows us to predict the reliability of a system from the reliability of its components without actually testing the system: for most complex hardware systems, there would not be enough time in the expected lifetime of the system to do the testing needed to experimentally confirm the reliability. Typically, expected test time to confirm a reliability value is an order of magnitude greater than that value. Thus, to confirm a mean time to failure for an aircraft autopilot of 10,000 years, we need 100,000 years of testing (one autopilot for 100,000 years or 100,000 autopilots for a year). However, because of hardware reliability theory is composable, we do not have to do this. We can get a trustworthy prediction by experimentally testing components and by using analytical models to predict the reliability of the autopilot without running 100,000 years of tests.

Does a similar composition principle hold for software? No! Or if one exists, it has not been found yet. The only way to infer the reliability of a piece of software is to measure it in use (either real or simulated) and there is no known theoretical model that allows one to infer the reliability of a software system from the reliability of its components. It is reasonable for a user to expect that his operating system will not cause unknown data corruption more than once in every ten years. But to assure that to statistically valid certitude would require 100 years of testing; and because of the vast variability of system configurations and user behaviors, what is learned from one case cannot be transferred to another.

So even if we could build bug-free software, we have no present means to confirm if we have or have not achieved our goal, or the quantitative extent to which we have failed to meet the users’ very reasonable quality expectations. This is an area of intensive research, but progress has been slow. The user is driving this. But they cannot have it both ways. They cannot, on the one hand, ask for ever-increasing sophistication and functionality, AND on the other hand, simultaneously not only demand that we maintain the reliability of the program's previous, simpler incarnation, but that we improve the reliability, and furthermore, prove that we have done so.

The above has addressed only the limited objective of reliability determination. But it is not the only composability issue. There are important composability questions for performance, security, accountability, and privacy, to name a few. For more general composability issues, the problem is worse and progress is even more meager. Composability, which is fundamental to traditional engineering disciplines, cannot be assumed for software.

2.6.2. Decomposition principle

"Divide and conquer!" The analysis of complex problems in engineering is simplified by this fundamental strategy: break it down into its components, analyze the components, and then compose the analyses to obtain what you want to know about the entire thing. We take it as given that in traditional engineering, decomposition, and therefore divide-and-conquer, is usually possible. Of course software engineers adopt this strategy to the extent that they can. But unlike traditional engineering, there are not as yet any *formal* decomposition methods. There is the beginnings of such methods, pragmatically useful heuristics, lots of folklore, but nothing rigorous yet. As laudable as hierarchical decomposition and top-down design might be, for example, they are nevertheless heuristic and do not have the mathematically solid foundation that, say, decomposition of a Laplace transforms in circuit theory has.

The biggest trouble is that when it comes to quality issues and bugs, the very notion of decomposition, and even its possibility disappears. This is so because the act of decomposing hides the bug for which we are looking. Two routines, A and B, by themselves work. Even if there is no direct communication between A and B it is possible that the combination does not work. Conversely, two routines A and B are buggy, but one bug corrects the other so that the combination does work. Divide and conquer and decomposition works and is useful for simple unit bugs. But for competent software developers it is rarely the simple unit bug that causes the catastrophic failure in the field.

2.6.3. Composition/decomposition reciprocity – analysis and synthesis

Composition and decomposition are opposite sides of the same engineering coin. Another slice at the same concepts is the idea of analysis versus synthesis. All traditional engineering fields have both an analytical side (tell me what I want to know about the behavior of this thing) and a synthetical side (tell me how to build a thing that will have the specified behavior). Traditional engineering fields alternate between periods of analysis dominance and synthesis dominance. That is, at any given point in time, one or the other dominates the new literature and the emphasis, especially in teaching. I will use electronics as an example. In the 18th century, there was not much to synthesize about electricity, other than to make it happen. Then people such as Franklin started to study it (analysis). The analytical view dominated, culminating in Maxwell's equations, which seemed to explain everything. Then, as electricity became an industry, the focus shifted to synthetical methods – how do I design? During the Second World War, design and synthesis outstripped analysis. It did not matter how a radar-tube (e.g., magnetron) or waveguides worked: we needed working radar, whatever the analytical principle behind it. After the war, the emphasis shifted back to analysis to explain how all those strange devices crafted by trial and error during war worked. Now, in semiconductor circuit design, synthesis appears again to have outstripped analysis, which is playing catch-up because the new synthesis tools depend on it.

