Software quality is evolving beyond static assessment to include behavioral attributes, such as availability and maintainability.

Jeffrey Voas and William W. Agresti

Software Quality from a Behavioral Perspective

Thanks in large part to early metrics, most old-school researchers and practitioners view software quality as a static assessment of the code’s structure. Fortunately, a new generation is realizing that software quality is more than a static trait; it also comprises non-functional, or behavioral, attributes, such as reliability and maintainability. Even Microsoft’s Bill Gates seems to have recognized this. Rumor has it that on 15 January 2002 he sent the following e-mail to all employees:

Every few years I have sent out a memo talking about the highest priority for Microsoft. ... Over the last year it has become clear that ensuring .NET is a platform for Trustworthy Computing is more important than any other part of our work. If we don’t do this people simply won’t be willing—or able—to take advantage of all the other great work we do. Trustworthy Computing is the highest priority for all the work we are doing. We must lead the industry to a whole new level of Trustworthiness in computing.

In light of this quality redefinition, the challenge facing software-quality research is to produce a metric that will guide developers in choosing techniques to achieve the desired product quality. “Quality” in this sense comprises some set of key behavioral attributes, which we believe are reliability (R), performance (P), fault tolerance (F), safety (Sa), security (Se), availability (A), testability (T), and maintainability (M). Software quality (Q) is thus a function of these combined attributes plus an error term (ε) that represents quality aspects these eight attributes can’t define. This error term is some quantitative, differential function that accounts for the fact that precise numerical assessment of many of these attributes is not possible. Thus, quality would be

\[ Q = f(R, P, F, Sa, Se, A, T, M) + \varepsilon. \]

This metric is simplistic, impractical, and (in its current form) not even accurate. It does, however, serve as a backdrop for examining the elements that a quality metric should have. Our equation illustrates the type of flexible functional form developers need to account for software development’s practical realities. It is consistent with the idea that to build quality software, developers must consider various tradeoffs among the key attributes, which some call the ilities of software development. For example, as security increases, performance decreases, because increased security consistently reduces performance. Thus, in our definition of quality, simultaneously increasing security and performance is unrealistic. Further, as fault tolerance increases, testability decreases, because if the software can more easily reveal hidden defects...
at test time, failures are more likely to propagate when the software is operational. Thus, attempting to increase both fault tolerance and testability is also unrealistic.

Quite simply, the software community needs a different quality model; this equation is an attempt to combine these nonfunctional attributes and give developers some framework for examining the impact of these attributes on software design and cost. To build high-quality software, developers must consider all the desired ilities, and they cannot maximize all of them for a single system. Further, each attribute comes with technical and economic requirements that often conflict, and a usable software quality metric must account for the tradeoffs among them.

Knowing the goals for each behavioral attribute before or very early in development, either quantifiably or qualitatively on paper, would be immensely valuable. Moreover, in defining these goals, developers must consider component requirements as well as those imposed by the system as a whole. As developers create requirements, they should discuss the economic tradeoffs necessary to achieve each attribute in the desired concentration. In this way, they ensure that the final product cost-effectively delivers the correct degree of each attribute. Finally, to achieve a particular level of any ility, developers must employ certain techniques, methodologies, tools, and processes, each of which has associated costs and benefits. To produce quality software in a practical setting, it is vital that software engineers begin to explore the economic issues related to those tradeoffs.

In this article, we invite the software community to view quality in more practical terms and propose several modifications to our basic equation, which we believe will enhance its flexibility and utility over time.

**QUANTIFYING QUALITY**

Underlying our equation is the idea that a particular software system can achieve a specific quality level by combining each attribute’s contributions. That is, a particular equation will describe the degree to which the software contains a particular attribute. We measure an attribute’s contribution in units. For example, quality for a particular piece of software might comprise one unit of reliability, 20 units of performance, three units of fault tolerance, two units of safety, zero units of security, five units of availability, three units of testability, and 18 units of maintainability. The units suggest how much of that attribute is present in the software. So hypothetically, if we assign 100 as the maximum for any attribute, then for these eight attributes, 800 is the maximum score, but any score of less than 800 is achievable by many different means.

The units are not necessarily equivalent, meaning that one unit of reliability is not necessarily equal to one unit of performance or one unit of any other behavioral attribute. Moreover, for ilities such as security, a unit is rather vague, possibly not even quantifiable or measurable.

Because developers typically measure each attribute differently, we normalize attribute values so that they lie in a range from zero to one. As a result, overall quality also lies in that range. Within the normalized range, the highest value for an attribute (one) is its maximum achievable level—as if that attribute were the sole focus of quality, and any tradeoffs with other attributes are in this attribute’s favor. For example, if maintainability’s value is one, the software embodies the best maintainability practices of the software-engineering community. In the extreme case, it might mean that the organization developing the software has taken the following steps:

- Independent experts examine the software and propose alternative restructuring and coding.
- The organization conducts experiments to identify if these alternatives made a difference. For example, test subjects (representative maintenance programmers for that software) might complete a suite of anticipated and representative maintenance actions to modify, enhance, and repair the software.
- The organization, in concert with maintenance programmers, develops a support environment for maintenance tools, conducts usability experiments with the environment, and modifies the environment until everyone judges it to be as supportive as possible.
- Another independent expert review reevaluates all the changes and iterates further until no one can imagine a more maintainable piece of software.
- The expert assessors award maintainability a value of one.

