Editor: Warren Keuffel ■ wkeuffel@computer.org

Practical Programming Advice

Diomidis Spinellis

The Pragmatic Programmer: From Journeyman to Master *by Andrew Hunt and David Thomas, Addison Wesley, Reading, Mass., 2000, 0-201-61622-X, 321 pp., US$34.95.*

Early in my programming career, I was lucky to come across Brian Kernighan and P.J. Plauger's *Elements of Programming Style* (McGraw-Hill, 1978) and Jon Louis Bentley's *Programming Pearls* (Addison Wesley, 1985). These books deeply influenced me. Since 1988, when the last volume of *Programming Pearls* was published, no book has matched the insight and empathy to programming of those works. Although *Elements of Programming Style* is a classic, its Ratfor (rational Fortran) examples made me hesitant to recommend it to colleagues and students.

Fortunately, the last two years have blessed us with a new edition of *Programming Pearls*, Kernighan and Rob Pike's the *Practice of Programming* (Addison Wesley, 1999), and now Andrew Hunt and David Thomas's *The Pragmatic Programmer* (www.pragmaticprogrammer.com). The progress of technology has brought with it new tools and approaches. Scripting languages, client–server computing, graphical user interfaces, integrated development environments, object orientation, Web applications, and Internet time all provide new opportunities and challenges for today's programmer. *The Pragmatic Programmer* addresses these issues and more, making the authors' experience accessible to all programmers.

Accessible, Self-Paced Learning

The book targets those who know how to program and want to improve their skills. Structured in self-contained sections, it can be read in random order. This lets readers selectively read and apply sections most pertinent to their situation. For example, some sections assume knowledge of Perl. Because the authors rightly consider plain text processing an important skill for addressing complexity, readers can read those sections when they have learned Perl (the authors recommend learning one new programming language each year).

The authors typically condense their discussion of many topics in the form of a memorable one-line tip. These span from the functional, "learn a single editor well," to the practical, "coding ain't done till all tests run," to the deep, "abstractions live longer than details."

Hunt and Thomas have a talent for making concepts accessible and for motivating readers by using inspiring analogies. Revision control systems are fittingly compared to an "undo key" that spans an entire project's lifetime; all programming artifacts, not just source code, should therefore be placed under revision control.

On the subject of refactoring—which both in academia and industry has not received the attention it deserves—Hunt and Thomas assert that software construction shares more common traits with gardening than with building construction—a recurrent analogy in software engineering circles. Plants can thrive, but they need pruning. Minor adjustments can make the garden more aesthetically pleasing, but activities such as monitoring the garden's health, moving plants around, and fertilizing never stop.

Finally, when discussing specifications, Hunt and Thomas ask readers to write a specification for tying shoelaces, the morale being that some things are easier done than specified.

0740-7459/00/$10.00 © 2000 IEEE

Broad Range of Practical Advice

The range of issues *Pragmatic Programmer* covers is large—their common thread is ways to become a better programmer. Thus, we read about human development and social issues, design concepts and methods, a wide variety of tools and approaches for using them, defensive programming, techniques for programming in the large and in the small, requirements, and project management. The pragmatic advice offered similarly varies from the very technical (when debugging, you can sometimes determine the culprit for variable corruption by reading the memory in the variable's neighborhood) to the humane (sign your work and be proud of it).

Although many excellent books cover software design, most of them, written by the very people behind a specific methodology, are necessarily narrow-focused. Refreshingly, we can turn to Hunt and Thomas for a broad and accessible anthology of approaches that work. For example, they suggest when to structure an application around services, offer details on when and how to implement a model-view-controller structure (mind you, not just for graphical presentation), and explain how to use blackboard techniques.

By now, we all know the problems of the rigid waterfall model development processes; it is encouraging to see that the authors provide concrete advice on how to incrementally develop software. The *tracer code* approach they advocate is based on the development of architecturally complete—if not fully functional—code for demonstrating and concept-testing a software system. (The approach gets its name from another analogy: the tracer bullets used to identify a weapon's firing direction). This development mode—also advocated in a slightly different form by the Extreme Programming approach ("implement the most important features first")—complements rapid prototyping as a way to obtain early feedback during development.

A Well-Rounded Offering

The text is accompanied by a number of interesting exercises and aptly termed challenges. It was difficult to resist the temptation of thinking of an answer and comparing it against the one provided at the end of the book. (Which has a higher bandwidth: a 1-Mbp communications line or a person walking between two computers with a full 4-GByte tape? In your answer, document the constraints used.)

