

# Resource **Reviews**

## GENERAL KNOWLEDGE

### **Commonsense Reasoning**

Eric T. Mueller.

2006. Morgan Kaufmann

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

432 pages.

ISBN: 0-123-69388-8

CQSE Body of Knowledge  
area: General Knowledge

*Review by Robert W. Ferguson*  
[rwferguson@gmail.com](mailto:rwferguson@gmail.com)

One of the most difficult problems with the development of autonomous robots centers on the term "commonsense reasoning." This subject belongs to the larger domain called artificial intelligence, but that domain is now so large that one must focus on narrower subjects. The fundamental ideas of commonsense are based on everyday experience. For example, if one sees a colleague walk into his office (which has no other doorway), then one knows, by following him into the office, he will be there. The everyday examples of such logic are numerous and support much daily activity; however, commonsense is not easily codified. Mueller's book is a thorough description of the research in the formal logic approach to the problem.

The core of the approach is a representation called "the event calculus." Traditional logic learned in geometry class cannot address this type of problem. For one thing, traditional logic assumes no time-dependent context. Another problem with traditional logic is that the only allowable values are "true" and "false." People sometimes make decisions based on other values such as "better" and "worse."

The author defines commonsense reasoning in the first paragraph of the book. It is a process that takes information about a particular scenario and then makes inferences about other aspects of this scenario based on general knowledge of how the world operates. The core problems of the research are readily identifiable in the definition. One must characterize "knowledge" of "how the world works," "scenarios," "information about scenarios" and "inferences about events." A simple example of the formalism is the commonsense law of inertia, which essentially says that things will stay the same unless they are affected by an event. The inertia law, however, is often violated so there is a set of formal axioms that describe how a "fluent" may be released from the law of inertia. A "fluent" is a property in the world that varies with time.

This is a very formal book, suitable for researchers but not for casual readers. The author is patient and thorough, however, so the book is accessible to amateurs.

After reading about the subject it seems that the formal logic approach is going to meet a number of practical limitations. The number of statements used to support knowledge about the world would be extraordinary. In fact, in the discussion of a formal approach to emotions, Mueller suggests the use of 78 axioms. This leads one to believe that people do not use a formal structure for commonsense reasoning. Rather, they use some form of experiential patterns. People with better commonsense than others are better observers of context and have more patterns that are more readily accessible.

Nevertheless, commonsense reasoning is an important area of

study. Mueller's book will be valuable to those involved in this field.

Robert W. Ferguson is a senior member of the technical staff at the Software Engineering Institute where he works in software engineering measurement and analysis. He has a master's degree in mathematics and about 30 years of experience in software development and project management.

## SOFTWARE QUALITY MANAGEMENT

### **Improving Almost Anything: Ideas and Essay**

George Box and Friends.

2006. John Wiley & Sons, Inc.

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

598 pages.

ISBN: 0-471-72755-5

CSQE Body of Knowledge  
area: Software Quality  
Management

*Reviewed by Jayesh G. Dalal*  
[jgdalal@gmail.com](mailto:jgdalal@gmail.com)

When this book was received the *SQP* book reviewers were asked if it should be reviewed. As evidenced from the following comments, the support for the book was lukewarm. My reasons for going ahead with the review are simple. In the last few years Six Sigma methodology has gained considerable popularity and *SQP* has published several articles describing its application. In those articles, however, statistical tools and related material are not well presented. The author, George Box, presents statistical methods in a way that almost anyone can understand and appreciate, I thought that a review of this book would be of value to *SQP* readers, and I have no

reason to change my mind after reading it.

This book is a revised and updated version of a previous publication, and is based on previously published articles and papers. It is divided into six parts:

- Part A—Some thoughts on process and quality improvement
- Part B—Design of experiments for process improvement
- Part C—Sequential investigation and discovery
- Part D—Control
- Part E—Variance reduction and robustness
- Part F—Songs

Most *SQP* readers should read all of Part A, since the information is readily applicable to the software domain. The chapter titled, "Scientific Method: The Generation of Knowledge" should be of value to any professional. Part B deals with design of experiments. Starting with Chapter B.7, readers might want to read the first and last sections of each chapter. Then, if they find the topics discussed to be of interest, they can read that chapter. For parts C through F I suggest a similar strategy. Within each part of the book the treatment of statistical concepts in the earlier chapters is less demanding. In the table of contents, the chapters where presentation of statistics is more demanding are "starred."

A quick read should help readers appreciate the value statistical methods could provide in software process/product design and development activities. I may be biased, but I cannot think of a better book. The readers who actively use statistical tools and methods will find that the book teaches them their appropriate and effective applications.

Jayesh Dalal is an ASQ Fellow and past chair of the ASQ Software Division. He has worked for more than 30 years in U.S. communications, electronics, and metals industries in the areas of process and

quality development and improvement. He also has served on the board of examiners for the Baldrige Award.

**International IT Governance:  
An Executive Guide to ISO  
17799/ISO 27001**

Alan Calder and  
Steve Watkins.

2006. Kogan Page International  
(<http://www.koganpage.com>).  
366 pages.  
ISBN: 0-7494-4748-6

**CQSE Body of Knowledge  
area: Software Quality  
Management**

*Reviewed by Scott Duncan  
sduncan@westfallteam.com*

While the title says "IT Governance," the authors state that "the main website . . . provides extensive information on IT governance practices and procedures, while the book continues to focus on the information security aspect . . ." Also, the authors state that "the book is written primarily with a Microsoft environment in mind," though they believe the principles they describe could apply more broadly since the standards are not system specific. The focus of the book, however, seems to be "how to implement an ISMS (Information Security Management System) capable of certification to ISO/IEC 27001:2005."

