Agile and the Definition of Quality

Gerald M. Weinberg, Author

Abstract. To convince people of the value of Agile, we need to produce new software that is full of wonderful features that the old software didn't possess while functioning exactly the way as the old software did.

Introduction

Some Agile writers have called me "the grandfather of Agile." I choose to interpret that comment as a compliment, rather than a disparagement of my advanced age. As a grandfather, much of my most influential writing was done long before the Agile movement appeared on stage. As a result, newcomers on the scene often fail to see the connection between those writings and today's Agile movement.

I use my blog to correct that situation, with a series of articles relating specific material to Agile basics. I started with an essay about my definition of "quality"—often quoted by not always understood.

A Bug in the Family

My sister's daughter, Terra, is the only one in the family who has followed Uncle Jerry in the writer's trade. She writes fascinating books on the history of medicine, and I follow each one's progress as if it were one of my own. For that reason, I was terribly distressed when her first book, Disease in the Popular American Press, came out with a number of gross typographical errors in which whole segments of text disappeared. I was even more distressed to discover that those errors were caused by an error in the word processing software she used—CozyWrite, published by one of my clients, the MiniCozy Software Company.

Terra asked me to discuss the matter with MiniCozy on my next visit. I located the project manager for CozyWrite, and he acknowledged the existence of the error.

"It's a rare bug," he said.

"I wouldn't say so," I countered. "I found over twenty-five instances in her book."

"But it would only happen in a book-sized project. Out of over 100,000 customers, we probably didn't have 10 who undertook a project of that size as a single file."

"But my niece noticed. It was her first book, and she was devastated."

"Naturally I'm sorry for her, but it wouldn't have made any sense for us to try to fix the bug for 10 customers."

"Why not? You advertise that CozyWrite handles book-sized projects."

"We tried to do that, but the features didn't work. Eventually, we'll probably fix them, but for now, chances are we would introduce a worse bug—one that would affect hundreds or thousands of customers. I believe we did the right thing."

As I listened to this project manager, I found myself caught in an emotional trap. As software consultant to MiniCozy, I had to agree, but as uncle to an author, I was violently opposed to his line of reasoning. If someone at that moment had asked me, "Is CozyWrite a quality product?" I would have been tongue-tied.

How would you have answered?

The Relativity of Quality

The reason for my dilemma lies in the relativity of quality. As the MiniCozy story crisply illustrates, what is adequate quality to one person may be inadequate quality to another.

Finding the Relativity

If you examine various definitions of quality, you will always find this relativity. You may have to examine with care, though, for the relativity is often hidden, or at best, implicit.

Take for example Crosby's definition:

"Quality is meeting requirements."

Unless your requirements come directly from heaven (as some developers seem to think), a more precise statement would be:

"Quality is meeting some person's requirements."

For each different person, the same product will generally have different "quality," as in the case of my niece's word processor. My MiniCozy dilemma is resolved once I recognize two things:

a. To Terra, the people involved were her readers.

b. To MiniCozy's project manager, the people involved were (the majority of) his customers.

Who Was That Masked Man?

In short, quality does not exist in a non-human vacuum, but every statement about quality is a statement about some person(s).

That statement about quality may be explicit or implicit. Most often, the "who" is implicit, and statements about quality sound like something Moses brought down from Mount Sinai on a stone tablet. That's why so many discussions of software quality are unproductive—it's my stone tablet versus your Golden Calf.

When we encompass the relativity of quality, we have a tool to make those discussions more fruitful. Each time somebody asserts a definition of software quality, we simply ask, "Who is the person behind that statement about quality."

Using this heuristic, let's consider a few familiar but often conflicting ideas about what constitutes software quality:

a. "Zero defects is high quality"

1. to a user such as a surgeon whose work would be disturbed by those defects
2. to a manager who would be criticized for those defects

b. "Lots of features is high quality"

1. to users whose work can use those features–if they know about them
2. to marketers who believe that features sell products

c. "Elegant coding is high quality"
1. to developers who place a high value on the opinions of their peers
2. to professors of computer science who enjoy elegance

d. "High performance is high quality"
1. to users whose work taxes the capacity of their machines
2. to salespeople who have to submit their products to benchmarks

e. "Low development cost is high quality"
1. to customers who wish to buy thousands of copies of the software
2. to project managers who are on tight budgets

f. "Rapid development is high quality"
1. to users whose work is waiting for the software
2. to marketers who want to colonize a market before the competitors can get in

g. "User-friendliness is high quality"
1. to users who spend 8 hours a day sitting in front of a screen using the software
2. to users who can't remember interface details from one use to the next

The Political Dilemma

Recognizing the relativity of quality often resolves the semantic dilemma. This is a monumental contribution, but it still does not resolve the political dilemma; More quality for one person may mean less quality for another.

