

Resource Reviews

GENERAL KNOWLEDGE, CONDUCT, AND ETHICS

Waltzing with Bears: Managing Risk on Software Projects

Tom DeMarco and Timothy Lister. 2003. Dorset House (<http://www.dorsethouse.com>). 208 pages. ISBN: 0-932633-60. (CSQE Body of Knowledge area: General Knowledge)

*Reviewed by Ralph Young
Ralph.young@northropgrumman.com*

The authors provide a fresh approach to risk management. The typical benefits of risk management are viewed as a needed approach to assume reasonable "worthwhile" risks in order to achieve greater rewards. It should be noted that the authors' experience is primarily one- and two-year projects with 10 or fewer staff members (p. 105).

A valuable contribution of the book is a set of questions that evaluate whether risk management is actually taking place. Readers can use this test to guide their conclusions concerning whether additional investment in risk management is warranted.

Additional wise counsel provided in the book is that the responsibility for risk management needs to be recognized by the party that will have to absorb the costs. Often this is at least one level higher than the project, yet frequently risks are not identified, managed, and mitigated at that level. Another important insight is that risk mitigation steps may cost a fraction of the cost of a failed program, yet most often, high risks are not managed and mitigated. Another insight is that program or project failure may be

jeopardized by the failure of only one of many subprograms or subprojects, and taking the effort to identify those at highest risk and mitigating those risks is wise. Still another important and useful insight is the risk added as a result of the standard industry practice of accepting changes to requirements and adding new requirements after launching other technical work.

As in other areas of software development, management sponsorship and support is critical. If management cannot or will not consider the real risks, this renders risk management ineffective and the project team powerless to manage and mitigate risks.

The authors advocate the practice of applying probability characteristics to estimates, for example: the most likely date for delivery is the beginning of April 2004, but one probably does not want to commit to any date before mid-year, and if he or she wants to offer assurance that the date will be met, promise no earlier than the end of 2004.

They emphasize the importance of looking at the assumptions underlying estimates, noting that most project managers do a poor job identifying tasks that might have to be done. Speaking of assumptions, the authors identify a particular category of assumptions: showstoppers (p. 67). Showstoppers are beyond the responsibility and authority of the project. They need to be addressed by higher levels.

The authors suggest that a good way to energize one's risk management program is to identify approximately 20 problems encountered on a half dozen projects in one's organization over the past few years. Trace each problem back to its root cause and

identify these as risks. The authors identify five core risks of software projects:

- Inherent schedule flaw
- Requirements creep
- Employee turnover
- Specification breakdown
- Poor productivity

They characterize an elapsed time multiplication factor based on varying percentages of uncertainty for four of them (specification breakdown is treated as a binary effort—it either happens or it does not). [Reviewer's note: such risks can be overcome through the process of partnering—see <http://www.ralph.young.net>]. The authors take a dim view of projects that have not taken into account of these five core risks. In addition, they advocate use of a defined process to identify other risks in addition to the core risks.

Ralph Young is the director of software engineering, systems, and process engineering, Defense Enterprise Solutions, Northrop Grumman Information Technology. He is the author of *Effective Requirements Practices* (Addison-Wesley 2001).

SOFTWARE QUALITY MANAGEMENT

Leading Culture Change in Your Software Organization: Delivering Results Early

Rita Chao Hadden. 2003. Management Concepts, Inc. (<http://www.managementconcepts.com>). 400 pages. ISBN: 1-56726-123-X. (CSQE Body of Knowledge area: Software Quality Management)

*Reviewed by
Carolyn Rodda Lincoln
Carolyn.Lincoln@titan.com*

Leading Culture Change in Your Software Organization: Delivering Results Early is a how-to book for those who are beginning the journey of software process improvement. The author has experience in all parts of software development and has learned how to successfully implement best practices. This book is a distillation of Hadden's thoughts on how to exercise effective leadership so that valuable change occurs and improves the position of a software organization in the marketplace. Her advice addresses the four main interest groups: executives, middle managers, practitioners, and customers.

The main part of the book includes seven chapters on best practices and overcoming barriers to culture change. There are also eight appendices with sample artifacts, procedures, a plan, a decision paper, and metrics. Hadden suggests that an organization should start with five practices: 1) work with customers to agree on requirements and perform change control; 2) plan before executing the project; 3) track progress by work products and deliverables (not just milestones) and take corrective action; 4) peer review key work products; and 5) manage configuration items. Those practices should show significant project benefits to help overcome the natural resistance to change.

Most of the book is a list of 65 barriers to software process improvement and methods to overcome them. Examples are "Heroes do not want to give up Firefighting" and "training is viewed as a luxury." For each, Hadden provides recommended actions to address. For the first example, she says to educate practitioners that using best practices will allow them to be a different kind of hero. Her overall point of view on change is that it is valuable because it makes professionals more adaptable and

therefore more successful. This is a great motivator if people can be convinced.

Leading Culture Change in Your Software Organization is a good book for organizations that recognize that something must be done to improve their software development but are not sure what to do. It is also good for those who are already on the journey but seem to be "stuck." The author provides both encouragement and practical advice on what to do next. Many other books address practices that meet specific models such as the Capability Maturity Model Integration. Hadden concentrates on "quick hits" that will provide early success and also address the "people" side of process improvement.

Carolyn Rodda Lincoln is an ASQ-certified quality manager and member of the DC Software Process Improvement Network. She is currently employed as a QA director for Titan Corporation in Fairfax, Va. She holds bachelor's and master's degrees in math.

Interpreting the CMMI: A Process Improvement Approach

Margaret K. Kulpa and Kent A. Johnson. 2003. Auerbach Publications

(<http://www.auerbach-publications.com/home.asp>). 414 pages.

ISBN: 0-8493-1654-5.