The book begins with some basic reasons why one should care about information security material, some brief comments about Sarbanes-Oxley, and then a short summary of the two standards. The two standards are internationalized versions of Parts 1 and 2 of the UK (BSI) standard 7799:

- ISO/IEC 17799—"Information Technology—Code of Practice for Information Security Management"

- ISO/IEC 27001—"Information Technology—Specification for Information Security Management Systems"

The authors note that IT evolves so fast that guidance in the standards, especially 17799, "may be inadequate to deal with newly identified threats and vulnerabilities and the most current response to them." They also state the book may "go beyond the current requirements of the standard in a number of areas, mostly to do with the Internet and e-commerce."

The rest of the book goes step-by-step through the clauses of 27001 in describing what the authors think would be needed to meet the intent of the clauses from a compliance perspective as well as extensions where IT security issues go beyond the standards. The authors provide a comprehensive coverage of the standard, since this portion of the book is about 308 of the book's 366 pages. Of course, the authors caution readers to be sure "any processes they implement are appropriate and tailored for their own environment."

Finally, the last chapter of the book briefly covers hiring an auditor and what to expect from an audit.

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.

**Jumpstart CMM®/CMMI®  
Software Process  
Improvement using IEEE  
Software Engineering  
Standards**

Susan J. Land.

2005. IEEE Computer Society  
Wiley-Interscience  
(<http://www.wiley.com>).  
175 pages.

ISBN: 0-471-70925-5

# Reviews

---

## CQE Body of Knowledge areas: Software Project Management; Software Quality Management

*Reviewed by Joel Glazer  
joelglazer@ieee.org*

This is a short review of a valuable book. Short, because most of the information in the book is contained in tables that compare and map various IEEE standards to the Capability Maturity Model (CMM®)/Capability Maturity Model Integration (CMMI®) goals, processes, and practices for level 2 key process areas (KPA). It is valuable because if one is using or is familiar with the IEEE standards and plans to implement CMM®/CMMI® model for process improvement, this book can save time, money, and aggravation.

The book consists of six chapters and two appendices:

- Chapter 1, "Introduction and Overview," briefly explains the CMM® and CMMI®. It also defines what the IEEE software engineering standards are and how they came to be.
- Chapter 2, "Summary of SW-CMM®," describes the maturity levels and the KPAs at each maturity level, the structural elements of the SW-CMM®V1.1, the appraisal framework of the CMM®, and the software capability evaluation, which is used in support of software acquisition.
- Chapter 3, "Summary of CMMI-SW® (Staged)," describes the continuous versus staged models, specific, and generic goals and practices that support the process areas for the maturity level; the appraisal of the CMMI®; the appraisal requirements for CMMI®; and the standard CMMI® appraisal method for process improvement (SCAMPI), and the three appraisal classes A, B, and C. Although CMMI® practitioners have de-

emphasized the term "key," it is used throughout the book.

- Chapter 4, "Differences Between CMM® and CMMI-SW® (Staged)," describes the initiating, diagnosing, establishing, acting, and learning (IDEAL) process improvement model, variations in emphases within the process areas, the difference between maturity level and capability level.
- Chapter 5, "IEEE Software Engineering Standard," addresses each of the seven staged CMMI®-SW level 2 process areas and what IEEE software standard is associated with that capability. The author shows how the various IEEE standards fulfill the goals and objectives of the CMM®/CMMI® KPAs.
- Chapter 6 describes how to use IEEE standards in support of achieving software process improvements. It also points out to potential pitfalls in implementing CMM®/CMMI® too hastily.

The book is easy to read with the CMM®/CMMI® terminology explained in simple language. It is written from the perspective of IEEE and is primarily for companies that are currently using the IEEE software engineering standards in their development approach and are contemplating adopting the CMM®/CMMI® for process improvement.

The book suffers from inconsistent use of terminology and editorial lapses. In particular, where differentiation between components of CMM® and CMMI® would be expected, there are references to one of the models using terms from the other model. Readers new to CMM® or CMMI® are cautioned that these oversights might lead to some confusion over their absorption of CMM®/CMMI® material. There are several glaring typographical errors as well.

Recently, support for CMM® appraisals is diminishing and will soon disappear altogether. A new version of CMMI® is anticipated

toward the end of 2006/beginning of 2007. This will make part of this book obsolete. The IEEE-CMMI® mapping, however, will most likely still be useful to facilitate mapping to the new version of CMMI®.

Joel Glazer is the current ASQ Region 5 Software Division councilor. He has more than 30 years of experience in the aerospace engineering, software engineering, and software quality fields. He has dual master's degrees from The Johns Hopkins University in computer sciences and management sciences. He is a member of IEEE and an ASQ Fellow.

## SOFTWARE ENGINEERING PROCESSES

### Refactoring Databases: Evolutionary Database Design

Scott Ambler

and Pramod Sadalage.

2006. Addison-Wesley

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

ISBN: 0-321-29353-3

### CQSE Body of Knowledge area: Software Engineering Processes

*Reviewed by Scott Duncan  
sduncan@westfallteam.com*

As the authors note, Martin Fowler originally defined *refactoring* as "a small change to your source code that improves the design without changing its semantics." In this book, the authors apply this concept to database design, using examples in Java, Hibernate, and Oracle to illustrate the database schema changes and their implications for associated application code changes. Indeed, the latter is one of the difficulties faced when database changes are made, that is, the application code using the database usually must change or be reviewed to determine that no changes are required.

