About Quality

Quality is an elusive term. And it is especially elusive in the software field. The elusiveness of the term is the primary focus of that wonderful book *Zen and the Art of Motorcycle Maintenance* (Pirsig 1974). The main character of the book, an academic looking into the real meaning of the word, went mad seeking a workable definition!

No matter how smugly we look at that madness—certainly none of us would ever go mad about this—the fact of the matter is that we in software have done very little better. Oh, we do an “I’ll know it when I see it” kind of thing about quality. But the fact is, we neither agree on a workable definition nor agree on whose responsibility quality in the software product is. And even if we could agree on a definition, we have yet to figure out how to measure how much quality we have achieved in any given software product. Let’s take each of these considerations in turn.

No workable definition? There is enormous disagreement in the field about what quality is. Worse yet, there are people in the field who believe there is no disagreement but who, in fact, are supporting a definition that is flat wrong. In this book, I present the definition that I prefer in Fact 46 (quality is a set of seven attributes) and then list the definitions that I feel are wrong in Fact 47. Warning: You may not agree with my positions on this.

Whose responsibility is quality? Most of the books and courses on software quality take the position that achieving quality is a management task. But if you could peek ahead at my definition of quality, one based on the attributes that make up quality, you would find some highly technical things. Modifiability, one of those attributes, is a matter of knowing how to build software in such a way that it
can be easily modified. Reliability is about building software in ways that minimize the chance of it containing errors and then following that up with an error removal process that uses as many of the multifaceted error removal options as makes sense. Portability is about building software that can easily be moved from one platform to another. These and all the other "-ilities" of quality are deeply technical—they require a heavy dose of technical knowledge to carry off successfully. I would assert that, because of all of this, quality is one of the most deeply technical issues in the software field. Management's job, far from taking responsibility for achieving quality, is to facilitate and enable technical people and then get out of their way.

Why can't we measure quality? Because not only is quality itself elusive, but those attributes from Fact 46 that make it up are all too elusive, too. It is nearly impossible to put a number on understandability or modifiability or testability or most of the other quality -ilities. The fact that we can put numbers to reliability, and to some extent efficiency, does not change the fact that the slope leading to measurable quality is pretty slippery. Some years ago, the U.S. Department of Defense funded a study of measuring the -ilities (Bowen, Wigle, and Tsai 1985). The resulting three-volume report contained a lot of worksheets to be filled out (they took 4 to 25 hours apiece) and checklists to be followed. Unfortunately, when the smoke of doing all of that had cleared away, you weren't much closer to quantifying quality than you were when you started the exercise.

So, where are we going in this material on quality?

- I try to set the record right on what quality is and what it isn't.
- I take an overview of some aspects of reliability, like the characteristics of errors and their makers. In particular, I revisit some spin-offs from the earlier facts about the error removal phase of the life cycle, raising some of those spin-offs to the level of facts in their own right.
- I take an overview of some aspects of efficiency. When efficiency is important, we will see in the facts that follow, it is really important. There are some decades-old lessons about efficiency that deserve to be raised to the level of facts here.
- You sharp-eyed readers may note that, of the seven -ilities, I have split out only two of them (reliability and efficiency) for further emphasis. That should not lessen your view of those other -ilities. It's just that there are significant facts about these two that are both fundamentally important and oft-forgotten (that is, after all, the theme of this book).
Sources

In addition to the two sources listed in the References section that follows, see


References


QUALITY

Fact 46: Quality is a collection of attributes.

Discussion

There are a lot of different ways of defining software quality. Here I want to present the definition that’s stood the longest test of time.

Quality in the software field is about a collection of seven attributes that a quality software product should have: portability, reliability, efficiency, usability (human engineering), testability, understandability, and modifiability. Various software people provide somewhat different sets of names for those attributes, but this list is pretty generally accepted and has been for almost 30-something years.

What are those attributes about?

1. Portability is about creating a software product that is easily moved to another platform.
2. Reliability is about a software product that does what it’s supposed to do, dependably.
3. Efficiency is about a software product that economizes on both running time and space consumption.

