MANY THANKS FOR the interesting “view from the bridge” in respect to the health of software engineering (F. Shull, “Progression, Regression, or Stasis?” IEEE Software, Jan./Feb. 2014, pp. 4–8), with which I completely concur. I believe we have already reached (exceeded?) the limit of what our engineering learning and tools can reliably deliver, and a step-change is necessary for the future.

Some years ago (Jan. 2008) I delivered a keynote speech to the British Computer Society on the unpreparedness of our industry for the future challenges (http://www.bcs.org/content/conWebDoc/17049). Since that time, we’ve better organized a number of important hurdles that we as an engineering profession need to clear in order to improve our ability to reliably step up our capabilities.

Although the following clearly comes with a UK bias, I know from conversations with my international colleagues that this, for the most part, also relevant elsewhere. In my world, this extends significantly beyond the purely software engineering boundary, but because most delivered mechanical systems today derive their performance or functionality from software, it must extend in some part to the system design and constraints that software and electronics impose, by association, on those systems.

Systems Engineering

Systems engineering is a poorly understood and much maligned skill. For software intensive systems, it has often grown from software engineers who have taken on a wider role, but still leaves them in a position of systems that are insufficiently analyzed, poorly documented, and lack engagement of all stakeholders. What’s more, these systems rely on assumptions borne of legacy systems, or worse, oversimplistic understanding and lack of common language, terminology, and diagramming for consistent, unambiguous, communication of intent (for example, SysML) to component engineers.

The lack of an ability to transfer analysis (requirements) into appropriately synthesized component subsystems (specifications) or disciplines (in embedded-systems terms, I consider these to be the software, electronics and electromechanical systems of sensors and actuators). For example, seamless transformation of SysML to UML for software or to VHDL for electronics also considerably hampers the seamless work flow.

The short-term life of many of our products and their rapid evolution means that a lack of consideration exists for the robustness of required designs in light of component subsystem changes (for example, performance enhancement through-life) or component obsolescence; change of use, or abuse, by users; potential for modification for repurposing; changes to security threats; and frequently, the required enterprise change of bringing the system into operation while replacing legacy systems.

We need mechanisms that can unambiguously and completely relate the
user’s purpose to the component requirements of the system.

**Software Engineering**

There appears to be a lack of understanding that the quality of a system containing software (robustness, reliability, cost of ownership, and so on) is directly related to the quality of the software development process, assuming a rigorous and complete definition of the system’s software components. That those systems are expensive to engineer (for example, aerospace and nuclear standards) and that they

- attempt to reduce the introduction of implementation errors,
- try to remove errors as close as possible to the point of introduction, and
- use as many techniques as possible to detect errors based on the possible mechanisms of introduction,

means that it’s possible to deliver high-fidelity translations of the specifications into software systems. Validation starts early (on simulation, prototypes, and so on) to ensure that we meet the user’s purpose. It must also continue in parallel with all the component implementations and their integration and must not be seen as a serial activity that signifies “product complete.” Verification must be seen as relentlessly removing any errors we introduce, using whatever technology we have at our disposal.

Unfortunately, economics pushes hard against the removal of some of these techniques, which are onerous and lengthy, as does uninformed project management who wish to “see progress” rather than get the specifications right. Add to this the unrealistic expectation that you should use the same rigor of process throughout the entire development cycle (for example, using production standards that make prototypes expensive or inadmissible to be ignored once the requirements have been solicited and validated) and you have a recipe for economic disaster. Yet, if we apply this same thinking to a mature manufacturing or mechanical engineering context, we can see that it’s directly analogous!

We need to mature in our thinking of development cycles, using lessons learned from manufacturing, mechanical engineering, and silicon engineering. At the moment, processes appear to be selected because they’re “common to that industry” or are considered best practices, rather than because they explicitly serve a business purpose.

**Safety and Security**

Socially, we’re becoming ever more dependent on the software-enabled systems around us. Removing Wi-Fi, mobile, satellite, or Internet connectivity today is analogous to removing electricity and would cause widespread disruption. That we have built complex infrastructural systems, with little or no back-up solutions, and that these systems are therefore fragile heightens our sensitivity to both safety and security because we’re dependent on their flawless execution.

Unfortunately, many of these systems rely on legacy components that weren’t written with the existing expectations of functionality and aren’t easily engineered out. (Imagine rearchitecting a city if the electrical, freshwater, or foul-water systems were incapable of supporting a new, highly popular feature.)

Safety is, by component, relatively easy to assure through design by analyzing its componentry for modes of failure and the systematic use or abuse by users. Security
must address ever-changing threats that deliberately (and usually maliciously) exploit weaknesses of the design at component, subsystem, or system levels. Most systems are architected with a defined threat-model that secures against known mechanisms of attack and are therefore intrinsically at risk from newly identified threats.

Security must address ever-changing threats that deliberately (and usually maliciously) exploit weaknesses of the design.

We need better mechanisms for specifying components’ safety and security attributes that we can use to evaluate integrated systems, changes of system use, substitutions, and that enable us to identify and mitigate appropriate risks via other system measures.

