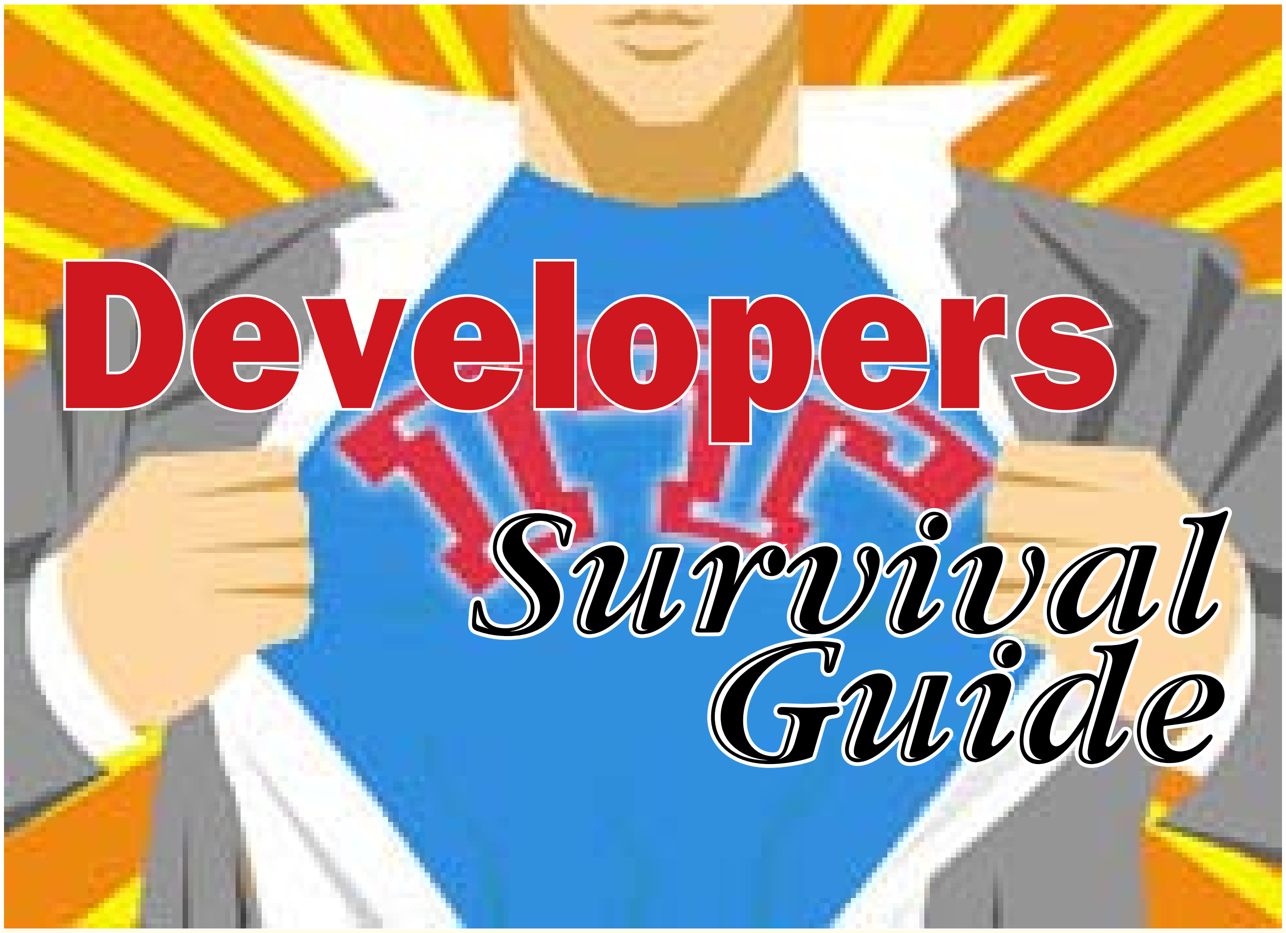
InfoWorld

SPECIAL REPORT



Developers Survival Guide

Table of contents

INTRODUCTION

Programming trends to watch 2

BEST PRACTICES

7 deadly sins of developers 7

Sidebar: 10 business skills every IT pro must master 6

Earth to developers: Grow up! 9

Beware these developer errors. 11

Sidebar: IT job spotting: Top 20 metro areas for tech jobs 12

Don't be afraid to rewrite code 16

TRENDS

Programming languages on the rise 13

INTRODUCTION

Programming trends to watch

New tools, techniques, and troubles are changing how developers work

By Peter Wayner

Depending on your perspective and proximity to the bleeding edge, the world of programming evolves either too fast or too slow. But whether you're banging out Cobol or hacking Node.js, one fact remains clear: Programmers must keep an eye on the latest programming trends to remain competitive in ever-shifting job markets.