4. Human engineering (also known as usability) is about a software product that is easy and comfortable to use.

5. Testability is about a software product that is easy to test.

6. Understandability is about a software product that is easy for a maintainer to comprehend.

7. Modifiability is about a software product that is easy for a maintainer to change.

I have not presented these quality attributes in any kind of prioritized order. In fact, it is not possible to do that in any meaningful way. That is, there is no general, correct order in which one should try to achieve the software quality attributes. However, that is not to say that the attributes should not be ordered. For any one project, it is vitally important to establish a prioritized list of these attributes from the outset. For example, if a product is to be built for a marketplace in which it is to run on many platforms, then portability should be at or near the top of the list. If lives depend on the successful operation of a software-controlled product, then reliability must be at the top of the list. If a product is expected to have a long, useful life, then quite likely the maintenance attributes—understandability and modifiability—need to be at or near the top of the list (it is interesting to note that two of these seven attributes are explicitly about maintenance). If a product runs in a resource-starved environment, then efficiency likely belongs at the top of the list.

For example, a common prioritizing for an average project could be as follows:

1. Reliability (if a product doesn’t work right, it doesn’t matter much about the other attributes)

2. Human engineering (the heavy emphasis on GUIs these days speaks volumes about the importance of usability)

3. Understandability and modifiability (any product worth its software salt is probably going to be maintained for a long period of time)
4. Efficiency (I'm a little embarrassed at how low I've placed this; for some applications, this will be number 1)

5. Testability (coming in next to last doesn't diminish the importance of the attribute that can lead us most directly to reliability, which I have placed at number 1)

6. Portability (for many products, portability doesn't matter at all; for others, it probably belongs at the top of the list)

Don't be surprised if my sample ordering doesn't match yours. When I first wrote my book on software quality (Glass 1992), one of the reviewers for that book kept trying to change the (arbitrary) ordering I had used; he wanted it to match his own prioritized set of beliefs (which, incidentally, was very different from my own). I think trying to make a generalized ordering of the quality-ilities is something like creating a good software design: If any two software people agree, they probably constitute a majority.

Controversy

There are several controversies about this fact, emerging from the following questions:

1. Is this the right definition of quality?

2. Is this the right list of attributes?

3. Is there a correct ordering of these attributes?

With respect to controversy 1, there are many software folks (including some experts) in the "this is the wrong definition" camp. Most of those people use a definition among the ones I present in Fact 47. When I discuss that fact, I will tell you why I think these people are simply wrong.

With respect to controversy 2, there are software folks who take issue with the attributes in my list. One software expert, for example, argues strongly against the inclusion of portability on the grounds that if you look at the attributes of quality in other product fields, this one isn't among them. That's an interesting and, I would assert, erroneous argument. Quality attributes shouldn't be expected to be field-independent any more than their ordering should be project-independent. For example, an important quality attribute for automobiles is "fit and finish."
There are many products, including software, for which that has no meaning. There are other people who simply offer different names for the attributes in the list I have provided. On that matter, I feel much less strongly. I don’t care what you call these things, as long as the concepts they represent are included in your definition of software quality.

With respect to controversy 3, we have already discussed that. There is no correct, generalized ordering, I would assert, and arguing about one is vaguely reminiscent of arguing about how many angels can dance on the head of a pin.

Sources

The earliest and best known instance of this particular attribute-based definition of the quality attributes is found in the work of Barry Boehm.


My favorite elaboration on Boehm’s work is found in the work cited in the Reference section that follows. An even (slightly) earlier instance of an attribute-based definition of quality is found in


Reference


**Fact 47**

Quality is not user satisfaction, meeting requirements, meeting cost and schedule targets, or reliability.

Discussion

So many different definitions are offered for the meaning of the word *quality* in *software quality*, that sometimes I despair. The reason I despair is that so many of those alternative definitions are clearly wrong, but their advocates clearly believe they are right.

