duce systems. It will cut development costs and reduce time-to-market, provided you do it with a clear understanding of the issues involved. If the software team doesn't have in-depth knowledge of the component, the risks involved with using it are quite high. Unfortunately, most teams do not have the luxury of building their knowledge. The resources are usually unavailable to do an adequate Risk-Misfit study, and technical staff are not trained in providing the required data to get management to allocate such resources. After reading this book, software engineers and programmers should be able to present their case for acquiring the necessary competence for the component selection process.

Often, when selecting components, performance and reliability considerations are forgotten in the face of an exciting user interface or a cool feature. Sometimes, those responsible for selecting components like that they are easy to demo, ignoring the critical aspects that are not so readily apparent. The example used to illustrate the Risk-Misfit concept in Chapter 10 lists the highest risk as *Accept degraded performance*, with considerable risk. The example assumes the cost of degraded performance as zero, meaning we can ignore this risk. By refining the Risk-Misfit concept further to quantify the cost associated with *do nothing* repair strategies, we can deal with a more realistic scenario. For example, with the degraded performance, what would be the cost in terms of not being able to meet the performance goals? The Risk-Misfit analysis is compromised if potentially high-risk elements are not clearly understood in terms of their impact.

The first half of the book deals with the unique challenges of using COTS components and is well written. The new issues and proposed ways of dealing with requirements engineering, the architectural abstraction "ensembles," the design notation of "ensemble blackboard," and the risk-driven discovery process are all valuable tools the practitioner can use.

However, among the book's intended audience, only the system architect or the chief engineer with an academic bend of mind will be able to read through the entire book. The case study was a little too long. The project manager or the CTO will not have the patience to read all the chapters, but even if they only get through the first few, they will learn valuable insights into software development using components.

Shantha Mohan is president of Kaveri Inc., a software management and technology consulting company. Contact her at shantha@kavericorp.com.

Business Metrics for Software

Robert C. Larrabee

Practical Software Measurement: Objective Information for Decision Makers *by John McGarry, David Card, Cheryl Jones, Beth Layman, Elizabeth Clark, Joseph Dean, and Fred Hall, Addison-Wesley, 2002, ISBN 0-201-71516-3, 277 pp., US$54.99.*

To many people, the field of software metrics is self-contradictory and confusing. Adding to the confusion, business leaders are currently awash in the return-on-investment fad. Just what are good software measures anyhow, and who should I trust to tell me? Do I implement metrics top-down, or bottom-up? *Practical Software Measurement: Objective Information for Decision Makers* answers these questions in a strategic business context.

More enterprises are beginning to understand that business strategy must determine metric selection.

Strategic implementation

If I am an IT project manager, I might have forgotten what the terms *defect density* and *cyclomatic complexity* mean. I might vaguely remember that these measures are connected to software maintainability, but I'm not sure exactly how anymore. I am less confused about product quality, however. If I instruct my managers to report code quality to me, I don't necessarily care about arcane tech-speak. I do care about time-to-market—I want to know when I can ship the product. More enterprises are beginning to understand that business strategy must determine metric selection. If the measures I look at derive from top-down strategic business goals, then I have insight into progress toward these goals.

The authors have approached this topic from a model-based perspective. Actually, they use two distinct, complementary models. The Measurement Information Model (MIM) is the strategic model guiding selection of appropriate measures. The Measurement Process Model (MPM) is the tactical and business representation; it assists in fine-grained execution of the strategic measurement process. According to the authors, you need both models to accurately and appropriately measure your software project.

Patterns and templates

The top level of the metric development roadmap is the MIM, which tells me what measures I need and why I need them. Like all good models, the MIM can lead me to insight about the measurement problem. Then, I develop the MPM for execution. The

MPM is a Shewhart/Deming (PDCA) process itself. The authors provide an excellent taxonomy for this decomposition process. Finally, at the bottom of this measurement tree are the "measurement constructs"—examples of fill-in-the-blank tables representing various metrics categories (for example, customer feedback, process compliance, effort, and financial and schedule performance). These constructs thoroughly address almost any measurement a project manager could need. This book also offers good boilerplates (examples), so the ramp-up time is minimized.

A systems engineering solution

One of the first "Aha!" insights for systems engineers is the discovery of aggregation and decomposition. By using decomposition, intractable problems become comprehensible. I can solve this huge, intimidating problem if I decompose it properly. Moreover, part of correct decomposition entails traceability: when I decompose down a layer, I must also be able to trace back up a layer. The authors implicitly follow these systems engineering patterns in their metric development roadmap.

Another feature of systems engineering is requirements engineering. The inherent G-Q-M (goal-question-metric) paradigm of *Practical Software Measurement* follows this necessary precept.

This software metrics field has matured substantially in the past 10 years. Earlier works included *Practical Software Measurement for Project Management and Process Improvement* (Prentice Hall, 1992), *Applied Software Measurement* (McGraw Hill, 1996), and *Measuring the Software Process* (Addison-Wesley, 1999). These texts are still valuable, but *Practical Software Measurement* is an excellent business wrapper for them all.

Some of the book's authors work on the ISO 15939 Software Measurement Process standard, and on the Practical Software Measurement project funded by the US Department of Defense (www.psmc.com). This Web site offers information on practical software measurement, including a tool and training to facilitate the processes described in this book.

Robert C. Larrabee works for ARINC Engineering. Contact him at larrabeerc@ieee.org.

Software Radio: RF Engineering's New Era

Dharmendra Lingaiah

Software Radio: A Modern Approach to Radio Engineering *by Jeffrey H. Reed, Prentice Hall, 2002, ISBN 0-13-081158-0, 592 pp., US$92.00.*

Software Radio was written due to the lack of a comprehensive resource for building radios based on digital signal processors (DSPs). This reference book fills this gap in academia and industry by explaining the interaction of key subsystems in software radio architectures.

Hardware to software

Radio frequency engineering traditionally involved using analog and mixed-signal design techniques that let experienced RF engineers work on complex RF circuitry. The inherent high costs of designing, testing, and building these systems enforced specialization in the RF domain. RF engineering was predominantly hardware-based systems consisting of amplifiers, filters, mixers, and oscillators. Because of these reasons, RF engineering mainly attracted a specialized group of engineers, either having broad RF research experience or years of practical analog and mixed-signal design industry experience.

The inherent high costs of designing, testing, and building these systems enforced specialization in the RF domain.

However, radio engineering saw a paradigm shift in the early 1990s when Joe Mitola III of Motorola coined the term *software radio*, meaning reprogrammable or reconfigurable radios. Reprogrammability meant that you could change radio functionality without making changes to the radio hardware. This idea opened up a new era in radio design, in which you could use a single piece of programmed radio hardware to support a wide range of frequencies, air interfaces, and application software.

Generally speaking, programmable hardware meant using a DSP or any programmable microprocessor. So, the author now defines software radio as, "A radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software."

Readers should understand that an upgrade to a conventional radio design doesn't exist. When hardware dominated radio design, you had to abandon the old design and start from scratch.

Now, the focus in radio design has shifted to software, along with improving the hardware component design. Thus, we see that a wonderful new era has arrived that moves radio design from being fixed and hardware-intensive to being multiband, multimodal, and software-intensive.

What engineers need to know

Software Radio covers most RF issues and techniques that engineers

Continued on p. 95