From JavaScript everywhere to everything on the JVM, the times and the tools are a-changing. So too is the way programmers work, thanks to the rise of frameworks and walled gardens, as well as a shift away from openness. Concerns around bandwidth, energy, and scalability are finding a place at the programming table, as are parallelism and the video card. There's so much happening that you might find yourself thinking of going back to school, if only traditional education wasn't fading from relevance.

Every so often, our understanding of the ways and means of programming needs to be renewed or replaced just like an operating system or a library. Here we offer a look at 11 recent trends that capture how programming is changing.

PROGRAMMING TREND NO. 1: THE JVM IS NOT JUST FOR JAVA ANYMORE

A long time ago, Sun created Java and shared the virtual machine with the world. By the time Microsoft created C#, people recognized that the VM didn't have to be limited to one language. Anything that could be transformed into the byte code could use it.

Now, it seems that everyone is building their language to do just that. Leave the job of building a virtual machine to Sun/Oracle, and concentrate your efforts on the syntactic bells and structural whistles, goes the mantra today.

As a result, the list of JVM-dependent languages is long and growing. Ruby lovers like how much better JRuby works under heavy loads. Scala and Cloture allow developers to write code that is more functional and often faster than Java when running on the same JVM.

Even Java-heads like to use the JVM without writing Java. Take the scripting langauge Groovy, which is fully integrated with the JVM and Java API. Write Groovy shorthand, and if you also adopt Grails, you can enjoy Rails-like coding-by-convention. Need to link in Java libraries? Voilà. It's designed to work seamlessly, giving you all the power and stability of Java without the curly brackets.

PROGRAMMING TREND NO. 2: JAVASCRIPT IS NOT JUST FOR JAVASCRIPT

The JVM isn't the only cross-platform solution open to all comers. JavaScript, the langauge your kid sister uses to add an alert box to her band's website, is not just for JavaScript coders any longer. The list of languages that cross-compile to run on the fancy, just-in-time JavaScript engines is even longer than the list that runs on the JVM.

Take Google Web Toolkit. You write Java code like you're writing for the Swing framework from the '90s, and the GWT compiler turns it into JavaScript that runs in a browser on a desktop, smartphone, or tablet. There's no need for a Java applet plug-in or JVM on the client because JavaScript in the browser offers machine independence.

One of the newer arrivals is CoffeeScript, a shorthand language that's compiled down to JavaScript by inserting all the punctuation that scripting-language users hate to type. The idea is so popular that there are already CoffeeScript spin-offs like Coco, Parsec-Coffee-Script, and Contracts-Coffee-Script, each of which adds its own sophisticated metaprogramming structures to make it easier to spin out elaborate code.

Some extensions are so successful they've almost become languages unto themselves. Think of all the Web developers banging out workable code with jQuery, without remembering or knowing anything about JavaScript scoping.

If that's not enough, there are experiments linking pretty much any language to JavaScript, including Ruby, Python, Lisp, Scheme, Haskell, and OCaml.

PROGRAMMING TREND NO. 3: PLANTATIONS EVERYWHERE

There's a dark side to these tightly integrated stacks of code: the walled garden.

The Internet began with the premise that there would be no gatekeepers. Every packet would be delivered to its destination, with our data free to wander. Alas, that promise is eroding, and not because the ISPs are increasingly turning to traffic shaping or deep packet inspection technologies.

These days, everyone seems to be retreating to walled gardens, where everything is safer and simpler. If you want to develop for the iPhone, you'll have to write code to Apple's vague specifications, then Apple — and Apple alone — will decide whether it will run on its machines. It's not up to you, the programmer, and it doesn't matter what the users say, either.

It's not just Apple. Creating games for Facebook means getting Facebook's permission to connect to Facebook users. It doesn't matter how many people click the Like button if Facebook decides to lock out your code.

Only a naive programmer thinks that other companies won't follow along.

There are deeper problems with walled gardens, beyond loss of control. Purveyors of walled gardens could very well keep the lion's share of the income derived from the work of independent developers.

These walled gardens also threaten to balkanize the coding world into separate camps according to language. One look and you can see programmers moving from stubborn individualists in the open frontier to hired hands. Welcome to the new plantation.

PROGRAMMING TREND NO. 4: NO CODE IS AN ISLAND

A friend once told me he was heading to the woods in Northern Michigan where his father and uncles built a cabin by themselves. It was theirs and theirs alone. They hauled the wood and the rock, and they could sit afterward and gaze at the sunset with deep pride in their accomplishment.

Writing a program used to be like this. Push the compile button, and after it churned, the code would take over your machine. Sure, it was interacting with the OS layer, but it was easier to point to a tangible thing that you built, just like that cabin. See that file with the EXE suffix? I built that one, mom.

That distinction is disappearing. Our code is living increasingly in ecosystems. Many PHP programmers, for instance, create plug-ins for WordPress, Drupal, Joomla, or some other framework. Their code is a module that works with other modules.