---

Before getting into actual database refactoring, there are several chapters covering fundamental database design and configuration management concepts, the general topic of (database) refactoring, testing schema changes and deploying them into production, and strategies for refactoring databases. Starting with chapter 6, the authors spend the rest of the book offering some 70 examples of database refactorings, which they divide into the following categories:

- Structural, to “change the table structure”
- Data quality, to “improve the quality of the information contained within the database”
- Referential integrity, to “ensure that a referenced row exists within another table and/or ensures that a row that is no longer needed is removed appropriately”
- Architectural, to “improve the overall manner in which external programs interact with a database”
- Method, covering “code refactorings . . . applicable to stored procedures, stored functions, and triggers”
- Transformations, to “change the semantics of your database schema by adding new features to it.”

Finally, an appendix briefly covers UML data modeling notation followed by an equally brief glossary and short list of references.

From a design and quality engineering perspective, this book offers not only examples of the refactorings, but also examples of queries useful in analyzing databases to determine the extent and nature of the changes that would need to be made. For example, to address the issue of tables “indirectly coupled to the columns of other tables via naming conventions,” the authors offer a simple three-line SQL query showing how to hunt for column names sharing similar text strings. In

this way, the book is helpful even to those who are not specifically making such changes by suggesting some of the techniques useful in doing so. This suggests appropriate questions to ask during design and other quality reviews about how such refactorings were determined, analyzed, and performed.

### **Software Engineering: The Implementation Phase**

**Claude Petitpierre.**

2006. EPFL Press

(<http://itiwww.epfl.ch/~petitp/BOOK>). 349 pages.

ISBN: 2-940222-10-X

**CQSE Body of Knowledge area: Software Engineering Process**

*Reviewed by Uma Reddi  
reddi1@lnl.gov*

This is not a run-of-the-mill book. Java language is the medium used to discuss the implementation phase of software engineering. While this idea is novel in its approach, it limits the book's audience to those who are well-versed in that language. All examples given in the companion site are also of Java. The authors claim that by presenting programmatic aspects and basic functioning of the concepts it improves the readability and ease of comprehension. Maybe that is the case if one is an experienced Java programmer. Another paradox is that an experienced programmer may not need a book on the implementation phase that only touches fundamentals.

Many aspects of Java are discussed in the beginning chapters, including some basic features and creation of graphical user interfaces using a Java development environment called Eclipse. Many topics related to Java language are also

covered in later chapters, including Enterprise Java Beans and multi-threading. Thus, this book comes out as a little book made up of many different Java books that cover the same themes and topics in much more elaborate manner.

The book's relevance to software quality is nonexistent in the direct sense. Some ideas discussed in the book can be stretched to find some loose and superficial connections to the topics covered in the CSQE Body of Knowledge (BOK). For example, chapter 10, “The Development Process,” comes somewhat close to some of the topics covered in the BOK. However, even this does not go into any lengthy discussions as the author, like in other topics, gives it also a simplistic treatment.

This book is not relevant to the software quality aspects that are the mainstream of ASQ. It may be useful to those who are interested in seeing several functions of Java language described in one book. I do not recommend this book to ASQ readers based on these observations.

Uma Reddi is currently with Lawrence Livermore National Laboratory. She has bachelor's degrees in electrical engineering and math, and a master's degree in electrical engineering. She is a Senior member of ASQ.

### **Applying Domain-Driven Design and Patterns, With Examples in C# and .NET**

**Jimmy Nilsson.**

2006. Addison-Wesley

(<http://www.awprofessional.com>). 528 pages.

ISBN: 0-321-26820-2

**CSQE Body of Knowledge area: Software Engineering Processes**

*Reviewed by Ray Schneider  
rschneid@bridgewater.edu*

I've been in love with patterns and the patterns movement since I first stumbled upon it more than a decade ago. It had that "just right" feel that Christopher Alexander called "the quality without a name" (QWON). It is about a higher level of abstraction than just another abstract data type. It is language independent and practice-centric because, as Jimmy Nilsson points out in this great book, a fact that people have to be reminded of, over and over: "An important point about patterns is that they aren't *invented*, but rather *harvested* or *distilled*." In other words, patterns are about the real world of practical necessity and things that work. That doesn't mean they are obvious or easy to apply. That's why books that show how patterns are actually applied in real-world work are so helpful.

Everyone has been subjected over the years to the "the next great thing," the latest "silver bullet," and any number of other true religions. Sooner or later software development has to grow up, mature, and steady up on what really works and sharpen the tools instead of inventing new ones, whether languages, methodologies, standards, or whatever else is happening now. Nilsson owns and runs a consulting company and blogs at [www.jnsk.se/weblog/](http://www.jnsk.se/weblog/). He's coined the term "Lagom Process," by which he means a balanced process (lagom means balance in Swedish) with just enough of the right stuff. He mixes in patterns, agile processes, and various other ingredients in just the right mixture to be balanced for the task on hand.

*Applied Domain-Driven Design and Patterns* pulls together test driven development (TDD), domain driven design (DDD), patterns, agile processes, a variety of Nilsson's friends, and a nice conversational presentation packaged in a book that has been carefully crafted with figures, code, in-line notes, and running tabs that tell readers

what section of the book they are in so that they can quickly go to other parts.

There may be no higher praise for a book than that it was fun to read. This book is filled with fun and little moments of "oh, that's interesting" or "I'll have to try that." I heartily recommend this book for anyone who wants to see how patterns play out in a domain-model, data-centric world and have fun at the same time.

Ray Schneider holds a bachelor's degree in physics, a master's degree in engineering science, and a doctorate in information technology. He has more than 35 years of product development and applied research and development experience working in 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.

