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.

There are a few narrow exceptions to this general principal. There are always cases where some piece of software that is critical to your application has already been written but lacks a key bit of functionality — functionality that might constitute a competitive advantage for your company. Let's say that your application needs a calendar control, and you have one that does just about everything that you need. However, you have just been told that there is a deal in the works with a customer in the Middle East that could mean big money for your company. The calendar control that you have been using doesn't support alternative calendars such as those used in much of the Middle East. Now you might just find it a competitive advantage to write your own calendar control. Sure, it's not core to your business, but if it means getting a deal that you might not get otherwise, then it adds business value.

Unfortunately, it is situations such as this, coupled with the inherent desire to write more code, that makes it difficult to decide when to buy and when to build. It is often not a simple question to answer. It almost always comes down to money. I think that this is one of the core issues that separate ''computer science'' from ''software engineering.'' Software engineers always have to keep in mind that they cost money. Academic ''computer scientists'' have a much different value proposition to consider. Coming up with activities that are grantworthy is vastly different (although no less difficult) from trying to decide which software to write and which to buy to save your company time and money. Writing software that could have been purchased at less cost is a good way to get your project canceled. No project owner is likely to question the need to write software that adds business value to your project if the value is real and demonstrable.

The part of the equation that can be particularly galling to software engineers is that the 80% solution often ends up being just fine. It's hard to let go of that last 20% and accept the idea that the overall project will come out just fine even if every piece isn't 100% the way you want it to be. The last 20% is almost never worth the money that it costs your project's sponsors. This means that if a component that you can buy off the shelf solves most of your problem, it's almost certainly more cost-effective than trying to write it yourself.

Creating a Competitive Advantage

So what should you be writing? The parts that constitute a competitive advantage for your organization. Software is a competitive business, and the part of any project that you have to do yourself is the part that will beat the competition. If you are writing banking software, that's probably the part that talks to the bank's mainframes, and not calendar controls. If you are writing inventory management software, it's probably the algorithms that decide when to order new parts, not the drivers for a bar code scanner. The important part is to focus on the key piece of your project, which provides the most business value to your customers. That is, after all, our real job. Writing code is fun, but what we really do is solve problems for business owners.

It can be very difficult to decide which parts you really should write yourself and which you can leave to someone else. Sometimes the choice is clear. Most of us wouldn't consider writing an SQL query engine just so that our project could use a database. We would use MySQL or purchase a commercial database product. That one seems pretty clear. An SQL engine intuitively feels too big to take on, and it would require such specialized skills to implement that it is easy to see why it makes sense to buy one. Low-level components also tend to be easy to write off. Even if the product must communicate over the WAN, most of us wouldn't start by writing a TCP/IP stack. It's been done. Our value proposition as software engineers lies in solving problems and writing code in a way that *hasn't* been done.

Base Class Libraries

One of the most productive activities you can spend time on is learning your base class library (BCL). Every language and development platform comes with some kind of base class library. Whether that means the .NET BCL, the standard Java libraries, the basic Ruby class library, or the stdlib for C++, you will receive a sizable return on investment if you spend some time getting to know all of its ins and outs. The more you know about your BCL (or basic application programming interfaces (APIs) or built-in methods or whatever they happen to be called), the less likely you are to find yourself writing code that has already been written. It will take some time. Over the years, the code most of us write has moved toward ever-higher levels of abstraction. The base libraries have grown larger and larger over time, and continue to do more for us. That enables us to focus more on business problems and less on plumbing. We don't have to write windowing procedures or bubble sorts, so we can concentrate on complex algorithms and better user experiences.

