Learning from Experience

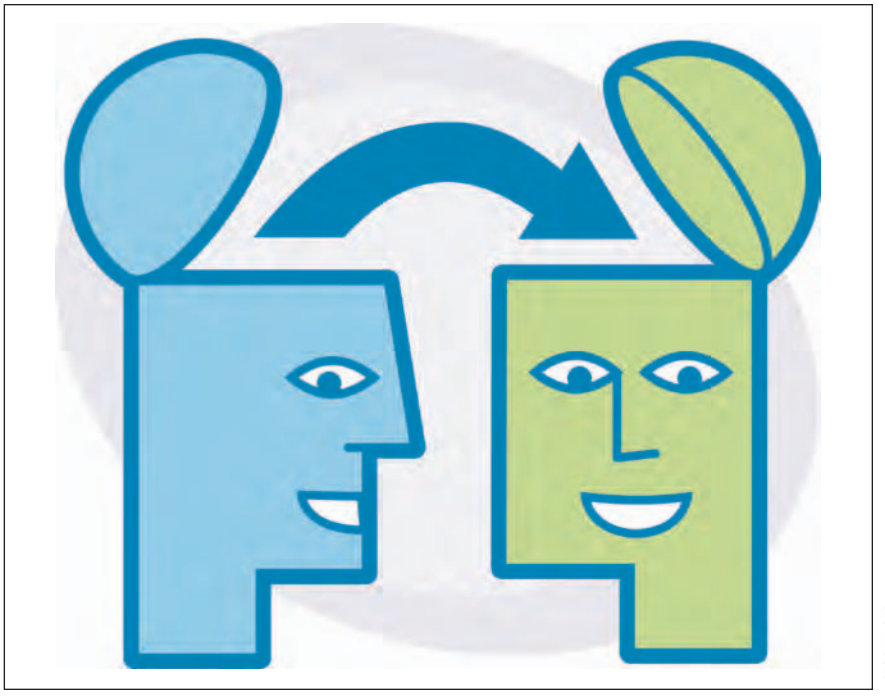
by Michael Bolton

The Social Life of Information [1] describes researchers at Xerox who were puzzled by the reluctance of copier repair people to use the repair manuals and the online information system. Anthropologists, rather than traditional process experts, examined the ways in which the repair people did work, learned skills, and exchanged knowledge. The analysis noted:

> What looked clear and simple from above was much more opaque and confusing on the ground. Tasks were no longer so straightforward and machines, despite their elegant circuit diagrams and diagnostic procedures, exhibited quite incoherent behaviors … Consequently, the information and training provided to the reps was inadequate for all but the most routine of the tasks they faced. Although the documentation claimed to provide a map, the reps continually confronted the question of how to travel when the marked trails disappeared.

In a process model, machines work and break down predictably. You get an error code, and then you replace the part indicated by the error code. Yet machines are idiosyncratic and behave differently based on their age, the conditions of their parts, the interactions between them, which parts have been replaced and which ones haven't, the environment in which they're used (hot, dry, damp, dusty, heavy or light traffic), and so forth.

So how *did* the repairmen cope? First, they learned about the machines as farmers learn about their cattle. They recognized (as the process model didn't) that each copier has peculiarities, strengths, and weaknesses. The repairmen's experience and skill allowed them to recognize general problems versus machine-specific ones. And what did they do when they were stuck? The anthropologist knew (as the managers didn't): The repairmen went to lunch. And breakfast. They met *before* breakfast (the managers assumed that the repairmen's days started at 9:00 a.m., but the anthropologist knew differently). They met for dinner, met for coffee, and played cards. And they talked about work incessantly. They developed collective knowledge, discussed it, refined it; they were resources for one another. What they were trading was not mere information, but *knowledge*.



ISTOCKPHOTO

A while ago, I went to work for a financial institution. On my first day, my new manager showed me around and introduced me to the other testers and to the developers. "Here's your desk, and here (*THUMP!*) is the spec for the product you'll be testing."

This 120-page document had been prepared by professional writers—not bankers nor testers nor programmers. The writers wrote clearly, but they didn't really comprehend the process that they were describing. Bits of the document helped with rapid learning, but not much. As a typical specification, it was meant as a reference, rather than a tutorial. It was authoritative, rather than friendly.

