A Collection of Desert-Island Books on Software Engineering

Colin Potts

College of Computing, Georgia Institute of Technology

potts@cc.gatech.edu

The prospect of being hanged in a fortnight, wrote Samuel Johnson, concentrates the mind wonderfully. The same, I suppose, can be said about the imminence of being stranded on a desert island with a collection of software engineering books. With a limit on the number of books one can take and the stipulation that they all be related to software engineering, the compulsive reader has to make some hard choices and engage in a fair amount of dodging and weaving to convince his editor that the books' relationship to software engineering is not too tenuous. One has to choose one's vestigial connections with civilization carefully. The sad fact, when forced to do this exercise, is that very few software engineering books are even worth reading - let alone being cast away with. Here, though, are six that I think are.

Terry Winograd's recent collection, *Bringing Design to Software* (Addison-Wesley, 1996), is a collection of fourteen delightful essays by various authors on the notion of design as it applies to software. Software engineering has rather co-opted the term 'design' to refer to the boxes and arrows that methodologists say you must draw before you start writing code. But this isn't the sense in which the word is used in many other design and engineering disciplines. Using this broader, and more creative, use of design in software, is very illuminating. The key design decision underlying *Visicalc*, for example, was not what to represent in a module chart but the centrality of the spreadsheet consisting of interdependent cells. The World-Wide Web is designed, for good or ill, around the notion of pages containing embedded URLs. Unix has a few central design abstractions, such as a hierarchical file system and the pipe-and-filter model of software tools. These design concepts are what will be remembered by software historians, and the significance and centrality of none of them would leap out from the page if they were buried in a "design" diagram.

Winograd's collection contains more articles about design for usability, than software engineering as it is usually understood. But, as Mitch Kapor points out in the first essay (and the only one not written specially for the book), software design is rather like architectural design, and, as Vitruvius said two thousand years ago, buildings are judged by their "firmness, commodity, and delight". Kapor claims, and most of the the other writers implicitly agree, that we have spent too much effort addressing the firmness of software, and not enough attention to its commodity (its usability and usefulness), and still less to its capacity to delight. If you are jaded with weak and half-hearted analogies from "real" professions to software engineering, *Bringing Design to Software* may inspire you to think about what we design and how we evaluate it in a fresh way. In particular, you may start questioning the definitions of software "quality" in widespread currency that measure quality exclusively in terms of absence of defects, and particularly the absence of defects of firmness.

Commodity comes down to the degree to which the designed artifact "fits" its domain of application. A classic treatment of what it means for information systems to fit their application domains, is William Kent's *Data and Reality* (North-Holland, 1978). The subtitle of Kent's book, *Basic Assumptions in Data Processing Reconsidered* tells the reader immediately of its subversive intent. Written at a time when the relational model was barely emerging from the database research literature, Kent was one of the first authors to imply that designing information systems was similar to doing applied analytic philosophy, and he was the first to write about it accessibly. It matters, for example, when we consider two objects in a domain as "the same" and when they are different versions of a thing existing in some relationship to each other. It matters that we distinguish symbols and identifiers from the things that they denote. It matters that an information system can be said to "contain" information that is not explicitly present in its database, files or data structures but which can be computed therefrom.

Kent goes further, claiming a kind of Whorfian hypothesis for modeling languages:

> A model is more than a passive medium for recording our view of reality. It shapes that view, and limits our perceptions. If a mind is committed to a certain model, then it will perform amazing feats of distortion to see things structured that way, and it will simply be blind to the things which don't fit that structure. [p. 93]

(That the Whorfian hypothesis applies to artificial languages is readily apparent after talking to a devotee of object-oriented programming for no more than five minutes.)

Although he is concerned with the fit between system and world, Kent's notion of Vitruvian commodity is purely technical. A more socially-informed view is expressed in Bo Dahlbom and Lars Mathiassen's *Computers in Context: The Philosophy and Practice of Systems Design* (NCC Blackwell, 1993), a choice of mine that was pre-empted recently by John Dobson in his desert-island column. The authors of this book take the view that system designers are not so much engineers (or architect-designers) as agents of organisational change, and their book deals in great depth with what such change really means. In the English-speaking philosophical tradition, change is a simple and asymmetrical concept: You do something which causes an effect. This view has subtly permeated much software engineering practice and research. But, claim Dahlbom and Mathiassen, it is wrong. Taking both Hegelian and Marxist views of dialectic processes as a starting point, and in the process explaining Hegel more clearly than anyone else and removing the ideology and political predictions from Marx, they discuss how organisational change through systems intervention is fuelled by the attempt to resolve contradictions in an organisation (what, in the language of engineering, we usually refer to as fixing problems). But systems interventions always lead to new contradictions, and these, in turn, may be addressed by further interventions in a never-ending and seldom convergent process.

Developers who wonder why nobody uses a system, or why users use it in a sub-optimal or unintended way would do well to read this book and take a hard look at the contradictions that they think their design interventions resolve, the degree to which they really do resolve them, and the further contradictions that they engender. Software engineering researchers who find themselves complaining that it is difficult to "get people to use" the fruits of their research might well ask the same questions.