This is another area where the 80% solution is usually good enough. For example, the .NET BCL comes with a ReaderWriterLock implementation that does exactly what you would expect it to do. It allows multiple readers to access the same resource without blocking, and it causes writes to be done one at a time. Additionally, a write operation blocks all readers until the write is finished. I have heard it said that the .NET Framework 1.1 implementation is susceptible to a condition whereby it basically blocks everyone, even if there is no write operation pending. I've never seen it happen in my code, but the developer who told me about it had looked into the code and proclaimed that it was a theoretical possibility. The solution he proposed was to write his own ReaderWriterLock that would theoretically be faster and not prone to this problem. I would never use this as a reason *not* to use the ReaderWriterLock in the BCL. The fact is that the one in the BCL is done. Every line of code I don't have to write (and more importantly maintain) saves my team time and money. I would absolutely go forward with the BCL version unless I found out that my code was actually experiencing the theorized problem, and that it really had an impact on my users.

That second part is just as important. If I looked at my application while it was running and saw that once a week I got a deadlock on a ReaderWriterLock that caused one web server to recycle itself, I would probably go right on using the BCL class. The fact that a handful of users would have to relogin to my site once a week wouldn't make it worth spending the time and effort to rewrite the ReaderWriterLock, or worse, to try to come up with an alternative solution that worked around it. Consider your actual Service Level Agreement with your customers before deciding to write a single line of code you don't have to.

Of course, your application might not be able to tolerate that kind of failure, in which case you would have to make that hard choice about rewriting or working around the problem class. Making those decisions is what project sponsors pay us for. Before you decide, though, take a hard look at what is good enough versus what you would like to have in a perfect world. The reality is that almost always, you can live with the less-than-perfect solution and still deliver valuable functionality to your customers.

To know which components you really do have to write from scratch, you should be intimately familiar with your platform's libraries. That doesn't necessarily mean that you have to know every method of every class. It means that you should have a broad understanding of which functionality your application might need is provided by the framework, and which is not. That way when it comes up in a project meeting that the project really needs functionality XYZ, you can make a high-level estimate based on whether the class library already has that functionality or you will have to write your own. It may turn out that the BCL already has only 60% of what you need, in which case you may be able to reuse parts of its implementation. Or you might have to start from scratch. Either way you will be making an informed choice.

Open Source Software

Whether to incorporate Open Source software (OSS) into any given project is a huge and potentially controversial issue. Legal, moral, or other questions aside, let's pursue it from the buy-versus-build perspective.

Like any other third-party component, Open Source or other freely available software may save you a lot of time and money by replacing code you would otherwise have to build and maintain yourself. Truly Open Source, or even ''source available'' software is additionally attractive because if any issues come up, you can probably fix them yourself.

On the other hand, there are good reasons for not choosing an Open Source component. Before you decide to adopt any Open Source component, take a good look at the code. There are some very well-designed and built Open Source software projects, and there are some less-well-designed and built ones. If incorporating an OSS component into your project (legal issues aside) means that you are adopting a body of code that you didn't write, it may turn out to be more trouble than it is worth. If the code looks overly complicated or poorly designed, or is difficult to build or test, you may want to think about writing that part yourself rather than relying on the OSS project. You could easily end up spending more time and effort dealing with buggy or unreliable code that is difficult to debug than if you had started from scratch.

A common argument in favor of OSS projects is that there is a whole community of people who can help you solve any problems you have with the project. For a large and active project, that is certainly true. However, just because a project has been released to the OSS world doesn't mean that there is anyone around who is working on it or who knows how it works. Before adopting any OSS component, see how active the community is, how often new releases are made, and how active the source base is. That will give you some hints as to how much help you can expect if you have problems.

There are some very useful, and even a few brilliant, Open Source projects that you may be able to take advantage of to save you time and money on your project. Just keep in mind that ''freely available'' isn't the same thing as ''free.'' ''No charge'' does not equal ''no cost.''

Taking Advantage of Your Platform

As software in general has become more complex, so too has the operating system software that powers computers. More and more services are provided by the operating system ''out of the box,'' and those services are there for you to take advantage of. Operating systems provide simple services such as encryption, as well as potentially complex services such as LDAP directories, authorization systems, and even database systems. Every function, subsystem, or application provided by an operating system platform represents work that you don't have to do, and code you don't have to write or maintain. That can save time and money when building an application.