(CSQE Body of Knowledge area: Software Quality Management)

Reviewed by Prem Ranganath
prem.ranganath@compware.com

The credibility and value of a book on process improvement is vastly improved when the authors have more than academic experience in implementing a model-based process. Kulpa's role in the authoring and review team for the Capability Maturity Model Integration (CMMI) and Johnson's

experience in implementing CMMI and leading audits is evident in the approach taken by the authors in structuring the content of this book. The book is divided into 20 chapters and several appendices. The appendices include useful templates, examples, and checklists that make this book a portable CMMI toolkit.

This is a book that describes the vast scope of the CMMI by starting with an introductory approach. It defines some basic terminology such as process, models, institutionalization, and using the Goal-Question-Metric (GQM) approach. For readers who are familiar with CMM 1.1, the opening sections may sound redundant, but they certainly provide a good background prior to delving into the details of the CMMI.

Kulpa and Johnson do not take a very prescriptive approach because it does not work when trying to apply a model for process improvement. However, their approach is focused toward problem solving. This makes it easier for readers to not only grasp the nuances of the CMMI but also learn how and when to apply the different practices. A simple example of this approach is Chapter 10. Using a Q&A-driven approach, the authors address the often-debated topic of "Is the CMMI relevant for small organizations?" This style is easy to read and readers can decide to read only those questions that are of concern to their organizations. The authors do a good job of conveying the message of tailoring the CMMI for small organizations in this chapter. As several small and medium organizations still grapple with the idea of using CMM 1.1, a question on the real motivation for an organization to move to CMMI should have been addressed as well.

As expected, each level of the CMMI and its associated key process areas (KPAs) are discussed in great detail. A positive aspect here is that the book also covers the organizational and people-related issues to address on the path to

CMMI. A significant challenge of any process improvement initiative is to rally the support of the people. Therefore, at least three chapters discuss topics related to establishing a process improvement organization, organizing improvement teams, and defining procedures and policies.

Any discussion of the CMMI has to address the process of appraisals. This book has assigned an entire chapter for the different Software Engineering Institute (SEI) approved assessment methods. In Chapter 11, there is considerable discussion on the Standard CMMI Appraisal Method for Process Improvement (SCAMPI), the only CMMI Class-A Appraisal method. This is a very heavy chapter in terms of content. The authors start with a discussion on terms and follow it up with a frequently asked questions section. They thereby make it easy to understand for new and seasoned enthusiasts of the CMMI.

A big surprise comes in the form of two chapters that discuss popular metrics for a CMMI effort and statistical process control (SPC), respectively. The chapter on metrics organizes a collection of metrics for KPAs at each level of the CMMI, thereby simplifying the process of accessing metrics based on relevance.

I was particularly intrigued to find a chapter on SPC in a CMMI book. With the model's renewed focus on the GQM model, a discussion on SPC is certainly useful. Again, experienced CMM/CMMI practitioners may find the chapter on SPC to be too basic. Since metrics have such a significant role in validating the success of an improvement effort, the chapter includes a discussion of control charts. This is just right for improvement teams seeking to implement a formal quantitative approach to tracking progress.

The appendices are an invaluable resource, making this book a handy toolkit for CMMI

planning, implementation, tracking, and assessment. There are several checklists included that are sure to simplify a CMMI implementation. So, whether the reader is a novice to the world of CMMI or if his or her interest is in developing an approach to implement CMMI, this book will be a useful investment.

SOFTWARE ENGINEERING PROCESSES

Lean Software Development: An Agile Toolkit

Mary and Tom Poppendieck. 2003. Addison-Wesley (<http://www.awprofessional.com>). 218 pages.

ISBN: 0-321-15078-3.
(CSQE Body of Knowledge area: Software Engineering Processes)

Reviewed by Scott Duncan
sduncan@tsys.com

This is another good book on agile methods. It provides "a toolkit for translating widely accepted lean principles into effective, agile practices." Mary and Tom Poppendieck state that, while there are manufacturing principles that can be applied to software, we seem to be applying manufacturing principles that manufacturing has long since given up using or applies only in very specific situations. In the process, the Capability Maturity Model (CMM), as well as the PMI certification program, are described as giving "the wrong flavor to a software development program." (In the last chapter, Instructions and Warranty, the authors say to "work with" the CMM, "be wary of" the CMMI, and "be careful with" PMI.) Noting classical manufacturing's emphasis on upfront decision making to avoid costly rework, the authors describe Toyota and Honda as having addressed this problem by

not making "irreversible decisions in the first place; delay[ing] design decisions as long as possible, and when they are made, mak[ing] them with the best possible information to make them correctly...then mak[ing] it [the product] as fast as possible."

After more introductory comments the book is divided into a chapter on each lean principle plus an "Instructions and Warranty" chapter at the end with advice to "look for the balance point of the lean principles":

- Eliminate waste—don't do anything that doesn't add value (as perceived by the customer) to the product, but that doesn't mean throw away all documentation.
- Amplify learning—development is about learning and production, about reducing variation (so lean "development" is not based on the same ideas as lean "production"), but that doesn't mean one keeps changing his or her mind.
- Decide as late as possible—noted previously, but also includes building in "capacity for change," but that doesn't mean procrastinating.
- Deliver as fast as possible—rapid (and iterative/evolutionary) development provides reliable feedback quickly, allowing for delaying of decisions and learning, but that doesn't mean rushing and doing sloppy work.
- Empower the team—they know best what to do "when equipped with the necessary expertise," but that doesn't mean abandoning leadership.
- Build in integrity—perceived (get inside the mind of the customer), conceptual (the product hangs together "as a smooth, cohesive whole"), and (maintain) usefulness over time, but that doesn't mean big, upfront design.
- See the whole—do not allow people to maximize their specific

piece of the work to the detriment of the overall effort (which can be aggravated further in situations where organizations contract with one another), but that doesn't mean ignore the details.

In the "Instructions and Warranty" chapter, the authors address "the right amount" of feature analysis, traceability, and user interaction design. They also discuss resistance (to change) being about "a perceived threat to a largely unconscious belief system," which has "led to success in the past, so it will fight back with many varieties of self-fulfilling prophecies."