Sometimes the fragments are even smaller, just bits of code dropped into fields. Many Drupal modules can be customized with PHP, for example. The programmer is just filling out forms with snippets of code, not building something new that stands alone.

When these mechanisms work, the results can be uplifting. More often than not, the results are mixed. While your snippets can work well with what's under the hood, they often need plenty of debugging. In many cases, the errors come from deep inside the system where you're not supposed to be looking. And there is little documentation because no one expected you to interact with the system this way.

In the worst examples, the errors come from someone else's snippets, and there's no way to debug the two simultaneously because they own their code and you own yours.

These tools often work for small extensions and simple tools that have been anticipated by the authors of the original framework. Anything else is an invitation to hit-or-miss debugging cycles. There's no substitute for having all the source code available to be read and traced with your own stack, but that's becoming less common.

PROGRAMMING TREND NO. 5: OPENNESS RETREATS

For all the success of open source software, the ability to engage in pure tinkering is slipping away in many corners. The success of the iPhone has everyone looking to find ways to wall off the commons. Sure, the new car computer systems are built with Linux, but don't for a second think you'll be typing "make" and deploying to your car.

Even if we concede that it would be creepy and dangerous to reprogram your brake system, why can't we hack the nav system? The car companies are touting how their fancy computer systems integrate with your phone, but they're not open the way your desktop is open. Hardly anything is as open as the desktop used to be. Even desktop systems may be more locked down, with some wondering whether the new secure booting infrastructure for Windows 8 will make it impossible to run alternative OSes.

This is bound to limit innovation in the future. After the garage hackers and programmers finish building tools that put a smile on their faces, they turn around and create companies that do the same tasks for the average person. Slicing off the open source experiment in this area destroys the aftermarket. And it becomes harder for companies to hire the programmers they need because open source tinkering produces skilled programmers that can fill jobs.

There remain glittering exceptions, usually when the code is shared between programmers. Some projects like Apache still thrive and attract the kind of financial support they need to pay top talent. Github and Sourceforge continue to add more projects. Others work well for developers experimenting with the bleeding edge. But there are few examples of pure openness succeeding with the end consumer, who seems drawn to the siren call of proprietary gardens.

PROGRAMMING TREND NO. 6: BANDWIDTH IS NO LONGER FREE

Web programmers have grown up believing bandwidth is free and getting ever faster. No need to worry about slow download times — in a year, everyone's connection will be zippier, and the problem will disappear. Unfortunately, those days are over, thanks to more and more ISPs adding bandwidth caps and metering.

Regardless of whether you see this as a need to crack down on bandwidth hogs destroying the commonwealth or as a power grab by those who own the pipes and, by coincidence, want to sell pay-per-view video feeds, bandwidth is something programmers need to worry about consuming.

This will change many of the gimmicks built around the cloud because traffic from your home machine to the cloud will be metered. Will radio stations be able to stream every bit that we hear and still make enough money on the pennies from ads? Will online backup be viable?

Optimizing bandwidth consumption when designing apps is becoming imperative. Minimizing JavaScript files and CSS files isn't just for speed; it also saves bandwidth. If programmers don't heed this trend, users of their code could be driven away by higher bandwidth charges in the near future.

PROGRAMMING TREND NO. 7: ENERGY IS NO LONGER FREE, EITHER

The cost of keeping a computer plugged in has never been an issue. It never mattered how much energy your rack of servers sucked down because the colo just sent you a flat bill for each box.

No longer — energy consumption is a big issue, whether you're programming for smartphones or the server farm. The biggest limitation of my Android phone is that it can drain its battery in 8 hours doing nothing but sitting there. Design an app that eats up battery power faster than GPS features do and watch downloads of your app plummet.

The problem is less understood by server programmers, who could always take power for granted. You worry about speed, but rarely the cost of energy for completing a database transaction. Google is one of several companies in front of this issue, investing in finding the lowest-cost electricity to do extensive searches. It's likely the company is already deciding how to fine-tune a search based on energy costs and how much ad revenue the search will generate.

Cloud computing is helping make this issue more obvious. Some of the more sophisticated clouds — like Google App Engine or Amazon S3 — don't bill by the rack or root password. They charge for database commits and queries. While this is a new challenge for most programmers, it's making the cost of energy more transparent. Get ready to start thinking about the cost of each subroutine in dollars, not in lines of code or milliseconds of execution time.

PROGRAMMING TREND NO. 8: TRADITIONAL EDUCATION NOT RELEVANT

Ask any project manager and they'll say there's not enough talent from top-tier computer science departments. They may go so far as to say they would hire a new CS major from a top school without reading the résumé. But ask this same desperate project manager about a middle-aged programmer with a degree from the same school, and they'll hesitate and start mumbling about getting back to you later.

Indeed, it isn't unheard of to find major technology companies complaining to Congress that they can't find Americans capable of programming, all while defending themselves in age-discrimination lawsuits from older programmers with stellar résumés and degrees from top universities.