Reading the specification carefully and critically was hard. Its subject was a payment-processing system that included a payment-notification option. The specification called the person who was sending the money the "payer" and the person receiving the notice the "payee." The payee could designate some *other* person to accept the payment and could send that person appropriate information to pick it up. At this stage in the description, the name for the payer magically turned into "sender" and the person who got the money was called the "receiver." So, the payee received the notification but wasn't necessarily the one who got paid. Meanwhile, the receiver got paid but didn't necessarily receive a notice from our system, while the payer and the sender were presumably the same person. Very confusing. The glossary contained a bunch of highly technical banking terms but nothing clear about this sender/receiver/payer/payee business. I needed other strategies

to help me learn quickly.

I asked my manager to describe the roles and the flow of the transactions through the system and to sketch it on a whiteboard. I frequently had him pause to explain things. Later that same day, I had the director of development show me the same process. His account seemed, in places, to contradict my manager's account. I went back to my desk, drew up a concept map and a sketch of the workflow, and asked my manager to critique it. He pointed out several mistakes in my understanding. I asked about connections to external systems and about potential broken pathways on my map. I double-checked to make sure that I could expand and understand the abbreviations and acronyms.

The conversations and whiteboard diagrams helped me learn about the overall architecture of the system. Now I wanted more fine-grain detail. The shop used FitNesse, which allows people to enter tables of examples that contain function names, input data, and expected output. Small chunks of code, called fixtures, link the data in the tables with actual functions in the product, and on the push of a button, FitNesse executes the functions, fills in the actual results, and color codes them for fast interpretation. FitNesse is both a design tool and testing tool, allowing business people or testers to create the tables of examples and programmers to write and test the code to handle the examples properly. FitNesse is also a wonderful requirements tool. Examples can be interspersed with narrative descriptions, diagrams, pictures, sketches, comments—anything that helps understanding. That makes FitNesse a potential learning tool, too.

In our FitNesse wiki, there were … tables. There also were some titles and the odd paragraph of description here and there, but mostly there were tables. The developers were busy writing code and making the tables work. The testers were fleshing out the tables, adding test ideas, and attempting to learn the intricacies of XPATH to try to parse HTML documents. Describing the product wasn't on people's priority lists. I could see a fairly obvious error in one of the test ideas in FitNesse and wanted to correct it. Upon adding a particularly harsh test with a very long string, I found that FitNesse truncated my input. Suddenly I found myself buried in the FitNesse documentation, with new terms to learn, a structure, a syntax, and exceptions and gotchas. As I worked those out, I got help from the developers, the other testers, and the manager. We learned mostly by experimentation and by conversation. I added detail and description where it felt useful.

I needed a tool to generate fictitious credit card numbers and another to convert data from ASCII into EBCDIC (some big banks still use antique encoding systems). Off I went to learn some more Perl. The Perl documentation's descriptions were often incomprehensible, but the examples were clear and adaptable. I made plenty of coding errors at first, but I learned and worked more quickly as my hands got dirtier.

When we needed to add new features, we held meetings in which business analysts, developers, and testers explored, drew, discussed, questioned, conjectured, proposed, and sometimes argued. Drawings helped us understand the flow, conversation helped us work out problems, and arguing helped us refine ideas. The group's culture was to argue, sometimes passionately, but to avoid making the argument personal. We realized that we had much to learn from each other. It was reassuring to see that the programmers themselves often got confused about that sender/receiver/payer/payee stuff.

In his book *Black Box Software Testing* [2], Boris Beizer asserts, "Documents (if they are read) are a more efficient way to transmit process details to individuals unfamiliar with the process and the culture." I used to believe that, but now I believe that *experience* with process and culture—perhaps *supplemented* by documents—is the most efficient way to transmit process details. For this financial group, preparing comprehensive documentation represented *opportunity cost*—time that could be used for more valuable activities. The group reckoned that new testers came on board only rarely, that the ability to learn rapidly was simply a job requirement, and that the culture would support just-in-time training. It did.

Cultures, products, and problem solving can be learned. Training and mentoring can help. But complex products, human organizations, and human skills can't be captured, written down, and then read into someone's head like a computer program. Real knowledge is socially constructed and experiential. Don't mistake the requirements document for the requirements; don't mistake the process manual for the process. **{end}**

References

[1] Brown, John Seely, and Paul Duguid. *The Social Life of Information*. Harvard Business School Press, 2000.

[2] Beizer, Boris. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

When deciding what you're going to document for other people, how do you decide what's going to be helpful and what they'll learn for themselves?

Follow the link on the StickyMinds.com homepage to join the conversation.