## **Software Requirements Encapsulation, Quality, and Reuse**

**Rick Lutowski.**

**2005. Auerbach (<http://www.auerbach-publications.com>). 248 pages.**

**ISBN: 0-8493-2848-9**

## **CSQE Body of Knowledge area: Software Engineering Processes**

*Reviewed by Ray Schneider  
[rschneid@bridgewater.edu](mailto:rschneid@bridgewater.edu)*

*Software Requirements Encapsulation, Quality, and Reuse* by Rick Lutowski gives an insightful view of what could be styled an agile requirements methodology that encapsulates requirements in a formal way and that is simultaneously lightweight and user accessible. In a brief acknowledgments section Lutowski thanks the original members of the Naval Research Laboratory (NRL) Software

Cost Reduction (SCR) project, especially David Parnas and David Weiss. Most software practitioners, he says, are unaware of the SCR team and its accomplishments, but it is from this earlier work that the "seeds of freedom were sown." *Freedom* is the name that Lutowski gives to this requirements methodology, which evolved from the in-house software development team's work on the Space Station Freedom Project (SSFP).

Lutowski provides a fast insightful read starting with an overview chapter that quickly sketches the freedom methodology. The key is that freedom defines requirements as "the black box view of the software system." This may be a surprising point of view to some, especially if the definition one has always heard is "a Software Requirements Specification (SRS) is a document containing a complete description of *what* the software will do without describing *how* it will do it." Requirements are about *what* not *how* we've been told. Alan Davis points out what he calls the "what versus how" dilemma, which briefly defined is that "one person's *how* is another person's *what*." *Freedom* doesn't have this problem. Chapter 2 "Information-Hiding Secrets" gives you a fast update on the rationale behind information hiding and its virtues. In chapter 3, "What Are Requirements?" readers are introduced to freedom's perspective on software in three views based on Harlan Mills, an IBM Fellow, and software engineering research professor's 1988 paper on box-structured systems.

Chapters 4 through 14 examine the nuts and bolts of the freedom requirements process. Freedom's view is of external stimulus and responses sets that are grouped using cohesion to control complexity and then structured into a functionality tree, which is further elaborated into responses and behavior tables. The final goal is an interface prototype,

---

which acts as an active requirements specification that the customer can evaluate and which becomes the encapsulated requirements specification for the system.

Freedom stresses customer interaction through a point of contact (POC) and speaks of teaming up developers and customer POC's into pair specification teams. In chapter 10, "Responses and Behavior Tables," Lutowski discusses the ideal language for recording response behavior. He looks at natural language, formal methods, and program design languages.

I finished the book all fired up and ready to apply the freedom method to real-world problems. Some of the things that are appealing: 1) It's object oriented, but not restrictively so; 2) it's management neutral and life-cycle neutral; 3) it's lightweight and still formal enough to meet most desires for formality; 4) it emits usable artifacts that actually advance the project; and 5) I think it's agile. For those who are involved with requirements specification then this is a must read. For those involved with software development and are fed up with bad design direction disguised as requirements, this is a must read. I liked this book a lot and heartily recommend it.

### **The Unified Modeling Language User Guide, Second Edition**

Grady Booch, James Rumbaugh, and Ivar Jacobson.  
2005. Addison-Wesley  
(<http://www.awprofessional.com>). 475 pages.  
ISBN: 0-321-26797-4

**Verification and Validation for Quality of UML 2.0 Models**  
Bhuvan Unhelkar.  
2005. John Wiley & Sons, Inc.

(<http://www.wiley.com>).  
271 pages.  
ISBN: 0-471-72783-0

### **Component-Based Software Testing with UML**

Hans-Gerhard Gross.  
2005. Springer-Verlag  
Berlin Heidelberg  
(<http://www.springer.com>)  
316 pages.  
ISBN: 3-540-20864-X

### **Software Evolution with UML and XML**

Hongji Yang.  
2005. Idea Group Inc.  
405 pages.  
ISBN: 1-59140-462-2

### **CSQE Body of Knowledge areas: Software Engineering Processes, Analysis, design, and development methods and tools**

*Reviewed by Ray Schneider*  
([rschneid@bridgewater.edu](mailto:rschneid@bridgewater.edu))

These four books describing and using the Unified Modeling Language (UML) 2.0 cover the software life cycle. One of the UML's virtues is that one can pick and choose what he or she needs. The whole enchilada is big and a bit of a shotgun marriage of many diagrams and methods that existed long before UML came on the scene. It also leaves out important diagrams such as Petri-nets, data flow diagrams, and others. Its primary advantage is that it is a standard, but standards are also straitjackets. The "unified" part is a bit like the courses of a picnic lunch being unified by being in the same basket. Some ongoing research seeks to make the unified part more meaningful.

*The Unified Modeling Language User Guide, Second Edition* covers UML 2.0, but except for relatively

minor changes, is very similar to the original edition published in 1999. For those who have either not heard of UML or have heard of it and never used it, UML is a graphical language consisting currently of a toolkit of 13 diagram types supporting the four main purposes of UML, the visualization, specification, construction and documentation (VSCD) of software systems.

"There ain't no such thing as a free lunch," so it isn't enough to just know about UML and be able to identify the diagram types. One must be able to do it well and *Verification and Validation For Quality of UML 2.0 Models* by Bhuvan Unhelkar focuses in on what readers need to make sure that they're doing the right thing and doing it right.

Hans-Gerhard Gross, in his book *Component-Based Software Testing with UML*, goes beyond the usual VSCD applications of UML to the realm of testing using a component meta-model to guide model-driven, contract-based testing with a goal of overcoming problems such as testing a component to be employed in a new context striving to overcome limitations set by black-box testing.