**Composite systems**

These steps illustrate that quantifying quality can be problematic even for a monolithic software system. The difficulty increases when the system is a system of systems or a system of components.

Figure 1 shows two components in a series—a very simple system. In a composite system, each component has its own equation, so the system in the figure should have the equations

![Figure 1. System with two connected components.](image-url)
We view components as stand-alone entities, with each component potentially having a different quality level. It is meaningless to assess these behaviors outside the context of the system where the components will reside, so the problem becomes how to determine the quality of the composite system.

Two issues are at stake. The first is the interplay among the different degrees of the same attribute when the two components connect: How much reliability, for example, does the composite have as the result of $R_1$ and $R_2$?

The second issue is more important and more difficult to resolve: What is the cross-functional consequence of the various attributes? Will availability $A_1$ negatively impact fault tolerance $F_1$ in the composite? If so, you can’t simply assume that the degree of fault tolerance achieved is some function of $F_1$ and $F_2$, because a closer look reveals that someone has designed $A_1$ in a way that decreases $F_1$ when the two components connect. In other words, component 2’s availability characteristic diminishes the fault tolerance that component 1 provides.

Bear in mind that this is a tiny two-component system in a series. A real system, composed of commercial off-the-shelf software and hardware, is highly unlikely to have values for any illy attribute unless the vendor supplies them, and even then the numbers would be suspect. Add distributed processing and Internet transactions, and the composition problem quickly becomes intractable. It is not enough to quit here, however. As we describe later, the software-engineering community must at least partially resolve this problem before it can truly claim any success in achieving trusted computing.

**What color is $Q$?**

Another interesting issue is the measuring stick for quality. How do you represent quality before you normalize attribute values? We stated earlier that quality is in a range from zero to one, but its interpretation and meaning are complex. Is quality better represented as an integer, a floating-point value, a probability, or possibly an 8-tuple such as $(1,2,10,4,3,7,8,22)$? What about a color scheme? Whatever the measure, quality must represent some relative metric that lets the measurer determine if it is increasing or decreasing, and the metric must be revealing enough to serve as the basis for accurately predicting the software’s behavior.

Suppose a color-coding scheme did represent quality levels. Then green, yellow, and red could represent high, medium, and low quality. In this format, quality is not an absolute value, which means that developers can observe how it changes over time, even if the software itself does not change. For example, the value of security could change as the organization discovers new threats, even if developers decide not to modify the software itself—a phenomenon some refer to as software rot.

This example highlights a significant implication. If quality can change quickly relative to new threats or changes in the operating environment, then quality is based not only on the attribute set but also on the execution environment. This in turn means that the environment can potentially modify any attribute’s value for a given piece of software. Thus, software probably cannot have a single quality valuation, but instead needs a series of quality valuations, based on different environments.

We offer the following formulation for characterizing this aspect of quality:

$$\text{Prob} (Q \geq 0.7 \mid E = e_2 \text{ or } e_1) > 0.9,$$

where $e_2$, $e_1$, and $e_1$ are specific operational environments for the software.

This formulation recognizes the inherent difficulty in stating that quality will be 0.4, 0.7, or any other precise value. All you can say, at some confidence level, is that you require evidence from tests, demonstrations, and experiments to convince a review panel that the quality will be at least 0.7 some percentage of the time when the software is in those specific operational environments.

Even so, requiring overall quality to meet a prescribed level might not be sufficient, given that quality is a composite measure. To address this aspect, you could also require the software to meet a specific attribute level. The formulation

$$\text{Prob} (Q \geq 0.7 \text{ and } M > 0.8 \mid E = e_2 \text{ or } e_1) > 0.9,$$

for example, imposes an additional requirement on maintainability $M$.

The second probabilistic formulation could encourage fruitful applications of optimization theory that aim to guide decisions about quality. For example, you could maximize a particular attribute, say maintainability, subject to achieving some acceptable level of quality, and given resource expenditures constrained to some value. Or you could focus on the software’s acceptable quality even if it is ported to several environments. In this case, you would minimize costs while specifying that, say, quality be greater than 0.6 across environments.

**Weighting priorities**

Arriving at a more accurate picture of quality requires weighting each attribute according to its importance to the system into which the software will be embedded. We give
each of the eight attributes a $w_i$ value in the range of zero
to one and place them in this linear equation:

$$Q = w_R R + w_F F + w_Sa S_a + w_Se S_e + w_SA A + w_FT + \sum w_M M,$$

where the sum of the weights is 1.0.