This fact lists four of those alternate definitions. Each of them is quite appealing; each is about something important to the software field. But none of them, I would assert, is a correct definition of quality.
For a long time, I believed that something was wrong with those other definitions, but I couldn't put my finger on what that was. It wasn't until I heard a speaker from Computer Sciences Corp. provide what he called the relationship between all of these terms that I was able to point to something that clearly demonstrated that those other definitions were wrong. Here is the relationship that speaker provided:

User satisfaction = Meets requirements + delivered when needed + appropriate cost + quality product

This is a nicely intuitive definition of user satisfaction. A user will be satisfied if he gets a product that meets needs, is available at an appropriate time, doesn't cost a fortune, and is of reasonable quality. But what it shows, if you analyze it properly, is that all of the important entities in the relationship are distinguishable from one another. Note that one of those entities is quality. That says, I would strongly assert, that quality is clearly different from all of those other things.

Note that all of these entities are really important ones. Saying that quality is not the same as these other entities does not diminish their importance. It merely says that quality is about something else entirely. Meeting requirements and achieving schedule and cost targets are vitally important, but they are not about quality. User satisfaction is about quality, but it is also about some other pretty important things as well.

That takes care of most of the things that quality is not, in this fact. One remains, however—reliability. Many software experts equate software quality to the existence, or lack thereof, of errors in the software product. But, as we can see from Fact 46, quality is certainly about errors—that's what "reliability" is all about—but it's also about so much more.

Those experts who equate quality with lack of errors often know better. Right after a discussion in which they acknowledge all of the other attributes of quality, you may find them proceeding to discuss errors as if that were the only thing quality is about. Equating quality to reliability is seductive, since reliability is such an important focus of software quality (I grudgingly put it at number 1 in Fact 46), but doing so neglects some pretty darned important other attributes.

**Controversy**

This fact is a controversy all by itself! You will continue to hear people discuss quality as if it were user satisfaction or meeting requirements or achieving estimates (how that one, which I believe has nothing whatsoever to do with quality, ever got into this list, I have no idea) or being reliable. And those discussions will
be supported by intense conviction. That does not diminish the fact that they are wrong.

Source
I prefer not to provide sources for these alternate and erroneous views of software quality since (a) in doing so I would be reinforcing those erroneous beliefs, and (b) I would have to confront the beliefs of some people whose names you would likely recognize. The important thing here is for all of us to accept the correct definition of quality and leave the erroneous ones behind. Until we do that, it will be difficult to discuss the subject of software quality in any meaningful way.

RELIABILITY

Fact 48  There are errors that most programmers tend to make.

Discussion
It should probably come as no surprise that some kinds of software errors are more common than others. Anyone who has ever participated in a code inspection will probably remember someone saying something like, "oh-oh, it's another one of those (brand-X) errors." The fact of the matter is, human beings are susceptible to doing certain kinds of things wrong. Off-by-one indexing, Definition/reference inconsistency. Omitting deep design details. Failing to initialize a commonly used variable. Neglecting one condition in a set of conditions.

What is odd is that few researchers have explored this matter. My source for this particular fact, in addition to my own experiences, is a paper published by a German researcher that was presented at, and published in the proceedings of, a little-known conference in Bremerhaven, Germany (Gramms 1987). That researcher called these errors "biased errors" and said they resulted from "thinking traps." This is odd because you would think that these biased errors would be among the ones that error removal techniques would focus on. Inspections, for example, could include them in their checklists. Tools to isolate and identify them could be built. Tests would be set up explicitly to trap them. Researchers could study additional ways of avoiding or detecting them.
There is another interesting facet of these biased or common errors. Among the concepts included in fault-tolerant programming (building software that tries to trap its own errors as they occur) is something called N-version programming. N-version is based on the idea that N diverse solutions to a problem from N separate teams of programmers will be unlikely to replicate the same errors; these diverse solutions, operating in conjunction with one another, could identify and vote out as erroneous any results from one solution that did not match the results of the others. But the fact of biased errors suggests that more than one of those N versions might indeed contain the same error, thus negating some of the power of the N-version approach. (This is a problem to those software development communities in which ultrareliable solutions to critical problems are essential—aerospace and rail systems, for example.)

Controversy

Although the phenomenon inherent in this fact would surprise few in the software field, it is little acknowledged—and therefore not the cause of any controversy.

Source

I know only about the German Computing Society conference (cited in the Reference section that follows) because I spoke at that same conference. My own presentation, into which I incorporated some of Gramms’s findings, was


Reference

Fact 49  Errors tend to cluster.

Discussion
Try on for size this collection of statements about where software errors are found.

