Metrics Say Quality Better than Words

Tom Gilb

Traditional software engineering jumps too quickly from high-level quality statements, such as "user friendly," to design ideas.

Quality requirements such as usability, reliability, security, and adaptability drive software system costs. Functions and use cases are far less interesting and, in many cases, state only the functionality that stakeholders already have—albeit with less quality than they want. Software engineers and project managers must quantify quality requirements to manage project results, control risks and costs, and prioritize tasks intelligently. Otherwise, we risk continuing our already embarrassing reputation for software project costs and results.

Traditional software engineering jumps too quickly from high-level quality statements to design ideas. To be clear about design, we must add a step to quantify the quality levels required. Words are ambiguous and lead to misunderstandings. Compare "I want a user-friendly system" to "I want a system that reduces the number of accidental errors made by novice users to fewer than five per 1,000 transactions." You can engineer and prove the latter.

I've defined 10 principles for quantifying quantification and a notation system for them:

1. Stakeholders expect many different system qualities, but developers must focus on the most critical top 10, until they achieve these in practice. And they should start with the top three.
2. Decompose complex qualities such as usability, maintainability, and adaptability into requirement subhierarchies, and select those that matter most to your project.
3. Give each quality requirement at least one defined scale of measure ("Scale"). All qualities vary in a scalar manner, so you can always quantify qualities along a defined scale of measure.
4. State the current benchmark levels—that is, "Past" performance levels.
5. Specify one or more target levels for each quality requirement. I advise my clients to use "Goal" for a committed target, "Stretch" for an ambitious target that's not essential, and "Wish" for stakeholder interest in the level but with no developer commitment yet to achieving it.
6. All specified and delivered quality levels can vary—from worthlessly bad levels, to tolerable, satisfactory, and beyond-satisfactory levels. Specify the constraint levels first. For example, use "Fail" (for pain levels), and maybe even "Catastrophe" for total system failures.
7. Consider future quality level trends, and set your targets to compete with trend levels. You can use a "Trend" specification to quantify your best estimate of degrading quality levels of your systems, or perhaps to specify improving levels of competitors' systems.
8. As quality levels near the state of the art, costs will likely increase exponentially. I advise clients to use a "Record" specification to capture the state-of-the-art level of any given quality. This knowledge of what is

Continued on page 66

Subjective Quality Counts in Software Development

Alistair Cockburn

I keep hunting for evidence that quantified quality requirements have a demonstrable ROI for increasing stakeholder value. I'm just not finding it.

Since the early 1990s, I've interviewed dozens of project teams worldwide. I started when I was looking for things to include in IBM's methodology for object-oriented projects, but I've continued the practice for 15 years now. IBM gave me no restrictions—only to find out what mattered—and I still use the same guidelines. I ask each team to tell me the project's history, what they liked and didn't like, what they would do differently next time, and their top-priority items to attend to.

Quantified quality requirements are stunningly missing from the results. Not a single project team said, "If only we'd had quantified quality requirements, our project would have turned out differently." I finally asked a project leader the question directly. I was hopeful, briefly, when he replied, "Yes, we should have quantified the response time requirement. It was terrible and made the product unusable."

Sadly, his answer contains his own rebuttal: they never needed to quantify response time. "Terrible" means that it exceeded all reasonable bounds. Programmers don't have to be told "our definition of interactive is 3 seconds." They already know what "interactive" means because they use computers daily.

The presence or absence of the number "3" didn't cause this project's failure. It was a basic process failure. The system was obviously interactive, so the group should have run user tests for being "acceptable" with respect to response time. I expect that they would have found "acceptable" to have different values at different places in the user interface. In other words, "3 seconds" would have been the wrong number anyway.

This gets us to the difficult issues with quantified quality requirements.

First, most projects succeed or fail on the basis of two or three primary factors. Many researchers have studied and listed these factors. Lack of a good sponsor, lack of interaction with real users, lack of a qualified lead designer, sloppy design and testing habits—all these are on the list. Quantified quality requirements hasn't yet been on any list I've seen.

