

Resource Reviews

SOFTWARE QUALITY MANAGEMENT

AntiPatterns: Identification, Refactoring, and Management

Phillip A. Laplante
and Colin J. Neill.

2006. CRC Press.

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

304 pages.

ISBN 0-8493-2994-9

Pattern Languages of Program Design 5

Dragos Manolescu,

Markus Voelter,

and James Noble, eds.

2006. Addison-Wesley

Professional.

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

596 pages.

ISBN 0-321-3194-4

CSQE Body of Knowledge
areas: Software Quality
Management and Software
Design Methods

*Reviewed by Ray Schneider
rschneid@bridgewater.com*

I have a confession. I love patterns. Perhaps more to the point, I love antipatterns. Patterns are about the distillation of experience. What works repeatedly, is a pattern. What doesn't work, but seems to be pervasive often pretending to work or to be the way things truly are, is an antipattern. Antipatterns are often pernicious for much the same reason as Gresham's Law. Simply speaking, bad money drives good money out of circulation. Antipatterns do something similar. Bad

practices drive out good practices, so it is very important to isolate and understand antipatterns in order to eliminate them.

I have several books with the term "antipatterns" in their titles. Among them is Brown, Malveau, McCormick, and Mowbray's 1998 classic *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Laplante and Neill's *Antipatterns* proposes to extend and complement Brown et al. with a new catalog of antipatterns including management antipatterns and environmental or cultural antipatterns that poison the environment and stifle success. What makes antipatterns helpful is that it is often easier to stop doing things that are dysfunctional and destructive than it is to begin doing things that are beneficial.

Before patterns there were a few books that touched on antipatterns. Some examples are *Parkinson's Law* (1956) by Cyril Northcote Parkinson, *The Peter Principle* (1970) by Laurence J. Peter and Raymond Hull, and *Augustine's Laws* (1982) by Norman R. Augustine. Laplante and Neill, following Brown et al., continue this tradition in pattern form. Seeing these dysfunctional patterns will often produce bitter-sweet memories. I was constantly reminded of incidents from my own career that could have been handled better if I'd recognized them for what they were.

Antipatterns begins with a chapter that explains patterns and antipatterns, followed by six chapters that catalog patterns in an expanding scale starting with individual human patterns and antipatterns. Chapter by chapter the scope grows to encompass, finally, environmental antipatterns that can

result in destroying the effectiveness of entire organizations. Chapter seven closes the book with general advice to individuals for dealing with antipatterns.

I found the book insightful, entertaining, and compulsive. Once I got started it was hard to put down. Each antipattern is presented in the form of a colorful name, a summary of the central concept, a vignette, some explanation, followed by some advice. The advice includes a bandage—something that can be done to partially address the situation—self-repair, more extensive advice called refactoring, some closing observations, and an identification checklist to help people recognize antipatterns.

For example, in chapter two there is a taxonomy of *corncocks*, a name used by Brown et al. to designate difficult people. Laplante and Neill classify corncocks into seven phenotypes or antipatterns drawn from a book, *Coping with Difficult People*, by Robert Bramson. Readers will probably recognize some of these from their own experience: hostile aggressives, indecisives, whiners, negativists, clams, bulldozers, and superagreeables. Each is further subcategorized. For example, hostile aggressives are divided into sherman tanks, snipers, and exploders.

Another illustration from the management section is the antipattern of fruitless hoops, the manager who keeps asking for more and more pointless work, usually to reassure him or herself that the team is on the right track. This is contrasted with plate spinning, which is the intentional assignment of meaningless work. The first is due to lack of confidence and the second is malicious. Both are destructive.

In the section on environmental antipatterns there is the example of fairness doctrine, an excuse not to deal with real difficulties such as incompetence, in the name of being fair, to the destruction of team morale.

In all, this book describes—in humorous detail—more than 50 antipatterns and variants in antipatterns. I thoroughly enjoyed and heartily recommend it. Any software developer, team leader, or manager at any level will benefit from reading this book. Readers will find this an excellent resource with good advice and great insights.

The second and longer book I reviewed is *Pattern Languages of Program Design 5 (PLPD5)*, edited by Dragos Manolescu, Markus Voelter, and James Noble. Unlike *Antipatterns*, *PLPD5* is about things that are functional and helpful. In the introduction, Norman Kerth offers some well-chosen questions: 1) What did we do well, that we must not forget? 2) What did we learn? 3) What would we do differently next time? 4) What still puzzles me? and 5) What was my greatest joy?