- “Half the errors are found in 15% of the modules” (Davis 1995, quoting Endres 1975).
- 80% of all errors are found in just 2% (sic) of the modules” (Davis 1995, quoting Weinberg 1992). Given the quote that follows, it makes you wonder if 2 percent was a misprint.)
- “About 80% of the defects come from 20% of the modules, and about half the modules are error free” (Boehm and Basili 2001).

Whatever the actual numbers, it is obvious that errors tend to be found in clusters in software products. Note that people have known this fact for several decades—the Endres quote is from 1975.

Why would this clustering of errors be so? Could it be that some parts of a program are considerably more complex than others and that complexity leads to errors? (That’s my belief.) Could it be that the coding of programs is often divided up among programmers, and some programmers tend to make more errors, or discover fewer of them, than others? (That’s certainly possible, given the individual differences we mentioned in Fact 2.)

What’s the message of this particular fact? If you find a larger than expected number of errors in some program module, keep looking. There are quite likely to be even more there.

Controversy

The data here is sufficiently clear and of sufficient long standing that there is no controversy over this fact that I am aware of.

Sources

The sources supporting this fact are listed in the References section that follows.
References


Fact 50

There is no single best approach to software error removal.

Discussion

Talk about getting redundant! I made this point several times, several facts ago. The reason I repeat it here is that it deserves to be a fact all its own, not a part of some other fact.

What’s the underlying meaning of this fact? That there is no error removal silver bullet. That there is unlikely ever to be one. That testing, of whatever flavors, is not enough. That inspection and review, no matter whose definition, is not enough. That proof of correctness, if you believe in that sort of thing, is not enough. That fault tolerance, for all its value in critical software, is not enough. That whatever your favorite error removal technique, that’s not enough either.

Controversy

The controversy here comes largely from the hypesters. The advocates of silver bullets will continue to make exaggerated claims for whatever technique they are selling, including error removal approaches. The fact that these advocates are consistently wrong will not keep them from stirring up these same fires in the future.

Source

This fact is one of the main themes of

Fact 51 Residual errors will always persist. The goal should be to minimize or eliminate severe errors.

Discussion
More redundancy! But, once again, I repeat this fact because its deserves to be a fact of its own, not piggybacking on other facts that lead us to this one.

There will always be residual defects in software, after even the most rigorous of error removal processes. The goal is to minimize the number and especially the severity of those residual defects.

When the subject of software errors arises, it is vitally important to introduce into that discussion the notion of error severity. Severe errors must be eliminated from software products. It would be nice to remove all those other errors too (for example, documentation errors, redundant code errors, unreachable path errors, errors in numerically insignificant portions of an algorithm, and so on.), but it's not always necessary.

Controversy
There is no controversy about whether there are common residual errors in software products. But there is huge controversy about whether that must remain true. Realists (some would call them pessimists or even apologists) believe that the situation won't change (because of all that complexity stuff we've been talking about). Optimists (some would call them dreamers or even advocates) believe that error-free software is within our grasp, given a sufficiently disciplined process.

One recent study (Smidts, Huang, and Widmaier 2002) casts important light on this issue. Two practitioner teams using very different software development approaches (one traditional, at CMM level 4, and one avant garde, using formal methods) were unable to build a fairly simple product that met a required goal of 98 percent reliability (even though both teams worked to “generous” cost and schedule constraints).

By now, you have probably made your own choice as to which side of this controversy you are on. I won't belabor the point further.

Source
This fact is another of the main themes of

Following are some interesting quotes that touch on this matter. About residual errors:

- “Disciplined personal practices can reduce defect introduction by up to 75%” (Boehm and Basili 2001).
- “About 40-50% of user programs contain non-trivial defects” (Boehm and Basili 2001). (Note that this quote is also about defect severity.)
- “You will not find all the bugs” (Kaner, Bach, and Pettichord 2002).

About severity:

- “Almost 90% of the downtime comes from, at most, 10% of the defects” (Boehm and Basili 2001).

References


EFFICIENCY

Fact 52 Efficiency stems more from good design than from good coding.

Discussion

Ever-optimistic programmers have generally believed, over the years, that they can code their way to an efficient product. That’s why assembler language, for example, has persisted so long in the history of the field. We deal with the issue of assembler language in Fact 53. Here, I want to gore the sacred cow of code over design.

