Alex Groce (agroce@gmail.com), Oregon State University

Andrew Hunt and David Thomas' *The Pragmatic Programmer: from journeyman to master* is not a novel, it is not journalism, and it is not essentially "literary." That's fine, and even very good. A column on classics of software engineering should *occasionally* discuss books that are primarily written for people actually writing software to read and use. Nonetheless, this is not a book only for hardcore programmers (this distinguishes it from McConnell's excellent but heavyweight *Code Complete*, and puts it more in the class of books like *Programming Pearls*). There is some source code in this book, but not a lot; much of the advice given is either suitable for people aiming to do anything resembling programming in a "pragmatic" way, or will help educated non-programmers in any field understand what it is that software engineers *do*.

What's a pragmatic programmer, anyway? According to Hunt and Thomas, there are two basic characteristics of pragmatic programmers (and 5 less basic ones, listed in the preface): pragmatic programmers care about their craft, and pragmatic programmers think! about their work (the exclamation mark is theirs, not mine). At first glance, this really has almost nothing to do with the philosophical pragmatism of William James and Charles Sanders Peirce, and sounds a little bit like "Who is this book for? This book is for cool programmers. If you're a really bored programmer who hates programming this book isn't for you!" After finishing the book, however, it becomes clear that the "get things done" rejection of either settling first-principles questions or rejecting all first-principles questions that characterizes pragmatic philosophy is truly central to this book's approach to programming. That is to say, Hunt and Thomas are not dogmatically committed to any methodology (formal or process-based) of software engineering, and primarily explain things by giving examples, elaborating from pithy quotes, and appealing to intuition and experience. They also don't reject all "big ideas" and wallow in a shallow positivism. There's something pragmatic (in both the day-to-day and philosophical meanings) in that approach, and it goes beyond just "good programmers who care should read this book."

The book covers a lot of ground in 259 pages. Elements of agile methodology are in here (tracer bullets, emphasis on prototypes and feedback and avoiding requirements/spec bogs). There's a lot of emphasis on tools (source code control, code generators, really automating builds, and a heartening love of plain text and its manipulation). This is a book *written* for the working programmer, but it is definitely a general software engineering book, with plenty of attention to design, requirements, and (in a broad sense) process rather than just the nuts and bolts of "just coding" (which doesn't really exist, and is very hard to do, to boot).

Testing and debugging are taken seriously and the advice given on those topics is worth the price of admission --- as a "testing guy" I think that using assertions, dying on failure, *really* making sure you add a test to catch every last bug, being ruthless, using as much automation as possible, and contracts are just the right topics to hit given the range of the book. A surprising amount of the things covered in a specialized book like *Lessons Learned in*

Software Testing show up here. The lessons aren't narrowly technical. Instead, they're useful to any work where testing is really important, and centered on the notion that in testing making something fail isn't a bad thing, it's basically the goal of the enterprise.

Perhaps the most repeatedly emphasized idea, however, is DRY. **Don't Repeat Yourself**. The principle of a "single point of truth" is emphasized over and over (ironically, on this point the authors have good reason to repeat themselves, and they do so at length). In fact, a lot of the motivation for orthogonality, for good documentation, for code generators, and so forth is provided by DRY. And DRY really is a critical principle, distinct from any particular language or philosophy of software development. There's no reason to repeat yourself in a project, other than that bad tools make it necessary in some cases.

What makes the book a pleasure to read is partly that all of this ground is covered using three basic techniques for communication (and communication gets its own substantial sections in the book) well. First, there are the succinct, catchphrase reminders, set off from the text as tips, and even gathered in one place at the end of the book. Examples (picked by randomly opening my text and picking the first one I hit) include:

Treat English as Just Another Programming Language
Prototype to Learn
Always Use Source Code Control
Coding Ain't Done 'Til All the Tests Run
Don't Program by Coincidence

There are 70 of these, and most of them stand alone as good reminders (a few don't make much sense without the book's explanation of terms -- you probably don't know what it means to program by coincidence, but Hunt and Thomas do tell you).

Second, there are the quotes. These are less frequent, and seldom stand on their own, but they are often amusing, sometimes thought-provoking, and help make the context of each section seem larger than just a narrow point about programming. The quotations include choice words from the prophet Jeremiah, Ralph Waldo Emerson, Woody Allen, Arlo Guthrie, and Alfred North Whitehead. They're not strictly necessary, but they make the book fun rather than merely useful.

Third, since there is very little source code in the book, the primary method of exposition is analogy and anecdote, which is memorable and convincing. The authors draw examples from history, literature, and other areas, but use enough examples drawn from their own experience on large, real-world systems to ensure the skeptical reader will believe this is testimony from the battlefield, not airy pontification. That's pragmatic.

One definition of a good idea is that when someone who is good at something reads it, they will say "hey, that's what I do already!" in many cases. By that measure, *The Pragmatic*

Programmer is full of good ideas, and worth reading. Even if you already do much of what the book advises, it'll help you understand why you do it. For example, there was much more need in 1999 (when the book was written) to convince people they should start using source code control. Today's programmers are more likely to have religious wars over which source code control tool to use than to ever argue for no source code control. The section on source code control still gives insight into *why* we need it, and has novel insights --- for instance, source control can tie in to DRY and keeping documentation and source both generated from a single point of truth. That's why this is a good book for software engineers: even when it preaches to the choir, it helps the choir see why it's singing.