Beyond initial development is a world of evolving software and evolving methods. *Software Evolution with UML and XML* by Hongji Yang explores research being conducted by the multinational software research community as it strives to grapple with many of the problems software evolution imposes. Reading *Software Evolution with UML and XML* will give readers a snapshot of what the future holds for model-guided techniques.

*The Unified Modeling Language User Guide, Second Edition* is a lot like the first edition. In fact, it's almost identical, with most of the text and figures duplicated. There are new chapters on components and artifact diagrams, but it was mostly just cutting the pie a little differently. The book is a rather

# Reviews

---

formula production. Each chapter is divided into four sections: Getting Started, Terms and Concepts, Common Modeling Techniques, and Hints and Tips.

For those who have the first edition, this isn't going to be an epiphany. It's mostly same-old, same-old and even the new stuff is not all that important.

*Verification and Validation for Quality of UML 2.0 Model* is just the book to help one use the UML effectively. The book starts out by offering an overall quality strategy for using UML. The primary roles of user, business analyst, system designer, system architect, project manager, and quality manager are presented by the author as he divides modeling into three model spaces: the model of problem space (MOPS), the solution space (MOSS), and the background space (MOBS). The first models the analysis of the problem, the second the design of a solution, and the third the application of real-world constraints to the architecture of the system. UML is used to address these three modeling spaces. Telling the difference between good models and poor models is what creates quality. Each kind of UML diagram is evaluated against the objectives of the three model spaces.

The author did a great job of partitioning UML to show what diagrams address what aspects against a modeling space demarcated by structural and behavioral aspects along one dimension and static and dynamic considerations along the other. Then each UML diagram type was briefly illustrated using an example drawn from the overarching case study.

Quality is assessed based on syntax, semantics, and aesthetics exploring the correctness of the modeling elements, the completeness and consistency of the diagrammatic models of the underlying reality at the intermediate level, and symmetry and consistency

of the entire model. Unhelkar calls these levels the artifact, diagram, and model levels and uses a height metaphor to represent the views as syntactic at the ground-level view, semantic at the standing view, and aesthetic at the bird's eye view. The model stages are illustrated using worked out templates and checklists given in the appendices.

Each chapter of this outstanding book ends with discussion topics, which readers can use to help confirm their own understanding and coverage. Readers will still need other UML guides and reference books, but with the tools here will help them climb the learning curve rapidly. I highly recommend this book.

*Component-Based Software Testing with UML* by Hans-Gerhard Gross uses UML in a nontraditional but emerging role—testing. In a sense it directly complements Unhelkar's book, since V&V is a kind of testing. Ina Schieferdecker in the Foreword comments, "Of particular interest to me is the use of the UML 2.0 testing profile in this book and its application and extension to the special case of built-in tests." Gross begins with a review of component-based development. He quickly introduces a UML component meta-model that puts the component in the center of the development focus. The application is a composition of components leading to a mixture of top-down and bottom-up analytical (decomposition) and synthesis (composition) activities. The focus on testing highlights important problems illustrated by the ARIANE 5 failure where a previously tested component, being reused, failed due to the change in context.

Having set the stage, Gross describes the KorbA method in chapter 2. Two basic dimensions (abstraction—concretization and composition—decomposition) are navigated using four basic activities: decomposition, embodiment, composition, and validation applied in a spiral development fashion.

Readers might be tempted to look at the 2002 book, *Component-Based Product Line Engineering with UML*, by Colin Atkinson, et al., for a wider ranging view of the KorbA method. Chapter 3 contrasts model-based testing with more traditional testing. UML models, especially behavioral models but including structural models, guide the testing. A vending machine example is introduced in chapter 2 and carried along throughout. The fundamental concepts of the UML testing profile are introduced and explored. It all comes together linking model-based testing and test modeling in chapter 4, which covers built-in contract testing.

A component should be able to test itself. The development of that capability is simply another aspect of component development. Various depths of testing and testing architectures (implicit and explicit) are explored and illustrated using the vending machine example. Testing components acquire components to be tested in a kind of client-server relationship. It's further explored in chapter 5 where implementation languages and existing component technologies are discussed.

*Software Evolution with UML and XML* is an interesting but not entirely unified book. It consists of 12 chapters under the fairly broad rubric of *software evolution*, and most of the chapters use UML and XML in some capacity in addressing software evolution. The topic is intrinsically important since the cost of software change is very large, commonly cited at 70 percent to 80 percent of the total budget, although this is generally attributed to feature creep during development. Of more concern is the longer-range evolution as major version change comes into play. The table of contents lists the many authors of the individual chapters. This is an international group including the U.K., Spain, Italy, China, Singapore, and Australia. Three of

---

the chapters are written by U.S. researchers. The papers are all of a research character and many are quite theoretical. There is no continuity thread that unites the papers, so they are each rather set-pieces on aspects of software evolution.

This book accomplishes two things: 1) it gives a snapshot of what the research community is doing; and 2) it provides ideas from the synergism that arises when one is reading about other people's ideas. For example, chapter 8 presented an innovative approach to testing using patterns by linking scenario patterns and verification patterns. That was a neat idea. I didn't read anything in this book that made me think that I had to own the book, but I enjoyed it. Each chapter was a different adventure in trying to understand what a different research team or individual was attempting.

To sum up—these are all good books. If I had lots of money for books I'd probably buy them all. On a limited budget my first choice would be *Verification and Validation For Quality of UML 2.0 Models*. My second choice would be *Component-Based Software Testing with UML*.

