What is Software Quality?

Peter J. Denning

A Commentary from Communications of ACM, January 1992.

Quality and dependability of large software systems are among the most common topics in discussions about computers. Organizations, both public and private, have invested huge sums in research to develop methodologies for producing software of quality and dependability. Schools have devoted large efforts to curricula in the engineering of software. Yet, many people remain dissatisfied with the progress of the field over the past decade.

The methodologies currently employed to produce "software of quality" are based on the notion that quality is strongly related to rigor in the specifications and texts that appear throughout the software design process. According to this view, quality will be achieved when software is produced by a process consisting of four stages: obtain a clear and comprehensive requirements statement, construct a formal specification from the requirements, derive the programs from the specification, and demonstrate convincingly that the implemented program meets the specification. Program construction techniques that maintain tight relationships between program structure and dynamic behavior are considered to be essential. Readability, modularity, modifiability, style, adequacy of comments, and good tests are considered to be important guidelines. Failure to achieve software of quality is attributed not to possible limitations in the process itself, but to inadequate knowledge. Once the process is completed, the designer delivers the software to the customer and declares the job done.

I believe that this understanding is too limited to enable us as a discipline to systematically fulfill promises to deliver software of quality and dependability. I propose to reframe the question, from "What is software quality?" to "How do we satisfy the customers of our software?" By making concerns of customer satisfaction central among the criteria for judging software, new actions will appear for making reliable and dependable software.

Our current understanding treats quality as a property that can be built into a system by following certain rules and procedures. But this understanding inclines us to forget that "quality" and "dependability" are assessments others make based on their experience with how well the software helps them do their work. The question on the minds of the users of software is not, "Is this software well structured?" but "Does this software help me get more work done? Can I depend on it?" The greater the level of satisfaction, the more likely the customer is to say the software is of good quality and is dependable.

An immediate and obvious consequence of this shift of the question is that the software designer does not declare that the job is done. Instead, the customer declares satisfaction (or dissatisfaction) with what the software designer has delivered. The job of the software designer is not done until the customer has declared satisfaction.

One can distinguish three levels at which a customer can declare satisfaction:

(1) *All basic promises were fulfilled*. The customer assesses that the producer has delivered exactly what was promised and agreed to. This might be called "basic integrity".

(2) *No negative consequences were produced*. The customer uses the product for a while and encounters no unforeseen problems that cause disruption and perhaps serious losses. The customer assesses that the product's design has been well thought out and that it anticipates problems that were not apparent at the outset. That this level is distinct from the previous one can be seen in the common anecdotes, "This is what I asked for, but it's not what I want!" and "It does what they promised but I wish I'd have asked about what it would do in this situation!"

(3) *The customer is delighted*. At this level the product produces no negative consequences and, in fact, goes well beyond the customer's expectations and produces new, unexpected, positive effects. The customer expresses great delight with the product and often promotes it among others. The customer says the producer is a partner, understands the organization, and contributes to the well being of the customer.

With the new eyes given by these distinctions, I offer several observations.

(a) There is a level of satisfaction lower than the first level in the preceding list; I call it "cynical satisfaction". We see a lot of it in the computer industry. Many producers repeatedly make extravagant and hyperbolic claims about their wares. Virtually all software licenses explicitly disclaim any warranty or other responsibility by the producer. Customers have learned to discount the claims and purchase the software anyway. They find ways to use the software to make their work more productive. They say they are satisfied "once you factor out the producer's claims and factor in reality." But this is not a stable situation, for the cynical customers will transfer to another producer as soon as a more realistic offer is made.

(b) The first level has been made unnecessarily difficult to achieve because the language used to describe business and organizational processes is different from the languages currently used for formal specifications. For example, business processes are cyclic and comprise networks linked by requests their participants make of other people; formal specifications use logic notation to describe input-output relationships of software components. Business processes have deadlines and are triggered by time-dependent events; temporality is difficult to express in the notation systems used for formal specifications. The customer and designer would be in a better position to know they are in agreement if the language of the specifications contained the same distinctions as the language of business processes.