Design

Just as it behooves every developer to know all the features provided by his or her base class libraries, it behooves every architect to learn about all the services provided by each operating system. Before designing any new system, it’s important to isolate those features that you can rely on an OS to provide and those that you will have to construct yourself, or buy from a third party.

As with any other buy-versus-build decision, the key to this process is understanding a sufficient percentage of your requirements early in the design process so that you can decide if the services provided by the platform are ‘‘good enough.’’

I recently hosted a discussion session that centered on Microsoft’s Authorization Manager, which ships as a part of its Windows 2003 Server operating system. Windows Authorization Manager provides a role-based authorization system that can be integrated into other applications. You can add users, organize those users into groups, and then assign those users and groups to roles. Those roles, in turn, are assigned rights to operations or sets of operations. When your application runs, you can ask Authorization Manager if user X can execute operation Y, and get back a yes or no answer. Authorization Manager represents exactly the kind of software it would be nice not to have to write. It is very useful as part of another application that requires security, but it is functionality that is ‘‘generic’’ enough not to provide what for most applications would represent a ‘‘competitive advantage.’’ One of the participants in the discussion asked, ‘‘Why would I use the one provided by Microsoft when I could just write it myself?’’ The implication was that if he wrote it himself, he would have full control and could achieve that perfect 100% solution instead of compromising by using the Microsoft implementation. My response was exactly the opposite: ‘‘Why would I write it myself when Microsoft’s already done it for me?’’

Do I really need more than what is provided? In all likelihood, no. Even if there were ‘‘nice to have’’ features that were missing, I might choose to work around those deficiencies. Why? Because there is nothing about a simple role-based authorization system that gives my product an advantage over the competition. It then becomes more advantageous for me to save the time and money it would take for me to write the 100% solution. Delivering software faster and cheaper represents more of a competitive advantage in many cases than adding additional features.

Risk

There is always a risk, however. If you bank on functionality provided by the operating system (or any third-party software, for that matter), you could find yourself backed into a corner. Months into your development process, it might turn out that some functionality that is critical will be impossible to implement using the third-party software. If that happens, the only alternative is to write it yourself after all, only now you have to do it with time and resources you didn’t budget for at the beginning of the project.

The best way to mitigate this risk is by building a ‘‘baseline architecture.’’ If you can build out as much of your design as possible in ‘‘width’’ rather than ‘‘depth’’ (meaning that you touch as much of the macro-level functionality of the system as you can without going into detail), you will have a much better estimate of whether any third-party systems will meet your needs. It is particularly important to include third-party or operating system functionality into that baseline architecture because that is where the greatest risk often lies. As long as you can implement enough critical functionality in your baseline, you can be reasonably sure you won’t end up backed into a corner later on.

Building a baseline architecture is discussed in greater detail in later chapters.

Services

Operating systems provide a wide range of services. Some of those services are in the form of simple method calls, such as encryption or text formatting. Others — such as messaging or networking services — are more complex. Once upon a time, applications had to include their own routines for 3D graphics, but now those are provided by subsystems like OpenGL and DirectX. Some operating systems even provide whole applications such as an LDAP directory (Microsoft's Active Directory Lightweight Directory Services is one example) or a database (such as the Berkeley DB that ships with many *nix systems).

The one thing that all of those services have in common is that they represent functionality that can be exploited by developers writing non-operating-system software. Every time you can take advantage of such a service, you are not writing code that you will have to maintain someday. That shaves valuable time off of your development schedule, and can save a great deal of money in maintenance. Supporting code already written can be much more expensive than developing it in the first place, so code not written is that much cheaper.

