ISSN: 1742-5921

***eLucidate* Vol. 5 Issue 1, Feb 2008**

Book Review: The Inmates are Running the Asylum

Alan Cooper, 1999

It seems there three universals in this world: truth, falsehood, and what they teach you on a computing degree. We live in a world surrounded by IT, and yet those who have studied IT formally seem often least able to understand user requirements, and hence to create software that genuinely meets our needs. Cooper's book, although published almost ten years ago, provides an idea why that might be. The author himself has a highly respectable track record as a developer – he was responsible for Visual Basic, subsequently sold to Microsoft, so he can claim some understanding of the programming process, and of the programming mentality. So if he says that programming alone is not sufficient, then you are right to take notice. Everyone with an involvement in IT, whether as a user, or as an information professional as a sponsor and influencer could benefit from his assessment of how programmers think.

Cooper is worth reading because he doesn't just preach the Microsoft-wrong, Apple-right message, or even the currently accepted marketing gospel, that of "customer right, developer wrong". His view is considerably more subtle than that, and even if I have some reservations about his solutions, I agree with his diagnosis of the problem. For, as he points out, customer-led design can lead all too frequently to feature overload, to designing for nobody in particular, but for everyone in part.

For Cooper, the zapper to your TV is symptomatic of the problem. It represents "cognitive friction abandonment" when you encounter a problem where the parameters seem to change while you are dealing with it, so that in the end you simply give up. It's true, I only press two or three of the buttons on my zapper. How many programs do we give up on because we can't be bothered to learn the details?

According to Cooper, programmers design for expert users, while marketeers design wizards for beginners; yet most of us are neither beginners nor experts but in the "perpetual intermediate" stage. We know a bit about IT, but not that much, and we can't be bothered to learn more. When we get on a plane our instinct is to turn right, into the welcoming (and instrument-free) cabin, while programmers would instinctively turn left, to the cockpit, where there is a plethora of dials and controls. The solution is not to wish that we, the users, knew more, but to create programs that mean we don't have to know any more than we do currently.

Programmers see things differently. For programmers, the world does not know as much as it should about how programs work. The author recalls asking a classroom of programmers how many of them had taken a clock or mechanical device to pieces when they were younger – most put their hands up. And how many of them, continued Cooper, had managed to put it back together again? Almost none of them. In other words, he explains, programmers willingly trade failure for understanding. What does it matter if the program doesn't quite work, if I know understand what it does? Given that attitude, it's hardly surprising that software works so poorly.

A further insight into software development is "negative feedback". For most of history, adding more features to a product has cost time and money. So you don't add more features to something unless there is clear and ongoing demand for it. But to add extra features to software is more or less free, once the development is done – it adds no cost to the product, and if the product now does more, surely this is a benefit? Of course, the opposite is true. Being presented with several commands is almost always complex, and it usually takes better design to remove controls than to add them. Cooper points out how Microsoft, with the biggest budget of any software company to get its products right, nonetheless loads them with little-used and unnecessary features – unnecessary, that is, for most of us. In all releases of Office up to 2003, minor features and core features would be placed next to each other in drop-down menus, thereby making the selection of any feature more difficult.

There must be a better way than this, cries Cooper, and of course he provides the answer. "Interaction design" is his term for a rigorous design process that should take place before a line of code is written, much like a Hollywood film spends months in preproduction before the cameras roll – it is cheaper, and more effective, when managed in this way. Interaction design involves the use of personas, and Cooper's book provides one of the most convincing justifications for personas I have read. These are imaginary actors whose use of the software is the use case so beloved of modern design teams. Intriguingly, Cooper insists on naming them, which is sensible, but also demands we should have at most three personas for one software program, and in fact implies we should ideally reduce that to one. That's a tough challenge indeed!

In practice, he slightly disproves his own recommendation with the first case study he gives, a project for in-flight entertainment. Passengers want to choose their in-flight movie, and this program was to provide that choice, at the back of every seat on the plane. The solution they came up with satisfied two groups of users. For the technically innocent, who had no knowledge of selecting from menus, they simply lined up all the film posters in one long horizontal band on the passenger's screen. All you did was turn the single interactive scrolling button until you reached a film you wanted, and selected the film. But for more sophisticated users, they provided a horizontal bar at the bottom of the

screen with additional options. So it was possible to provide a use case for two different personas in the one program.

In this way, some of the author's more dramatic recommendations look decidedly risky. Can you really create a successful car that has been rejected by 80% of the focus groups on which it was tested? Cooper's example is the car that meets all needs fairly well – the family user, the single person with shopping, and so on. But by meeting everyone's need partly, the car will never be perfect for anybody.

I'm not entirely convinced by that one, but I am inclined to agree with many of the book's proposals, since so many of Cooper's insights are spot-on. Why is it, asks the author, that computers are so stupid? They throw away information about their user even when the computer has only had that one user in its entire lifetime – yet after years of use, it still doesn't know anything about you.

If you want to know how programmers think, then this is the book for you. In a hundred years, when the terms Microsoft and Apple are like hieroglyphics carved in stone and only understood by archaeologists, the insights about programmers in this book will survive. Like many of these revolutionary guides to software, this book is better at describing the disaster than solving it. We can all agree that existing software is horrendous. But Cooper's vision where companies abandon deadlines and give full control to the interaction designers who can work in a team without developers or outside interference to create a product design is optimistic, and based to some extent on the classic consultant's cry "trust me – I can solve it". I don't think even designers have all the answers to the problems we expect software to solve. But for a guide to pointing out the worst excesses of developer-led IT, this book has no equal.

Michael Upshall