Some of this may suggest that education doesn't have the same value it used to hold. Older workers with degrees that used to be valuable are saying companies want only young, unfettered bodies that will work long hours. It leaves you to wonder whether it's the age and implied lower pay expectations, not the knowledge that makes fresh college graduates so desirable.

Others are simply moving beyond such questions, looking instead to exploit what they see as a market distortion caused by our infatuation with the four-year diploma. Venture capitalists are paying top talent to skip their undergraduate years. Others are actively recruiting people with odd degrees and pushing them through a boot camp that teaches them practical skills, not the theoretical analysis common in university courses.

The most prominent rejection of a traditional university education is the program run by PayPal founder Peter Thiel. He's recruiting top programmers who are just leaving high school and paying them to "stop out" of college. The kids get a job and he gets young, malleable talent.

Others are looking at the staggering rise in tuition and suggesting that shorter, more focused education makes economic sense. Paying off a degree from a top-flight university over a 40-year career can easily consume $1,500 per month ($250,000 at 6.8 percent). Online courses and training from the vendors themselves can be dramatically cheaper.

One article from Fast Company asked the question, "Why hire a PhD when a self-taught kid is just as good?"

PROGRAMMING TREND NO. 9: ACCURACY FADES AS SCALABILITY TRUMPS ALL

Years ago, ACID ruled the database roost. The challenge was to build a bulletproof machine that always gave a correct and consistent answer when queried. Hurricanes, nuclear weapons, and errant janitors unplugging the rack could not scramble the database. The big customers were banks, hotels, and airlines, and they wanted to make sure bank accounts and reservations were consistent and correct.

Today, the industry is trying to find an easy way to store ephemera from our lives. From the places we visit to the toss-away comments between friends, the goal is to find a fast and efficient way to store endless tidbits from everyone on earth.

The smartest people approaching this problem quickly realized they could make their job dramatically easier by cutting corners and blithely ignoring any glitch. If some status update disappeared, who would notice? If somebody checked in to a service while at a coffee shop and failed to be crowned mayor of that coffee shop, it wasn't a big deal because they would probably return again tomorrow. After the new class of data caretakers recognized that they could save a fortune on compute cycles and infrastructure simply by loosening requirements, they started building NoSQL and other so-called data stores.

Now, saving time and money by trading away accuracy rules the Web. Try searching for an older email message with some of the Web-based tools. They're quietly leaving some of the older ones out of the index. This often reflects a slow erosion of standards for search. Google, for instance, quietly ended the ability to use true boolean searches with the plus sign. Expect to see more and more Web engineers subtly tossing aside the fanatical commitment to accuracy once common among database administrators.

PROGRAMMING TREND NO. 10: REAL PARALLELISM BEGINS TO GET PRACTICAL

Computer architects have been talking about machines with true parallel architectures for years, but the programmers in the trenches are just starting to get the tools that make it possible.

The parallelism is appearing in two prominent areas: multinode databases and Hadoop jobs. Some mix the two.

Most NoSQL data stores offer to help spread the workload over multiple machines. Some offer automatic sharding, which splits the data set into pieces, synchronizes the machines that host a given piece, and directs queries to the right machines as necessary. Some offer duplication or backup, a feature that's a bit older; some do both.

Hadoop is an open source framework that will coordinate a number of machines working on a problem and compile their work into a single answer. The project imitates some of the Map/Reduce framework developed by Google to help synchronize Web crawling efforts, but the project has grown well beyond these roots.

Tools like this make it easier than ever to toss more than one machine at a problem. The infrastructure is now solid enough that the enterprise architects can rely on deploying racks of machines with only a bit of fussing.

PROGRAMMING TREND NO. 11: GPUS TRUMP CPUS

Was it only a few years ago that the CPU manufacturers created the chips that fetched the most money? Those days are fading fast as the graphics processors are now the most lustworthy. It's easy to find kids who will spend $300 on their entire computer and operating system, then $600 on a new video card to really make it scream.

The gamers aren't alone in their obsession with video cards. Scientists who need high-powered computation are reprogramming GPUs to analyze protein folding or guess the secrets of the smallest particles.

Nvidia runs conferences for nongamers using the devices, and they're selling video cards by the palletload to scientists who want to build supercomputers. Oak Ridge National Laboratory, for instance, plans to put 18,000 Tesla GPUs from Nvidia into one room so that they can call it the fastest supercomputer.

They're presumably going to build elaborate models for the Department of Energy, not to brag about the frame rate they get while playing Doom.

Peter Wayner is contributing editor of the InfoWorld Test Center.

10 business skills every IT pro must master

IT pros must "develop business skills" to succeed. Here are the most valued skills you can cultivate.

The plain fact is that not all business skills are important for IT, which is just as well. If we needed them all, we wouldn't be IT professionals anymore. We'd be too busy learning to be accountants, copywriters, sales reps, recruiters, and purchasing agents.

Ignore all that. Here are the 10 business skills that truly matter to your career in IT -- a lot.