In terms of risk management, the larger the scale of functionality you rely on the OS providing, the riskier it is. If all you need is an encryption routine, the risk is very low. It either works or it does not. If you are relying on an OS-provided LDAP directory, on the other hand, you had better do some baselining to make sure that it really does what you need it to do before committing to it. That is a much more risky proposition, and it must be approached with considerably more caution.

As with any other buy-versus-build decision, take a look at the features provided by your operating system and decide which parts of your software are core to your business and which parts you can rely on someone else to write and maintain. Write only the code that provides your software with its competitive advantage, and leave the busy work to other people whenever you can.

Third-Party Components

In addition to features provided by base class libraries and operating systems, there's a third category of code you don't have to write. That is third-party components provided by ISVs (independent software vendors) — which basically means everyone except us and the operating system vendors. There are a number of types of third-party components, ranging from small things like networking libraries and sets of UI components, all the way up to very expensive and complex applications such as messaging systems like IBM's MQ series.

The reason to choose any of those third-party solutions is that they represent work you don't have to do. That has to be balanced against the risk associated with not being in control, although there are often ways to mitigate that risk. Many component vendors will provide you with source code at an additional cost, which is a great way to lessen risk. If you have the source code, it is easier to diagnose and possibly fix any problems that may come up, or to add new features that are specific to your application. Other vendors, particularly smaller ones, are happy to make changes to accommodate your needs in exchange for a commitment to purchase *X* units, and so on.

Evaluating third-party components can be tricky. Every vendor will tell you that its product will solve all of your problems, so the only way to know for sure is to try it out. Purchasing a third-party component based solely on the demos given by technical salespeople is just asking for trouble. Get a trial copy, a sample license, or whatever will allow you to get that software in your hands for real testing. Try it out in what you think is a real scenario for your business to see if it really does what it is supposed to do.

Take the time to evaluate not just the product itself, but also the documentation, samples, and developer support. The most feature-rich of third-party components is useless if you can't figure out how to program against it or can't get support from the company you purchased it from.

One of the most common places to use third-party components is in the user interface. User interface code and components such as calendars, date pickers, and grids are excellent candidates for the use of third-party software. UI components are necessary for most applications but are hard and extremely time-consuming to develop. A good UI control should be easy to use, feature rich, and support complex user interaction, all of which means lots of code to write, and a great deal of testing. It would be nice, then, to just buy someone else's controls.

There are dozens of vendors for every platform providing rich suites of UI widgets. Unfortunately, the user interface is often where applications have the most detailed and stringent requirements. Most companies have a very specific vision of how users will interact with their applications, and very detailed requirements regarding look and feel. The details of how an application looks and interacts with the user can end up being a competitive advantage and differentiates applications from one another in terms of user perception. For that very reason, UI components are among the hardest to shop for. It can be very difficult to find components that provide all the functionality you want and are still configurable enough to match your application's look and feel. Luckily, control vendors are often willing to work with you to meet your needs, including writing custom code or a customized version of their components. Those are details that need to be worked out ahead of time.

It is also very important to spend some time working with the different suites available to evaluate their ease of use, documentation, customizability, and feature set. Don't assume that you can learn everything about a set of components by getting the demo or looking at sample applications. Try them out to see for yourself before committing to any specific suite. If source code is available, try to get a sample so that you can evaluate how easy it would really be to make changes or debug problems if you had to. You can save large amounts of time and money by relying on off-the-shelf user interface components, but you do have to choose carefully from among the available options.

Summary

Writing code is fun, and it is what we do, but ultimately it is not what we get paid for. Providing business value to your project's sponsors can often best be achieved by purchasing someone else's code. It can be difficult to balance the savings incurred by not writing code against the risks posed by adopting and relying on code you don't own, however. Focus on writing the parts of your application that provide the most value to your project sponsors, and buy everything else if you can. Building a baseline architecture that incorporates the functionality of all the third-party software is one of the best ways to mitigate the risk of adopting third-party code and helps ensure that you find any potential problems early on.