## PROJECT AND PROCESS MANAGEMENT

### **The Cognitive Dynamics of Computer Science**

Szabolcs Michael De Gyurky.  
2006. Wiley Interscience for  
IEEE Computer Society  
(<http://www.wiley.com>).  
292 pages.  
ISBN: 0-471-97047-6

CQSE Body of Knowledge  
area: Project Management

*Reviewed by Scott Duncan*  
[sduncan@westfallteam.com](mailto:sduncan@westfallteam.com)

This is a most unusual book. Its title suggests a rather academic discus-

sion, and much of the content supports this. On the other hand, the author suggests that the book is about how he and his colleagues at Jet Propulsion Lab have been able to bring the cost of producing large, complex software systems "down to under \$10 per line of code" compared with \$100s or \$1000s typical of other organizations. In general, De Gyurky believes that the kind of performance he describes is made possible because of "several important factors:

- Leadership
- Management
- Communication
- Organization
- Understanding software development standards, architecture, and methodologies."

De Gyurky, however, does not head straight into a description of how these specific factors are employed on software projects. He first asserts that the direction in which computer science is headed is the development of autonomous cognitive systems, which, for example, could be embedded in "a true robot . . . that can be sent into space . . . with an intelligence similar to our human intelligence . . . and go to places where we humans cannot survive . . ." This leads De Gyurky into a discussion of the necessary architecture for such autonomy and, then, how his management success has been based on "the influence of the great classical philosophers" and their views on "how work is actually done" from a cognitive perspective. This all leads him to make the claim that "The field of computer science, as it applies to my profession as a manager and architect, is therefore cognitive dynamics and cognitive mechanics."

The first three chapters are devoted to making the philosophical case above, yet are a bit maddening to get through, since they periodically promise to get into more software project specifics then turn back to the philosophical discussion.

The book also uses stories about development and military experiences that do not always seem to advance an understanding of how the key factors "work." Some of this is due to redundancy and a sense that De Gyurky is justifying the importance or success of the experiences when his judgment worked successfully. Indeed, De Gyurky states that his "team here at JPL certainly has proven many times that no one builds quality systems and cheaper software than we do here." Perhaps the answer to better quality software, then, is simply to emulate De Gyurky and his team, presumably for the application of the key factors he identifies.

Some examples of ideas from the book include:

- Organize and staff around the project and its architecture—don't shoe-horn projects into a preset organizational hierarchy
- Have as flat and nonhierarchical an organization as possible
- Have an environment where people can "speak freely without fear of retribution"
- Have clear interaction/communication protocols for the software and the organization
- Use management and technical standards
- Have effective project control (which means being involved in the project and knowing, personally, what is going on)
- Select a methodology appropriate to the project, which includes considerations of "appropriate organization, communication, development, and control functions," not just life-cycle processes and sequencing
- Realize that "rapid development" and "prototyping" are different things

I think this book is worthwhile, but one must get beyond the first few chapters and some of De Gyurky's stylistic tendencies.

# Reviews

---

## **Software Evolution and Feedback**

Nazim H. Madhavji,  
Juan Fernández-Ramil,  
and Dwayne E. Perry.  
2006. John Wiley and Sons  
(<http://www.wiley.com>).  
575 pages.  
ISBN: 0-470-87180-6

**CQSE Body of Knowledge  
area: Software Project  
Management**

*Reviewed by Scott Duncan  
sduncan@westfallteam.com*

In 1985, Lehman and Belady published *Program Evolution, Processes of Software Change* based on analysis of large software system development data, such as the IBM/360 software. It addresses issues related to large system "evolution," that is, the development of "successive generations of artificial systems." Over the years, Lehman has pursued this work and attempted to interest others in it. *Software Evolution and Feedback* is a collection of 27 essays and articles by Lehman and others on the topics of system (evolution) growth and development project feedback.

Basically, Lehman and Belady state that certain kinds of software must inevitably evolve, since their continued usefulness to the user community depends on the addition of new capability over time. If they are not evolving, they likely are not being used much or only in a limited domain. This change in software is inevitable, and it is important to understand how this change comes about, how projects respond to it, and what it means for the (design) "integrity" of the software over time. Specifically, Lehman and Belady note that a system's "integrity" will deteriorate because of changes made to it in a manner inconsistent with its design/archi-

ture if there are not enough resources applied to maintaining "integrity." At some point, such changes result in a system that is almost impossible to understand and change reliably.

Among the topics covered by the essays are:

- Various definitions, explanations, and theories associated with software evolution and feedback
- Statistical modeling of software evolution
- Requirements change impact
- Evolution in and structural analysis of open source systems
- Feedback control
- Risk management and reliability
- Expert estimation and feedback learning
- Rules and tools for evolution planning and management

The book states it is intended for "researchers in software engineering: senior practitioners and consultants," "graduate students and junior practitioners," "thesis students" and "advanced undergraduates." From a software quality perspective, the ideas behind software evolution and feedback are important considerations when planning the kinds of design and V&V activities required to ensure system integrity over time as well as to understand the "forces" that affect how software comes to be and is changed.

## **Catastrophe Disentanglement**

E. M. Bennatan.  
2006. Addison-Wesley  
Professional ([http://www.  
awprofessional.com](http://www.awprofessional.com))  
288 pages.  
ISBN: 0321336623

**CQSE Body of Knowledge  
area: Software Project  
Management**

*Reviewed by Nels Hoenig  
nelsh@virtualteam.com*

This book is a survival guide for a software project disaster. The author uses a 10-step process of understanding the issues in place today and developing a strategy of how to move forward on the project to drive to success. The book includes summaries and exercises and appears to have been written with the MBA student in mind. The examples are topical and realistic both in the problems and the solutions used.