The computer industry is only 50 years old. It has (understandably) been dominated by synthesis – how to write working code – albeit guided by heuristics instead of formal synthesis tools. We, speaking for software developers, do not yet have an analytical infrastructure. We are only into the first round of synthesis-analysis alternations and it will take a few more rounds before we know what we are doing. It would be nice if we had a few centuries to learn how to do what we do, but our users will not let us. I do not offer this as an apology, but as an explanation. It is also a matter of setting realistic expectations; for software developers and for users. Users always want magic, so it is about time that we first admit to ourselves that we do not have firm guidance for what we do and perhaps then to our users that there are risks associated with ever-increasing complexity without benefit of either analytical principles or synthesis tools.

2.7. Quality of what?

In traditional engineering, quality is easy to define and measure. Quality metrics fall into two broad categories: structural (tolerances) and behavioral (operational failure rates). Also, there is generally an empirical relation between tolerances (or rather, the lack thereof) and failure rates. It is possible to say that if various parts are built to specified tolerances then it follows that the failure rates will be within specified bounds. The fact that such relations exist (be they developed from theory or determined empirically) is fundamental to statistical quality control of manufactured objects. There is no agreed way to measure software quality and despite close to 30 years of trying, no such way appears to be on the perceptual horizon. Here are some past proposals and what is wrong with them.

1. *Bugs per line of code.* There's no agreement that the most popular size metric, "lines of code," is the best metric to use (see section 2.8 below). Even if we adopted some program size metric such as compiled token count, what has bug density got to do with what the user sees? If most of the bugs are in low execution probability code, then what does it matter if the bug density is high? Unless, of course, it is life-critical software and that low probability code takes care of the one-in-million situation. Then it does matter. Bugs per line of code is a property of the software, but the failure rates the user sees is a property of the way that software is used: so we cannot measure the code's quality is unless we know how

it will be used. All bugs are not equal. Some bugs, or rather their symptoms, are more severe than others. That also depends on expected user behavior. No existing quality measure today takes bug symptom severity into account. Finally, what is a bug? The answer to that one leads to deep ethical and philosophical issues debated for 4,000 years and software engineers and/or quality experts are unlikely to end the debate. So scratch *that* metric.

2. *Defect detection history.* Track the product over a time and note the mean time between successive defect detections. When that time reaches a specified value, declare the software fit for use. That is not even a measure of the software. It is a measure of the stamina, imagination, and intuition of the test group. The defect detection rate could get small because the testers ran out of ideas or are incompetent.
3. *User perceived failure rate.* This is the most promising kind of measure, but it is as much a measure of the user as it is a measure of the software. I almost never use the graphics features of this word processor and I have never used the mini-spreadsheet in it. This word processor supports 34 languages, of which I use only one (American English). Overall, I probably use less than 30% of the features and 98% of what I do depends on only 10% of the features I do use. The very flexibility of software and our ability to pack it with features means that any given user's behavior is unpredictable and therefore, so is any usage-based quality measure.

I could go on, but it would be a redundant recitation of software engineering's state of ignorance and disunity when it comes to how to measure quality. The software quality issue is not that of the quality of a manufactured object whose design is the result of engineering, but of the quality of the engineering process itself. There is no evidence that civil engineers make fewer mistakes than software engineers do. In fact, software engineers probably make fewer mistakes than any other engineering discipline. Ask the critic the next time they decry software engineering's lack of suitable quality metrics, what metric do they use to judge the quality of their *engineering* in contrast to the metric they use for the quality of their products?

It is a fundamentally new problem that has first surfaced in software, but that will undoubtedly become more important in other fields of engineering as the complexity of engineered products inevitably increases. We see this already in aviation. It is well known that contemporary commercial aircraft disasters can rarely be attributed to a single cause, but results from unfortunate conjunction of several causes. For example, the accident is caused by: a failure of component X **AND** abnormal weather **AND** the foreshortening of runway 25 **AND** the loss of the NOTAM that should have warned the pilot of that fact **AND** . . . FAA accident reports are instructive reading because each factor must be examined and a recommendation for its prevention made. Imagine if we had to do a bug postmortem like FAA accident investigations and distribute specific recommendations *for that one bug* to all concerned programmers? Do it for every bug found? How much software would then get produced?

2.8. Quantifiability

Quantification in engineering is generally attributed to Galileo, although the Egyptians, the Mayans, the Romans and later the Arabs were darn quantitative centuries before. Whatever the genesis of quantified engineering, it has been a fundamental part of the engineering paradigm for at least four centuries. It is *assumed* in traditional engineering fields that anything of interest can be quantified – that is, reduced to numbers; and if it cannot be quantified, it is not engineering [Gilb 1995].

Not a bad assumption if it has been true for several centuries and has always served engineers well in the past. But it is merely an *assumption* – a cherished belief, a pragmatic observation – but not an immutable fact. There is no evidence that this *assumption* of quantifiability applies to software at all. And there is considerable evidence that it does *not* apply.