For this fact to make sense, you have to think about where inefficiency comes from in the typical software product. Among other places, inefficiency emerges from external input/output (I/O) (for example, slow data accesses), clumsy interfaces (for
example, unnecessary or remote procedure calls), and internal time wastage (for example, logic loops going nowhere).

Let's tackle I/O inefficiencies first. Computers are infinitely slower in fetching and replacing data on external devices than on anything else they do. Thus a penny saved in designing I/O manipulation is a dollar earned in application speed-up. There are lots of data format choices, and which you choose determines the efficiency of the resulting program more than any other choice you make. I used to try to convince academic computer scientists that the main reason for their bread and butter courses in data and file structures is to learn about which of those approaches are most efficient for which kinds of applications. After all, given the simplicity of sequential and even hashed data structures, why would anyone have ever invented linked lists and trees and indexing and the like? And why would we have data caching, a logic inefficiency solely introduced for increasing data efficiency? Because, I would tell them, data structures represent a tradeoff between increasing data structure logic complexity and improving the efficiency of data access. (Some computer scientists, convinced that efficiency is yesterday's problem, see data structures as being about interesting options for data arrangement and nothing more.)

So at design time we go to great lengths to choose just the right data structure. Or file structure. Or database approach. Why waste a lot of energy prematurely coding an inappropriate data access scheme?

Interface and internal inefficiencies pale to insignificance compared to I/O inefficiencies. Still, it is possible, through poorly designed looping strategies, for a coder to make a program's logic wheels spin an inordinately long time. (There are even "infinite loops" that never end.) Perhaps the worst offender is mathematical iterative algorithmic approaches. Slow or nonexistent algorithmic convergence can waste whole bunches of computer time. Close behind is inefficient data structure access (data does not have to be on external storage for its access to be problematic.) Once again, a little design time consideration of an efficient algorithm can be far more effective than a slickly efficient coding solution.

So what's the bottom line here? If there is a need for efficiency in a project, it must be considered early in the life cycle—well, prior to the beginning of coding.

Controversy

To some eager programmers, coding is the most important task of software construction, and the sooner we get to it, the better. Design, to people in that camp, is simply something that puts off the ultimate problem solution activity.

As long as these people continue to address relatively simple problems, (a) they will probably never be convinced otherwise, and (b) there may be nothing
terribly wrong with what they are doing. But it doesn’t take much problem complexity before that minimal-design, quick-to-code approach begins to fall apart. (Recall Fact 21, about how quickly problem complexity drives up solution complexity?)

What I am saying is that most of the controversy about this particular fact is between those who see design as being of little value and those who see it as an essential prelude to coding. The Extreme Programming movement, which advocates simple design approaches that evolve quickly into coding, is undoubtedly refueling this particular controversy. So is Extreme Programming’s emphasis on ongoing “refactoring” to fix inefficiencies and errors in the design after it is coded.

**Sources**

This is another one of those facts that has been known for so long that it is nearly impossible to trace it back to some kind of published roots. Any software engineering textbook (and those have been around now for 30-something years) will make this point.

To understand the argument for rushing design to get to code and refactoring it later to repair errors made in that rush to code, see the following Extreme Programming literature:

- Beck, Kent. 2000. *Extreme Programming Explained*. Boston: Addison-Wesley. See the material on “simple design” and “refactoring.”

The most thorough discussion of refactoring can be found in


**Fact 53**

High-order language (HOL) code, with appropriate compiler optimizations, can be about 90 percent as efficient as the comparable assembler code. Or even higher, for some complex modern architectures.

**Discussion**

The debate about high-order language (HOL) versus assembler language is ages old in the computing field. Devotees of each have been known to go to extreme lengths to try to get their point of view to prevail on a particular project. The data needed to settle that debate has been known almost as long. The sources that follow include studies dating back to the 1970s. So although the HOL versus assembler debate feels as fresh as the edutainment application domain, where today’s
most die-hard assembler supporters seem to reside, the wisdom of the software ages is probably sufficient to make that debate go away.