The answer to question one is, in fact, the motive for patterns and pattern languages. People don't want to forget the things they did well. The pattern movement is one of the most exciting, because it shows computer science maturing and trying to capture its own culture and patterns of successful work. *PLPD5* offers readers 19 chapters of patterns divided into six groups called parts. Part I, design patterns, has three pattern languages. The first, dynamic object model, shows how to dynamically add new types without programming new classes. The second, domain object manager, provides a compound design pattern to isolate clients from the details of diverse repositories and makes an explicit tradeoff between complex implementation and easy usage. The pattern is a further refinement of the repository pattern found

elsewhere. The third chapter in Part I is called "Encapsulated Context." The chapter describes patterns to avoid the use of global data and extensive pointer passing by providing context classes. All three of the pattern languages described in part one incorporate brief example code to illustrate the patterns. Other parts of the book focus on different domains. Part II focuses on concurrency and real-time patterns. Distributed systems are discussed in Part III, domain-specific patterns in Part IV, architecture patterns in Part V, and a special treat for hardcore pattern lovers is meta-patterns (patterns about patterns), which are discussed in Part VI.

In Part II, the "triple-t" is a system of patterns for reliable communication in hard real-time systems. The patterns presented are considered a basic set for reliable hard real-time system implementation. The focus here is on communication and not scheduling, so it is not a complete pattern space. Each of five patterns is described in careful detail.

The first prescheduled periodic transmission, also known as time division multiple access, includes a discussion of why the popular Ethernet protocol won't work. Time-triggered clock synchronization deals with how to keep the clocks accurate without a central clock (a single-point failure mechanism). Sync frame addresses the problems of getting started and the synchronization of new nodes entering the net. Bus guardian shows how to defend the system against noisy nodes or nodes slightly out of sync. Finally, the pattern temporal application decoupling shows how to use buffers to decouple the production and processing of data from the transmission devices. A brief appendix, which includes known uses, concludes this detailed and important pattern set.

Part III includes the comparand pattern, which addresses the problem

of interpreting objects as being the same for different contexts. I tend to see this problem treated as the simple dichotomy of identical because the same thing is pointed at (reference semantics) or identical because the things have the same values (value semantics). The treatment shows that the problem is more difficult than such a classification suggests and the strategy trades off complexity and flexibility.

There is an interesting pattern language in Part IV on interactive Web-based applications. The chapter provides a roadmap for building highly dynamic, personalized, and content-centric Web applications.

In Part V every chapter is fascinating ranging from patterns for plug-ins, grid computing, component and language integration, and framework development.

Finally, Part VI provides meta-patterns for pattern writing, language design, the process of pattern shepherding, and a bonus titled "Patterns of the Prairie Houses," a chapter that reverse engineers the essential themes in Frank Lloyd Wright's Prairie Houses, resulting in a fragment of a pattern language interpreting Wright's designs, their justification, and their motivation.

As diverse as this book is, there is always another interesting chapter addressing an important design problem. Some I read because I knew the domain and wanted to see what the authors said. Others I read because I wanted to learn about the domain addressed. It was a rich and enjoyable learning experience.

Both of these books have found a permanent home in my pattern library. I heartily recommend *Antipatterns* for its humor and insight into the things that go wrong at the individual, group, and organizational levels, and I recommend *Pattern Languages of Program Design 5* for its rich, valuable, and wide-ranging contribution to the pattern literature.

Reviews

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 ENGINEERING PROCESS

Software Engineering Handbook

Jessica Keyes.

2003. Auerbach Publications:
A CRC Press Company.
(<http://www.auerbach-publications.com>). 874 pages.
ISBN 0-8493-1479-8

CSQE Body of Knowledge
area: Software Quality
Management and Software
Engineering Processes

*Reviewed by John D. Richards
john.richards1@amedd.army.mil*

This is a large, comprehensive volume that serves as an excellent reference tool for software engineers and managers of software projects. The author set quite a task for herself as put forth in the foreword:

"This book was written to push the information technology industry up that learning curve in one fell swoop. Collected here are 65 chapters, 191 illustrations, and 19 appendices filled with practical (the keyword here is practical) techniques, policies, issues, checklists, and guidelines, and complete "working" examples on methodology, quality, productivity, and reliability." (Page xvii).

This work was collected over the author's 25 years of experience as a consultant and professor of

computer sciences. Her experience includes business and technology. She was managing director of research and development for the New York Stock Exchange and has been an officer with Banker's Trust and Swiss Bank Company. She has published more than 200 articles and is the author of 16 books on business issues and technology.

