Books

Software Architecture in Practice, Len Bass, Paul Clements, and Rick Kazman, Addison-Wesley Publishing Co., Reading, MA, 1998. 452 pp. (ISBN 0-201-19930-0).

The book *Software Architecture in Practice* provides definitive answers to the question: What is software architecture? Written for organizations without established architecture practices, this book answers fundamental questions and issues often posed by newcomers to software architecture. The following paragraphs summarize the major ideas in the book.

The Architecture Business Cycle (ABC) is a simple reference model, which is used throughout the book. On any project, a handful of architectural influences affect the designers of the software architecture. Four kinds of influences include: end users, developers, technologies, and experience. The ABC is used to describe each case study in a nutshell. A key message of the ABC is that the resulting system qualities feed back upon the architectural influences.

The book defines software architecture as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." The term *structure* means a model or view of a system. Components are the architectural objects in each of these views. System structures address the architectural influences from the ABC, such as stakeholder requirements. The book's definition of software architecture is sufficiently general in purpose to be compatible with well-known architecture approaches, such as the Zachman framework and open distributed processing. Unfortunately, these approaches are not covered because they are considered too advanced for the book's target audience.

The book defines architectural concepts by establishing several formative taxonomies, including system qualities, architecture styles, and unit operations. *System qualities* are overall characteristics of software systems that are affected by architecture. Example system qualities include: performance, functionality, and modifiability. *Architecture styles* are categories of architectures that share similar components and general patterns of interchange. Some well-known architectural styles include: batch processing, layered architectures, and event-based systems. *Unit operations* are applications of fundamental architecture principles. Examples of unit operations include: architectural separation, abstraction, and resource sharing.

One of the more advanced topics covered is the Software Architecture Analysis Method. This is a technique for evaluating and comparing architecture designs. The technique includes identification of architectural change cases, followed by an assessment of their system impacts. The results are compiled into a simple tabular form and scored.

According to the authors, *Software Architecture in Practice* represents the cumulative results of architecture research at the Software Engineering Institute (SEI), an influential organization in U.S. government software policy. Interestingly, this book contains sections indicating SEI support for the use of CORBA** and design patterns. This book is a follow-on to the SEI's previous work, *Software Architecture: Perspectives on an Emerging Discipline* (Prentice-Hall, 1996), which reported earlier research results. Both books have a nonprescriptive, nonjudgmental philosophy about software architecture. This philosophy is compatible with diverse software approaches, but the books prescribe no specific methods or standards for designing software architecture.

Software Architecture in Practice is an important contribution because it establishes a foundation of architectural concepts that should be acceptable to most organizations. For organizations without software architecture practices, this book provides useful explanations of fundamental terminology and a

0018-8670/99/$5.00 © 1999 IBM

compelling rationale for architecture-based development.

Thomas J. Mowbray
Blueprint Technologies, Inc.
McLean, Virginia

**Trademark or registered trademark of Object Management Group.

Surviving Object-Oriented Projects: A Manager's Guide, Alistair Cockburn, Addison-Wesley Publishing Co., Reading, MA, 1998. 250 pp. (ISBN 0-201-49834-0).

At a glance, *Surviving Object-Oriented Projects* appears to be a new contribution to the collection of "little" books for managers, the kind of book whose 200 pages (plus appendices) can be read on a coast-to-coast flight. In fact, its subtitle (*A Manager's Guide*) and hand-drawn style of diagrams are frankly derivative of David Taylor's original little-OO-for-managers book from the same publisher. However, these surface similarities are misleading; this book is not a "lite" primer on object technology. It is also not a textbook on project management, per se. It is more of a vehicle for attitude adjustment based on pragmatic advice, warnings about potholes to avoid, and many anecdotes from real projects that Alistair Cockburn has been involved with over the years.

There is much to like about this book, starting with its basic philosophy. While many people have pessimistic expectations for projects that are trying "OO" (or any other new technology) for the first time, Cockburn claims that success is possible, even likely, even on your first try, if you:

- Avoid the pitfalls (enumerated in this book) that have been "unpleasant surprises" for other projects.
- Establish a habit of delivering a work product, even if the first drop is what Cockburn calls a "bubblegum" release. This habit plants the seed of thought that "this team delivers" in the minds of the customers and the development team itself.
- Realize and accept that you *will* make mistakes and be prepared to change in order to fix them. Even with good intentions and foreknowledge of the pitfalls, the apocryphal "boogeyman" is out there—what Cockburn calls WYDKYDK ("what you don't know you don't know"). You can't plan for it, because you don't yet know what it's going to be. Cockburn advises: Give yourself permission to be wrong, but also be prepared to do *whatever it takes* to fix it when you find WYDKYDK. "To get your project to succeed," he warns, "you may have to change some of your closest-held wishes and beliefs."