The definitive data emerged from a flurry of studies conducted in the mid-1970s. The reason for that flurry was that the debate had become a critical issue in an emerging application domain, Avionics software (software to control the electronics of air- and spacecraft). And the findings of that flurry, nicely summarized in Rubey (1978), were “The reported . . . inefficiency of HOLs in avionics applications has been between 10 and 20% in nearly all reports.” (That study went on to note that optimizations provided by optimizing compilers could add at least 10 percent to code efficiencies and that tweaking an HOL program after it is written, which is easy compared to tweaking assembler language, can add another 2 to 5 percent.)

So why has the debate raged on throughout the intervening decades? Because, in spite of that data and because of the undoubted advantages of HOL for most coding, there are times when assembler really is the better choice. In other words, the advantages of HOL are highly task-dependent; some tasks are much harder to code efficiently in HOL than others.

What are the HOL advantages? Today, naming them seems almost unnecessary since they are generally known and accepted, but here goes anyway.

- It takes far fewer lines of HOL than assembler code to solve a problem; thus productivity is dramatically higher.
- HOL code finesses whole error-prone areas of difficulty, such as register manipulation.
- HOL code can be portable; in general, assembler cannot.
- HOL code is more easily maintained (understood and modified).
- HOL code can be more readily tweaked to increase its efficiency where needed.
- HOL code can be written by less-skilled programmers.
- HOL compilers can optimize use of modern architectures, such as pipelined or cached.

What are the assembler advantages?

- HOL statements do not necessarily track with hardware features. Some assembler code can take advantage of what the hardware provides and be simpler than its HOL equivalent.
• Similarly, HOL can be awkward for interfacing with assembler-focused operating system features.

• Tight space and time considerations sometimes dictate the ultimate in efficient solutions.

Regarding assembler advantages, a nice study of the use of assembler in a tightly constrained application can be found in Prentiss (1977). On this project, the original solution was coded entirely in an HOL, and then—following a study of resulting efficiency problems—20 percent of it was recoded in assembler. That 80/20 ratio quite likely represents the maximum amount of assembler code that any system, even today, should need.

Controversy

Sometimes it feels as if this controversy may never go away! Assembler is sort of seductive technology—what "real programmer" doesn’t want to get truly intimate with his or her computer and operating system? But, in fact, the controversy was really pretty well resolved back in the 1970s. The problem is, most of today’s programmers aren’t aware of yesterday’s Avionics studies.

Sources

The sources used for this fact are listed in the following References section.

References


Fact 54
There are tradeoffs between size and time optimization. Often, improving one degrades the other.

Discussion

Somehow it seems like anything that makes a program more time-efficient will also make it more size-efficient. But that is not true.

For example, consider something as trivial as a trigonometric function. Most trig functions are coded as algorithms; their code consumes very little space, but their iterative nature means that they will take some time to execute. An alternative way of providing trig functions is to build into the program a table of values and then simply interpolate among them to get a result. Interpolation is far faster than iteration but a table will be far larger than that iterative code. (This solution was actually employed on a space system I was involved with, where timing mattered a whole lot and size didn’t.)

As another, perhaps more contemporary example, consider the Java programming system. Java code is not compiled into machine code; instead, it is represented in the computer as something called byte code. Byte code is much more compact than the equivalent machine code, and, as a result, Java programs tend to be very size-efficient. But time-efficient? No way! That byte code, because it is not machine code, must be interpreted while the program is executing, and interpretation sometimes adds a factor of 100 to execution time.

Thus the search for efficiency is one of compromise. The number of time efficiencies that also result in size efficiency are few in number. (An interesting point, made by Rubey (1978) in the context of HOL efficiency studies, is that size efficiency is much more easily measured than time efficiency. The former, in general, can be measured statically, while the latter must be measured dynamically. Thus, for better or for worse, it is sometimes easier to provide size efficiency than time efficiency.)

The bottom line here? If you’re seeking that quality attribute called efficiency, be sure to keep in mind what kind of efficiency you most care about.

Controversy

This is simply a seldom-articulated fact, with very little controversy attached to it. Most people who care about efficiency already know about it; those who don’t, however, can make some grievous errors in failing to consider it.
Source

The source for this fact is listed in the following Reference section.

Reference