This mention of research and the effects it has (sorry, I meant to say the dialectic in which it participates) leads me to the ground-breaking, thought-provoking, and occasionally infuriating methodological manifesto: *Naturalistic Inquiry*, by Yvonna Lincoln and Egon Guba (Sage, 1985). Lincoln and Guba had social science, and especially educational research in mind in writing their book. It was published at a time when many social scientists had started to criticize the narrowness of quantitative, survey-based or experimental research methods and when supporters of those methods were conducting an aggressive campaign to criticize the unscientific nature of the newer, qualitative methods.

That debate has largely been resolved in educational research and has resulted in an uneasy co-existence of quantitative and qualitative research methods, with qualitative methods having achieved a degree of respectability.

The book is in two halves, only the second of which is desert-island reading. (The first being a polemic against positivism that may have been necessary in the social sciences ten years ago and may even be worth skimming for its relevance to software engineering today.) In the second, more constructive half of their book, Lincoln and Guba discuss in detail the techniques that are necessary to ensure the trustworthiness of naturalistic research results: how to design naturalistic inquiry (including site selection, inquiry phasing, instrumentation and logistics), how to implement an inquiry (including how to gain entry and negotiate consent, building trust and rapport, and qualitative data collection), and how to establish the trustworthiness of one's results (including prolonged and persistent engagement, triangulation, member checks, thick description, and audit).

In software engineering, we have been slow to attempt any research into outcomes. What there has been has followed a standard experimental paradigm and has rested on a tacit base of positivist assumptions. Lincoln and Guba's book is an excellent guide to how one can increase confidence in an inquiry without resorting to a spurious and rhetorical aping of the natural sciences. Its guidelines can also be used during development, because requirements analysis and system evaluation are themselves processes of naturalistic inquiry.

None of this would probably convince Michael Jackson, whose *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices* (Addison-Wesley, 1995) is a wonderfully lucid argument that software engineering really is engineering, but of a completely new kind. Software is description, not artifact, so software engineering is an engineering of descriptions. These descriptions fed into a general-purpose computer give rise to virtual machines, and it is these soft machines that are really the fruits of our labour.

There is little new in this view of what software and software engineering are. What is new is the uncompromising distinction that Jackson would enforce between descriptions of the world and descriptions of the designed machine. Many mistakes in design stem from confusing these two intrinsically different types of description.

Jackson's book is a sign of things to come. It is written as a series of linked essays, arranged alphabetically, and to be read by following cross-references or in any of a number

of sequences laid out at the beginning of the book. Although implemented as a book, it is designed as hypertext, and I would hope ACM Press, which owns the copyright, will publish it electronically as a hypertext. Linear writing styles have evolved for hundreds of years, and most hypertexts, with their choppy, fragmented style suffer in comparison. This book is a model for future hypertext writers in any technical field.

Speaking of model examples brings me to my last selection, which I suspect will come as a surprise. It is Robert Martin's *Designing Object-0riented C++ Applications Using the Booch Method* (Prentice Hall, 1995). "Yet another methodology handbook," I can hear you thinking. But you're wrong. Most methodology textbooks tell you about a method and its notations, illustrate their subject matter with some small examples, and may present more or less dogmatically some rules to follow in applying the method in practice. The problems with this pleasantly simplistic presentation are that the notations are usually too elaborate, the examples can't be used as a source of ideas for the reader's own problems, and the rules are too vague to apply unambiguously. This leaves one with the suspicion about many of these books that the author hasn't actually used the method that he or she is selling.

Martin's book is refreshingly different. The Booch method has a rich vocabulary of concepts, but Martin's treatment is appropriately lightweight (though no more than necessary). The examples are extended and realistic. The rules and guidelines are embedded in and justified by the examples. And the content and presentation unerringly tells me that the author knows what he is talking about (which instantly cuts its field of competitors down to four or five).

What really makes me go overboard [*sic*.] about this book, though, is that Martin describes the method as an unfolding process, not as a way of producing diagrams. In doing so, the book has changed my mind about a key issue in object-oriented design. I had always been sceptical about the centrality of dynamic modeling in object-oriented design, as opposed to viewing object models as glorified entity-relationship models. Martin's book neutralised my scepticism in a way that I think should be used by all authors of methodology handbooks as a model of exposition: the progressive disclosure of more and more complicated facets of a problem that undermine obvious intial solutions. By doing this, the reader is exposed to types of reasoning and the pitfalls to avoid in the contexts where this new knowledge is most likely to be evaluated carefully and to be assimilated for future use. Unlike many methodology books, this one does not pretend that design is easy or that the result is more important than the process of deriving it.

So, there we are: six books that could mitigate the absence of hearth and home, not to mention the pain of missing ICSE and the baseball playoffs. I must confess that in the concentrated state of mind that my imminent exile has caused, I would sooner take Samuel Pepys's unexpurgated *Diary* than any of the six, but that would be against the rules. I doubt, though, whether any of them will be remembered for even half as long.