(c) Formal specifications are a language for specifying interfaces between different subsystems, not between systems and users. For this reason, the makers of commercial, packaged software seldom offer formal specifications to their customers; many makers

do not even use them internally. Few customers ask to see formal specifications and fewer still have the specialized training required to read them. Customers determine satisfaction partly from listening to reviews and opinions of others and partly from their own personal assessments of whether the software delivers what has been promised by the sales literature and by sales people.

The requirements statement, whether in a formal notation or not, can serve as the written promise of the software producer to the software customer. The customer expects that the delivered software meets the requirements. The requirements are useful only insofar as they lead to a satisfied customer.

(d) Dire negative consequences can arise in an endless number of ways. One common pattern arises when a large number of copies of a program interact over a network, producing "collective phenomena" that were unanticipated by the designers. An example is the stock market crash of October 1987, ascribed by many experts to computerized, programmed trading: A large number of computers, programmed to sell when prices dropped by more than a preset amount, automatically issued sell orders, which drove prices down and triggered more selling by other computers. Another example is the recent series of telephone network outages caused by the same bug being present in the many copies of the switching software.

Another common pattern of negative consequences arises around computer hardware failures. Business executives may make determinations, based on cost, about how much they are willing to invest in backup computer systems. Even if the producer of the computer systems delivers what was asked, the organization may find itself in hot water with its customers when the hardware fails and the organization cannot provide service. The consequences are often compounded by customer-service agents telling customers that the computers cause the problem, when in fact the problem was caused by an unadvertised no-frills decision of the organization's executives.

A third pattern of negative consequences arises around mistakes by users themselves. Users are more likely to be satisfied at the second level by operating systems that allow them to undo commands to delete files or by editors that make checkpoint copies of files.

A fourth pattern arises when a user takes an unanticipated action or unforeseen circumstances arise. The response of the software in these cases may be disastrous or costly.

A fifth pattern arises when users change their expectations, often in response to the new level of productivity made possible by the software itself. If the designer is no longer available to help modify the software for the user's new expectations, the user is likely to become dissatisfied.

A good software systems designer will bring long experience with many different customers to a new customer. That designer may propose to include functions that the customer did not ask for but which will spare the customer unwanted future problems. The designer will continue to work with the customer after the system is installed in order to modify the system in case negative consequences are discovered. These actions -- anticipation and continued availability after delivery -- are essential for a software producer to earn the customer's satisfaction at the second level.

(e) Very few software systems have produced genuine delight. Some examples include the second-generation UNIX system, the Apple MacIntosh, Lotus 1-2-3, the Quicken accounting system, Microsoft Excel and Windows, and PageMaker document system. In each case customers found they could complete much more work with the system than without, much more than they expected.

But delight is ephemeral if based on the software itself: Having mastered the new working environment, the user will expand horizons and expect more. Will the software producer be ready for the growing customer?

Delight arises in the context of the relationship between the customer and performer. The delighted customer will say that the performer has taken the trouble to understand the customer's work and business, is available to help with problems and to seize opportunities, may share some risks on new ventures, and generally cares for the customer. Organizations that are well known for cultivating fiercely loyal customers include IBM, Microsoft, and Digital.

(f) It is worth noting that the marketplace itself has promoted a general improvement in commodity software. Positive reviews in software magazines can lead to many new sales quickly, while negative reviews can kill a product. Microsoft, for example, has demonstrated a particular deftness at improving its products in response to feedback from customers and has set up its own networks to obtain this feedback.

I have argued that software quality is more likely to be attained by giving much greater emphasis to customer satisfaction. Program correctness is essential but is not sufficient to earn the assessment that the software is of quality and is dependable. Other methods and approaches are needed to generate satisfaction at the second and third levels.