1

Buy, Not Build

The very basis of our jobs as developers is to write code. If you can't write code, there is no work for you to do as a software engineer. We spend long years learning to write better, tighter, faster, more elegant code. There comes a time, however, when what we really need to do is *not* write code. It turns out to be one of the hardest parts of the job: learning when to say no and letting someone else do the work. Why should you say no? Isn't writing code what we do? Yes, it is what we do. Yet the reality of the business of software engineering is that business owners or project sponsors or whoever holds the purse strings don't hire us to write code. Sounds odd, doesn't it? But it's true. They don't hire us to write code. They hire us to solve problems using software. Sometimes (in fact, most of the time) that means we get to write code. But more often than we like to think, what we really should do is let someone else do the fun part.

As software systems have become increasingly complex, it has become impossible for any one developer to know everything there is to know about building a single system. We'd like to believe that isn't the case, but unless you are working on a very small project devoted to a very narrow domain, it's true. Someone who is an expert in crafting HTML to look exactly right in every browser and still knows how to write kernel-mode hardware drivers might exist somewhere in the world, but he certainly isn't in the majority. There is simply too much to know, which naturally leads to specialization. So we end up with some developers who know the ins and outs of HTML, and others who write device drivers.

Even in the case of a moderately sized commercial project, no average team of developers will be able to write all the software that it needs to accomplish its goals — nor should it. We get paid to solve problems that businesspeople have, not to write code because it's fun. Okay, the fact that for most of us writing code also happens to be fun comes as a nice bonus, but our business is really *business*.

So what's the point of all that? It is that before we write a single line of code for any project, we should be asking ourselves one simple question: *Is this code providing business value to my customers?*

At the core of that question lies an understanding of what business value means to your organization. For example, if your company sells online banking software, then the core business value that you are delivering to your customers is *banking*. When it comes time to build your banking web site,

and you need to ask the user what day he wants his bills paid, you might want to add a calendar control to your web page. That certainly makes sense, and it will make the user's experience of your software better. It's tempting (because we all like to write code) to immediately start writing a calendar control. After all, you can write one in no time, and you always wanted to. Besides, nobody else's calendar control is as good as the one that you can write. Upon reflection, however, you realize that there is nothing about being able to write a calendar control that provides value directly to online banking customers. ''Of course there is,'' you say. ''The users want a calendar control.'' True, they do. But the business of a company that builds banking software is not to build calendar controls.

Cost versus Benefit

As I said, this is one of the hardest parts of this job to really get your head around. We've all encountered and suffered from the ''not invented here'' syndrome. Every developer thinks that he writes better code than any other developer who ever lived. (That's healthy and exactly the way it should be. It leads to continuous improvement.) Besides, some stuff is more fun than other stuff. It would be fun to spend time writing the perfect calendar control. But what business value does it provide? If you work for a company that builds control libraries and competes daily with a dozen other control vendors, then by all means you should be working on the perfect calendar control. That's your business. But if you write banking software or health care software or warehouse management software, then you need to let those control vendors worry about the calendar controls.

Luckily, there are developers who specialize in writing user interface (UI) controls so that the rest of us don't have to. The same principal applies to a very wide range of services and components. It's not just things like controls or serial drivers.

As the complexity of software has increased, so has the level of support provided by operating systems and base class libraries. Only 10 years ago if you wanted to write a windowing application, you had to start by writing a message pump, followed closely by windowing procedures for each and every window in the application. Most of us would never think of writing code at that low a level now because there wouldn't be any point to it. It's been done; problem solved. If you're writing in C#, there are now not one but two window management systems (Windows Forms and Windows Presentation Foundation) that handle all the work in a much more complete way than you would ever have enough time to do yourself. And that's okay. As software becomes bigger and more complex, the level of problems we write code to solve has moved up to match it. If you wrote a bubble sort or a hashtable yourself anytime in the last five years, then you have wasted both your own time and your employer's money (unless you happen to be one of the relatively small number of developers building base class libraries).

The key is evaluating the benefits versus cost at the level of individual software components. Say that you write banking software for a living, and you need a hashtable. There happens to be a hashtable built into the class library that comes with your development environment, but it doesn't perform quite as fast as it could. Should you write a 20% faster hashtable yourself? No. The time it takes you to write a better hashtable costs your employers money, and it has gained them nothing. Another aspect of the increasing complexity of software is that performance issues have changed in scope. Coupled with the fact that Moore's Law (the number of transistors that can be placed in an integrated circuit doubles every two years) still seems to be working, increasing complexity means that the extra 20% you get out of that perfect hashtable is likely to mean absolutely nothing in the greater scope of your project.