Books

Beyond Software Architecture, Luke Hohmann, Pearson Education, Inc., Boston, MA (2003). 314 pp. (ISBN 0-201-77594-8).

Luke Hohmann's interesting book of practical advice on software projects definitely bears reading. It is not the book I expected following a quick glance or even when I began reading. It is not really a book on the technical aspects of software architecture, nor is it entirely a book on the management aspects of software projects. It falls in between these obvious categories and gives a lot of practical advice on how projects should be run and how they should not be run.

The first two chapters start out with fairly high-level generalities about business plans, product management, and quality assurance, but you should not assume the entire book is at that level—there is a good deal more depth to it than that. The first item that stuck with me was the box on page 8 labeled "When you know it's going to fail." This box and the surrounding text give some candid advice on when a project should be abandoned or canceled. The author properly uses the phrase "resumé-driven design" to describe the case when the designers' technical background leads to poor architectural choices. You do not usually get this sort of advice in project management-type books.

Another piece of practical advice I will certainly remember is what the author calls "entropy reduction." He recognizes that some software projects have to meet a deadline even though this sometimes forces you to use some ugly, inelegant, or experimental code. He simply notes that this code should be carefully commented as REDO or even HACK and then should be replaced soon after version 1.0 ships. He also cautions that this phase of cleanup, or entropy reduction, must be limited in scope or else you will suddenly discover that in cleaning up hacks you are also allowing new features to creep in.

Hohmann introduces the concept of architecture at both the technical level and the marketing level, and I think these are useful distinctions. The marketing team's architectural view of a product is of necessity rather different than that of the developer, but this in no way indicates that the marketing view is shallow or simplistic. Rather the marketing view is outward facing and much less concerned with coding details than with how product features meet the needs of the market. That said, I wish he had avoided some rather grating terminology: *markitecture* and *tarchitecture* for the marketing architecture and technical architecture, respectively.

The chapter on portability challenges some of the shibboleths of portability, in particular, that you should write code that will run unchanged on all platforms (the "lowest common denominator") even though you may thus forgo certain advantages that some platforms may provide. The author suggests that you should not avoid using the powerful features of a database or operating system just because they are not available on another supported operating system or database. He further notes that "the only valid business case for creating portable applications is that you will profit by doing so." He in fact is indicating that you should not go to the effort of making applications portable unless you are sure that it will help you make money. He encourages you to consider the costs of supporting multiple platforms

by making a matrix of all the possible configurations and reducing that to a tractable number.

Hohmann describes a number of techniques for building software in accordance with layered architectures, techniques that I think bear some expansion. His description of *spiking* between the layers is probably good advice, but it needs an example to make is clearer. By *spiking,* he seems to mean implementing each feature in all layers before implementing the next feature—as opposed to building all the features at the first layer and then moving on to the next layer. The other ideas in that chapter are also quite useful.

Further chapters on branding, usability, installation, upgrading, configuration, logs, and release management provide useful insight into these common problem areas. The final chapter on security, written with Ron Lunde, is particularly useful, and every project manager should read it. It covers authentication, open and closed systems (such as internal and external e-mail), transaction security, and software security. The authors point out that a license key system is desirable, but you should make sure that the code you use to test for this key must not be so simple that it can be "patched around." They note that security through obscurity is one of the weakest security schemes because secrets may and will be leaked. One of the most persuasive points in this chapter argues against implementing software back doors. It is a far better technical and marketing message to say that your software is so secure that even you cannot break in than to say we will help you crack it if you need to.

Overall, this is a very useful book of advice for anyone participating in, or managing, a software development project, and I recommend it even if your team has only one member.

James W. Cooper
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

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