BEST PRACTICES

7 deadly sins of developers

Recognize the worst traits of programmers and save yourself from developer hell

By Neil McAllister

BEING A GOOD DEVELOPER TAKES A LIFETIME OF TRAINING AND practice. But without proper discipline, even the best programmers risk falling prey to their worse natures. Some bad habits are so insidious that they crop up again and again, even among the most experienced developers. I speak of nothing less than the seven deadly sins of software development. Read on to hear how lust, gluttony, greed, sloth, wrath, envy, and pride may be undermining your latest programming project as we speak.

FIRST DEADLY SIN OF SOFTWARE DEVELOPMENT: LUST (OVERENGINEERING)

Modern programming languages tend to add features as they mature. They pile on layer after layer of abstraction, with new keywords and structures designed to aid code readability and reusability -- provided you take the time to learn how to use them properly.

At the same time, the discipline of programming has changed over the years. Today you have giant tomes of design patterns to pore over, and every few months someone comes up with a new development methodology that they swear will transform you into a god among programmers.

But what looks good on paper doesn't always work in practice, and just because you can do something doesn't mean you should. As programming guru Joel Spolsky puts it, "Shipping is a feature. A really important feature. Your product must have it." Programmers who fetishize their tools inevitably lose sight of this, and even the seemingly simplest of projects can end up mired in development hell. Resist your baser impulses and stick to what works.

SECOND DEADLY SIN OF SOFTWARE DEVELOPMENT: GLUTTONY (FAILING TO REFACTOR)

Nothing is more gratifying than shipping software. Once you have a working product out in the wild, the temptation is strong to begin planning the next iteration. What new features should it have? What didn't we have time to implement the first go-round?

It's easy to forget that code seldom leaves the door in perfect shape. Then, as features accumulate with successive rounds of development, programmers tend to compound mistakes of the past, resulting in a bloated, fragile code base that's too tangled to maintain effectively.

Instead of gobbling up plate after plate of new features, restrain yourself. Evaluate your existing code for quality and maintainability. Make code refactoring a line item on your budget for each new round of development.

THIRD DEADLY SIN OF SOFTWARE DEVELOPMENT: GREED (COMPETING ACROSS TEAMS)

The excessive desire for wealth and power -- how else to explain the motives of programmers who compete with their own coworkers? It starts when other teams are left off email lists, then proceeds to closed-door meetings. Next thing you know, one team has written a library that reimplements more than half of the functionality already coded by another team.

Programming teams seldom reinvent the wheel out of malice, but lacking clearly defined objectives, they can easily latch onto responsibilities much broader than are strictly necessary. The result is a redundant, unmanageable code base, to say nothing of the budget lost to duplicated efforts. One of the top priorities of managing a development project should be to make sure each hand knows what the other is doing, and that all the teams are working toward a common goal. Share and share alike should be your motto.

FOURTH DEADLY SIN OF SOFTWARE DEVELOPMENT: SLOTH (NOT VALIDATING INPUTS)

The list of basic programming mistakes is long, but the sin of failing to validate input is so pernicious that it bears special consideration. Why this seemingly amateur error still crops up in code written by experienced programmers is baffling. And yet, many commonplace security vulnerabilities, from buffer overruns to SQL injection attacks, can be traced directly to code that operates on user input without validating it for correct formatting.

Modern programming languages provide many tools to help coders keep this from happening, but they have to be used properly. Remember, a Web form that uses JavaScript to validate its inputs can be easily sidestepped by disabling JavaScript in the browser or not using a browser to access it at all. Input validation should be baked into the core of your application, not sprinkled onto the UI. Anything less is simple laziness.

FIFTH DEADLY SIN OF SOFTWARE DEVELOPMENT: WRATH (NOT COMMENTING CODE)

What act could be more hostile to your fellow programmers than failing to comment your code? I know, I know: Well-written code is its own best documentation. Well, guess what? Those methods you wrote at two in the morning last Thursday weren't exactly well-written code. (And if you're a Perl hacker, you owe me nine Hail Marys.)

It's easy for programmers to forget that the code they write today may live on long after they've left the job. To the programmers who replace them falls the unenviable task of figuring out what each snippet of code actually means. So have mercy, and leave them a few hints.

But remember, unintelligible comments or commenting too much can be as bad as not commenting at all. Comments like "this is broken" or "don't touch this ever" aren't much help to anybody. Code is its own best documentation of what it does; comments should be there to explain the why.

SIXTH DEADLY SIN OF SOFTWARE DEVELOPMENT: ENVY (NOT USING VERSION CONTROL)

It's hard to believe there are still software projects that exist as a directory tree on a file server, jealously guarded by one "master maintainer." Scattered around the office are duplicates of this tree on individual developers' workstations, each slightly different -- though no one knows exactly how.