There are many other kinds of formal structures that cannot be quantified in the ordinary sense of simple numbers, or even vectors of numbers. For example: partly ordered sets, general relations, graphs. Quantification implies comparison (e.g., either $A > B$, $B > A$ or $A = B$): furthermore, in most engineering, quantification means strict numerical comparison. But some things just do not compare that way. The general rule is partial-, rather than strict-ordering. There are many (infinite) ways to order things and the strict ordering of traditional engineering quantification is merely the oldest and the simplest. Furthermore, our understanding of structures in computer science makes it clear that we cannot willy-nilly assume that strict ordering applies. While it is always possible to tag numbers onto partially ordered structures (e.g., leaf count, node count, depth, etc.) such numbers may not capture what it is we want to capture by the use of numbers, no more than "lines of code" captures what we mean by "complexity."

The fact that there is a huge literature on software metrics and that software developers gather a lot of numbers about their products, does not mean that such metrics are fundamentally correct, accepted, or even useful. There is a lot of ongoing research. There have been many attempts to establish axiomatic foundations for software metrics; but none are without flaws and without controversy. The state of the software metrics art is at best in the pre-Galilean stage. What metrics there are, do not scale, do not reliably transfer from one project to the next, do not compose, to mention only a little of the ongoing controversy. Furthermore, it may be that any notion of software metrics as we know them, are fundamentally flawed [Bauer 1995]. At best, the options are uncomfortable: what design restrictions must be adopted so as to make software quantifiable (an as yet unanswered, indeed, almost completely uninvestigated question) and at worst, drop the entire notion of quantification for software and replace it with something else – also as yet undiscovered. To promote the idea that quantification of software at present has the same solidity as quantification in traditional engineering is a distortion of the facts, misleading, and potentially dangerous.

Let us leave the formal math issues aside and restate it this way. To insist on strictly ordered quantification (i.e., ordinary numbers) is to eliminate from consideration most ways of measuring things in computer science. The *assumption* that interesting and important aspects of software can always be represented by numbers (traditional quantification) gets in the way of developing the kind of quantification appropriate to software, if any such quantification exists – and it may be that *any* notion of quantification will be incorrect. This is the second hardest paradigm shift of all.

2.9. Adequate knowledge

The most difficult paradigm shift of all is letting go of the notion that we can have adequate knowledge. That too is an *assumption*. The Eighteenth Century Rationalist [Ferm 1945] model of the universe, championed most eloquently by Rene Descarte, is with us yet. The Rationalist model holds that the universe is a giant clockwork whose gears and pinions are the physical laws. If you make enough measurements of enough things, then in principle everything is predictable. That is, it is always possible to get the information you need for any purpose whatsoever – whether it is worth doing is another issue that we will not discuss here – but it is, in principle, always possible.

Old Rene did not have to deal with the Heisenberg uncertainty principle; with Godel's Theorem; with chaos. The Heisenberg uncertainty principle tells us that you cannot make such measurements even if you wanted to. Godel's theorem tells us that if you had the measurements, you might not be able to do the calculation (or know that you had finished the calculation). Chaos tells us that even insignificant errors or variations in our measurements could lead to arbitrarily divergent predictions. The theoretical ideal of adequate knowledge is based on very shifty foundations. It applies on the common scale of physical objects: but not for the very small (atoms and particles), not for the very big (the universe), and not for the very complex (software).

The above *merely* fundamental problems aside, software has an uncertainty principle of its own. We cannot predict how the changes in our software will change the users' behavior. Sometimes we get complacent and think that we can. A few years ago, who would have questioned the ultimate, ongoing, perpetual dominance of the software industry by Microsoft? I did not. Did you? Yet, today, the arena has shifted to the Internet and Microsoft looks more like a frightened elephant besieged by a pride of hungry lions than Godzilla.

That is a dramatic example, but it happens daily on a smaller scale. We change our software in response to perceived market demands, which in turn affects the market and the users' behavior, making any quantification based in whole or in part on user behavior next to useless.

That the users' behavior is unknowable is not the only barrier to adequate knowledge. There are many others, of which combinatorial explosion is perhaps the biggest. As Danny Faught [Faught 1996] likes to quote in his Internet signature: "Everything is deeply intertwingled." The 'intertwingling," meaning the combinatorially intractable potential interaction of everything in software with everything, provides a practical barrier (even with big computational iron) to what could be analyzed even if all the facts were known and none of the fundamental problems discussed above existed.