Chapter 6, "Step 4, Evaluate the Team," was the most interesting chapter in the book. Most project failures are not the result of poor technology or bad developers but a lack of communication and a working team structure. A project leader needs to be a strong leader, able to inspire confidence, and help the team drive to success. A software project also needs people who believe in their project leaders and respect their opinion to accomplish the project. This places a heavy burden on the project leader to understand the project and be able to delegate, motivate, and trust the team beneath him or her to make the project a success.

The author also makes a valid point that this team evaluation is not a performance review and that employees must not feel threatened by the process if the real issues are to be found. The evaluator needs to keep above any of the personality conflicts that are likely to exist when a project reaches this state.

After the introductory chapters, each step in the process is contained in a chapter and each chapter is broken into the key areas of the task.

The process includes three basic reports in the process:

1. At Step 4 Team Overview: A summary of the state of the project and the team originally on the project. It includes a

---

comparison of the published status to the actual status

2. At Step 6 Midway Report: Details on the new plan being put in place and any new skills needed are identified
3. Summary Report: Even if the project is cancelled, a report that covers the findings and lessons learned to improve future projects

Most people, statistically anyway, will be on one or more projects in the state of disaster during their career. I found the book relevant and a useful addition to my bookcase.

Nels Hoenig is an experienced software professional who has participated in many roles on successful projects. He holds numerous certifications, including CSQE, PMP, MCP, IT Project+, and others. He recently was the technical editor of the book, *How to Cheat at IT Project Management*.

### **Why Most Things Fail: Evolution, Extinction, and Economics**

Paul Ormerod.

2005. Pantheon Books  
(<http://www.pantheonbooks.com>). 255 pages.  
ISBN: 0-375-42405-9

CSQE Body of Knowledge  
area: Project Management

*Reviewed by John E. Moore*  
[Moore.john.e@cox.net](mailto:Moore.john.e@cox.net)

The title of this book is somewhat of a misnomer because it is not as much about the failure of "things" as it is about the failure of "classes of things." In other words, while there are some examples of specific business or product line failures (such as the "New Coke"), the book focuses on failure at a higher level, such as the failure of government policies to effect economic improvement. So what does a book that

finds analogies between economic failure and the extinction of species have to do with software quality? Quality is all about avoiding or preventing defects and thus minimizing the likelihood of system failure. If people can understand why systems fail, perhaps they can do a better job of preventing failure.

One of the most interesting observations that Ormerod makes is that it is very difficult to determine the exact cause of failure in many cases. He uses examples from game theory to show that even for a fairly simple model, "it becomes hard, virtually impossible to trace the precise reasons why an agent becomes extinct."

Ormerod's book gives some serious food for thought, not just in the discussion of failure, but also in the fundamental limitations that one has to plan and organize policies and procedures to prevent it. Ormerod compares the extinction of species (which, except for humans, have no ability to think about the future to prevent their extinction) with the failure of businesses (which have humans on the staff whose job it is to think full time about the future). He shows strong parallels, which he summarizes as the "iron law of failure."

At first glance, one might think that all this talk of failure would be depressing. But Ormerod's goal and his writing are not depressing. His writing is very understandable to anyone with a first or second year college education and an interest in economics and biology. It's fascinating to explore with him as he challenges basic assumptions about the way economies and business processes work. For instance, he shows that some of the "best practices" taught in business schools are, in fact, not supported by the real world and can actually encourage failure rather than avoid it.

But what does this have to do with software quality? By substituting the term "software system" for

"business" during the read that I came away with many "aha" moments. Ormerod is not a pessimist, but what he does show is that most systems are inherently complex because of interactions with the outside world and they are dynamic—they change over time. Sometimes people's complicated efforts to effect improvement in complicated systems actually hasten failure rather than prevent it, because they fail to recognize these inherent properties of the systems one is trying to control. I recommend this book to anyone who wants to challenge his or her preconceived notions about why most things fail and what to do about it.

John Moore is a systems engineer who provides process improvement and project management support for projects and proposals at a major systems integrator in the Northern Virginia area. He has master's and doctorate degrees in mechanical engineering from the University of Utah.

### **Software Project Management: Measures for Improving Performance**

Robert Bruce Kelsey.

2006. Management Concepts  
(<http://www.mngtconcepts.com>). 199 pages.  
ISBN: 1-56726-173-6

CSQE Body of Knowledge  
areas: Software Project Management; Software Metrics, Measurement, and Analytical Methods; Software Quality Management

*Reviewed by Heather Kujak-Coon*  
[kujak-coon.heather@co.la-crosse.wi.us](mailto:kujak-coon.heather@co.la-crosse.wi.us)

*Software Project Management: Measures for Improving Performance* is an excellent guide for project and software managers who are interested in beginning the collection of

metrics in their organization or on a specific project. This book provides a progression from simple metrics to more complicated metrics; it explains each thoroughly enough to allow readers to implement on their own. The author states, "I don't talk about statistical process control because I'm assuming most readers won't have any defined and consistently followed processes. Nor is there anything about creating metrics dashboards or scorecards, because I'm assuming that no one in your boardroom cares." This is a refreshing approach, since many books assume a metrics program or process improvement program is already in place, which often is not the case.

The author starts by describing some of the basics of software measurement, such as ways to use software measurement and what one can accomplish with it. He goes on to describe the general architecture of a measurement program, including the goal-question-metric paradigm, what a base vs. derived measure is, and the steps to implement the architecture. The next three chapters describe various metrics, what one would use them for, how to interpret the results, and how to implement them. These chapters are especially useful because they give readers examples of real metrics they could collect, rather than leaving it up to them to think of on their own. This is particularly helpful for those who are just starting on a metrics program, as it can be difficult for beginners to determine what metrics to start with. The last two chapters of the book discuss effective ways to present metrics and the all-important "human side" of the metrics collection.