Perhaps my description of this book so far has already done that for many of you? Perhaps you already think agile methods are just an excuse for hacking? The authors suggest that one "write design summary documents for maintenance support only after you have finished your coding — otherwise you're going to have to write them twice." What?! Write the design after the code?! Well, it's design "for maintenance support," not implementation, and they certainly say it should be written (that is, it's not an optional activity). Perhaps the fear that such discipline cannot be counted upon and must be enforced in some way is what concerns folks. I would guess that many of the proponents of agile methods believe their approaches require more, not less, development discipline than expected in many organizations, and are not "excuses" for sloppy development behavior.

This book, like so many from the agile methods community, is very helpful in explaining the sense of discipline and development philosophy embraced by that community. It helps explain what agile methods are "getting at." I could spend a lot of time on each chapter. Instead, download some of the authors "toolkit" papers and presentations at

<http://www.poppendieck.com/presentations.htm> for an overview.

Scott Duncan has 30 years of experience in all facets of internal and external product software development with commercial and government organizations. For the last nine years he has been an internal/external consultant helping software organizations achieve international standard registration and other national software quality capability assessment goals.

Code Reading/The Open Source Perspective

Diomidis Spinellis. 2003. Addison-Wesley (<http://www.awprofessional.com>). 494 pages. ISBN: 0-201-79940-5. (CSQE Body of Knowledge areas: Software Engineering Processes)

Reviewed by Ray Schneider
rschneid@bridgewater.edu

Reading seems logically prior to writing. Yet Diomidis Spinellis's book, *Code Reading*, is the first book I have seen that focuses primarily on reading code as a skill central to developing quality software. This is simply a book one has to buy if he or she is serious about becoming a first-class programmer.

Dave Thomas, a coauthor of *The Pragmatic Programmer*, writes in the preface to *Code Reading*: "This book should be included in every programming course and should be on every developer's bookshelf." The author in the introduction credits those who have emphasized the social dimension of writing code and the virtues of writing code to be read. The litany of those who have recommended code reading is a veritable Who's Who of software luminaries. To achieve mastery in any field that requires artistry, one must study the work of masters to acquire equivalent mastery. One must see how something is done well before he or she can have any

hope of doing it better.

Code Reading is organized into 11 chapters that build from relatively elementary topics to more sophisticated ones. It illustrates them with a CD-ROM, packaged with the book, which contains 484 megabytes of open-source code ranging from ACE and Apache to XFree86. Each section concludes with several exercises. A determined reader can use them as the basis of a self-study program, or a teacher can use them to create assignments in a graduate-level course.

The book itself is simply a testimony to the fecundity of the open-source movement. Rather than a book one reads chapter by chapter, this is a book that consists of many deep pools. Readers will want to dive into and thoroughly explore each one. Marginal icons, a black triangle with an exclamation point, warns of elements that are dangerous that one should be wary of. A white box with an imbedded "i" identifies common coding idioms. Figures throughout the book give code fragments, tables, and illustrative diagrams. These help propel the reader deeply into the topics ranging beyond simple code writing to standards, documentation, architecture, and tools.

As readers progress through the book they will find references to the source code in footnotes. This allows the brief discussion in the text to direct the reader to in-situ examples of the use of the idioms, structures, and patterns in real production code.

Coding standards include discussion of file naming and organization, indentation including warnings about the nontransportable aspects of using tabs and spaces, tools that use instructions embedded in comments, and other aspects of code formatting. Naming conventions showcased are drawn from four sources: 1) Java and

Windows coding conventions; 2) Unix and GNU coding standards; 3) methods used in the BSD World of documenting interfaces; and 4) the Hungarian naming notation commonly used in the Windows world in C, C++, and even Visual Basic.

Architecture is not stressed enough in programming education. The 72 pages devoted to it in *Code Reading* will not solve that problem, but it is a great summary of the area covering all the major architectures that are generally distinguished. This includes central repositories and distributed approaches using client-server and blackboard protocols as well as the more general rpc (remote procedure call) methods. Other architectures briefly summarized include data-flow, object-oriented, layered, and event driven. Perhaps more important, there is discussion of the abstractions used to implement such architectures.

No discussion of code reading would be complete without a discussion of the tools and methods used to facilitate effective code reading. Chapter 10 provides an overview of using regular expressions, program editors as code browsers, tools like grep for code searching, and diff for locating code differences. The compiler itself is explored as a way of tapping the source code for important information.

Of course, this is only a quick look at three chapters. *Code Reading* is not a casual read. It is a project or a map to mastery. It will leave readers inspired but also frustrated at how much there is to know before they are masters. Novice and journeyman programmers will find this to be the mother lode, a rich vein of pure ore to mine for mastery. Managers might consider buying this book for their programmers and making a commitment, encouraging them to acquire and learn the methods and

tools featured in the text.

Chapter 11 is titled "A Complete Example." Much of the knowledge gleaned is pulled together here. A realistic miniproject illustrates how code reading and analytical methods can be applied. The illustration produces an added enhancement to the *hsqldb* database, a nontrivial 34,000 lines of Java source code. This step-by-step illustrative example shows how to accomplish such a goal.

This is a book that I hope will create a trend or perhaps a movement toward the development of a deep understanding of creative code mastery. All programmers (even those who are already masters) could benefit from having this book on their bookshelves. It is a tool they can use themselves or support their efforts to mentor others. It is time to begin teaching code reading as an enabling skill leading to programming mastery.

Ray Schneider is a licensed professional engineer in the state of Virginia. He has more than 35 years of product development and applied research and development experience working for government, the defense industry, and small business. He is currently an assistant professor in the Mathematics and Computer Science Department of Bridgewater College in Bridgewater, Va.

