David Budgen

The Loyal Opposition

Software Design Methods: Life Belt or Leg Iron?

EDITOR: Robert L. Glass • Computing Trends • rglass@indiana.edu

Do software design methods have a future? In introducing the January-February 1998 issue of *IEEE Software*, Al Davis spoke of the hazards implicit in "method abuse," manifested by a desire to "play safe." (If things go well, you can take the credit, but if they go wrong, the organization's choice of method can take the blame.) As Davis argues, such a policy will almost certainly lead to our becoming builders of what he terms "cookie-cutter, low-risk, low-payoff, mediocre systems."

The issue I'll explore in this column is slightly different, although it's also concerned with the problems that the use of design methods can present. It can be expressed as a question: Will the adoption of a design method help the software development process (the "life belt" role), or is there significant risk that its use will lead to suboptimum solutions (the "leg iron" role)? (At the risk of being immediately categorized as a grammatical pedant, I will use "method" to mean "a way of doing something," rather than using the more pretentious-sounding "methodology," which more correctly means "study of method.") To address, but not necessarily answer, this question, I'll first consider what designing involves in a wider context, then compare this with what we do, and finally consider what this might imply for the future.

The Design Process

Developing solutions to problems is a distinguishing human activity that occurs in many spheres of life. So, although the properties of software-based systems offer some specific problems to the designer (such as software's invisibility and its mix of static and dynamic properties), as individual design characteristics, these properties are by no means unique. Indeed, while largely ignored by software engineers, the study of the nature of design activities has long been established as an interdisciplinary topic in its own right, with a well-established academic journal (*Design Studies*).

Continued on page 133

Continued from 136

Studying the problems of design in different domains has produced three concepts that are particularly important in the context of the arguments that I am putting forward:

♦ The need to assume the likely outcome of design in developing the form of a solution.[1] I sometimes liken designing to trying to reverse-engineer something that has not yet been developed! In other words, if we had a solution of this form, what do we think its elements and structure might look like? (I often adopt the analogy of designing a clockwork mechanism for a watch to illustrate this. Given a description of its external form, how do we develop the escapement mechanism and the balance wheel?)

♦ The "wicked" nature of any design process.[2] In a wicked problem, a solution's different aspects are so extensively interconnected that in adopting a particular solution to any one part of a problem, the resulting interactions with the problem itself might make the task of solving it even more intractable. The original concept arose in the context of social planning, but many characteristics of a wicked problem (such as the lack of a stopping rule and the absence of true or false solutions) are readily recognizable facets of software development.

♦ The opportunistic nature of much observed problem-solving activity.[3] Basically, this means that as a solution's form emerges, the problem-solving strategy is adapted to meet the new characteristics that are revealed.

These three concepts challenge the oft-encountered belief that good software engineering design solutions will most likely come from systematically following a prescriptive procedural method. However, we can perhaps take comfort (admittedly of a somewhat limited kind) in that workers in other disciplines also recognize the difficulties that are implicit in design activities!

How Can Design Knowledge Be Transferred?

Here indeed lies the rub. Back in the days (the late '60s and early '70s) when people recognized that a systematic approach to software development was needed to cope with larger-scale projects, it became necessary to find ways of promulgating and encouraging the adoption of desirable practices, such as structured programming. A procedural form (do this, then do this, then...) was one that readily lent itself to this role and had the further advantage that it could be relatively easily conveyed through books and courses. Indeed, methods employing this form, such as Michael Jackson's JSP (Jackson Structured Programming) and the early work of Ed Yourdon and his coworkers, met some real needs. By the late '70s, use of the procedural form was perhaps not so much established as entrenched. Even so, there were "good" practices that did not readily lend themselves to such a form. Perhaps the most obvious one was (and still is) information hiding, for which no satisfactory form of procedural development practice has yet been devised.

In reaction to such shortcomings, the approach to method development in the '80s was essentially "pile on more" (more diagrammatical forms, more models, and, alas, more complexity). A few years ago, I performed an analysis of design methods that involved modeling the transformations between design model states for different methods.[4] This analysis revealed a marked increase in the complexity of both the states and the transformation activities for later design methods. Arguably, much of this complexity stems from what I consider to be the paradox of object orientation, which seems to provide excellent paradigms for analysis and implementation but presents major difficulties for the designer!

Although we academics might be reluctant to admit this, procedural design methods also provide a relatively tractable basis for teaching design and, equally important, to help devise examination questions. (To continue in this vein of honesty, while teaching *about* design might be pedagogically attractive, if by no means easy, knowing how to *use* a design method looks better on a student's CV!) Only in the '90s have we seen attempts to develop other paradigms for transferring design knowledge, mainly through patterns and architectures. However, our present portfolio of software design practices still has little that really addresses those design characteristics I identified previously.

As a final comment on our practices, consider the observation of both Fred Brooks[5] and Bill Curtis and his colleagues[6] that software development often depends on a small number of exceptional designers who "think on a system level." (The additional observation from Curtis that such people might not be particularly good programmers is also significant in this department's context.) One resulting question is, "How did these designers acquire their expertise, and how much did observation, experience, or use of methods contribute?" Could this possibly imply that developing generic design skills might be more important than using methods to transfer procedural knowledge derived from the experiences of others?

The Empirical View

Given that the adoption of systematic evaluation practices in software engineering has been, and continues to be, a slow process, it is perhaps not surprising that little work has been published that evaluates design methods. Indeed, the very idea of conducting any form of evaluation of how a design method is used raises questions about what can usefully be evaluated. Should we be concerned with a method's ease of use, or with the quality of solutions produced (and the criteria for deciding this), or with scalability to larger problems, or…?

