What would you do if you believed your company was in the business of manufacturing, or marketing, or services, and you woke one day up to find you are really in the business of software?

“We are at the point where the hardware in our systems is OEM’d, the devices are assembled in China, and distribution and support are outsourced. What we do is design the system and write the software that goes in it.” This observation, by a senior executive at a U.S. electronics company, is becoming common. People who have grown up in manufacturing have a gut-level understanding of what a manufacturing business needs. Executives who have come up through the ranks of marketing know how a marketing company works. Senior managers who were once field managers for a service company are well aware of the needs of a service organization. But what happens when what you do is software, and you’ve never done software?

We are increasingly moving into the knowledge economy described by management guru Peter Drucker [3]. But if the critical asset of the present and future is knowledge, where are we going to keep it? If Drucker is correct, and he usually is, this asset must be created, stored, used, built upon, and carefully protected. Of the five knowledge storage media available to us [2], only one—software—has the necessary attributes of persistence, manageability, and executability that can qualify it as a truly viable corporate knowledge asset repository. Arguably, a primary duty of any corporate executive is the responsible management of corporate assets. Ergo, one of the most important jobs of any executive in a modern organization must be the husbanding of the software resource. But this is by no means a universally acknowledged responsibility. Some companies and some executives, however, are starting to realize that, not only is managing software important, it is what they do.

An Experiment in Learning
A while ago I conducted an experimental three-day program to “teach” executives about software. The target audience was executives and senior managers of companies going through the transition from their core businesses to software. At times through this program, we had financial executives, marketing types, hardware and manufacturing people, an occasional techie, and even—my favorite—corporate counsel. The subject of the symposium was: this is what you need to know if you are an executive and you find your company is “going software.”

The first day of the program was lecture oriented. It involved an
It is in programming that they stood to learn the most—it is by creating executables that they would personally acquire the greatest amount of knowledge.

Most had never attempted to make a computer do anything much beyond word processing. A couple of participants had almost never touched a keyboard. Nevertheless, we imposed a firm schedule deadline (in advance of any detailed requirements, of course) for a functioning system by 3:30 p.m.

Oops (Part One)
The first time this program ran, we had an oops. One of the more technical VPs started up the computers only to get a network diagnostic error. This caused some consternation: what if the machines didn’t work? How can the teams complete their task? Will the program be ruined? Close to panic (it was the very first time this program had run, after all), I dashed to the back of the room and was about to sit down to try to fix the problem when it occurred to me: this was not my problem. I sympathized with the group, hoped they would be able to solve the problem, and reminded them of their deliverable due at 3:30 p.m. Then I left the room.

A marketing VP followed me out and was more than a little angry at being left in the lurch: “How can you expect us to do this job, if you don’t provide us with the necessary resources! We don’t know anything about network
maintenance!” He ranted for perhaps two minutes before he listened to what he was saying. He stopped, thought a moment and then carefully observed, to himself more than to me, “...so, if we don’t have dedicated network resources available to projects, each project will have to learn network maintenance instead of learning what the customer needs.” An Archimedean moment. He turned and went back into the room to help his team solve the network problem (which they did).

Planning, Requirements, and Design
The early stages of the task were project planning and requirements gathering. These executives did uniformly great project plans, and quite good process planning too. However, they did not actually use either of them. Requirements gathering was very variable; some groups did almost none, preferring just to dive in with development and see what happened. Those groups that did solid requirements gathering mostly focused on what might be achieved in the next or even next-next generation product, rather than in this version.

The design of a Web site bears some similarity to the architectural design of larger systems. Good architectural practice dictates that development occurs in the context of an overall design that manages consistent look, navigation, and behavior. This is true of good Web sites and good systems of any type. It never happened in any group.

Oops (Part Two)
I have often described this program to software engineers. For some reason, the idea of turning the tables on the bosses seems to appeal to many. From these engineers, I have had suggestions that we pull the plug on the server at, say, 3:15 p.m., knowing that no one will have thought to make a backup. People have suggested that we force an upgrade of the operating system halfway through the day, or simply change the core requirements or the development environment once or twice. Someone offered that we should simply switch project team members randomly a few times. However, we found we never needed to introduce additional oops’s—they were generated aplenty by the executives themselves. At various times we had overwriting of peoples’ programs, lack of documentation or coordination, accidental deletion of code and Web pages, contradictory development, and version control problems. It was never necessary to introduce artificial snafus—except interruptions.

