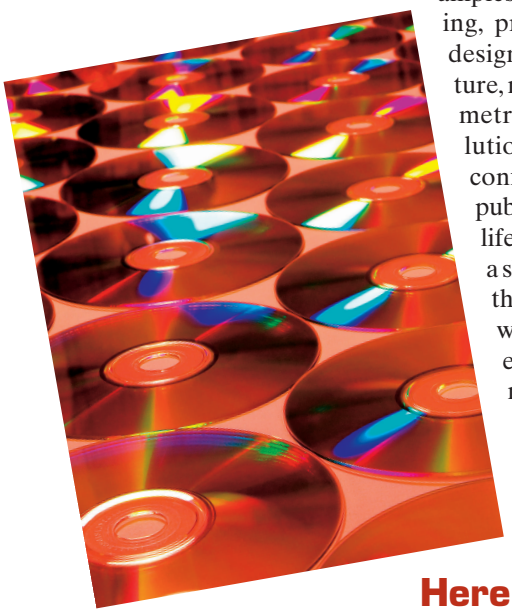
A Baker's Dozen: 13 Software Engineering Challenges

Jeffrey Voas

Software engineering has developed over the last few decades into a discipline of many diverse areas of interest. Examples include testing, programming, design, architecture, maintenance, metrics, and evolution. Specialty conferences and publications proliferate, stay for a short time, and then disappear, while software engineering remains as nontraditional engineering—part craft, part art, and part logic.



Here are 13 challenges facing the software engineering research and practitioner community, and hints on what to do about them.

No one has a magic recipe for building highly reliable software that can successfully be applied to all projects and organizations. Software engineering suffers from a nuance factor that may never enjoy a workaround capability. In this article, I look at 13 major research challenges for the software engineering community and provide hints on how to solve them.

To begin, I should note that the notion of a grand challenge is not new. Here are a few examples of grand challenges:

- The Grand Challenges in Global Health initiative, established on 27 June 2005, helps fund scientific research to fight diseases that kill millions of people each year. The initiative has offered grants totaling US$436.6 million for a range of innovative projects (http://www.grandchallengesgh.org).
- In October 2004, SpaceShipOne became the first privately funded spacecraft to reach suborbit, nearly 70 miles above Earth. A year later, "Stanley," a Volkswagen Touareg modified by Stanford University students, covered some 130 miles of desert without a human driver. It was guided by computer programs and sensors (*Christian Science Monitor*, 12 Jan. 2006; http://www.csmonitor.com/2006/0112/p13s01-stss.html).
- In 2001, the National Research Council defined the grand challenge in environmental science this way: "… the challenge to understand how the Earth's major biogeochemical cycles are being perturbed by human activities; to be able to predict the impact of these perturbations on local, regional, and global scales; and to determine how these cycles may be restored to more natural states should such restoration be deemed desirable" (http://newton.nap.edu/books/0309072549/html/14.html).

What is suggested here is that there is both a desire to be the creator of these difficult tasks as well as people who wish to demonstrate their abilities against them. Unfortunately, the challenges I discuss here will not have the politics, notoriety, or sex appeal of other grand challenges. However, I believe that for computer scientists and computer engineers, these grand challenges present enough work to be solved to be considered "grand" challenges in their own right.

CHALLENGE 1: SOFTWARE QUALITY

Software quality is something similar to beauty; it is in the eye of the beholder. Therefore, software quality is mostly intuitive, not formal. Thus, you will know it when you see it. The first challenge to the software engineering community is to define *quality*. Is it based on standards, people, and processes? Or is it

1520-9202/07/$25.00 © 2007 IEEE

defined on the quality of the product? If you take the latter position, then you must consider the attributes of quality: reliability, safety, security, testability, maintainability, performance, availability, fault-tolerance, recovery, and survivability.

A software system developed to address these attributes would be considered of high quality, but that is nontrivial to achieve.

Further, many of these attributes are hard to define and measure. That creates tension, a problem to be discussed later. Consider the following statement: "It is easier to build correct software than it is to ever know that you have done so." That may seem odd, but it's true.

Hint: Start with a basic perspective such as "quality means fit for purpose," and then start decomposing the terms *fit* and *purpose*.

CHALLENGE 2: RETURN ON INVESTMENT

If your software system works as desired and continuously, but you paid too much for it—given that you later learned of another system that would have been equally satisfying for half the price—would you be pleased with your ROI?