The handbook is divided into three sections. The first section consists of 20 chapters on software engineering. The chapters cover a project from feasibility and cost/benefit analysis, through plan writing, requirement elicitation, and outsourcing decisions to testing and documentation. This section is very detailed, providing examples, charts, checklists, formulas, and so on necessary for running a project.

In the second section, in the author's words, "We change gears from method to metrics." The focus is on such things as productivity, quality, and reliability. This section consists of 45 chapters. It covers such topics as the Baldrige Award, Corbin's methodology for establishing a software development environment, Motorola's Six Sigma defect reduction effort, Byrne's reverse engineering technique, and the IEEE framework for measures, to name only a few. Each chapter has an abstract that further increases the value of this handbook as a reference.

The third section contains 19 appendices, providing examples of such things as: sample project plan, sample data dictionary, sample cost/benefit analysis worksheets, and test plan. All these assist the project manager in being efficient in his or her efforts, in that forms or templates do not have to be created but copied and populated.

This is not a volume that is normally read cover to cover but rather kept on the desk for frequent reference. It is well-organized, complete, easy to use, and an invaluable resource for project managers who do not want to "reinvent the wheel" but rather improve on it.

John D. Richards works for Cambridge Systems, Inc, at the Project Management Division at the United States Army Medical Information Technology Center (USAMITC). He is a senior member of ASQ, a CQE, and a CQA. He has a master's degree in psychology and an advanced master's and doctorate in education.

Design for Trustworthy Software

Bijay K. Jayaswal
and Peter C. Patton.

2007. Pearson Education, Inc.
(<http://www.pearsoned.com>).
805 pages.
ISBN 0-13-187250-8

CSQE Body of Knowledge
area: Software Engineering
Processes

*Reviewed by Scott Duncan
sduncan@westfallteam.com*

In the preface, the authors state that "Any quality method employed to improve software reliability and hence trustworthiness [has] to be applied *as far upstream as possible*" since it is design, not manufacturing, that characterizes software. Therefore, their goal is to offer "a framework of tools, techniques, and methodologies . . . based on the principles of transformational leadership, best practices of learning organizations, management infrastructure, and quality strategy and systems" focused on software design. To do this, they have chosen to apply Taguchi methods "used in the context of other upstream customer-oriented methods, such as analytical hierarchical process (AHP), quality function deployment (QFD), TRIZ, Pugh concept selection, and failure mode and effects analysis (FMEA), all of which may be applied before a single line of code is written." (It should be noted, however, that at other places, the authors say their design for trustworthy software (DFTS) approach uses things like "the iterative

robust software development model, software design optimization engineering, and object-oriented design technology" to address "producing trustworthy software.")

The book is divided into five parts and 27 chapters:

- Part I: Covers contemporary software development process, including life-cycle models, an introduction to Taguchi methods, software quality metrics, financial perspectives (that is, cost of software quality (CoSQ)), and organizational leadership.
- Part II: Covers tools and techniques, including the seven basic and seven "new" quality tools, the analytical hierarchical process, poka yoke for software, the 5Ss, QFD, TRIZ, Pugh concept selection, FMEA (that is, risk assessment), object-oriented and component-based technologies, and N-version programming.
- Part III: Covers the DFTS approach, including quality measures and statistical methods, a discussion of robustness, applying Taguchi methods, verification and validation, and integration and maintenance.
- Part IV: Covers deployment of DFTS, including "organizational preparedness" and "launching" DFTS.
- Part V: Provides six case studies (by other authors as noted), including CoSQ at Raytheon's Electronic Systems Group (Herb Krasner), IT "portfolio alignment" (Ernest H. Forman), application of QFD to "unprecedented software" and another on "blitz" QFD for software project management (both by Richard E. Zultner, who also authors Chapter 11 on QFD and "voice of the customer"), service and product quality at MD Robotics and Universal Studios (Andrew Bolt and Glenn Mazur), and "resources for QFD and other quality methods" (Glenn Mazur).
Because the book is a textbook, it broadly surveys many basic quality

and software engineering topics. If readers would find ideas for applying traditional quality methods and tools within a software development context of interest, then this book may be of value. For those who already have books on general quality methods and tools as well as software engineering, however, much of this book will simply be a repetition of that material.

Scott Duncan has been a quality/process improvement consultant since 1994. He is the ASQ Software Division's standards chair and Web site chair. He has been a member of U.S. TAG for SC7 Software & Systems Engineering standards since 1996. He is a member of IEEE's Software & Systems Engineering Standards Committee, chair of IEEE 90003 adoption, and chair of IEEE 1648 Working Group on agile methods.