For instance, if our goal were "total quality," we'd have to do a summation over all relevant people. Thus, this "total quality" effort would have to start with a comprehensive requirements process that identifies and involves all relevant people. Then, for each design, for each software engineering approach, we would have to assign a quality measure for each person. Summing these measures would then yield the total quality for each different approach.

In practice, of course, no software development project ever uses such an elaborate process. Instead, most people are eliminated by a prior process that decides whose opinion of quality is to count when making decisions?

For instance, the project manager at MiniCozy decided, without hearing arguments from Terra, that her opinion carried minuscule weight in his "software engineering" decision. From this case, we see that software engineering is not a democratic business. Nor, unfortunately, is it a rational business, for these decisions about "who counts" are generally made on an emotional basis.

Quality Is Value To Some Person

The political/emotional dimension of quality is made evident by a somewhat different definition of quality. The idea of "requirements" is a bit too innocent to be useful in this early stage, because it says nothing about whose requirements count the most. A more workable definition would be this, "Quality is value to some person." By "value," I mean, "What are people willing to pay (or do) to have their requirements met."

Suppose, for instance, that Terra were not my niece, but the niece of the president of the MiniCozy Software Company. Knowing MiniCozy's president's reputation for impulsive emotional action, the project manager might have defined "quality" of the word processor differently. In that case, Terra's opinion would have been given high weight in the decision about which faults to repair.

The Impact on Agile Practices

In short, the definition of "quality" is always political and emotional, because it always involves a series of decisions about whose opinions count, and how much they count relative to one another. Of course, much of the time these political/emotional decisions—like all important political/emotional decisions—are hidden from public view. Most of us software people like to appear rational. That's why very few people appreciate the impact of this definition of quality on the Agile approaches.

What makes our task even more difficult is that most of the time these decisions are hidden even from the conscious minds of the persons who make them. That's why one of the most important actions of an Agile team is bringing such decisions into consciousness, if not always into public awareness. And that's why development teams working with an open process (like Agile) are more likely to arrive at a more sensible definition of quality than one developer working alone. To me, any team with even one secret component is not really Agile.

Customer support is another emphasis in Agile processes, and this definition of quality guides the selection of the "customers." To put it succinctly, the "customer" must actively represent all of the significant definitions of "quality." Any missing component of quality may very likely lead to a product that's deficient in that aspect of quality.

As a consultant to supposedly Agile teams, I always examine whether or not they have active participation of a suitable representation of diverse views of their product's quality. If they tell me, "We can be more agile if we don't have to bother satisfying so many people," then they may indeed be agile, but they're definitely not Agile (capital A).

Why People Don't Instantly Buy Into Agile Methods: A Catch-22

When "selling" their methods, Agile evangelists often stress the strength of Agile methods at removing, and even preventing, errors. I used to do this myself, but I always wondered how people could resist this sales pitch. I would plead, "Don't you want quality?" And, although they always said, "Yes, we want quality," they didn't buy what I was selling. Eventually, I learned the reason, or at least one of the reasons. Why can Agile methods be so difficult to sell?

Another Story About Quality

I've demonstrated how "quality" is relative to particular persons. To test our understanding of this definition, as well as its applicability, let's read another story, one that illustrates that quality is not merely the absence of error.

One of the favorite pastimes of my youth was playing cribbage with my father. Cribbage is a card game, invented by the poet Sir John Suckling, very popular in some regions of the

world, but essentially unknown in others. After my father died, I missed playing cribbage with him and was hard pressed to find a regular partner. Consequently, I was delighted to discover a shareware cribbage program for the Macintosh: "Precision Cribbage" by Doug Brent, of San Jose, CA.

Precision Cribbage was a rather nicely engineered piece of software, I thought, especially when compared with the great majority of shareware. I was especially pleased to find that it gave me a challenging game, though it wasn't good enough to beat me more than 1 or 2 games out of 10.

Doug had requested a postcard from my hometown as a shareware fee. I played many happy games of Precision Cribbage, so I was pleased to send Doug this minimum fee. Soon after I sent the card, though, I discovered two clear errors in the scoring algorithm of Precision Cribbage.

(Perhaps the word "precision" in the name should have been a clue. If it was indeed precise, there was no need to call it "precision." The software would have spoken for itself. I often use that observation about product names to begin my evaluation of a project. For instance, whenever a product has the word "magic" in its title, I steer clear of the whole mess.)