What is the advantage of building reliable software versus lesser quality software? Poor quality software creates business opportunities in the maintenance phase for the original developer. I recall examples where it was mandated to create lesser quality software since the profit margins on a delivered version were so low that you simply waited until the bugs started to fly. That is, the profit was in the extermination phase.

Fortunately, the software engineering community is working to develop ROI metrics using field data that is based on statistics, not anecdotes. However, the relationship between methods, approaches, and models with respect to the total cost of ownership economics is still immature because of many unforeseen variables. The empirical software engineering community is working diligently, but generic solutions are years away.

Hint: This is not so much of a research challenge but an observation that requires consistent journaling on projects so that lessons can be collected, aggregated, and disseminated to the larger scientific community.

> **"It is easier to build correct software than it is to ever know that you have done so."**

CHALLENGE 3: PROCESS IMPROVEMENT

Can clean pipes produce dirty water? Yes! (J. Voas, *IEEE Software*, vol. 14, no. 4, 1997, pp. 93-95; http://doi.ieeecomputersociety.org/10.1109/MS.1997.595964.) The software engineering community has embraced the notion that good software development processes make for better software products. In physical engineering that is true, but in software engineering, that is still debatable. CMM Level 5 does not guarantee high reliability software; it only suggests such. Also, this ties back to Challenge 2: How much higher quality can I afford—that is, where can I cut corners?

A few challenges related to software testing and verification have forced the software engineering community to acknowledge that it is easier to assess a process than to assess the software product, and that has steered the community toward assessing process far more frequently than assessing product.

Hint: Blend a variety of common sense do's and don'ts with any process-oriented scheme. If you see that some portion of the standard simply does not apply to you, document why, and move on.

CHALLENGE 4: METRICS AND MEASUREMENT

Metrics on projects are collected routinely, but their interpretation is dubious. Correlating the collected information to a project's true status is difficult. The metrics and measurement research and practitioner community has long suffered from this dilemma, and it is a reason why metrics are often collected and then discarded. If you do not know what the statistics mean, what can you make of them?

Hint: If you are going to collect metrics for reasons other than what your boss had in mind, try to get your organization to give you the time to train others in your group and at least feed the lessons derived from the collection back into future efforts.

CHALLENGE 5: STANDARDS CONFUSION

What standard do I follow? There are more than 1,000 existing software engineering standards. Some are geared toward testing, reliability, safety, security, and so on. Also, some standards are forced onto vendors by government regulation.

Regardless, software vendors and organizations that acquire software are forced to comply and/or abide with these standards, and the lack of harmonization of various software engineering standards makes this another one of the key challenges that software engineering faces.

Hint: Make sure that the standard you select fits your organization. General fact: You'll be far better off picking pieces and parts from various standards that apply to you than religiously following one to the letter of the law.

CHALLENGE 6: STANDARDS INTEROPERABILITY

This challenge is defined as follows: If I know nothing about two software components, but I do know something about the different standards that were used to develop them, can I sim-

ply look at the interoperability of the standards and then make an assertion about how the composed components will behave?

The question here is whether the composability of distinct standards may prove to be an easier prediction challenge than the prediction problem of component composability. Note that what I am suggesting here is difficult and rarely tackled. Why? Because some standards are more system-oriented versus component-oriented, and some are more process-oriented versus product-oriented. Once you look at those cross-product combinations and then throw in vertical domains such as medical, energy, transportation, and so on, the ability to harmonize any group into a single statement of trustworthiness is prone to failure, misinterpretation, and apathy.

Hint: The same advice for Challenge 5 applies here. Pick the parts that work and leave the rest behind.

CHALLENGE 7: LEGACY SOFTWARE

The next challenge is when to declare a legacy software system to be beyond repair. In the mechanical world, predictive decommissioning of worn-out hardware is well understood. Y2K was an example of where the software and IT community had no long-term vision concerning how long old Cobol systems would survive. The result: economic burden on IT infrastructures, and in some cases, panic.

The challenge here is to better understand the sustainability of systems before they are developed and to create design-for-maintainability paradigms and metrics that predict either when to let die or what life support should be rendered and at what ROI.