Maybe you have reasons for not implementing version control on your projects. Maybe it started small and just got out of hand. But powerful, effective version control systems are readily available today for free. There is no reason why you shouldn't make starting a code repository one of the first steps in any project, even small ones -- unless, that is, you can't stand to see anyone commit code changes but yourself.

SEVENTH DEADLY SIN OF SOFTWARE DEVELOPMENT: PRIDE (NOT UNIT TESTING)

It's often tempting to pat yourself on the back for a programming job well done. But how do you know it's well done? What are your metrics?

Unless you've validated your code against specific test cases, you have no idea whether it works as advertised and is completely free of defects. But all too many developers fail to produce unit tests for their code. They claim time spent testing is time not spent implementing features. In fact, some developers fail to even write QA testing into their project budgets.

What can I say, except that pride goeth before a fall? By the time defective code arrives in the client's hands, it's too late to undo the mistake. The more you plan for unit testing before your code ships, the more damage control you can avoid later.

Neil McAllister is a freelance writer based in San Francisco. He also writes InfoWorld's Fatal Exception blog.

BEST PRACTICES

Earth to developers: Grow up!

Rational programmers must step back and remember a few basic principles

By Neil McAllister

If you're a developer reading this right now, chances are you're an idiot.

That is, chances are you code regularly in a language that should never even have been invented. Or maybe you use the wrong IDE or the wrong text editor or some version control system that can't possibly do the job. Maybe you're dedicated to a programming methodology that never works, or your release cycles are set up all wrong. You might debug your code the wrong way, or you have no idea which optimizations to switch on in your compilers. Whatever it is, all your projects are destined for failure.

By the same token, chances are I'm an idiot, too.

Why are developers so quick to call each other idiots, anyway? Check out any developer forum or message board: It won't take long before you'll find some seemingly innocuous thread that has erupted into a full-blown flame war.

The list of hot-button topics is endless: Emacs versus VI. Java versus .Net. C++ versus Java. Eclipse versus NetBeans versus Visual Studio. Perl versus Python versus Ruby. Agile versus waterfall. Django versus Rails. Extreme programming versus Scrum. Git versus Subversion.

You'd be hard-pressed to find a more contentious group outside a "Star Trek" convention. Yet bickering about your favorite sci-fi shows is all in good fun. It's entertainment; it doesn't have any bearing on real life.

For many developers, on the other hand, programming is their livelihood. When decisions about tools and practices become polarized and zealotry takes the place of rational discussion, it not only wastes time, but lowers morale, causes communication breakdowns in other areas, and at its worst threatens the successful completion of critical objectives.

It's grown so bad in the app dev world that groups of developers have taken to issuing "manifestos," as if they were Central American revolutionaries. First there was the Agile Manifesto. Now others are trying to come up with the DevOps Manifesto (though the thought behind that particular revolution seems a little harder to articulate).

In that spirit, I'd like to propose a new manifesto for those developers who are tired of the partisan squabbling, flame wars, name-calling, and finger-pointing. For lack of a better name, I call it the Maturity Manifesto, and it's organized around a few guiding principles:

1. I WILL REJECT DOGMATIC THINKING ABOUT TOOLS, PRACTICES, AND PROCESSES

If you find yourself returning to certain websites for talking points about why your favorite tool is better than all the others, there's a good chance you don't really understand your favorite tool well enough to argue as fervently as you do. Talking points are a great way to convince developers why they should try a new tool or become expert in one they already know. They are less valuable when a team is evaluating a range of options for a specific, real-world task.

Also, resist the urge to reject a tool based solely on the vendor that supplies it. Your personal vendor preferences (or prejudices) may not actually reflect the best interests of the project.

2. I WILL VALUE FLEXIBILITY OVER REPETITION

Just because it worked last time doesn't mean it's the only way to do it this time. And did it really work as well last time as it could have? Has absolutely nothing changed -- neither the options available, the staff available, nor the expertise of the staff -- that might make an option viable now that wasn't viable earlier? Try to keep an open mind when team members raise new methods that haven't been tried. And even when the decision is made to reject a certain idea for this iteration of a project, don't be too quick to dismiss it if the same idea is proposed the next time.

3. I WILL WEIGH ALL CONSIDERATIONS BEFORE MAKING A DECISION

Be particularly wary of qualitative adjectives, such as "faster," "smaller," "easier," "more scalable," and "more robust" when arguing in favor of tools. Like synthetic benchmarks, they seldom tell the whole story. It may be true that the best-written C++ code will outperform the best-written Java code at the same task. But are the C++ programmers on your team really the best in the business? Does performance outweigh other considerations, such as code maintainability or ease of memory management?

For some projects, time to market might be the most important factor; in those cases, the "best" tool for the job might simply be the one that gets it done fastest. Make sure you understand your project's priorities before you argue in favor of any tool.