Six Sigma Software Development

Christine B. Tayntor. 2003. Auerbach Publishing (<http://www.crcpress.com>). 324 pages. ISBN: 0-8493-1193-4. (CSQE Body of Knowledge area: Software Engineering Processes)

Reviewed by Scott Duncan
sduncan@tsys.com

The first half of this book has little to do with software other than pointing out how "QA and the Capability Maturity Model (CMM) are important," but not as

comprehensive as Six Sigma, though they can be usefully employed along with it. There is a case study that covers the use of Six Sigma, but it uses a manufacturing example. This reinforces the unfortunate impression that Six Sigma isn't for software, which is what the book claims to refute. At the very least, it fails to provide a relevant software example.

The second half of the book moves through the traditional software development life cycle saying what kinds of Six Sigma tools and methods one might apply at different places. This does not provide a clear picture of how Six Sigma addresses software development other than as a set of tools/measures to drop in at various times during the life cycle. One could go to any number of software engineering texts and find the same, if not better, recommendations. But perhaps the most serious objection is that it takes the Six Sigma improvement methodology (DMAIC) and tries to align it with phases in the software life cycle. While it may not be the author's intent, it could suggest to those not already knowledgeable about Six Sigma, that a software development and process improvement life cycle are the same.

Some specific issues from the book supporting the overall impression:

- It says Six Sigma "differs from previous quality initiatives" because it focuses on "prevention rather than correction," but that is not a new idea from my perspective and certainly not new in the software field.
- It implies QA doesn't have a focus on "continuous improvement," "analysis and confirmation of facts before making decisions," or an "emphasis on teamwork" as does Six Sigma. The first is a questionable statement (perhaps derived from classic

manufacturing QA/QC, not software) and the last two certainly are not excluded as possible in QA.

- It says that "Although it can be considered a quality program, CMM is a specialized one"; however, that isn't the real point of the CMM at all (that is, to be a quality program). It is a maturity model for software development practice and has the IDEAL model as a process improvement approach. CMM is also critiqued as not stressing a "teamwork" approach or "fact-based decision making," but the former is questionable, and it's hard for me to swallow the latter when one considers what level 4 is about.
- It says "CMM applies only to the software process," but if that's what the book is about and that is the scope of the effort, then what's the problem?
- It says "CMM does not ensure that the right problem is being addressed" and that is true if one means just that the CMM deals with requirements after a broader systems engineering effort has already allocated the "right" requirements to software. Certainly there is, again, the IDEAL approach to improvement, which would not have to have this limitation.
- It devotes five chapters of case study, presumably to introduce Six Sigma "tools" and the DMAIC model, but, as noted previously, without any software focus.
- It says "although there are no Six Sigma tools specifically designed for system development, some have direct applicability." And that's fine, so why diminish their perceived applicability by making it sound like they have a deficiency to start with in their lack of software-specific design?

- It maps DMAIC phases to a generic (software) life cycle of project initiation (define, measure, analyze), system analysis (define, measure, analyze), system design (analyze), construction (improve), testing and quality assurance (improve), and implementation (improve and control). This seems to me to be an inappropriate joining of a development life cycle with a process improvement one. I think it's done just to set the stage for which Six Sigma (management/improvement) tools the author thinks could be used in a project.
- It lumps QA in with testing, reinforcing the wrong impression of what QA (vs. QC) should be in software. Part of the problem early on (even before the case study), is that the author seems to anticipate objections to applying Six Sigma to software and spends time arguing against them. However, they all seem like straw man arguments that one would have a hard time defeating. Some of the text merely explains to the readers, if they don't already know, what the argument is about, for example, what the Software Engineering Institute's CMM is. So the text seems to assume the people reading are not the one's arguing, but somehow will be faced with those arguments. Perhaps the author's concept of the audience for this book was nonsoftware quality folks who have to deal with implementing Six Sigma in an IT environment and need basic software life-cycle/process issues explained to them. As there is no obvious "who is this book for" statement anywhere, this is just a guess based on the way the book approaches the subject. The last few chapters of the book touch on overviews, without

significant Six Sigma content, of RAD, prototyping and iterative/incremental life cycles, client-server and Web development (and only two pages at that for this topic), packaged software procurement, outsourcing, and a few appendices on templates for a "project charter," functional process maps, process improvement ranking, FMEA analysis, "metric reliability assessment," plus an appendix of terms.

The book spends too much time on nonsoftware matters, lacks depth, and doesn't make it clear who would make best use of the content. Also, it seems to take a largely manufacturing Six Sigma background and overlay it on software rather than properly integrate it. As such, I cannot recommend this book as a good guide to applying Six Sigma to software.

PROGRAM AND PROJECT MANAGEMENT

Architecture-Centric Software Project Management: A Practical Guide

Daniel J. Paulish. 2002.

Addison-Wesley

(<http://www.awprofessional.com>).

280 pages.

ISBN: 0-201-73409-5.

(CSQE Body of Knowledge area: Project Management)

*Reviewed by Pieter Botman
p.botman@ieee.org*

It was a pleasure to briefly put aside the ongoing noise and hype surrounding life-cycle methodologies and consider the more fundamental (and less religious) topic of project management. Daniel Paulish's latest work is a look at project management techniques, strongly driven by software architectural concerns. Paulish has an informal and highly readable style.

Reviews

Discussion of project management techniques from the viewpoint of architectural analysis and development is certainly worthwhile. In major systems development, software architecture presents a major source of risk, and project management is, after all, largely concerned with the management of various types of risk. Architectural concerns appear through the course of any reasonably complex project, and addressing them effectively means paying careful attention at every step or iteration in the life cycle. Fortunately, the software project manager has a range of methodologies, organizational structures, processes, and tools from which to choose.

Where exactly does the typical project manager fit into these various methodologies? Does he or she rely upon a software architect to analyze technical risk, and therefore to drive the detailed development planning? It is ideal to have software engineering knowledge and wisdom embedded in, and applied by, the project manager, sitting at the point in the organization where important decisions balancing functional scope, time, quality, and cost/risk are made. However, in some organizations this may not be the case — the project manager may be given narrow scope, dependent upon the architect, team leaders, and other specialists, to analyze risk and make architectural decisions and technical plans.