Finding those two or three primary factors and taking care of them is orders of magnitude more important than chasing decimal places on quality requirements.

Second, many quality requirements can't be known until users review the system. Outside the software industry, quality researchers recognize that "objective quality" (including Philip Crosby's infamous "conformance to requirements") becomes important when manufacturing items in quantity. By that time, the "subjective quality" issues—those related to "user satisfaction"—have already been investigated and settled. Subjective quality issues predominate when deciding what to build in the first place.

Software development is all about deciding what to build, or subjective quality. We

Continued on page 66

Software development is about deciding what to build, and subjective quality issues predominate in this process.

point

continued from page 64

possible might help warn us about too risky ambitions for our Goal levels.

9. Specify the *conditions* under which each level is required—for example, by the end of next year, in China, for the teenage market. Use square brackets to identify these conditions: Goal [End of Next Year, China, Teenage Market] 20 percent.
10. Specify some means of testing and measuring the quality level. Use "Meter" specification. Consider the cost of carrying out the measurement.

Using these principles, we could give the following example of a "user friendly" requirement:

- User Friendly: "The number of errors a user accidentally makes."
- Scale of Measure: The average number per defined [Number of Transactions: Default = 1,000] of defined [Error Type] that a defined [User Type] makes when carrying out a defined [Transaction].
- Meter: A combination of system data-entry statistics and database analysis.
- Past [Incorrect Code, Novice, Order Entry, March 2007]: 52 <- Audit Report>.
- Goal [Incorrect Code, Novice, Entry, August 2008]: Fewer than 15 <- CEO.
- Goal [Incorrect Code, Novice, Order Entry, January 2009]: Fewer than 5 <- CEO
- Fail [Incorrect Code, Novice, Order Entry, January 2009]: Fewer than 15 <- CEO
- Stretch [Incorrect Code, Expert, Order Entry, January 2009]: Fewer than 2 <- Marketing Director

Yes, you could make some headway if you were simply told "be user friendly," but controlling costs and, importantly, controlling contract payments will improve with quantified quality specifications.

Once you've clarified and agreed to the initial quality requirements, you can identify potential designs to meet these quality levels. You can use analytical methods such as "impact estimation" (which quantifies the impacts of different designs on the quantified requirements) to help you iterate the requirements, designs, and costs until you find an initial satisfactory solution. Then you can use an "evolutionary" feedback project-management methodology to deliver the solution, in a series of steps, that let you actively seek quantitative feedback from each delivered step, monitor the real results, and modify requirements and design solutions—as relevant. Once you've reached a requirement's target level, don't spend any more resources on it. Reprioritize your limited resources on the most important gaps between the current levels and target quality levels for other requirements.

Today, most software projects don't even attempt to quantify their quality requirements. If we could adopt only one practice to help systems engineering, quantifying quality should be it. But an organization won't quantify quality requirements on the initiative of programmers. It happens only when management makes it the standard.

Tom Gilb is an international consultant, teacher, and author. *Competitive Engineering* (Elsevier, 2005) is his ninth book. His book *Software Metrics* (Winthrop Publishing, 1976) was the inspiration leading to CMMI Level 4. Contact him at tom@gilb.com; www.gilb.com.

counterpoint

continued from page 65

generally don't know what to require until we've built something and the users comment on what they see. Experienced product designers know that users often misreport what they will like and then change their minds after they see the result.

Third, subjective quality requirements vary across product design elements within a single system. Developers can tweak the quality in many ways as they design and test the system, using the idea that product design elements fall into three categories of subjective quality: baseline, linear, and exciter.

Baseline design elements (such as brakes on a car) simply must be there. Users become happier as they get more of the linear elements (examples being cargo space, engine power, gas mileage). They are happy to see an exciter (a sunroof, perhaps) but not worried by its absence.

We can often delight users by first reducing the set of baseline items to generate some play in the budget, and then interacting with the users to discover how to give them the most linear and exciter items their budget will allow. We can't name the optimal set of baseline, linear, and exciter product elements in advance, certainly not when we're writing the requirements. We discover it in dialogue as the system grows.

All this makes it less and less meaningful to produce numbers for quality items in-side the requirements document.