4. I WILL RECOGNIZE THE DEFICIENCIES OF MY TOOLS, EVEN ONES I PREFER

I have yet to see a language, method, or process that is equally good at everything. For example, C might be the only language that makes sense for a project like the Linux kernel, but it's lousy for text processing. Agile methods might work wonders in small groups but fall apart at enterprise-wide scale. Foolish programmers are those who agitate for a specific tool again and again simply because it's the one they understand best; wise programmers are those who are willing to set aside their favorite methods when they know they aren't ideally suited to the task at hand.

5. I WILL NOT MAKE THE PERFECT THE ENEMY OF THE GOOD

We've all seen terrible code, and we've all seen the completely wrong tool bent to a task it was never meant to handle. These are the worst-case scenarios. Other times, however, a less-than-optimal solution that achieves the objective is preferable to an ideal solution that's too difficult to implement in the time allotted. It's important that experienced developers raise concerns when they recognize problem areas, but once those issues have been aired, it's time to set the debate aside and focus on the goal. Some fights are better left for another day.

6. I WILL ADMIT MY MISTAKES, RATHER THAN COMPOUND THEM

We're all wrong some of the time. We all put our feet in our mouths some of the time. No matter how experienced you are, no matter how sure you are of your footing, sooner or later you'll probably meet someone who understands a problem or tool better than you do. These moments aren't defeats. They are opportunities. To be a good advocate, you should not only be self-assured enough to argue passionately for your position, but also self-aware enough to recognize a lesson worth learning.

Failing that, you can always go home and bicker on a message board -- and good luck to you.

Neil McAllister writes InfoWorld's Fatal Exception blog.

BEST PRACTICES

Beware these developer errors

Nothing guarantees app security as long as developers repeat their mistakes

By Neil McAllister

PROGRAMMERS OFTEN LIKE TO TALK ABOUT HOW A NEW TOOL OR a new version of their favorite platform will make coding faster, easier, or more elegant. Although this may be true, it ignores just how difficult and painstaking the process of developing quality software actually is, no matter what tools are used.

Case in point: the CWE/SANS list of the top 25 most dangerous software errors. Each year, the list's editors draw upon the experience of leading software security experts to rank programming errors by frequency, severity, and the likelihood that they will lead to exploitable vulnerabilities. This year's list was published this week, and the bad news is how few surprises it contains.

Not only is this year's list predictable, it's redundant. Of the 25 errors cited, far too many can be chalked up to the same fundamental misdeeds -- mistakes that have been around almost since the dawn of programming itself. Will we never learn?

THE SAME ERRORS, OVER AND OVER

Topping the list is "improper neutralization of special elements used in an SQL command," also known as the dreaded SQL injection vulnerability, the bane of Web applications everywhere. According to IBM's annual X-Force Trend and Risk Report, the frequency of SQL injection attacks increased 200 times between 2008 and 2009, and IBM's researchers have seen at least one "globally scaled" SQL injection attack each summer for the past three years.

SQL injection is usually the result of improperly validated user input, where the application parses form data into a SQL query without checking to see whether it contains potentially harmful SQL code. But SQL injection isn't the only way user input can go wrong. Of the top 25 errors list, roughly a quarter of them can be attributed to inadequate input validation, including OS command injection, buffer overruns, cross-site scripting, failure to validate directory paths, and uncontrolled output formatting strings.

Even more than input validation errors, this year's top 25 list is rife with application security blunders of all kinds. Some of them sound fairly esoteric, such as "inclusion of functionality from untrusted control sphere." But of all such errors, the highest-ranking one on the list is "missing authentication for critical function" -- in other words, the attacker was able to gain access because there was no lock on the door to begin with.

Developers make such errors for two main reasons. First, they may be operating under the mistaken assumption that a given function is too obscure to be vulnerable; they fail to grasp the extent to which attackers may be willing to analyze their application flows to find weaknesses. More often, however, they simply haven't considered how important a given function might be to the overall security of their application. As applications grow more complicated and their functions are distributed across multiple systems and resources, it's particularly easy to lose track of the big security picture.

WHAT WE CAN LEARN FROM MISTAKES

The full CWE/SANS list is detailed, comprehensive, eminently readable, and chock-full of specific, valuable advice. If you manage a software development project, you'd be well served to pass the link along to everyone on your team and encourage them to study it in depth. Even a cursory read, however, yields key insights that every developer should keep in mind.

First, know your tools, and don't accept their features blindly. Among the specific recommendations given in the CWE/SANS list are such gems as "If you are using PHP, configure your application so that it does not use register_globals." This particular advice is as old as the hills, and it has actually been the default configuration since PHP 4.2. As of PHP 5.3, the feature in question has been deprecated. Developers who persist in using risky platform features because they're there, despite countless recommendations to the contrary, deserve what they get.

Second, don't put too much faith in your platform just because it's said to be more secure. For example, managed languages such as Java and C# eliminate the possibility of buffer overruns by doing bounds-checking at runtime. That means Java and C# programmers are shielded from the third-ranked error on the CWE/SANS list. But neither Java nor C# does anything to protect you from SQL-injection vulnerabilities caused by poorly validated user input, which rank even higher on the list than buffer overruns. Any platform is only as secure as the code that runs on it.