Paulish wisely avoids overly broad and simplistic pronouncements concerning methodological and organizational choices, claiming that the project management approach presented is "somewhere between heavyweight processes described within the CMM and the RUP and lightweight processes such as extreme programming."

I found Paulish's approach to be

significantly closer to the plan-driven side of the spectrum. While it would not be appropriate to revisit the entire plan-driven vs. agile debate here, it seems that his approach is consistent with a structured, CMM-oriented planning and control approach to project management, and appropriate for projects of considerable size, complexity, and technical risk.

Paulish draws upon his experiences in "mid-sized" projects, typically with 20–200 staff members and lasting a year or more. Much of his project experience comes from his work at Siemens, clearly a plan-driven organization with an engineering culture. His project management practices assume a sharing of responsibility (and close collaboration) between the software architect and project manager. Both are engaged in development of requirements, from different perspectives, and while the architecture itself remains the ultimate responsibility of the architect, the project manager participates in reviews and drives the planning that flows from that architecture.

The book does not follow conventional project management texts in the selection, or in the relative weights, of topics. After opening the book with an explanation of his core beliefs and background, the author provides a section for each of the four "primary activities": planning, organizing, implementing, and measuring.

Paulish uses the software architecture and design work of Hofmeister, Nord, and Soni in *Applied Software Architecture* as the basis for his rationale, descriptions, and terminology. I was surprised by the overuse of the term "component" in nonqualified form (that is, conceptual components, source components, deployment components). Project managers are of course not architects, and need

clearly defined and universally accepted terminology.

The planning section contains a strong explanation of the numerous linkages between software architecture and project management. Paulish presents many good software project planning practices, including an overview of estimation tools and techniques. He stresses repeatedly that while project planning continues in parallel with the development of the high-level design (architecture), reliable estimates and project plans can only be presented upon completion of the high-level design. His discussion of the "global analysis" (requirements analysis) process is perhaps his best illustration of the way in which a project manager and a software architect need to work together.

The organizing section dwells upon some of the social and cultural issues in project management, including team structure and roles, promotion of positive cultural values, and team building. The chapter on global development is especially interesting, as it addresses the challenges of geographically dispersed teams and multicultural teams.

The implementing and measuring sections return to the nuts-and-bolts aspects of project control. These are woven into chapters with themes such as "trade-offs and project decisions," "incremental development," and "creating visibility and avoiding surprises." While he touches on or introduces many basic concepts in project tracking, code and effort metrics, and quality metrics (defects per lines of code), Paulish doesn't go into great depth in terms of detailed software practices or classic project management theory.

A broad and informal approach is certainly acceptable in a book on software project management — no single text could hope to present all the interrelated aspects of software

engineering and project management. And, Paulish has chosen to use this book to expand upon and emphasize the architecture-driven aspects of project management.

Paulish's experiences at Siemens allow him to put forward many useful insights and illustrative anecdotes. His discussions on cultural and social issues are given significant emphasis, and, in fact, are perhaps the strongest aspect of the book. He also has pointed suggestions for project managers dealing with some highly charged political situations (arguing with senior management about estimates, schedules, and priorities).

Paulish closes the book by saying that he is embarking on another very large (1 MLOC) project within Siemens, that he intends to apply his project management techniques on this project, and hopes to report on this endeavor. Realistic, detailed, public case studies are valuable to all software engineers.

The book reinforces good project management thinking, and is certainly worthwhile as additional reading for all project managers, even if only for its practical case studies. Those needing more depth in some areas might wish to pursue more specialized and detailed books on project management.

Pieter Botman is an independent consultant with more than 20 years of experience assisting companies in the areas of software process assessment/improvement, project management, quality management, and product management. Botman is a professional engineer registered in the province of British Columbia.

Fast Forms for Managing Software Projects

Kathleen Demery, Monica Lusk. 2001. Management Concepts, Inc.

(<http://www.managementconcepts.com>). 539 pages.

ISBN: 1-56726-095-0.

(CSQE Body of Knowledge area: Program and Project Management)

Reviewed by
Carolyn Rodda Lincoln
Carolyn.Lincoln@titan.com

Fast Forms for Managing Software Projects is not what one would think of as a book. It is a three-ring binder of forms that a person can use for managing software projects. If one needs to create a checklist or a log, this is a good place to start.

There is a very short introduction, and then the rest of the binder is a print out of Word and Excel forms. The forms are grouped by subject. There are seven forms for project initiation, 22 for planning, 10 for execution and control, 11 for close-out, and five for general use. There are also tables showing how the forms map to the Project Management Book of Knowledge (PMBOK) and the Capability Maturity Model (CMM). Since the PMBOK focuses on project management, the PMBOK is covered fairly well; there are at least eight forms for each area of the PMBOK. The CMM is a much broader model, but the forms do address software project planning, software project tracking and oversight, and software subcontract management. The information for each form includes the purpose, background, references to the PMBOK and CMM, instructions, a blank form, and a sample. It also comes with a CD-ROM containing software to install the forms on a workstation or network as Microsoft Office templates.

Fast Forms for Managing Software Projects is a good resource for new project managers who want to follow good processes but have no examples to follow. The forms are also useful if the organization has a standard set of forms, but the particular one needed is not available. The PMBOK and/or CMM can provide models for what to do, and the book gives concrete examples of how to do it. The biggest drawback to this book is that the forms are for software only and map to CMM rather than the Capability Maturity Model Integration (CMMI). Any organization that is beginning its process improvement journey at this point will probably be using CMMI. Other than that, this book can help an organization get past the initial problems of needing to document project management and not knowing how to do it.

Modernizing Legacy Systems

Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. 2003. Addison-Wesley

(<http://www.awprofessional.com>). 334 pages.

ISBN: 0-321-11884-7.

