*## Column: Editor's Gossip Column*

# A Modest Proposal—The Sequel

*Whil Hentzen*

No, I'm not going to suggest you start eating your children. Yet.

Let's suppose you wander into the offices of Blackstone in Boston, Flash Creative Management in New Jersey, RDI or IMG in Chicago, The Power Store in San Francisco, a certain custom shop in Milwaukee, or, oh heck, virtually any firm involved in software development anywhere else in the country. Take one of the principals to lunch, and have them spill their guts to you: "What's your most pressing problem for the next six months?"

You'll hear the same thing from each person you talk to: The difficulties in finding qualified technical talent. And it's getting worse. A recent estimate by Steven Levy at the Center for Continuing Study for the California Economy estimates that the unemployment rate for technical specialties is less than 1%. For those of you who slept through Econ 201, this is merely frictional unemployment—folks who have quit their Chips R Us job at noon and are starting with We Be Processors after lunch.

My shop has been scared to answer the phone for months—it might be a potential customer with more work. We'd like to do more projects, but *where do we find the people?* Here are the available scenarios:

1. Hire experienced people. Well, this is pretty much a losing scenario. There simply aren't that many experienced (and competent) people around. Those who are around are really expensive. (And, frankly, there are a lot of expensive people out there who aren't at all experienced.)
2. Hire newbies at a lower rate and train them. Lots of danger here as well. First, while the cash out of the door each week isn't as high, the training costs are much higher, and there's always the risk in losing one of those folks just about the time they become productive. Yeah, I've heard the mantra "treat them well and they won't leave," but each person is different, and reasons for leaving aren't always logical. A parent gets sick, a spouse gets transferred, a 23-year-old just doesn't have enough experience to realize that the grass is pretty darn green around here. A small shop or department only has to lose a couple of newbies this way to get hurt pretty badly.
3. Get more done with the current people—make them work 80 hours a week. Yeah, right. Any shop that requires Herculean efforts from its employees on more than an occasional basis is being run by a bunch of morons.
4. Get more done with the current people—improve their productivity. Hmmm, how might that be done?

Here's what I've been thinking about. Let's suppose that we could look at software like it was widgets. In the olden days (circa 1880), virtually all factories were peopled by craftsmen. Al and Bob are both making widgets, but they both did things their own way. They were artisans, craftsmen, individuals. And while you'd prefer a painting or a symphony constructed this way, the final product embodying the artist's soul and passion, you don't want your toaster, automobile, or payroll system built subject to the frailties and idiosyncrasies of an individual. You don't want your inventory analysis system to exhibit flair, personality, and heart.

Frederick Taylor came along around then, and transformed the construction of widgets from an art to a science. He was the father of the time-and-motion study—the genesis of the folks in the white coats, stopwatches, and horn-rimmed eyeglasses who determine how long it should take to punch a hole in a widget and to move that widget with a hole to the next station.

He broke down each job into the smallest possible components, analyzed each piece, and then assembled the job again, eliminating waste and reducing the chance for error and risk. The result was more work per unit of time. And those who bought into it made more money, got hurt less often, and produced better products. The downside? I'll get to that in a minute.

It's 1997. Why aren't we doing this with software?

Some firms are trying. They create analysts, who meet with customers, determine what needs to be done, and write up specifications. The specs are then turned over to the mole people—programmers who are locked in tiny little rooms, fed a steady diet of Mountain Dew and Doritos, and are given T-shirts when a project is finished.

There are two problems with this scenario. The first is the silly argument that making manufacturing efficient dehumanizes people by trying to turn them into machines. A job was no longer an art—it was rote work. Put Framboozle A into Thingamabob B. Over and over. There was only one right way to do something. It robbed people of their individuality, of their contribution to the work. It turned human beings into mechanical drones.

I would agree, except for one point. If done properly, this system actually empowers people. What *should* happen is those folks with the ability to contribute to the art get involved with the aspects of the process that require creativity. And those who are given the grunt work? What can happen is the opening of opportunities to people who before couldn't have gotten involved at all.

Take the job that involved some boring and repetitive actions. There are plenty of people who need consistency and order in their daily routine. People who don't react well to change, who can't adapt to a multiplicity of conditions—for these folks, a "Taylor-ized" job can be a step up in their life. The key is to match the people with the task correctly.

Okay, so you've got images of Hell's Angels and Man Mountain Mikes pushing around large blocks of iron with their bare hands, and Wally Cox running around in a white shirt and penny loafers trying to examine what they do and not get killed in the process. Let's jump into the typical software development shop. Can we really afford to have some dweeb in a white coat and stopwatch measure the time it takes us to put a multi-column list box on a form?

Let's paint an ideal picture and then figure out how to get there from reality.

In the ideal shop, we'd have an analyst (that would be the department head or the owner), a couple of programmers of varying skill levels, and a QA person. Perhaps an administrative assistant as well. Suppose the company (or department) is smaller. Not as many programmers, and the QA person and admin assistant live in the same body. If the company/department is bigger, perhaps another analyst, a couple more programmers, and if it's any larger than that, then work groups get set up.

When a job comes in the door, the analyst goes off to the customer's site to write specs. Let's talk about specs for a minute. (Actually, I could talk about specs for several hundred pages, but my boss is already making those slashing motions across his throat.) Specifications are like blueprints. It's just that software isn't like metal. If you make a mistake with a ton of iron at 2,000 degrees, you've cost the company a *lot* of money. If you make a mistake with a class library that's going to be used by two dozen applications, well, hey, it's just software, right? It's not *real,* after all. All together now: nyuk, nyuk, nyuk.

So maybe blueprints are a good idea. And, like blueprints for a house, there are two kinds. The first kind is the set of renderings that the buyer sees—what the house will look like, inside and out. But it doesn't list every 2x4, electrical socket call-out, or plumbing joint. These renderings are equivalent to the functional spec—"what you can do with your software."

The second kind of blueprints is the internal set, which describes the thing being built down to the last bolt and quarter-inch measurement. We refer to these as the technical specs—they describe file layouts, internal processes and rules, ERDs, business objects, and so on.

Now that we have all this done, we can turn this over to the programmers, right? Well, maybe not 'til next week. We've got more work to do.

Fundamentals. Training. Getting good at the basics. How many of you have been through C&F with your programmers, page by page, making sure they understand the 200 tools you use most often? No? Any special reason? Maybe you're just assuming they know it all? (Not a good idea, trust me.)

How many of you have company coding standards? Using [] instead of () with array names. Naming conventions. When to use integer data types and when to use numeric. How to handle commenting. Agreement on when a view is more appropriate than tables.

Remember Taylor—that fellow I mentioned a few paragraphs ago? This training is required if you're going to break down software development into the smallest component pieces. Your programmers aren't going to be able to assemble a parent-child screen that processes multiple types of Medicare payments unless they understand that your company standards only use arrays and SQL SELECTS to populate combo boxes, and also understand when to use each.

OK, let's *assume* that this has been done. Anything else? Well, we have tight technical specs, and we have well-trained developers. There still seems to be a link missing. If you wander through a factory, you'll see detailed descriptions of how the work is to be manufactured. Step 1: Put Framboozle A into Thingamabob B. Step 2: Turn Thingamabob B on its back. Step 3: Put two screws into the open holes. Step 4: You get the idea.

All we have to do is create the same type of specifications for writing software. Break each page of the functional and technical specifications into step-by-step instructions and hand them over to the appropriate level of programmer.

Wow, that’s hard work. Maybe we should just start eating our children instead.