Book Reviews

Reliable Object-Oriented Software – Applying Analysis and Design

Ed Seidewitz and Mike Stark

Reliable Object-Oriented Software – Applying Analysis and Design, is written by Ed Seidewitz and Mike Stark, published by SIGS Books 1995, ISBN 1-884842-18-6, 368 pages, plus appendices and index, $45.

When I picked this book up, I feared that I might only have YABOO – Yet Another Book on Object-Orientation. But reading the book rewarded me with some interesting new thoughts on fairly familiar subjects. And it further reinforces some old thoughts on analysis, design, and the quest for universal answers.

Readers already familiar with OO will find nothing especially radical in the authors' introductory concepts and definitions. Readers not familiar with OO will find useful, plain language definitions not claiming to break new ground through the use of esoteric jargon.

(Let me get a mild complaint out of the way early. Although the title explicitly mentions 'analysis' and 'design', the authors do not demonstrate rigor or consistency in their use of the terms 'analyst', 'designer', and 'developer'. Sometimes it seems that 'developer' includes the other two, while at other times the authors draw clear distinctions between the three terms.)

The authors note that software builders operate in a less-than-perfect world, where some sense of discipline about roles can help alleviate some potential problems:

> "In reality analysis and synthesis occur continually and in parallel during software development and maintenance. However, a software-engineering approach must impose some discipline on these activities, organizing analysts and developers so that the analytic results properly feed synthetic development." (p 4)

Even though analysis and design typically occur with some overlap, we still must attend to their differences. I especially appreciated the authors' comments contrasting the intents of analysis and design. In reviewing the (much maligned) waterfall method and comparing it with spiral or incremental methods for developing software, they conclude that we cannot discover the one right way to do things for all times and all purposes:

> "Such a separation of analysis and design has the important advantage of clearly delineating what are analysis issues (understanding the functional decomposition of the problem) from what are design issues (creating control structures and modules). On the other hand, this separation requires a discontinuous transition from analysis to design that can prove to be a serious source of errors...
>
> The resulting blurring of analysis and design has the important advantage of supporting (even encouraging) a more incremental approach to development and promoting the continued consideration of problem-domain issues in design and implementation. There is, however the complementary danger of making design decisions during analysis." (p 105-106)

In addition, we see different levels of analysis and design. The authors suggest that an organization would serve itself well by having two clearly different focuses contributing to the overall creation of the whole suite of software systems, especially with regard to hoped-for reuse. They suggest dual tracks: a Domain Engineering Life Cycle which operates over an entire problem domain (perhaps a business area), managed in coordination with a System Development Life Cycle which operates per software system:

> "Domain analysis focuses on developing a general understanding of the problem domain and casting that understanding in object-oriented terms. System analysis, on the other hand, focuses on providing a complete specification of the system to be developed." (p 94)

This thinking can help an organization appropriately differentiate between the system and the project. Thus, the domain half of the organization can focus on the management of software systems as corporate assets, while the development half of the organization can focus on the projects which initially create those assets.

They further underscore this division by noting that each half of the organization will likely have its own value structure, and those value structures will likely clash:

> "Truly high levels of reuse must be based on a firm architectural foundation of trusted assets . . .. Unfortunately, experience has shown that a system architecture that may be just fine in the context of maintenance of a single system does not necessarily provide a good foundation for reuse beyond the initial system. Instead, it is necessary to design an architecture with reuse as a specific objective." (p 19)

The authors note that different sized software development efforts pose different problems. They suggest that the use of OO analysis and design techniques can help bring the software builders closer to the audiences for their work, especially through the use of the language of the problem domain. They point out that the idea of objects seems to hold more natural appeal to the non-technical staff:

> "On a large system-development effort, it is crucial to capture a common understanding of the

> system requirements between the developers and the ultimate customers of the system and to capture the inevitable changes in this understanding." (p 81)

Meanwhile, the builders may have to somehow unlearn the habits of separating data from processes, and learn instead to integrate these two into encapsulated objects. But the builders cannot abandon their ultimate need to exercise extreme precision in their language when required:

> "To guard against unintended ambiguity and inaccuracy, however, it is often very useful to think in terms of more formal concepts such as preconditions and postconditions, even when using a less formal style of specification. (p 165)"

The authors' pracatice of stopping short of lengthy discussions of coding considerations appealed to me. We have here no examples of how we might implement this object in C++ or Smalltalk. The book uses a pseudo-object language for the examples throughout the book section, even in the more comprehensive examples in the third section. I interpret this as a clever attempt to stay in the modes / roles of analysis and design without plunging into code examples. It further underscores the notion that you can do (or at least participate in) OO analysis and design without being fluent in an OO programming language.

I would, however, like to have seen more attention given to the idea of business domain analysis for the sake of business domain analysis. This book, like most, seems to assume that we can only undertake analysis when we have to confront a problem – hence 'problem domain' analysis. The idea that simply having these business models (not constructed subject to the 'tyranny of the project') adds value in itself seems generally lost. Think of the quickness and ease with which we could respond to problems as they arise if we had the business domain modeled already, in anticipation of problems! But perhaps we only have time for such proactive efforts in some alternate universe.

Reviewed by Michael Ayers, 3M/IT Education Svcs, 3M Center 224-2NE-02, PO Box 33224, St Paul MN 55133-3224 — mbayers@mmm.com

Bringing Design to Software

Terry Winograd, ed.

Bringing Design to Software is written by Terry Winograd, ed., and published by Addison-Wesley / ACM Press 1996 ISBN 0-201-85491-0, 320 pages, $29.00.

This new book edited by Terry Winograd includes fourteen chapters reflecting the thinking of fourteen different authorities on one common theme: the practice of software design. The contributors include the likes of Mitchell Kapor, Donald Norman, Peter Denning, and John Seely Brown.

In this thoughtfully compiled suite of pieces, they approach that one theme from a variety of directions. We have pieces on the art and science of software design; on the activities it entails; on good habits for designers; on desirable approaches to design; on cultures supportive of design; on comparing software design to other fields of endeavor. As a result we get a refreshing set of reminders that the world of software design has many facets, and we can choose to focus on any one of them. But if we strive to understand the world of software design better, we need to attend to all of them.

Kapor in 'A Software Design Manifesto' (written several years ago and reprinted here) talks of the need to elevate software design to a genuine discipline with its own special focus: "And the most important social evolution within the computing professions would be to create a role for the software designer as a champion of the user experience."

Another chapter offer opinions on the point of concentration for that discipline. Suggests David Liddle: "The most important component to design properly is the . . . user's conceptual model. Everything else should be subordinated to making that model clear, obvious, and substantial."

Still another chapter focuses not so much on the product of design as on the activities in the process of its creation. Gilliam Crampton Smith and Phillip Tabor point out that the proficient designer must demonstrate clear competence in the following activities (which they emphasize do not constitute absolutely sequential steps) – understanding, abstracting, structuring, representing, and detailing.

Other chapters offer suggestions on mindsets which a successful designer must adopt. Paul Saffo pointedly reminds us, with a wonderful phrase, "We do not use [software] tools simply because they are friendly. We use tools to accomplish tasks, and we abandon tools when the effort required to make the tool deliver exceeds our threshold of indignation – the maximal behavioral compromise that we are willing to make to get a task done."

More than one contributor brings up the ubiquitous question of quality. Peter Denning and Pamela Dargan write that software designers must focus on the use of the software within an action-centered context: "The [traditional engineering] process cannot offer a grounded assessment of quality, because many of the factors influencing quality are not observable in the software itself." Meanwhile, Michael Schrage notes that "The questions that organizations choose not to ask are just as important as the ones that they do ask. This point is particularly relevant in software development..." He maintains that the culture of an organization contributes to the quality of its products. And a successful software development organization must have a culture of prototyping, of genuine trial and error, in order the get the right questions on the table.

In a piece on thinking about what we do, Donald Schon writes about the philosophy of design. "Sometimes, we think about what we are doing in the midst of performing an act. When performance leads to surprise – pleasant or unpleasant – the