Another myth in the software legacy world is the notion of "size": the size of a fault, a fix, or the code. We all have intuitions here, and they are just that—intuitions. In short, size should not matter if a huge modification does not dynamically tickle most of the rest of the code. A tiny modification can have huge impacts if it touches most syntax of the code. And while the notion of static traceability is pretty well understood, dynamic traceability is not often applied for the same reason that testing numerous times with different test cases is intractable.

No matter how good the software has performed, if its environment has so drastically changed, then the software may need to be decommissioned.

Another important notion here is when the code itself is never modified, though the world around it is. No matter how good the software has performed, if its environment has so drastically changed, then the software may need to be decommissioned. Therefore it is vital to understand the difference between the volatility of the code and the environment.

Hint: Before any new project begins, have that discussion on how long the system and environment are expected to last and how much money is being set aside to keep it alive for that period of time. Also, keep the best documentation (software and environment) around just in case decommissioning does not occur as you had planned.

CHALLENGE 8: TESTING STOPPAGE CRITERIA

Testing is a huge cost: schedule slippages, failure to detect obvious faults, brand deterioration, and so on, are hurdles. You can test forever in theory to avoid several of these hurdles. However the real challenge is how to decide when to halt testing—that is, what is a reasonable testing stoppage criteria?

The underlying challenge is to develop metrics and models that define when testing should stop, based on criteria other than money and time constraints. Metrics and models should be scientific and mathematical. Recognize that the determination for when to stop testing safety-critical software is based on regulatory standards, but those standards are not applied commonly for general-purpose software due to large monetary and time costs.

Note that testing to determine when to stop testing is not the only rationale for performing testing. Testing also has a different issue, more related to structural unit testing, which requires its own test stoppage criteria. This occurs when a coverage metric such as branch coverage is mandated, or when a human or tool is simply unable to generate a test case to cover a branch. If the branch is truly unreachable, the problem is solved; if not, are you going to give up and stop?

Hint: Testing planning should occur during development and requirements planning. Make sure someone from the testing department is there on day one. Once there, continually explain if you think they are inching toward an inevitable untestable system. Be assertive and teach them what they need to modify to lessen that likelihood.

CHALLENGE 9: INTEROPERABILITY AND COMPOSABILITY

For this challenge, we must go back and reconsider Challenge 1. That challenge dealt with defining software quality. To compose two software units, it is essential to understand the notion of emergent behaviors. All of the attributes mentioned in Challenge 1 are partially or totally emergent once composed with other components that retain their own individual emergent behaviors.

Therefore, the fusion of different components into a single system must, at the least, consider for a single component

- the unique software version,
- the unique hardware that it resides in,
- the operational environment,
- the insider or outsider threat space,
- the policies and procedures to be enforced,
- the quality attributes it was developed to exhibit as behaviors, and
- time.

If you look at those seven entities for a single component, and then connect to a different component with its own different variations on each of those seven, you have a hard challenge in predicting how they will behave as two become one.

Also, consider the role that the interface(s) play in this union of the two. They can be as unreliable and damaging as the components being joined. As we have waited for years, we still wait for a solid, scalable theory to disambiguate such interconnections; we do not wish to return to the days of interface "spaghetti" logic.

Hint: Design for as much adaptability in components as is possible. Certain assumptions concerning functionality are not malleable; however, there are aspects that are more closely related to the external world and are not part of the core functionality. They can be designed for reuse with reduced amounts of rework when they are joined to other code components.

CHALLENGE 10: OPERATIONAL PROFILES

The problem with this challenge is that we have a difficult time agreeing upon what is the real operational profile. ("Toward a More Reliable Theory of Software Reliability," J. Whittacker and J. Voas, *Computer*, vol. 33, no. 12, 2000, pp. 36-42; http://doi.ieeecomputersociety.org/10.1109/2.889091). Most literature looks at this problem as the probability that a particular input vector is selected. There is nothing wrong with that perspective, but you still must determine, what a vector is.

The bottom line here is that software has many invisible users that also affect its executable behavior. The challenge is to identify who those invisible users are to better predict the software's operational reliability and behavior. And any forward progress in addressing this challenge will also be beneficial to creating an international standard for software product certification.

If you cannot conceptualize some aspects of the resulting target environment(s), then you are not likely to create a product that will work in many of them.