PROJECT AND PROCESS MANAGEMENT

Defining and Deploying Software Processes

F. Alan Goodman.
2006. Auerbach Publications (<http://www.auerbach-publications.com>). 248 pages. ISBN 0-8493-9845-2

CSQE Body of Knowledge area: Project Management—Planning

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

Goodman aims to write "from a lay perspective for a broad reader base," providing guidance on process definition and implementation. However, the term "process definition" can be overloaded. At the higher level, "process definition" can refer to the shaping of the broad software engineering project life cycle, or to the selection and tailoring of more specific process models and frameworks

such as the Software Engineering Institute Capability Maturity Model Integration (CMMI), Rational Unified Process (RUP), and ISO 12207. But this book is instead focused on process definition at a lower level: the real-world activities (he refers to "process elements"), which are actual tasks to be planned and scheduled. The book is generally silent on the use of established higher-level process models, and the tailoring down of such models and frameworks for use in defining tasks and activities in a real-world project. Readers seeking advice on the higher level or more abstract aspects of software process definition will not find much guidance here.

The book begins with plenty of motivation. Goodman speaks from experience when he cites the various risks of process documentation overkill "if you are tackling CMMI or ISO 9001 and don't have processes, you need this book."

The first section of the book contains the basics of process definition or construction. Goodman has lots of practical advice on the nuts and bolts of separating "what to do" (activity) from the "how to do it" (procedure) process elements. He carefully leads the reader through the connection of the activities elements in sequence, within phases, and also stresses the organization of input and outputs. This is the real nitty gritty of planning and scheduling, and will be recognized by any reader with a background in project management. The author takes care to logically connect (or map) each of the basic activities to their associated "how to" artifacts (such as guidelines, examples, templates, and procedures). He also takes care to connect (or map) each of them to higher-level process material, such as role definitions, corporate policies, process owners, metrics, QA, and compliance checks.

Goodman discusses some of the lower-level activities common within software projects, suggesting

Reviews

activities such as "create/update integration plan," "design unit," and "create/update unit test." He clearly understands many practical details of each of these tasks, as he discusses their respective connections to work instructions, inputs and outputs, and quality assurance/compliance checks. The higher-level software engineering wisdom, however, is frequently buried in the discussions of these lower-level tasks and activities.

The reader is left with a great deal of lower-level guidance on the construction of processes, elements, activities, and artifacts, all of which are cross linked into schedules, and also linked to the higher-level process information. All of this low-level detail is not easily absorbed in the absence of a high-level description. The payoff, however, comes in the subsequent chapters, where the relationships and mappings between process elements, and between elements and other process information, are implemented and published.

The second section, "Implementing the Software Process Model," addresses a scheme for intranet Web publication of the various process elements, making not only the tasks and procedures accessible and cross linked via hyperlinks to the other elements of the process model mentioned previously. This scheme is generic, relying upon no

specific tools or publishing mechanisms. Other chapters in this section address other applications related to the implementation of a fully linked and detailed set of process model artifacts on a corporate intranet, such as time collection, project schedule management, and subcontract management. I found these chapters to be the most useful in the book, worthwhile for project managers pondering the advantages and limitations of maintaining and linking on-line the various elements of their respective processes.

The author admits that the book is not an academic treatise, and his writing style certainly could not be called elegant. He typically gets directly to the point, but frequently uses awkward sentence construction and imprecise wording. He also includes a number of negative anecdotes or asides, wherein he cites "ignorant" managers who made "bad decisions" or "just didn't get it." These became irritating, added little value to the discussion of the topic at hand, and seemed unworthy of a professional. The book contains no formal glossary, although the author defines some of his terms in the first chapter. The book also contains no bibliography, although the author refers to ISO and SEI standards, and external documents from organizations such as the Software Productivity Consortium.

This book might be of some value to project managers who are beginning to define, publish, and link the various elements of their development processes. It contains bits of advice pertaining to the various process artifacts, their links, and the discipline of project management, but lacks an effective introduction of software engineering processes (and references to standard process frameworks such as ISO 12207 or CMMI). It might be that project managers in smaller firms will be able to contemplate the author's intranet publishing scheme, while those in larger firms will be saddled with a more specific, pre-existing publishing environment. This book will not be of great value to process specialists who deal already in process abstractions and existing process frameworks such as the CMMI.

Pieter Botman is an independent consultant, with more than 20 years of experience in software engineering and project/product management. He assists companies in the areas of software process assessment/improvement, project management, quality management, and product management. He is a Senior member of the IEEE.

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