(CSQE Body of Knowledge area: Project Management)

Reviewed by Scott Duncan
sduncan@tsys.com

Whenever I read a book about maintaining (older) systems, I inevitably recall the Lehman and Belady book, *Program Evolution* (1985). The authors of *Modernizing Legacy Systems* include it in their bibliography and a quote from Manny Lehman recommending this new book appears on its back cover. *Program Evolution* addresses the need to preserve architectural/design integrity when maintaining systems. *Modernizing Legacy Systems* notes that "legacy" seems to be applied to systems "when they begin to resist

modification and evolution," that is, when maintenance becomes increasingly harder to perform.

Modernizing Legacy Systems' subtitle is *Software Technologies, Engineering Processes, and Business Practices*, and the authors state that the book is intended to help someone:

- Decide whether a modernization or replacement effort is justified.
- Develop an understanding of the legacy system.
- Gain an understanding of, and evaluate the applicability of, information system technologies that can be used in the modernization.
- Involve the stakeholders...and reconcile their conflicting needs.
- Identify architecture's role in modernization.
- Estimate the cost of modernization.
- Evaluate and select a modernization strategy.
- Develop a detailed modernization plan.

The book begins by identifying factors that contribute to modernization project failure and by discussing various system evolution and reengineering topics. The bulk of the book uses a case study (called "the Beast") to explore these factors and topics. It goes into considerable detail regarding specific technical decisions made and specific technologies used in the transition.

The final chapter, "Recommendations," attempts to generalize from the case study to some principles (with my abbreviated characterization of the principle as I read the material):

- Find a better way (don't replace one legacy system with another because it seems like the easy way to go).
- Use commercial components (referencing to the Software Engineering Institute's COTS-based Initiative).

- Manage complexity (not much said on this at this point except to do it, that is, manage complexity).
- Develop and deploy incrementally (presumably to reduce risk and exposure).
- Software engineering skills (train people with old skills in new ones).
- Component-centric approach (what one cannot buy build for reuse).
- Architecture-centric approach (as the authors say "architecture can be used to encourage and enforce a component-centric approach").
- High levels of concurrent development (which the authors say is "possible only in a highly structured, mature environment").
- Continuous integration (to reduce risk by frequent checks that things can be made to work together at some level through the project).
- Risk-managed development (like complexity, one has to manage risk).

While the case study is admirable work and real-life examples are always good, the case study seems to take over the book and the principles at the end seem rather thin in that they cannot be argued with much. And, there is little guidance in applying them if the case study does not match one's situation. Indeed, the authors say in an 11th recommendation, it's "but one example, and the specifics of how this project was planned and managed may not be appropriate in your modernization effort."

SOFTWARE METRICS, MEASUREMENT, AND ANALYTICAL METHODS

Software Engineering Measurement

John C. Munson. 2003.
Auerbach Publishing
(<http://www.auerbach-publications.com>). 449 pages.
ISBN: 0-8493-1503-4.
(CSQE Body of Knowledge
area: Metrics and
Measurement)

Reviewed by Scott Duncan
sduncan@tsys.com

For those who are serious about measurement of software (products, processes, environments, and, even, people), this book provides a very rigorous coverage. I first encountered the work of John Munson more than a dozen years ago (along with his colleague Taghi Khoshgoftaar) while investigating work on defect identification using complexity analysis. At that time, Munson was pursuing large collections of data against which he applied significant statistical analysis. This book summarizes what he has learned about software measurement using such an approach over many years of research and states what is wrong about current (and historical) measurement practice in software.

In particular, Munson believes most measurement practice (and related standards work) is devoid of valid and reliable scientific method. He mentions, several times, that there is no way for researchers to communicate clearly using external measurement standards because there are no recognized standards for software measurement. ("NIST does not maintain any standards for software measurement; nor, for that matter, does any other national standards organization," and "NIST is not motivated to

establish standards for software measurement.") He also finds many attempts at standards definition wanting (for example, IEEE 982.1 - Dictionary of Measures to Produce Reliable Software, function point counting, and the general use of KLOC).

For example, Munson believes too much measurement effort (or the wrong kind) is expended under ex post facto conditions. That is, after data have been collected, people start looking at them to "see what they can find out" but, usually, without any reasonable statistical approach. While he notes that such "data can be very useful for data exploration...for examining sources of variation," he states this is an "intuitive," not a "scientific," approach and cannot result in "data to be analyzed and reported as scientific evidence." Among the problems with this sort of data is that the conditions for their collection are often not known, so one cannot say how valid or reliable they are. Munson says that if one is "beating the data (hard enough) with a statistical stick...you are almost certain to find that for which you are looking."

From a quality perspective, Munson says there are five simple software quality objectives:

- Learn to measure accurately people, process, products, and environments.
- Learn to do the necessary science to reveal how these domains interact.
- Institutionalize the process of learning from past mistakes.
- Institutionalize the process of learning from past successes.
- Institutionalize the measurement improvement process.

Regarding the first topic, Munson states that measurement of people "is one of the most important things we can learn to do in understanding the software

development process;" however, the "typical software practitioner simply does not have the educational background in psychology or sociology to attempt this very difficult feat." Since people create software and "we know so little about how they do it," that makes measurement of people important. But "our knowledge of [people] attributes is so weak" that the "very best we can do at the present time is to build good models that control for the effects of people."

On the other topics, he says one can have significant success in affecting quality results. Product measurement deals "primarily with cost and rate data," for example, fault insertion, change requests, productivity, and process improvement. Product data (for example, from requirements specifications, high-/low-level design, source code, test cases, and documentation) has a lot to do with recording change and when change occurs. Environment measurements include operating system, development and operating environments; machine (software) stability; physical workplace (for example, privacy, interruptions, access to information); and administrative stability (turnover).