Performance
It’s easy to get carried away with component specifications when most information systems require appropriate analysis of considerations including information flow, temporal behavior, error environment, communication delays, traffic scenarios, queuing models, processing throughput, sensor lag, and actuator response times to define the required communication link speeds, bandwidths, processor speeds, memory and bus widths, and transfer bandwidth, pipeline, and cache variations. Most of these attributes are composed of constraints borne of laws of physics and design performance and therefore require system performance to iteratively reevaluate chosen implementation constraints. This means that the work of the system engineer and especially the system architect can’t be completed until the final implementation solution is available and integrated; such facts are often overlooked.

We need annotation mechanisms at the system-architecture level that initially identify performance budgets that are gradually (over the development life) replaced with empirical results, while maintaining relationships that help the system architect understand shortfalls (or over-capacity) to resolving system delivery regarding performance and mechanisms for targeting improvements.

Lack of Engineers
In the UK particularly, this is often blamed on an insufficient numbers of graduating engineers. The reality is that the problem occurs somewhat earlier in the academic food-chain and isn’t unique to software but applies more generally to science, technology, engineering, and (applied) mathematics—the so-called STEM topics.

In UK primary schools, science isn’t an assessed part of the curriculum, which in turn means it’s rarely a primary skill for primary teachers. In turn, this means that these teachers generally have simplistic knowledge and little enthusiasm, for science and engineering topics, which means pupils are uninspired. This then biases their selection of topics for secondary education and ultimately at the university level. Survey results show that unless UK students have parents who are scientists or engineers, it’s unlikely that they’ll select science or engineering topics as their discipline. The fact that engineering in the UK isn’t glamorous or well-paid and has a very poor professional recognition also suppresses its uptake.

The UK primary and secondary education “ICT” curriculum largely teaches how to use PC applications (mainly Microsoft) as examples of software engineering, although there are attempts at redressing this. Some secondary schools offer classes on robotics or simple programming run by enthusiastic teachers. UK universities running computer science, software engineering, or electronics courses often have large remedial programs on intake and appear to vary widely in their output standards. This is hardly good grounding for the future expectation of cyber- and information-enabled, constantly connected, and context-aware living.

We need to inspire youngsters to enter our intellectually stimulating and challenging career world, where technology evolution means constant learning and growth in an industry that now pervades nearly every product. This seems highly unlikely to change and is thus a large future employment prospect. (The near-term shortfall of engineers, globally, should also help with the lackluster compensation!) We also need to raise the bar in educating our graduate engineers and in their initial industrial place-
ments to continue their learning into real applications and endeavor to move them quickly to the system level with solid foundations, good abstract thinking, and superlative communication skills.

Stuart Jobbins
MSc CEng FIET MIEEE
Chief Executive, Sofintsys

Quality Software

FIRST, PERMIT ME to express my gratitude to Forrest Shull for the sacrifices he has made to be an excellent editor in chief for IEEE Software. The entire software community owes him a debt for the high standard to which he has created and holds the content of the magazine. Thank you!

This message is in response to the From the Editor column in the Jan./Feb. 2014 issue. I don’t know objectively whether we are progressing, regressing, or standing still. Within the context of the special issue on software quality, I think the assessment is probably pretty dismal, as nearly all remnants of high-quality software have been removed from the marketplace and any processes that created high quality—in training, in education, and among developers—is also gone. How many courses are there on “preventing software errors”? Especially compared with the number of courses on a specific language, such as JavaScript, that is error-prone by design? Or, in the acquisition community, how many courses are they are on how to buy high-quality software?

From my perspective, a turn came in the mid-1990s. For example, Digital Equipment Corp. wrote some of the highest-quality operating and database systems. In 1998, it was acquired by Compaq, a commodity equipment provider that wrote no software. The year before that, Compaq bought Tandem Computers, a Silicon Valley enterprise famous for its fault-tolerant NonStop hardware and software line that were used, for example, by the New York Stock Exchange. Compaq disbanded both DEC and Tandem, and nowadays it’s very difficult to identify any major software providers that optimize product quality.

In fact, it appears that we have gone the other way, as the major software providers—for example, Microsoft, Google, Oracle, and Nintendo—have horrible product quality. They’re optimizing other things—primarily time to market. They each have incredible technical debt. Organizations like the Software Engineering Institute, NASA Goddard Space Flight Center, and ISO were independently trying to improve software product quality. Which of the enterprises are still in the business of quality improvement and which are not?

To respond to one of Shull’s questions on a deeper level, I, along with many others, wonder whether we’re undertaking assignments for which the complexity exceeds our abilities. Like everyone else, I’m influenced by the last thing I read, and at this time it’s X-Events: The Collapse of Everything by John Casti. The volume was recommended to me by Barry Boehm because he knew of my interest in requisite variety, which can be seen as the matching of how we organize ourselves to achieve a result inside some environment, some part of the real world. So, one gloss on software product quality is that we’ve undertaken such complex operations because we view them as necessary to control or respond to the complexity we see in the world, and our ability to organize ourselves tomatch the world’s complexity is less than what’s required.

Some observe—and I owe this to James Thompson, director of major program support of the Office of the US Assistant Secretary of Defense for Systems Engineering—that governments do indeed acquire high-

Nearly all remnants of high-quality software have been removed from the marketplace.

Stan Rifkin
Master Systems