Another point deals with the question, When in the life cycle should we be concerned with nailing down our expectations on the operational profile and the environment? For example, should we first fix the limitations on the target environment, and then look at what software to build for the environment, or vice versa? Or is there a happy medium here?

Hint: If you cannot conceptualize some aspects of the resulting target environment(s), then you are not likely to create a product that will work in many of them. Pretend you were going to build an outhouse outside a residence. What if you came to find out later that the occupant was an elephant? You need to get full disclosure as to what the environment will be as soon as the information is available, and voice concerns if needed.

CHALLENGE 11: DESIGNING IN

This challenge addresses designing in the attributes of software quality mentioned in Challenge 1. These attributes are desirable, however not all are achievable. Some are emergent, and some conflict, such as security and performance. Also, there are cost trade-offs for each—that is, you spend more money on attribute *x* and then have less for *y*. Therefore, one of the toughest of the 13 grand challenges deals with the attributes' technical and economic trade-offs that result in software quality. There is also another challenge here: How do you build in the attributes from the beginning of development and, in particular, build in the ones that cannot be quantified?

Hint: This is easier said than done, but by asking these questions to a client early, you will at least force them to think about things like, Do I care more about availability than security? Do I care if the system has a mean-time-to-repair of two days?

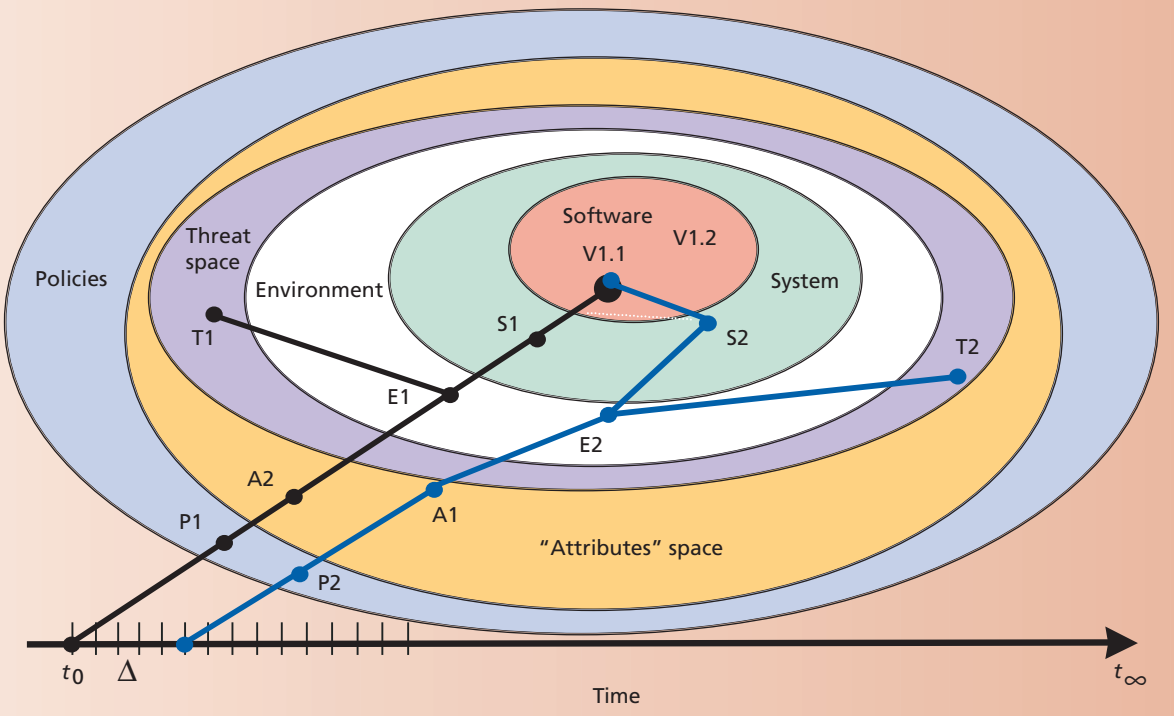
CHALLENGE 12: PRODUCT CERTIFICATION

In the physical world, certifying physical attributes of physical entities is not so difficult. In the virtual/software community, certification becomes a different challenge.

The challenge is how to certify a nonphysical entity; it retains observable behavioral aspects, but they are highly influenced by the environment, forcing them to be emergent, and appear erratic, thus making predictions suspicious. Nonemergent qualities of nonphysical systems can be assessed statically, but those are of lesser value than dynamic ones.

