# Books

**Design Patterns: Elements of Reusable Object-Oriented Software**, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Publishing Co., Reading, MA, 1995. 395 pp. (ISBN 0-201-63361-2).

A best-seller about designing objects? You bet! It's not Judith Krantz, but this book is going like hotcakes. Should you get this book, too? Yes. If you only read one programming book in the next five years, this is the one. It will change the way you program and the way you think about programming.

What are patterns? Patterns depend on the observation that software engineers apply the same solutions over and over. After a few years of professional development, we all put together a bag of tricks, which we apply to new situations. Far from being narrow-minded, it turns out that we all carry around some subset of the same bag. Each pattern records one of these "tricks."

The *Design Patterns* book (familiarly called the GOF book, for "Gang of Four") focuses on patterns that help design objects at a sort of middle scale. The decisions aided by these patterns will help you decide how to divide responsibility between two or three objects at a time.

Did I say "objects"? What if you don't "do objects"? Does this book have anything to say to you? You bet. The underlying assumption of patterns—that as software engineers we rarely create wholly new software structures, that focusing on the things we do over and over will make us more productive—is just as true whether we express ourselves with data flow diagrams, COBOL, C, C++, or Smalltalk. You use patterns, whether you choose to articulate them or not. The examples in this book will help you express yourself, regardless of your medium of expression.

Which plays into the real value behind patterns: communication. Programming languages and environments have gotten good enough that communicating with the computer is no longer the bottleneck in development. Communication between developers, between developers and users, and between developers and managers has become the new bottleneck.

Patterns assist communication by raising the level of discussions. The names of the patterns become part of the spoken vocabulary of developers. By choosing to solve common problems the same way, developers are able to help each other more easily—in design, implementation, review, or maintenance. Instead of having long arguments about how to solve a problem, once a pattern has been identified, the solution is straightforward.

So what about the book? *Design Patterns* presents 23 patterns common to designers working with objects. One of the things I like best about the patterns the authors chose is that each one had to be observed in two independent pieces of software. These are the tricks that everybody uses, or at least everybody should use.

Where could *Design Patterns* be improved?

Many of the patterns betray a bias toward C++. I think this mostly reflects the authors' experience. At the same time, some patterns would have been clearer without it. For example, *Composite* (my pick for number-one cool pattern) explains that if an *Account* contains a collection of *Transactions*, an *Account* should respond to the same messages as its *Transactions*. The pattern is written in terms of inheritance (*Account* and *Transaction* have a common superclass), which is how you implement it in C++. A language that doesn't con-




0018-8670/95/$3.00 © 1995 IBM

fuse subclassing and subtyping does not need the superclass.

The back inside cover has the most interesting bit of information in the book. It shows how all the patterns relate to each other. While fascinating in the glimpse it gives into the authors as designers, it was pretty clearly added late in the game. Patterns should work together more closely to solve big problems. Which of these patterns deserves early consideration? Which are mutually exclusive, or at least alternatives for solving a problem? Which are more common? These are questions that the reader is left to answer with experience.

Related to this is the problem of audience. I get the feeling that the authors wrote this book to help their own practice of design as much as to communicate with others. If you are a smart designer, you will get a tremendous amount out of this book. If you are a beginning designer, you will have to work at it. Some patterns raise more issues than they resolve. I like patterns that take a firmer stand, giving more readers concrete guidance at the risk of being less inclusive.

I've taught patterns to lots of folks, and here's the one absolute invariant: if you don't know the problem, you can't understand the solution. Each pattern helps you solve some common, repeating problem in software engineering. When I try to teach students a pattern, if they are already solving the problem as the pattern suggests then learning comes easy. If they have experienced the problem the pattern is solving, learning is possible. If they have never seen (or noticed) the problem, then the pattern is useless, impossible to learn. So, the only way to learn the patterns is to tie them back to your experience.

If you work alone, I suggest that you skim this book so you sort of know what all is there. Then read one pattern in detail that seems particularly applicable. Go back to your code and see where the pattern exists, or where it should exist that it doesn't. Change your code so that it conforms to the letter of the pattern (naming conventions, etc.). Keep this up, one pattern every few days, until there aren't any more patterns that seem to apply to your work. Six months later skim the book again and see if your "design vocabulary" is ready to expand.