If we examine the (admittedly limited) empirical material available, two conclusions emerge:

- Studies of actual design activities have observed only limited use of method practices and have clearly indicated that experienced designers are highly likely to employ opportunistic strategies.[7]
- Studies of method adoption suggest that a method's practices might be modified significantly in use.[8] (You can view this as either positive—if you believe that design practices should not be overconstrained—or negative—if you are concerned about ensuring consistency of practice to aid future maintenance!)

One distinct peculiarity of software design is the extent to which commercial interests have dominated the codifying of associated practices. Many of the more widely known design methods, such as Structured Analysis/Structured Design, JSP, and Objectory, have been developed and marketed largely by consultancies and commercial organizations. This situation has few parallels in related areas such as requirements elicitation or software testing. Although this clearly suggests that industry in particular perceives a real need for design skills, it does not create the most objective forum for considering the question of evaluation. We need to increase the use of empirical studies in this area, while accepting that evaluation is itself a wicked problem, for which we should therefore not expect to obtain true or false answers.

Where Next?

If we accept these arguments, we might conclude that software design methods are at least straining the limits of their effectiveness, and indeed might have overshot them. So what other factors might influence our search for more effective approaches to developing design expertise?

One factor must be the question of how software will be developed in the future. Procedural software design methods are implicitly predicated on a handcrafted approach to software construction, and driven primarily by technical factors. However, the growing emphasis on reuse and components and the associated influence of organizational factors[9] offer major challenges to this assumption. Designing a system to reuse existing components becomes a very different process (but not necessarily a less creative one). It also leads to the complementary question of how we should design components for reuse. As these cultural shifts begin to establish themselves, the use of opportunistic forms and the adaptation of method practices I identified previously will likely become even more marked, suggesting that reliance on methods will not remain a realistic option.

So, if the software design method might be becoming an anachronism (if it isn't already), what other means are available for transferring design experience and knowledge?

♦ *Design patterns* (or idioms). These perhaps offer a closer analogy to the way that we teach design in a programming context ("this is the type of problem where this looping construct is appropriate…"). However, the descriptive forms used to record design solutions usually lack the well-defined syntax and semantics of a programming language. So, although design patterns should be a part of design education, they will not be sufficient in themselves.

♦ *Design architectures*. The concept of software architectural style might yet prove to be more useful than that of the pattern, although it might also provide the syntactic and semantic framework for patterns. (But if so, please, can we agree to stop using such grammatical monstrosities as "architecting" and "architected"?) Architectural concepts seem able to provide a powerful framework for teaching how design solutions can emerge for a given type of problem.

♦ *Tools*. To date, the use of tools to support design activity has largely reflected pencil-and-paper practices, but with the disadvantage of having a less adaptable form. This disadvantage has probably led to some of the evident disillusionment with CASE tools. A further barrier to success seems to be visualization. Just what does an object or procedure (method, subprogram, and so on) look like when viewed as a 2D or 3D projection? Despite these problems, tool support will likely be needed to make both patterns and architectures widely usable and to provide the speed of development increasingly demanded. Can such tools also help develop and transfer expertise?

To return to the question with which I started, is not now the time to accept that the life belt has become somewhat waterlogged and is likely to act more as a leg iron? And if, as a corollary to this, we consider that procedural methods have no future, what might take their place? How can we break the mold—how do we stop pretending that designing software is largely a matter of following a set of well-defined activities, and recognize it as a creative process that requires us to find ways to develop the design skills needed to build the software systems of the future?

This leads to another question: How can we identify, grow, and encourage those talents needed for the great designers who will create elegant and effective solutions to problems? To make real progress, we need to find an answer to this question. (And note that, in the true spirit of the wicked problem, attempting to answer my initial question has merely led to new ones!) ❖

References

1. J.C. Jones, *Design Methods: Seeds of Human Futures*, Wiley Interscience, New York, 1981.
2. H.J. Rittel and M.M. Webber, "Planning Problems Are Wicked Problems," *Developments in Design Methodology*, N. Cross, ed., John Wiley & Sons, New York, 1984, pp. 135–144.
3. B. Hayes-Roth and F. Hayes-Roth, "A Cognitive Model of Planning," *Cognitive Science*, Vol. 3, 1979, pp. 275–310.
4. D. Budgen, "'Design Models' from Software Design Methods," *Design Studies*, Vol. 16, No. 3, July 1995, pp. 293–325.
5. F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, Apr. 1987, pp. 10–19.
6. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM*, Vol. 31, No. 11, Nov. 1988, pp. 1268–1287.
7. S.P. Davies and A.M. Castell, "Contextualizing Design: Narratives and Rationalization in Empirical Studies of Software Design," *Design Studies*, Vol. 13, No. 4, Oct. 1992, pp. 379–392.
8. J. Iivari and J. Maansaari, "The Usage of Systems Development Methods: Are We Stuck to Old Practices?" *Information & Software Technology*, Vol. 40, No. 9, Sept. 1998, pp. 501–510.
9. A. Lynex and P.J. Layzell, "Organisational Considerations for Software Reuse," *Annals Software Eng.*, Vol. 5, 1998, pp. 105–124.

David Budgen is a professor of software engineering and the head of the Department of Computer Science at Keele University. Contact him at the Computer Science Dept., Keele Univ., Staffordshire, ST5 5BG, UK; db@cs.keele.ac.uk.