After covering a variety of topics in the first two to three chapters regarding many of the problems Munson sees in current practice, the rest of the book deals with (the definition and use of):

- Validation of software measurements (different forms of validity and reliability)
- Static software measurement (pros and cons of Halstead "software science" and McCabe "cyclomatic complexity" measures)
- Derived software measures ("common sense [is] thrown to the winds in just about every derived software metric")

- Modeling with metrics (statistical techniques such as regression analysis, nonlinear models, multicollinearity, and canonical correlation)
 - Measuring software evolution (using static measurement data primitives to compare systems across builds and introducing fault measurement)
 - Software specification and design (setting up for the next chapter on dynamic measurements by understanding "just what we are measuring," that is, the requirements and design)
 - Dynamic software measurement (which is "a very expensive proposition," though it can help one "learn exactly where a system is fragile")
 - The measurement of software testing activity (to remove "the ambiguity in the goals and objectives of the software testing process")
 - Software availability (reliability models and estimation, availability, security, and maintainability)
 - Implementing a software measurement plan (building a measurement process, including a reporting system)
 - Implementing a software research plan (learning to "know" things about software development that are not "purely conjectural")
- Munson also provides two appendices:
- Review of mathematical fundamentals (matrix algebra, probability, discrete and continuous probability distributions, statistics, tests of hypotheses, and modeling)
 - A standard for the measurement of C programming language attributes (an offering to the industry that people can use in a reproducible fashion so data collected and reported can be verified publicly).

Reviews

For those who want a rigorous approach to measurement, this book would be an excellent choice.

Five Core Metrics—The Intelligence Behind Successful Software Management

Lawrence Putnam and Ware Myers. Dorset House Publishing (<http://www.dorsethouse.com>). 310 pages. ISBN: 0-932633-55-2. (CSQE Body of Knowledge areas: Software Metrics, Measurement, and Analytical Methods)

*Reviewed by Tim Kelleher
tkelleher@ilrd.com*

This book provides some information relevant to Section V (Software Metrics, Measurement, and Analytical Methods) of the CSQE Body of Knowledge. Considering this, I will critique the book against each section: Metrics/Measurement Theory, Process and Product Measurement, and Analytical Techniques.

The book contains a very good explanation for what the authors believe are the core metrics to monitor to improve software management. The book preaches size, productivity, time, effort, and reliability. These are the five core metrics that the book's title refers to. The book did provide many graphs, tables, and formulas to explain the different metrics and their relationship to one another. These were very positive to convey the authors' ideas.

The book touches on how metrics affect the psychology of people on the software development team, but it is not detailed enough. People are the most important resource for success and their emotional health is key to success. My favorite part of the book is the

recommendation to hire good people and retain them. However, the book does not emphasize the flip side, which is to remove the weak at the earliest identification. Weak teammates demotivate the strong employees.

The book does a very good job explaining software process and discussing different approaches to improving software development. In addition, it explains some of the downsides of software improvement programs. I especially like the opinions about the Capability Maturity Model.

A part of the book that could be improved would be to add more "how to" examples, such as detailed case studies and recipes for success. This approach may open new views and ideas to help the authors apply new metrics to the evolving software development methods.

This book could be useful for members of an organization that already employ these techniques. It would provide additional views and points of interest that would deepen their understanding for their metrics collection program. I would not recommend this book for any new organization or for individuals applying the latest software development methods of extreme programming or adaptive software development. If these new methods continue to expand in applications, then this book is obsolete.

I will not add this book to my library. In general, I found it difficult to read, since I cannot envision how this would really help an organization. Also, the author is very long-winded and attempts to cover too many topics. In today's high-tech, fast-paced business, short and direct with examples is better for conveying ideas. I have yet to work for a company that has, nor have I had a discussion with other colleagues who have, a successful metrics program. Most do not have any metrics at all.

Tim Kelleher is director of software development for Instrumentation Laboratory. He holds a bachelor's degree in biomedical engineering from Boston University and a master's degree in electrical engineering from Loyola College.

SOFTWARE VERIFICATION AND VALIDATION

Effective Software Testing, 50 Specific Ways to Improve Your Testing

Elfriede Dustin. 271 pages. Addison-Wesley (<http://www.awprofessional.com>). 271 pages. ISBN: 0201794292. (CSQE Body of Knowledge area: Verification and Validation)

*Reviewed by Hillar Puskar
Hillar.puskar@lmco.com*

This book consists of 50 separate items organized into 10 chapters, each focusing on a different aspect of software testing. It contains a lot of sound advice that is presented in a well-organized and easy-to-read fashion. The overall organization and table of contents covers the test planning effort in a logical order, so it is easy to find the items one is looking for. The reader can pick it up, easily find a chapter of interest, and read it without being overwhelmed. The book has an abundance of good information and it would make an excellent reference for most software test organizations. The author has written this book to be applicable to projects of all sizes and types. Those involved in software testing will find the book useful; I found this book to be at a slightly higher level than the many others and believe that it is a better reference for managers or experienced test engineers.

The material includes process-related and management-related topics. As Dustin points out, factors outside of the scope of testing will have a great impact on how successful the project is.

Nine of the 50 items are not new material—they are adapted or in other ways reused from the author's extensive writings. Dustin is clearly a leader in the field of software testing; this is her third book on this subject.

Subjects covered include best practices, problem areas to avoid, and solutions to problems that one has gotten into. The book gives techniques that let the reader build quality into the software instead of testing at the end for quality. It emphasizes the importance of including testing in all aspects of the software development life cycle. I often found myself nodding in agreement as I read this book.

Coverage of the following items stands out as being unique to this book; these topics can be overlooked on many projects and in other texts:

- Build vs. buy decisions for automation tools
- The importance of the roles and makeup of test team
- How to determine effectiveness of the test team

Although there are footnotes and references in each chapter, the references are not extensive and it is not easy for readers to learn more. Other books have included more references to relevant materials to expand on the topics that have been presented. There is a companion Web site, but it is limited in its usefulness unless one wants to buy a copy of the book. The site does not contain much more than the full table of contents and links to the Web pages for her other books. There is no overall "big picture" wrapup at the end of the book. Also, there is no obvious emphasis—are all 50 items equally important?

As a collection of items based on the author's experiences, *Effective Software Testing* is similar to *Lessons Learned in Software Testing*, which was reviewed in the March 2003 edition of *SQP*. I liked

both books enough that I purchased copies of them for my group. However, I end up grabbing *Lessons Learned in Software Testing* more often as a quick reference.