If you work in a group, do the same thing, but coordinate with your colleagues. I've heard from lots of organizations that have a Monday morning pattern reading group. They all discuss how last week's pattern changed their programming, then talk about the new pattern they've read in preparation for this week.

Where should you start? *Composite* will be a real eye-opener but it's easy to misuse. I suggest *State*, the pattern that transforms repetitive case logic into a message to one of several kinds of objects. You will most likely have had the problem (it's the "right way" to code in C). The solution will make your code much cleaner, easier to read, easier to maintain, easier to reuse.

Patterns are a hot topic, especially in the object world. Here are some places you can go if you want to get more involved:

- Wolfgang Pree has published a book about meta-patterns, which is more helpful to pattern writers than to pattern readers.
- The Pattern Languages of Programs (yes, PLoP) conference is held annually. Send e-mail to plop95@parcplace.com for more information.
- Check out the patterns World Wide Web home page (edited by GOF#2 Richard Helm) at URL http://st-www.cs.uiuc.edu/users/patterns/patterns.html.
- There are several active mailing lists for patterns. Check the home page for subscription information.

Kent Beck
First Class Software, Inc.
Boulder Creek
California

**The Mythical Man-Month—Essays on Software Engineering,** 20th Anniversary Edition, Frederick P. Brooks, Jr., Addison-Wesley Publishing Co., Reading, MA, 1995. 336 pp. (ISBN 0-201-83595-5).

Has it really been 20 years since *The Mythical Man-Month* appeared? It seems unlikely. The ground it covers—that managing a large software project is fundamentally different in nature from managing a small one, that producing a product is a different enterprise from producing a program, that people issues dominate technical issues in large projects—holds problems that vex us still.

And yet, has it only been 20 years? That seems unlikely as well. Familiar ideas from this book—the "second-system effect" that projects get to be a year late one day at a time, the surgical team, adding people to a late project makes it later—have always been a part of software engineering lore. Haven't they?

It has been 20 years, after all, and Fred Brooks (Professor of Computer Science at the University of North Carolina at Chapel Hill) has taken the occasion to look back and look around in software engineering. *The Mythical Man-Month* (20th Anniversary Edition) is a reprise of the original, plus new material, and reading it is like revisiting an old friend. Brooks fairly warns us that he's been doing other things than mainstream software engineering research in the intervening years. No matter: Most of us would rather depend on his conjectures than most other people's facts.

First, the particulars: The new book has 19 chapters, 4 of which are new for this edition. Chapters 1–15 repeat the 1975 edition of the book, untouched except for trivial changes. Chapter 16 is a reprint of Brooks's article "No Silver Bullet: Essence and Accidents of Software Engineering," which Brooks wrote in 1986 and which was picked up and popularized (complete with pictures of werewolves and distressed maidens) by IEEE *Computer* magazine in 1987. The paper argues that taming the complexity involved in building software is *inherently* a difficult intellectual task, and most in-vogue technological approaches only address the nonessential aspects of the problem. Chapter 17 continues the theme of "No Silver Bullet," in which Brooks recounts and responds to the significant discussion that the paper generated in the software engineering literature. (Bottom line: "NSB" predicted no order-of-magnitude improvement from a single technological innovation in ten years, eight of which have expired. The bet appears safe, although there are promising approaches.) Chapter 18 summarizes and distills, in note form, the propositions of the original book in order to, as Brooks writes, "invite arguments and facts to prove, disprove, update, or refine those propositions." Chapter 19, the newest of the new material, is an essay that revisits, extends, and updates the key theses of the original work.

Brooks's prose is disarming, like a stroll across the Chapel Hill campus. His North-Carolina-meets-Harvard writing style, reminiscent of his lectures, is relaxed and congenial, but not unscholarly or gimmicky. He very well understands—intuitively, one suspects—the craft of gentle, persuasive discourse. As a teacher, Brooks has always known how to convey information that matters, and *The Mythical Man-Month* matters, a lot: You pay attention, because you're going to need to know this; this is all about what you do for a living.

The fact that the original edition is still in print, is still selling copies, and still arouses enough interest for a 20th anniversary edition attests to the timelessness of its material. This is because, Brooks notes, it is primarily about people and not about computers.

