> 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

designer may respond by reflection in action: by thinking about what she is doing while doing it, in such a way as to influence further doing." This reflection can lead to new insights just because of the inevitable complexity of the environment. Software design deals not only with the software itself but also with the business that uses it, and the people in the business, and the varied goals of those people, and... "A system is complex in the specific sense that, whenever I make a move, I get results that are not just the ones that I intend. That is, I cannot make a move that has only the consequences that I intend. Any move has side effects."

Winograd sets the tone for the book in a Preface (commenting on the genesis of this compilation) and in his Introduction. In addition, interleaved between the chapters, Winograd provides Profiles of specific products or organizations or practitioners. In a feature I copme to find more and more useful, he also includes an excellent bibliography. If you find any of the writers especially compelling, you can easily track down more work by that individual.

In his closing Reflections, Winograd writes "Quality is perhaps one of the most elusive of the terms that we have introduced. The assessment of quality is at once objective and subjective, personal and political. . . . . In every mature field of design, standards of quality are a focus of attention, even when they cannot be quantified and measured objectively."

I recently spent some months working with a group of people in my organization exploring issues of software design and software quality – and the relative dearth of designers and poor comprehension of quality. I wish this book had come out six months earlier. Anyone interested in striving to design and build quality software would do well to read this book. And, further, to follow up in more detail with specific authors in areas of special interest.

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

## Software Development Using Eiffel

Richard Wiener

*Software Development Using Eiffel* is written by Richard Wiener and published by Prentice Hall as part of the Object-Oriented Series, 1995, ISBN 0-13-100686-X, 425 pages, $35.25.

The intent of this book is to provide readers with an alternative language to C++. It introduces Eiffel along with object oriented analysis and design utilizing the Booch'94 methodology.

Chapter topics are intro to Eiffel, classes and objects, correct programs, generic container classes, inheritance, polymorphism, and C++ versus Eiffel. In addition to these chapters, included are case studies in object oriented analysis and design, an ecological simulation project, a heuristic game project, and a simulated annealing project.

This book is not the standard introduction to Eiffel, as it admits in the Preface. But it is a very good complement to it. In my opinion, this book is a good mix of: an introduction to OOA, the overview of the language Eiffel with a substantial amount of sample source code, and case studies in different areas of computer science. This book is intended for those persons seeking an alternative to C++ or Smalltalk. It is assumed the reader has had an introduction of object oriented concepts.

In summary, I found this book to be very informative, easy to read and understand, and professionally written and prepared.

Reviewed by Ronald B. Finkbine, Department of Computer Science, Southeastern Oklahoma State University, Durant, OK 74701— finkbine@babbage.sosu.edu

## Software Fault Tolerance

Michael R. Lyu (ed.)

*Software Fault Tolerance* is editted by by Michael R. Lyu and published by John Wiley and Sons 1995, ISBN = 0-471-95068-8, 425 pages, paperback.

The intent of this book is to collect the most recent advances in Software Fault Tolerance (SFT). This is a, generally, unknown or misunderstood area of software engineering. SFT is concerned deals with handling errors once they have occurred, as opposed to traditional software engineering areas of software fault avoidance or software fault removal.

This book consists of two major divisions: survey of techniques and models in SFT, and applications and experiments in SFT. In other words, part one is theory and two is applications. N-version programming and recovery-block concepts are covered in both the theory and application sections. Exception handling is covered in the theory section, as well as various modeling schemes. The application chapters, in addition, discuss SFT features in three commercial OS's, a set of software components to detect and recover from software faults, and software fault insertion testing.

This book is appropriate for graduate students and practicing software professionals. This book brings together state of the art research papers on an area of software engineering that is not well known. Most software professionals will find at least a portion (or maybe all) of this book to offer techniques that are a vast improvement to their own software development process. This book was professionally written and prepared.

Reviewed by Ronald B. Finkbine, CS Dept. Southeastern OSU, Durant, OK 74701 — finkbine@babbage.sosu.edu