Even the Point-Counterpoint topic itself, "quantifying quality requirements," already contains the hidden implication that there won't be much feedback about the requirements' quality. Lack of feedback is one of the known key failure factors. Take care of feedback from the users while you develop the system, and the quality characteristics should make themselves known in good time.

(Note: Noriaki Kano identified the three subjective quality categories in 1984. I used Mike Cohn's simpler words for them. Jeff Patton taught me how to move baseline items to the linear category.)

Alistair Cockburn is an internationally renowned project witchdoctor and IT strategist, several-time winner of the Jolt & Productivity book awards, co-founder of the agile development movement, and expert in project management. Contact him at acockburn@aol.com.

Tom Responds

I agree with Alistair: "Finding those two or three primary factors, and taking care of them, is orders of magnitude more important than chasing decimal places on quality requirements." But if any of those critical factors is a quality, then I suggest we need to agree on their order of magnitude, at least. That's critical. Nice words don't do the job as well as a number. Using numbers doesn't mean they must be exact. Using numbers doesn't mean they're static either!

I agree with another point Alistair makes: "Take care of feedback from the users while you develop the system, and the quality characteristics should make themselves known in good time." That's why my clients initially use approximate numbers ("Goal: 20 minutes"), then use feedback from rough measures on a weekly cycle to adjust their perceptions—not only to the reality of stakeholders' feedback but also to the technology's effectiveness, and the costs incurred. I assume it's well known that numeric feedback is a pretty powerful tool—it beats nice words. Of course, we need to listen to any useful nonnumeric feedback, too. Numeric doesn't mean "no words."

I understand when Alistair observes, "Quantified quality requirements are stunningly missing from the results." Of course they're missing. Led by nonengineer programmers, software people (I won't call them engineers or scientists) don't know how to quantify quality. And they couldn't care less, as long as they're well paid for their frequently failed projects. A project I saw last year had used $100 million over eight years to fail to achieve any of its eight major objectives. All were qualitative objectives, such as "increased robustness," "better ease of use," and so on. The project hadn't delivered any result to any stakeholder. The system developers weren't even made aware of the project's initial objectives. But they were having a good time spending millions, while delivering nothing the managers who funded the project desired. A fool and his money will soon be parted—by programmers. Such projects are common, we observe first hand.

We software people are so immature regarding a "numeric quality culture" that we don't even understand that lack of it might be a major problem.

Alistair Responds

Tom never argued that quantified quality requirements have a demonstrated ROI for increasing stakeholder value. He simply made the unsubstantiated assertion, "If there were one thing we should adopt to help with the engineering of systems, quantifying quality would be it."

The one thing? The nearest failure factor I find substantiated is "unclear requirements." Even that phrase is misleading. For example, consider the task of choosing a color (although the problems come with any topic). You could report each of the following seven situations as "unclear requirements":

1. No usage experts show up. The customer says, "Discuss requirements? Of course not—that's why we hired you!" Nobody knows if the color should be red or blue.
2. Users don't know what they want. They speculate the need for red but decide after they see the results.
3. Nobody asks, "Why red?" If the users work in dark rooms and want to protect their dark vision, then "dim red, to protect their dark vision" would be better than the quantified "720 nm 100 lumens."
4. Business changes. Red was correct last month, but orange is the color this month.
5. Developers don't design for change. They code with in-line constants. When it changes, they scream that the change will take too long, and "The requirements experts should have figured out the correct value at the start!"
6. Undetected feature creep. Someone slips in the need for a strip of red on the side. The product is late, and no one can find out where this requirement came from.
7. Developers don't listen. They decide that orange is really the color needed, no matter what was written.

None of these situations is fixed by expanding "red" into "720 nm 100 lumens." However, six are helped by improving the communication-feedback cycle.

To revise Tom's statement, "If there were one thing we should adopt to help with the engineering of systems, then improving the communication-feedback cycle would be it."

Quantifying quality requirements is like fixing the *n*th decimal place when the first digit is still unknown. It could become significant at some time, but there are order-of-magnitude more important things to attend to first.