One of the most useful features of this book is the "Applying What You've Learned" section at the end of each chapter. Each of these sections contains exercises to ingrain the lessons of the chapter in the reader's mind. The exercises instruct readers to complete them using their own situation as a basis;

if readers do these exercises, they will be well on their way to implementing metrics on their project or whatever situation they are applying the lessons to.

I recommend *Software Project Management: Measures for Improving Performance* to any software project manager who wants to collect better data on his or her projects or to any software practitioner interested in the value metrics can play on their projects.

Heather Kujak-Coon is employed by the County of La Crosse, WI, in the information technologies department. She has a bachelor's degree in computer science and a master's degree in software engineering.

## **TSP—Coaching Development Teams**

**Watts S. Humphrey.**

**2006. Upper Saddle River, N.J.**

**Addison-Wesley**

**(<http://www.awprofessional.com>). 416 pages.**

**ISBN: 0-201-73113-4**

## **CSQE Body of Knowledge area: Program and Project Management**

*Reviewed by Carolyn*

*Rodda Lincoln*

*Carolyn.Lincoln@l-3com.com*

*TSP—Coaching Development Teams* is about how to perform the coaching role for teams using the Team Software Process (TSP). It is one of several books in the Software Engineering Institute (SEI) series on software engineering about TSP and the Personal Software Process (PSP). PSP and TSP were developed by the author, Watts S. Humphrey, at the SEI to help projects ensure quality software products. They support other SEI products such as the Capability Maturity Model Integration (CMMI).

TSP is a set of processes for self-directed teams to manage their

software work after management has provided a goal for the system. The process includes forming the team, holding a team launch, and then maintaining the team, which is how the book is organized. It also includes sections on coaching and TSP extensions such as multiple teams and integrated teams.

The TSP coach role is different from the team leader. The coach does team member training, facilitates and guides the team launch, provides support throughout the project, and monitors the measures. A coach can support three to five teams. The team leader performs the usual management functions of team selection, setting goals and standards, reporting to higher level management, and maintaining communication and discipline. The team leader role is described in a separate book.

As with all of Watts Humphrey's books, this one is very well written. He provides abundant examples and has a clear style that is easy to understand. The text has frequent informative headings to keep track of the thread of the information, but each paragraph also contains the right amount of detail. It is obvious that he has many years of experience in software development because everything he says is eminently practical.

Having said that, the book is probably not going to be a best seller for the general ASQ audience. The subject is only of interest to those using PSP and TSP. Although the author says any type of professional team can use it, the explanations are very specific to software development teams using TSP. Even for those teams, other extensive training is required to use PSP and TSP. The book is a necessary but not sufficient resource for people who want to be TSP coaches, since the SEI has a program of training and practice to become an authorized coach. It could be used as an executive introduction to TSP coaching and as a reference for TSP teams.

---

Carolyn Rodda Lincoln is an ASQ-certified quality manager and quality auditor and a member of the DC Software Process Improvement Network. She is currently employed in QA and measurement for L-3 Communications Titan Corporation in Reston, VA. She holds bachelor's and master's degrees in math and was previously a programmer and project manager.

## VERIFICATION AND VALIDATION

### **Black-Box Testing: Techniques for Functional Testing of Software and Systems**

Boris Beizer.

1995. John Wiley and Sons (<http://www.wiley.com/WileyCD>). 294 pages. ISBN: 0-471-1204-4

### **CSQE Body of Knowledge area: Software Verification and Validation**

*Reviewed by Matthew Heusser  
Matt.Heusser@gmail.com*

Boris Beizer wrote several books on software testing; this one is intended to be an advanced college textbook for engineers and computer science students. This is an important distinction, because it assumes the reader has a background in programming, mathematics, discrete math, set theory and graph theory, and an ability to connect all five.

The author presented methods for determining test cases that start with reverse-engineering the requirements to determine logical paths, then attempting to write a test case for each path. After a few chapters of this, he admits that developing a directed graph of the entire system is not really feasible, and recommends decision trees instead, then goes back to graphing methods.

This book seems to take testing ideas directly from hardware engineering and chip design and apply them to software. As the book

states, most real testing problems are too complex to create directed graphs. The emphasis on reverse-engineering specifications to graphs may work for compilers, and expert and rule-based systems, but otherwise I see little value for the aspiring software developer. In fact, when the newly graduated software developer has to start his or her first real project with woefully inadequate specifications and GUI-based messaging, the entire testing model begins to break down. It may be of some value for students who will become computer engineers or who desire to work on compiler construction and other abstract, rules-based systems.

Like most textbooks on graph theory, the writing style is thick and obscure. The author does not address the "why" of each technique, or its strengths and weaknesses. Aside from the high-level graphing constructs and ideas like equivalence partitioning, very little of the material seems to apply to the CSQE Body of Knowledge or the CSQE exam. The CSQE exam does not cover test design techniques in much detail, but if one is *interested* in the formal, mathematical aspect of software test design, I would instead recommend Paul Jorgensen's *Software Testing: A Craftsman's Approach, 2<sup>nd</sup> edition*, as it is more practical, and less painful to digest.

Matt Heusser is a senior software developer. His published works include articles in *Software Test Performance Magazine*, *Better Software Magazine*, and *AngryCoder.com*. Heusser speaks on software development and testing topics, and is the founder and moderator of SW-IMPROVE, a software discussion list.

CMM® and CMMI® are registered trademarks of the Software Engineering Institute, Carnegie Mellon University.