Key to being able to follow this advice is the use of an incremental approach to development. Unless the project is scheduled and staged in multiple increments, there will not be the opportunity to find and fix WYDKYDK, or to improve less-than-perfect or incomplete initial deliveries. The use of incremental development is so crucial that Cockburn lists it as the first of his four critical success factors, and its absence as the first of two key failure indicators. (His other key failure indicator is the use of C++.)

The pitfalls and guidance expressed in this book are not "ivory tower" musings: Cockburn has collected experiences from many different projects, of all sizes, types, and purposes. Chapter 2 summarizes 11 real projects, describing the problem domain, size and experience of the staff, project type, duration, and a history of each project's successes and failures. Many of these projects reappear in the rest of the book as illustrations of the points being made. To evaluate the conclusions from these real project debriefings, it would have been helpful to have the project dates in the profile information. For instance, conclusions about the value of modeling tools, based on projects in the early and mid-1990s, may be less valid now, due to the standardization of a Unified Modeling Language (UML) and the maturation of the modeling tools that are available.

Cockburn's discussion of methodology is not a dry recitation of phases of development. His "big-M" methodology provides a holistic view of how deliverables, standards, tools, and techniques are tied into people, teams, roles, and skills. The discussion is always very people-oriented, and Cockburn's advice is generally pragmatic; he doesn't suggest things that are unachievable in a real-life project. He describes the value of "lite" methodologies where they would be most appropriate for a project. He also discusses the importance of *low-precision deliverables* in the early phases of development. I have often seen developers try to create preliminary "domain-level" models with details more appropriate to implementation-level class diagrams. (Sometimes this is done in an effort to "use everything" in a complex mod-

eling notation.) End result: Analysis paralysis, abandonment of the "incomplete" models, and frantic leap to code. Cockburn reminds us that low precision is sometimes the appropriate choice, in a section that should be required reading for every software engineer who becomes involved in "analysis" activities.

In keeping with Cockburn's focus on Humans and Technology (the name of his company), he gives lots of advice on training-related issues. He points out the high cost of training, and the higher cost of not training: "Yes, you really did spend something like $6000 per person to get three weeks of training. Or you spent double that letting your people wander around on their own without a teacher." (I'd more than double that.) There is also some interesting, and nonintuitive, advice about how to organize teams that include a large number of "newbies." He advocates putting all of the novices in a "training team" (or "day care"), under the guidance of a single expert developer. This training team has very light development responsibilities, while the bulk of the work product is assigned to the "progress team," made up of the experienced developers. This approach is contrary to the more typical strategy of forming a number of mixed teams of several novices and an expert. Cockburn makes a very good case for considering the training team approach.

A feature I like in this book are the eyewitness accounts, stories from the trenches provided by many industry colleagues. My favorite is Luke Hohmann's "Burn Some Pancakes." The appendices are also nice. Appendix A is structured as a set of 12 patterns for risk mitigation. These are excellent reading for all project managers, whether or not they are managing OO projects.

If I had to pick one thing that I don't like about this book, it would be Cockburn's treatment of the "small-m" methodologies, specifically the use of model-based development methods. Cockburn has a long-standing bias away from the use of graphical models. However, this is one area where OO projects may really differ from "traditional" projects, and managers need to be aware of the ramifications. *If* a team is using a model-oriented approach to object-oriented analysis and design, as described in the "popular" OO methods books:

- The schedule needs to accommodate the up-front time (before coding starts) for models to be developed, reviewed, and reworked.
- The team needs tools. Good modeling tools are expensive, and I've seen many teams try to do without. Cockburn's "scanner challenge" ("How many changes are required before the tool beats a pencil drawing scanned into Lotus Notes**?") is cute, but I groan when someone gives that kind of ammunition to managers who balk at spending money to give their teams the tools they need.

Bottom line: As an OO developer, would I want my manager to read this book? The answer is an unequivocal "yes." From this book, the manager would get the right attitudes and some heuristics for success. To provide more of the discipline of project management, I'd match it up with other "bigger" books, such as Goldberg and Rubin's *Succeeding with Objects*, or the IBM Object-Oriented Technology Center's *Developing Object-Oriented Software.* But I'd want my manager to read this one first to establish the attitude and belief that success is achievable. As the author says: "Success is better than failure. Although you can learn some lessons from failure, from success you learn how to succeed."

Susan Lilly
SRA International, Inc.
Fairfax, Virginia

**Trademark or registered trademark of Lotus Development Corporation.

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