Hint: To date, the only real tool we have is testing. Try off-nominal testing and testing on the boundaries. We have no official government or commercial lab that performs true product certification, and so here you are pretty much on your own.

Figure 1. A grand challenge of software engineering is centered around nonphysical offerings, or services.



CHALLENGE 13: SERVICES

The challenge that I wish to leave you with is shown in Figure 1. What we have is a simplistic view of why we have little or no trust in software-centric systems today.

Today, the notion of services is becoming a central theme in all aspects of life. While the days of bricks and mortar will always be with us, concerns about buildings collapsing and bridges falling down are far from what enters most minds.

The question then becomes something analogous to peeling an onion. With the notion that a nonphysical offering is in essence a service, the physical versus nonphysical is a grand challenge.

So what about services? Is there a calculus for how to manage trust for services? Or do all the same ideas for software apply?

Thus, the final challenge is a simplistic view as to why we have little or no trust in software-centric systems today.

Let's begin by unpeeling this onion in the figure to see what's inside. The core of the onion is the software, version 1.1, and the next layer is the hardware/system shell.

To fuel the software and hardware, external stimuli from the environment must trigger the software and the hardware so that execution occurs. At that point, you have a system that is operating under whatever assumptions were wired into the software and hardware, and selected from the environment. Normally these selections will be the more likely (higher probability) environmental factors that are expected to occur with some quantifiable frequency.

However, expected environments can be attacked from malicious environments. This is shown in the threat space layer in Figure 1. Note that the threats in that space are not always malicious and can simply be off-nominal events that do not occur frequently enough to be placed into the environment layer.

As we continue to peel the onion, we find that the four previously mentioned layers have not dealt with quality of service issues such as reliability, performance, safety, availability, and fault tolerance. Those too are attributes of the entire system that enhance or detract from the system's overall behavior.

It is also necessary to add a layer—policies—describing how the system is to be run, operated, accessed and controlled, and governed. The decisions within this layer come from upper management (CIOs, CEO, security architects, and so on) who will speak in a language unique to that layer. They rely on technical staff to make the translations from policies to attributes, threats, environment, hardware, and software. In short, there is a chain of command by which communication propagation occurs,

however not all propagation from the policy space needs to be unidirectional.

But the most interesting part of this problem of nonphysical systems is not the communication bridges between layers. The most interesting challenge is time. Time is the one part of this model that no one can freeze (and probably the threat space is a close second to lack controllability). In this figure, we see that at time t_0, we are subjected to a set of policies (P1), a set of attributes (A2), an environment (E1) that is challenged by a threat space (T1), and we live in a hardware world (S1) that operates software version 1.1. However, at time t_5, we are subjected to a set of policies (P2), a set of attributes (A1), an environment (E2) that is challenged by a threat space (T2), and we live in a hardware world (S2) that operates software version 1.1.

The challenge, as shown by the Δ in the figure, is that as time moves we want to know what else moves within a particular layer and how that affects the other layers upstream and downstream. Note that in this simple illustration, I opted to not use a different version of the software even though v1.2 existed, however, I did modify all other aspects of the enterprise.

So this problem, stated as succinctly as possible, is

- understand the dependencies between different members in the same layer,
- determine how two adjacent layers relate, and do so for all layers whether adjacent or not, and
- figure out how to determine a priori what Δ will mean for the entire enterprise based on understanding the lesser Δs.

Hint: Keep this perspective as you continually run across Δ problems in your everyday work. This simple view should better explain where falldowns occur.

In my opinion, the 13 challenges listed here are the hardest and the ones that need the most immediate research attention. They present great opportunities and define a roadmap for future exploration in the software engineering research community. Many researchers are currently working in these 13 areas, but the results have been slow in coming.

Note that I deliberately ignored major problems in software security. Software and computer security are important, but security, in my opinion, is a field unto itself, and these 13 are more software engineering-oriented.

The bottom line is that the easier problems in software engineering have been addressed in past years. This article addresses today's harder ones. Complexity today is so different than in the past. And scalability adds yet another set of worries that I did not delve into here. ■

Jeffrey Voas *is director of systems assurance at SAIC and an SAIC Technical Fellow. Contact him at jeffrey.m.voas@saic.com.*