How do the first (original) 15 essays hold up after 20 years? Fairly well; some better than others. The gems about people and organizations endure: why programming is fun, and why it's hard; why large projects are qualitatively different from small ones; why adding people to a late project makes it later; the overarching importance of conceptual integrity to the project as a whole, to its subteams, and to the user; using effective team organization and management discipline to achieve conceptual integrity. Many of the chapters read as fresh today as ever, except for illustrative references to now-antiquated machines, or to issues that have become moot under current technological paradigms. For example:

> The most serious objection [to self-documenting code] is the increased size of the source code that must be stored. As the discipline moves more and more toward on-line storage of source code, this has become a growing consideration. I find myself being briefer in comments to an APL program, which will live on a disk, than on a PL/I one that I will store as cards. (P. 175.)

But, borrowing the "No Silver Bullet" theme, these are *accidental* blemishes of time, easily updated or removed, if anyone cared to. We might have wished for an updated *Mythical Man-Month*; what we got instead was an appended one. The risk is that a new reader may be disenchanted, and may miss the book's *essential* wisdom as a result.

The *essentials* of most of the essays are still quite sound. "The Tar-Pit" (about why what we do is fun and hard), "The Mythical Man-Month," "The Surgical Team" (about team structures for large projects), "Aristocracy, Democracy, and System

Design" (about maintaining conceptual integrity), "The Second-System Effect" (about the dangers of over-designing), "Ten Pounds in a Five-Pound Sack" (about conserving memory), "Sharp Tools," and "Hatching a Catastrophe" (about project scheduling) all continue to resonate strongly.

Somewhat less resonant are the essays about documentation, especially parts that prescribe content. Unit/integration/system test documentation, design rationale, maintenance scenarios, process models, and network management and protocol specifications all have blossomed in the intervening years. And do we really want every team member to be able to access any project document, as Brooks recommends today? Do we really want coders to know in advance the test cases to which their programs will be subjected?

Brooks corrects a few *essential* errors of the 1975 essays himself, in the final chapter.

"It is a very humbling experience," Brooks wrote in 1975, "to make a multimillion-dollar mistake, but it is also very memorable." It is also very memorable to read a candid and instructive account of such an experience, and Brooks's honesty about his own excursion into the tar-pit was one of the best gifts of the 1975 edition. ("A ship on the beach is a lighthouse to the sea," he writes, quoting a Dutch proverb.)

For the 1995 edition, he has given us another candid self-assessment. In Chapter 7 ("Why Did the Tower of Babel Fail?", about communication in large programming projects), Brooks aired the idea (then advocated by Harlan Mills and others) that total disclosure between teams would illuminate errors and inconsistencies early. He compared that to David Parnas's "radical" idea that keeping implementations private would prevent inadvertent, harmful dependencies on changeable details. He came down firmly in the Mills camp ("How, then, should teams communicate with each other? In as many ways as possible . . . While [using precisely and completely defined interface specifications] is definitely sound design, dependence upon its perfect accomplishment is a recipe for disaster.").

Today, he writes simply and candidly, "Parnas is right, and I was wrong about information-hiding . . . I am now convinced that information-hiding, nowadays often embodied in object-oriented programming, is the only way of raising the level of software design."

The other *essential* thesis that Brooks undoes is the acceptance of the waterfall model of development that was implicit, but widespread, in the 1975 edition. The retrospective chapter argues eloquently and persuasively that new models are appropriate in the age of shrink-wrapped software running on millions of computers for which time to market is the dominating driver.

But the retrospective chapter does more than look back; it also looks around. Besides summarizing the effects of the micro-computer revolution on our development paradigms, Brooks argues more forcefully than ever that "conceptual integrity is central to product quality," and reasserts that "people are everything." And he leaves us with a new aphorism or two, including "it is far better to be explicit and wrong than to be vague" in capturing user attributes and needs.

And, we hope, he has seeded new debates. If teams are given free rein to pick their own tools (as he recommends), where is the conceptual integrity for the maintenance organization that will inherit the conglomeration? How can we make sure an incremental development approach such as Microsoft's build-every-night discipline (which he touts) does not simply reward components that arrive first, at the expense of imposing unilateral interaction requirements on their more complex siblings that can least afford them?

But *The Mythical Man-Month* remains a foundation piece. Besides being one of the founding fathers of software engineering, Brooks is a gifted essayist, and we hope that in the future he will turn his lighthouse beacon on the field a little more often.

Paul C. Clements
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania

Note—The books reviewed are those the Editor thinks might be of interest to our readers. The reviews express the opinions of the reviewers.