Hillar Puskar is a systems engineer with Lockheed Martin in King of Prussia, Pa. He has a bachelor's degree in industrial engineering from Lehigh University, and a master's degree in computer science from Stevens Institute of Technology.

Systematic Software Testing

Rick D. Craig and Stefan P. Jaskiel. 2002. Artech House Publishers (<http://www.artechhouse.com>). 536 pages.

ISBN: 1580535089.
(CSQE Body of Knowledge areas: General Knowledge, Conduct, and Ethics)

Reviewed by Noreen Dertinger
noreen.dertinger@cognos.com

As the title implies, *Systematic Software Testing* by Rick Craig and Stefan Jaskiel presents an in-depth, end-to-end compendium of software testing. This book is based mainly on IEEE software testing related standards (829-1998 Standard for Software Test Documentation and 610.12.1990 Glossary of Terms), *The Systematic Test and Evaluation Process (STEP™): An Introduction and Summary Guide*, as well as many other references and the authors' personal experiences.

This book has grown out of suggestions from their clients that Craig and Jaskiel write a book about their processes and methods for testing software. Their processes are based upon the STEP™ methodology, which was originally developed in 1985 by Bill Hetzel and David Gelperin as an approach to implementing the IEEE-829-1983 Standard for Test Documentation. These methodologies have been updated based on the latest versions

of the standards, 828-1998 Standard for Test Documentation and 1008-1987 Standard for Unit Testing. STEP™ does not depend upon any specific software testing tools. Rather, it provides guidelines that can be customized to meet the needs of individual organizations in order to produce quality software.

The authors have organized *Systematic Software Testing* so that it can be read from beginning to end, or any chapter can be read on its own for quick reference. Beginning with an overview of the testing process and the STEP™ methodology, this book goes on to cover key components of a testing strategy, such as risk analysis, test planning, analysis and design, test implementation, and test execution. The authors also deal with various aspects of the software test organization (including topics such as different types of test teams, office space, and allocation of work time and office space). They describe characteristics of good testers, hiring, retention of staff, and certification. On the test management side they explain various managerial roles, management vs. leadership, metrics, and more.

Systematic Software Testing concludes with chapters on improving the testing process (such as identifying areas for improvement and successfully implementing changes, overview of ISO certification, the Capability Maturity Model (CMM) and Test Process Improvement (TPI)), and final thoughts on initiatives that the authors believe every software testing organization should strive to undertake as a next step.

Written in a clear and flowing style that uses everyday language instead of jargon, *Systematic Software Testing* is easy to read and should be comprehensible to all levels of testers, managers, and developers, as well as anyone interacting with professionals

involved in the testing process. The case studies, based on Rick Craig's real-life experiences as well as experiences encountered by the authors' clients, do an excellent job of illustrating the material being discussed and enable the reader to understand why certain processes worked and others failed. Rather than attempting to provide verbose coverage of the topics at hand, the authors strive to provide enough of an introduction in a compact format to get readers started. They do this by recommending procedures and many supporting examples. They provide hints that allow readers to continue with more detailed research into the areas of their interest.

In summary, *Systematic Software Testing* will provide those looking to develop a well-defined testing process with a template and procedures to do just that. Readers whose companies already have well-defined testing processes will undoubtedly also be able to extract useful information to help them fine-tune their testing processes.

Noreen Dertinger earned a master's degree from the University of Ottawa, a certificate in information technology from the University of Victoria, and completed her CMMI certification. She has 15 years of experience in the software industry in development, configuration management, and quality assurance. Dertinger is a software quality control analyst with Cognos in Ottawa, Canada, working on Cognos ReportNet.

SOFTWARE CONFIGURATION MANAGEMENT

Configuration Management Principles and Practice

Anne Mette Jonassen Hass.

2003. Reading, Mass.:

Addison-Wesley

(<http://www.awprofessional.com>).

370 pages.

ISBN: 0-321-11766-2.

(CSQE Body of Knowledge area: Software Configuration Management)

*Reviewed by Carolyn Rodda
Lincoln*

Carolyn.Lincoln@titan.com

Configuration Management Principles and Practice is a book about configuration management (CM) in the Addison-Wesley Agile Software Development Series. CM and agile – is that an oxymoron? The author, wants to convince readers otherwise. The main message of the book is that CM is crucial in many areas of software development, but the degree of formalism (cost) needs to match its benefit. It has to be done, but it can be informal, delegated, and/or automated.

The author's purpose is to provide a broad view of CM without being a primer. She covers most CM topics without going into depth on any one. The book has five parts: 1) What is CM? 2) CM data, 3) Roles in CM, 4) CM in Practice, and 5) Improving CM. To emphasize that CM can be done in many ways, she gives examples with a high degree of formalism and examples with a low degree of formalism. She also

follows a four-part view of CM: identification, storage, change control, and status reporting. Interestingly, she leaves out auditing, which is usually considered one of the four. She considers auditing part of quality assurance and does not address it.

Since Hass is a process consultant, her point of view is heavily process oriented. The book includes mappings for the major process models to the chapters: Capability Maturity Model (CMM), Capability Maturity Model Integration (CMMI), BOOTSTRAP, and ISO 9001. Part 5 provides chapters on how to meet the CM requirements for capability levels 1-5 of the CMMI. Other major topics include CM tools and CM considerations for different software development life cycles such as agile and iterative. Despite being part of the Agile Series, the author covers many life cycles; she even mentions the formal CM required for safety-critical systems.

Configuration Management Principles and Practice accomplishes the author's purpose of providing an overview of the field, so it is particularly good for those who need to understand the basics of CM, such as developers and managers. It would also be beneficial as an introduction for a newly appointed CM person. However, he or she would need additional information and training to be able to perform his or her job. There are very few books available on CM, so this book is a welcome addition to the literature for this body of knowledge primary area.