Interruptions
Three times in the day, especially when it appeared the teams were working well, I would interrupt their work. Sometimes literally dragging people away from their terminals, I would force everyone to stop what they were doing, gather at the front of the room, view a few overhead diagrams, and discuss their projects’ status. The reason for the interruption was quite irrelevant, but the interruption was not. Why did I do this (apart from the fact that I could)? Many executives do not understand the cost of interruptions to the concentration and workflow of the teams they interrupt. By the third disruption, they were starting to understand, and were also starting to get quite angry with me. The status reports I requested at each interruption point were equally unpopular, and generally elicited impatient comments such as “we’re doing OK,” “we can’t exactly say how much longer it will take...,” “...we haven’t been collecting metrics because we’re too busy doing the work...,” and, of course, “...we’re about 90% complete.”

Coding
The environment the executives were using to develop their system was easy to learn and provided many of the functions of a “real” programming development system—they could construct Web pages and write simple supporting code quite easily. They could even generate animations with a little effort. Their creative instincts were supplemented by several CDs full of animated GIFs and assorted decorative tools. All the teams became enamored of the coding activity and their creations, sometimes to the point where each member of a team ended up recreating the same components. Often a team of five would produce five different home pages for their system, but little else. In fact, many
people said they so enjoyed the coding activity and the sense of creating they experienced, that they planned to continue it when they got back to work.

Testing and System Release
Very few teams created anything like a test plan. No team ever followed one. The final presentations were punctuated with comments such as “…this is all we were able to do in the time we had…,” “…we planned to add a menu here, but we didn’t get around to it…,” and my favorite: “uh… it’s not supposed to do that…”

Despite the fact that a functioning system that fulfills the basic requirements could have been created in about one to two hours using the wizard capability of the development environment, no team ever produced a system that actually worked.

Why We Do What We Do
The key thesis of this program is that knowledge is the asset of the future, and as companies transition to a knowledge basis, everyone in a company will start to move their knowledge into an executable software form. This is exactly the purpose of the system the teams are asked to create—just what kind of interactive and executable system would act as a valuable resource to the very people taking this program? The second part of the thesis is that software development is primarily a knowledge acquisition activity. As the executives realized they had spent the day doing what they often accuse developers of doing: playing, working on the “cool” stuff rather than the functional stuff, and hacking rather than carefully crafting, they started to understand the lure of the learning and creativity inherent in building systems. As senior executives, they knew a lot about the business, so by focusing on their role as users and specifiers of the system they would mostly rehash what they already knew, albeit in a different context.

Prior to attending this symposium, few of these senior managers knew how to program a computer. So it is in programming that they stood to learn the most—it is by creating executables that they would personally acquire the greatest amount of knowledge. Even though the stated job was to create a system, the lure of discovery took over. And so they succumbed to the siren song of learning and ignored their own restrictions of planning and process. They also found the reward of creating something that works. People became inordinately proud of what they had built, even if it didn’t actually do anything useful. These executives had discovered the two greatest motivators of the software developer: to learn and to create.

In businesses where software is the business or the means by which the business is conducted, the job of executives is not to stifle learning and creativity under a blanket of control and process and metrics and status reports. Their job is to create an organization that focuses these powerful forces, and fosters a climate in which they are nurtured and realized. They need to learn to channel this energy, not suffocate it.

The Chinese philosopher Confucius is credited with the aphorism: What I hear, I will forget; what I see, I may remember; what I do, I will understand. In many cases senior managers and executives don’t understand software because they’ve never done it. But as the knowledge economy becomes more entrenched and visible, and as software assumes its paramount role as the currency of the knowledge asset, they will.

Executive Code Postscript
Since no team in the program ever produced a system that actually worked, it has never been attempted, but the initial design of the three-day executive program on software included the provision to take a system developed by one group of executives and give it to the next group to maintain and enhance. Now wouldn’t that have been cruel?

References

Phillip Armour (armour@corvusintl.com) is a vice president and senior consultant at Corvus International Inc., Deer Park, IL.