Software Sustainment or Maintenance?

I work with a person who cringes when he hears the term *software maintenance*. For him, maintenance brings forth images of an angry general asking why he is paying to fix a system that he already paid for and expects to work. As you will read in this month's issue, software maintenance usually involves so much more and perhaps the term *software sustainment* is more descriptive.

As the CrossTalk staff prepared this month's selection of articles, I noticed one software sustainment topic that warrants additional discussion: knowledge retention. In order to sustain software using the techniques discussed in this month's issue, knowledge of the software/system is needed. This required knowledge must be adequately planned in addition to other resources required for adequate logistics support. Basically, someone has to know how the software works. Even if the number of fixes, enhancements, alterations, and other activities are minimal and just a few engineers could conceivably address these, the size and complexity of the system may be quite large, requiring more people to cover the needed knowledge.

I am fortunate to work closely with Dr. Randall Jensen, one of the leaders in the software estimation community. In a recent discussion, he pointed to heuristics from Dr. Barry Boehm in his book, "Software Engineering Economics." These heuristics assign complexity values to various software systems such as operating systems, accounting systems, operational flight programs (OFP), etc. For example, an OFP is assigned a value of 10. This 10 equates to (of all things) boxes of cards that an operator can handle for this system. One can imagine this information is quite old because computer cards are certainly before my time. Yet the statistic seems to have stood the test of time. By allowing 2,000 cards per box, or 2,000 source lines of code (KSLOC) per number, a typical OFP will need a knowledgeable person for every 20 KSLOC (2 KSLOC x 10).

Of course, it is cost prohibitive to have numerous people waiting around to make few alterations to the code. However, these people can spend their excess time supporting other systems that may have oversight from other experts.

There are numerous other sustainment considerations within this month's CrossTalk, beginning with Capers Jones' *Geriatric Issues of Aging Software*. In his article, Jones provides a much more comprehensive discussion of sustainment issues that must be considered when planning a logistics effort. We next share the F-35 Lightning II sustainment approach, including the contracting structure, in *Performance-Based Software Sustainment for the F-35 Lightning II* by Lloyd Huff and George Novak. In our final theme article, *Reference Metrics for Service-Oriented Architectures*, Dr. Yun-Tung Lau suggests including service time, scalability, availability, and reliability while measuring the usefulness of a fielded service-oriented architecture.

We offer two supporting articles this month, beginning with *A Primer on Java Obfuscation* by Stephen Torri, Derek Sanders, Gordon Evans, and Dr. Drew Hamilton. In this article, the authors caution on the attempted use of obfuscation to protect Java code, while adding some suggestions for consideration if Java is truly required in a secure system. Finally, Alison A. Frost and Michael J. Campo discuss defect containment in *Advancing Defect Containment to Quantitative Defect Management*.

Having worked on multiple sustainment efforts, I have experienced the daunting task of understanding the code that is being altered. What would have often been a simple job for a system I was familiar with, became more complicated and time-consuming as I needed to learn about the software before starting any changes. When this need is added to complex changes requiring requirements modeling, contracting, etc., the list of considerations is extensive and must be approached with educated knowledge.

Elizabeth Starrett

Elizabeth Starrett
Publisher