3. What we can we do about it

3.1. *The world in which we live*

All of us together are unlikely to solve the open problems of software engineering discussed above. We cannot wait for the unborn Nakamuras to publish their Five Laws of Software Engineering. We cannot call for a moratorium on software development (civil engineers tried that in the early 19th century when iron bridges collapsed all over the landscape, to no avail). Those are things we cannot do. We *can* start to change our own paradigms, then our managements' paradigms, and eventually, our users' expectations: I have no hope of ever changing the marketeers' way of thinking. Here are some of the ways in which we could do that.

3.2. *The development option*

We have tried, for fifty years now, with no notable success, to bring rationality to software engineering. We have made progress alright, but so have user demands. All of our productivity increases and methodology improvements have just barely kept pace with the ever-increasing complexity of the software we produce. When you couple that with justified but continually rising user dependability expectations and the increasingly lower technical level of our users (also expected), we have been, and continue to fall behind. At some point we must ask if the old way is the right way. Can we realistically expect Nakamura to come out with the theorem that solves everything or Smith with the tool the fixes everything up? I think not.

The bright light in the history of software engineering has been the recognition that we must forego the fully generalized potential of software and adopt design restrictions that makes it possible for us to understand our software. We have done it in the past; here are a few such restrictions:

1. Structured programming.
2. Strong typing and user defined semantic types.
3. Avoidance of global data.
4. Adoption of style rules and use of style checkers.
5. Encapsulation.

Each of the above are restrictions on the way software is written. Each exacts a toll in terms of execution time, program space, and most important of all, personal ego. We can probably add ten more to the above list of restrictions that are acceptable today. It is not whether this is the right list or the ultimate list, but that such a list exists and that programmers are willing to back down from the total freedom (actually, chaos) of four and five decades ago. The list will expand, of course, but will this do the job? No! Because a list of pragmatic design restrictions avoids the questions of deriving design principles from fundamentals (e.g., axiomatically). There has been no active search for the right design restrictions. People, programmers and researchers, have been saying "If we adopt design restriction X, then there will be fewer bugs." That is doing it backwards. The question should be "What are the design restrictions we need to make software constructable, testable, and safe?"

3.3. Do paradigm checks

When communications seems to be stalled and you are talking at cross-purposes in a crossed transaction, run down a checklist of paradigms. Ask "Are you assuming":

1. That software is easy to change?
2. Bug space locality?
3. Bug time locality?
4. Proportionality between bug and consequences?
5. Bug independence?
6. Proportional complexity growth?
7. Safety limit knowledge?
8. Composability?
9. Decomposability?
10. Agreed quality measures?
11. Quantifiability?
12. Knowledgeability?

If you are yourself assuming such things, then you should quickly come to terms with software reality. Until your own basic assumptions change, you are unlikely to change anyone else's.

3.4. Restructure priorities

The software development priority list that has dominated the industry from the beginning is:

1. Develop it as fast as possible.
2. Make it run as fast as possible.
3. Build it as tight as possible.
4. Put in as many features as you can.
5. Do it at the lowest possible cost.
6. Worry later. Bugs will get fixed.

These priorities do not even make my list. Here is an alternative set of priorities more in keeping with what we can and cannot do in software development. While one might argue that these new priorities apply only to life-critical software, I argue that they should take priority for all software because together, they are only a more formal way of saying "do we know what we are doing?" And we should be able to honestly say that we know what we are doing before we concern ourselves with development time, performance, size, and cost.

1. Can it be analyzed? Is its behavior predictable?
2. Can it be tested?
3. Does it have a composable model?
4. Does it work?
5. Have feature, component, and data interactions been reduced to the absolute minimum?
6. Does it have the features the users need (as contrasted to want)?

3.5. Public honesty about our ignorance

Speaking from the perspective of a member of the software industry, I say that we are a profoundly dishonest bunch. We software types lie to ourselves about what we can and cannot do, we lie to our managers about how long it will take to do it, they tell the marketeers that the delivery schedule (which was more the product of martinis than rational thought) they want is what they will get. Among ourselves we may hint that we do not know how to make something work, but we will reassure the public that it will work (somehow? somewhere? somewhen?) If users have unrealistic expectations then we are to blame. And therefore, it is up to us to educate them so that their expectations are aligned with our current abilities instead of our aspirations.

References

Bauer, R., Unpublished, privately circulated research results that claim to prove that no set of software metrics axioms can be consistent with measure theory.
Berne, E. (1964), *Games People Play*, Grove Press, New York.
Faught, D. (1996), Quote in Email signature attributed to Ted Nelson.
Ferm, V. (1945), *An Encyclopedia of Religion*, Philosophical Library, New York.
Gilb, T. (1995), Private correspondence.