One error in Precision Cribbage was an intermittent failure to count correctly hands with three cards of one denomination and two of another (a "full house," in poker terminology). This was clearly an unintentional flaw, because sometimes such hands were counted correctly.

The second error, however, may have been a misunderstanding of the scoring rules (which were certainly part of the "requirements" for a program that purported to play a card game). It had to do with counting hands that had three cards of the same suit when a fourth card of that suit turned up when the deck was cut. In this case, I could actually prove mathematically that the algorithm was incorrect.

So what makes this story relevant? Simply this: even with two scoring errors in the game, I was sufficiently satisfied with the quality of Precision Cribbage to:

a. keep on playing it, for at least several of my valuable hours each week

b. pay the shareware "fee," even though I could have omitted payment with no fear of retribution of any kind

In short, Precision Cribbage had great value to me, value that I was willing and able to demonstrate by spending my own time and (if requested) money. Moreover, had Doug corrected these errors, it would have added very little to the value of the software.

What's Happening to Quality?

My experience with Precision Cribbage took place some years ago, and occurred in a more-or-less amateur piece of shareware. Certainly, with all we've learned over the past few decades, the rate of software errors has diminished. Or has it?

I've conducted a small survey of more modern software. Software written by professionals. Software I use regularly. Software I paid real money for. And not software for playing games, but software used for serious tasks in my business. Here's what I found:

Out of the 20 apps I use most frequently, 16 have bugs that I have personally encountered—bugs that have cost me at least inconvenience and sometime many hours of fix-up time, but at least one hour for each occurrence. If I value my time at a conservative $100/hour (I actually bill at $500/hour), these bugs cost me approximately $5,000 in the month of August. If I maintain that average, that's $60,000 a year.

If I consider only the purchase prices, those 20 apps cost me about $3,500. In other words, over one year, the purchase price of the software represents less than 10% of what it costs me. (And these are selected apps. The ones that are even buggier have been discarded any time I can find a plausible substitute.) In other words, since quality is value, there's a large negative quality associated with this set of applications.

And that's only for one person. In the USA, there must be at least 100,000,000 users of personal computers. My hourly rate is probably higher than the average, so let's just estimate $10/hour, roughly minimum wage for the average person. That would give us an estimate $6,000/year per person for buggy software, which adds up to about $600,000,000,000 for the annual cost to United States workers. Even if my estimates are way off, that's not chump change.

Why Is Improving Quality So Difficult?

If they payoff is so huge, why aren't we raising software quality to new levels? We could ask the same question about improving auto safety, where tens of thousands of human lives are destroyed every year in the United States. You might think that's more motivation than any number of dollars, but it doesn't work that way. Unless the person killed in the car is someone we know, we've heard about so many traffic deaths that we've grown immune to the terrible cost. In other words, it's precisely because traffic deaths are so common that we don't get awfully excited about them.

And, I believe, it's the same with software failures. They're so common that we've learned to take them with an accepting shrug. We simply reboot and get back to work. Very seldom do we even bother to switch to a different app. The old one, with all its bugs, is too familiar, too comfortable. In fact, some people obtain most of their job security precisely because of their familiarity with software bugs and ways to work around them.

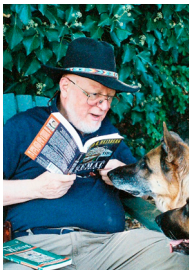
In other words, we're surprised that people don't generally feel motivated to improve quality because we vastly underrate the value of the familiar. And that observation explains an interesting paradox. Agile advocates are often so eager to prove the value of Agile methods that they strive to create products with all sorts of wonderful new features. But each new feature, no matter how potentially valuable, has a downside—a negative quality value because of its unfamiliarity. The harder we strive to produce "higher quality," the lower the quality we tend to produce.

It's a classic catch-22. To convince people of the value of Agile, we need to produce software that is full of wonderful features that the old software didn't possess, at the same time the new software functions exactly the way the old software did. No wonder it's so difficult to change the way we develop software.

Disclaimers:

Author's note: For this article, I've adapted some material from my book, "Agile Impressions" <https://leanpub.com/jerrysblog>

ABOUT THE AUTHOR



Gerald M. Weinberg is the winner of many awards for his writing. His works include "The Psychology of Computer Programming, An Introduction to General Systems Thinking, Becoming a Technical Leader," "Quality Software Series," "Perfect Software," and other books on computing, consulting, human behavior, writing, and technofiction. In addition to his works on software development, Gerald is also science fiction author. He was an architect of NASA's space tracking network and designed and built the first real-time multiprogrammed OS. Gerald is in the University of Nebraska Hall of Fame and is a Founding member of the Computing Hall of Fame.

E-mail: jerryweinberg@comcast.net