To keep the volume of the book manageable, many concepts and ideas are presented in a cursory style and some examples are rather contrived. I got the impression that the authors have enough experience to fill two books—one for each of them. Instead, as the next best thing, they provide references to practical books and pointers to free tools and documentation available on the Web. A selective bibliography complements the material offered. Although the books I mentioned in the first paragraph and Andrew Koenig's *C Traps and Pitfalls* (Addison Wesley, 1988) are unfortunately missing, the annotated bibliography can serve as a shopping list for populating a budding programmer's bookshelf.

I often found myself nodding and smiling while reading the book; the authors have successfully mined nuggets from the current programming practice field and made them available in an accessible and enjoyable form. I will certainly recommend this book to people whose code I will have to read or use.

Diomidis Spinellis is an assistant professor in the Department of Information and Communication Systems at the University of the Aegean. Contact him at dspin@aegean.gr.

Organizing the Rabble

Gerry Coleman

Introduction to the Team Software Process *by Watts S. Humphrey, Addison Wesley, Reading, Mass., 2000, ISBN 0-201-47719-X, 459 pp., US$49.95.*

Introduction to the Team Software Process is the latest in the Software Process Improvement series from the still-prolific Watts Humphrey. Divided into four parts (plus a comprehensive appendix), the book is designed as a process management course (the TSPi) for software teams. A software tool is also available from the publishers, which contains the forms and scripts used in the book.

Like Humphrey's earlier work, *A Discipline for Software Engineering* (Addison Wesley, 1995), which introduced the concept of a Personal Software Process (PSP) and presented practices that individual software developers could follow, this book is essentially designed for use in an academic environment.

Whys and Wherefores

The first part of the book covers why companies need a team software process. Part 2 describes how the TSPi works, Part 3 describes the various roles associated with a software team, and Part 4 looks at working patterns within teams. However, Humphrey suggests a sequence in which the chapters should be read, which coincides with the phases encountered during software development.

Like Humphrey's previous books, many of the ideas highlighted already reside within the public domain. What he has done heretofore, and very skillfully in my opinion, is incorporate those ideas into one coherent, practical whole. Although *Introduction to the Team Software Process* is no exception in this regard, it is very much an introduction to how software teams might use defined processes to their advantage.

The Meat and Three Caveats

But what is the main thrust of the book? Humphrey covers each development phase, from project launch through system testing, in meticulous detail. For each phase, he provides a process script and associated forms that describe the process the team should follow during that particular phase. Each phase then has its own entry and exit criteria and accompanying measures to let the team control the development. He also documents in detail the roles needed during projects, such as team leader, development leader, and quality manager. He primarily focuses on roles in the TSPi. The roles might rotate between team members or, as happens in many teams, one person might occupy more than one role. All of these ideas are well argued and clearly presented.

However, to achieve widespread acceptance and adoption by the software community, the TSP faces a number of significant challenges. Because of the hothouse academic training environment the book envisions, many team structures, formations, and experiments are possible. Yet, how many industrial software teams are self-directed and facilitate the members to choose their own roles? How many such teams operate within a framework where each team member appraises another's performance? And how many software teams operate in the cloistered, bubble-like environment the book describes, without recourse to or liaison with user departments, senior management, customer representatives, and so forth?

Definitely Maybe

Examining the TSP from a project manager's perspective, it is evident that you would have to customize substantially the documented approaches to use them successfully in a development context, where team and project sizes vary and existing corporate processes are often in place. Humphrey states that industrial software teams are carrying out trials using the TSP. As these results are published, it will be interesting to see to what extent the TSP was tweaked.

The TSP's forerunner and prerequisite, the PSP, has not achieved a significant impact on engineers, and based on published studies to date, its success rate seems relatively low and confined to a handful of companies. As the TSP requires practitioners to be PSP-trained, it is difficult to see how the software community can be convinced of the benefits of a new model whose predecessor encountered such marked resistance.

Nonetheless, the book is an excellent overview of the processes and practices that software teams should use. Many of the chapters—particularly Chapter 9 on integration and system testing and the Appendix on configuration management—are extremely well written and should be required reading for all software teams embarking on a development project.

Gerry Coleman is a lecturer in the Department of Computing and Mathematics at Dundalk Institute of Technology. Contact him at gerry.coleman@dkit.ie.