The weighting for each attribute depends on the type of
software. For a financial system, the weight for security
would probably be higher than that for safety, and the
weight for testability is likely to be zero. For a safety-critical
system, the key attributes would probably be reliabil-
ity, performance, safety, fault tolerance, and availability.
For an e-commerce system, the key
weighted attributes would be reli-
ability, performance, availability, security, and maintainability (main-
tenance on these sites occurs con-
tinually).

Weighting is a reminder not to
overdesign the system in favor of a
particular attribute. As developers
ensure that an attribute’s weighting is consistent with that
attribute’s importance, they avoid silly design decisions.
Creating a system with a large value for a security attribute
and a small weighting for security would be foolish and
expensive. This equation makes it easier to examine trade-
off issues when developers create system requirements.
Another useful analysis tool would be a financial tradeoff
model that defines the costs of building a system with just
enough of each attribute to satisfy the system’s quality
needs.

Software versus hardware

So far, we have focused on the software part of the sys-
tem. A quality metric must also account for the hardware’s
contribution to the attribute levels. For example, part of
performance is due to the order of the software algorithms
but part is also due to the choice of processor, available
memory, and so on. Thus, some of performance metric $P$
reflects how well the software interoperates with the hard-
ware. Again, an attribute’s valuation must account for the
anticipated hardware environment. The fastest algorithms
on the slowest processor will still yield a diminishing per-
formance attribute.

Project versus product risks

A quality metric must also consider the effect of risks.
This entails knowing the difference between risks that
affect the project (budget and schedule) and risks that
affect product behavior (portability and quality). Our
equation focuses on product behavior, but a reasonable
quality metric should also consider attributes such as
portability and readability. Only by including these can the
metric provide the total cost of owning a software system.

QUALITY AS A RECIPE

If the linear equation of weighted attributes is accurate,
we should be able to “solve for” single attributes. That is,

$$w_Sa S_a = Q - w_R R + w_F F + w_Se S_e + w_SA A + w_FT + \sum w_M M.$$

We can’t do that, however, because we’ve already said
that the units of each attribute are not equal. Adding four
oranges and four apples creates neither eight apples nor
eight oranges. It creates eight pieces of fruit. Even the orig-
inal general equation in which quality is a function of the
eight attributes plus the error term is more a recipe than
a mathematical absolute. The recipe’s ingredients are the
amounts of each attribute. By adding distinct ingredients, you
slowly create a completely new
entity with different units than
those in any of the ingredients. In
that respect, these equations are
more of a chemical recipe. Adding
two teaspoons of salt to two cups of water creates a new
entity, salty water—in essence “modified” water.

In this chemical analogy, quality is more a compound
than a mixture. Such a view acknowledges that, as a whole,
the software has emergent quality properties that are pres-
ent in ways unapparent in observations of an isolated
attribute. For example, a tablespoon of salt in isolation is
just salt. Adding it to something else produces different
results. Salted buttered popcorn is quite different from a
candy bar containing the same tablespoon of salt.

Given that quality is a chemical composition of distinct
ingredients, our linear equation presents a reasonable,
abstract model for discussing what software-quality behav-
iors really mean.

DERIVING MEASUREMENT SCHEMES

Regardless of whether or not the attributes have the
same measurement unit, a quality metric must be flexible
enough to measure attributes individually. Some attributes
lend themselves to direct numerical measurement. A
reasonable unit for measuring reliability, for example, is
mean time to failure. An organization typically uses test-
ing to quantify such numerical values.

For other attributes—such as fault tolerance, safety, secu-
ritiy, testability, and maintainability—direct numerical
measurement through testing is problematic. Fault toler-
ance, for example, usually refers to the use of design prin-
ciples such as redundancy and independence, and to the
system’s ability to rollback and recover from undesirable
runtime states. How do you place a numerical value on
this ability? For such attributes, an absolute value is infeasible.

Earlier, we described the possibility of color coding as a
way to portray relative quality. This scheme would also
work for individual attributes that require relative value. Ranking attribute levels according to the use of best practices is another possibility. A security ranking, for example, might consider the number of principles for designing more secure code. Independent experts could verify that an organization has correctly applied a certain number of these principles and would then rank the application’s security attribute accordingly. Similar schemes are possible for other attributes.

The bottom line is this: It is not enough to say, “Behavioral attributes are not directly measurable, so we can’t measure them.” As we have tried to show, you can work around relative values by formulating schemes from an attribute’s key indicators. Thus, you can still assign numerical values to various attributes and then normalize them; the greater the value in the range of zero to one, the more that attribute is necessary or the more you have achieved it. By having such a system during requirements elicitation, developers can immediately begin to determine what techniques, methodologies, tools, processes, and costs they will need to produce a system with that quality level.

Jeffrey Voas is chief scientist at Cigital. Contact him at voas@cigital.com.

William W. Agresti is an associate professor of information technology at Johns Hopkins University. Contact him at agresti@jhu.edu.