Third, data security is hard. Unless you're a specialist, cryptography seems like an arcane art, and it's tempting just to treat it simply as magic dust that you can sprinkle onto your applications to make them more secure. Similarly, it's all too easy to introduce backdoors in your authentication scheme if you don't treat security as a core principle in your software design process. Improper, inconsistent, or naïve application of security techniques is especially insidious because it fosters a false sense of safety even as it leads to serious vulnerabilities.

Last, and most important, the list reminds us that software vulnerabilities are everywhere, and virtually no development project is completely safe. With the pace of Internet attacks accelerating, now is not the time to cut QA staff or skimp on testing and code review. No matter what tools you choose, developing secure applications is challenging and laborious, yet critically important, now more than ever. Let's be careful out there.

Neil McAllister is a freelance writer based in San Francisco. He also writes InfoWorld's Fatal Exception blog.

IT job spotting: Top 20 metro areas for tech jobs

Dice.com reports the best areas for tech jobs, based on salaries and the number of open positions

After two straight years of flat wages, tech pros finally got a salary bump in 2011. The average annual wage for technology and engineering pros climbed 2% to $81,327 last year, according to new data from Dice.com. In some areas, salaries jumped even higher. Tech pros reported 12% gains in Austin, Texas, and Portland, Ore., for instance.

Here's a look at salaries and job opportunities in the top 20 metro areas for tech jobs, based on data from Dice.

TRENDS

Programming languages on the rise

Once niche programming language are gaining converts in today's enterprise

By Peter Wayner

In the world of enterprise programming, the mainstream is broad and deep. Code is written predominantly in one of a few major languages.

Programmers looking for work in enterprise shops would be foolish not to learn the major languages that underlie this paradigm, yet a surprising number of niche languages are fast beginning to thrive in the enterprise. Look beyond the mainstays, and you'll find several languages that are beginning to provide solutions to increasingly common problems, as well as old-guard niche languages that continue to occupy redoubts. All offer capabilities compelling enough to justify learning a new way to juggle brackets, braces, and other punctuation marks.

While the following seven niche languages offer features that can't be found in the dominant languages, many rely on the dominant languages to exist. Some run on top of the Java Virtual Machine, essentially taking advantage of the Java team's engineering. And when Microsoft built C#, it explicitly aimed to make the virtual machine open to other languages. That detail may help make deployment easier, but it doesn't matter much to the programmer at creation time.

Either way, these seven languages are quickly gaining converts in the enterprise. Perhaps it's time to start investigating their merits.

PROGRAMMING LANGUAGES ON THE RISE: PYTHON

There seems to be two sorts of people who love Python: those who hate brackets, and scientists. The former helped create the language by building a version of Perl that is easier to read and not as chock-full of opening and closing brackets as a C descendant. Fast-forward several years, and the solution was good enough to be the first language available on Google's AppEngine -- a clear indication Python has the kind of structure that makes it easy to scale in the cloud.

Python's popularity in scientific labs is a bit hard to explain, given that, unlike Stephen Wolfram's Mathematica for mathematicians, the language never offered any data structures or elements explicitly tuned to meet the needs of scientists. Python creator Guido von Rossum believes Python caught on in the labs because "scientists often need to improvise when trying to interpret results, so they are drawn to dynamic languages which allow them to work very quickly and see results almost immediately."

Scientific and engineering enterprises such as pharmaceutical companies aren't the only ones tapping Python for research. Many Wall Street firms now rely heavily on mathematical analysis and often hire university scientists who bring along their habit of coding in Python. Python is becoming so popular on Wall Street that there are even proposals to require the prospectus for a bond to include a Python algorithm for specifying who gets what return on the investment.

PROGRAMMING LANGUAGES ON THE RISE: RUBY

Some may argue that Ruby and Python are hardly "niche" languages, but the truth is, from an enterprise perspective, they remain promising tools all too often pushed to the margin. That said, Ruby, or more precisely the combination of Ruby with the Rails framework known as Ruby on Rails, is becoming increasingly popular for prototyping. Its entrance into the enterprise came on the heels of the Web 2.0 explosion, wherein many websites began as experiments in Ruby. 37signals -- one of Ruby's many proponents -- actually uses Ruby to deploy code.

The secret to Ruby's success is its use of "convention over configuration," wherein naming a variable "foo" causes the corresponding column in the database to automatically be named "foo" as well. As such, Ruby on Rails is an excellent tool for prototyping, giving you only one reason to type "foo". Ruby on Rails takes care of the rest of the CRUD scaffolding for you.

Ruby on Rails sites are devoted to cataloging data that can be stored in tables. Well-known examples include Web applications like Basecamp, Backcamp, and Campfire from

37Signals, a collection of websites that knits together group discussions, debates, and schedules. Ruby on Rails handles the formatting of these database tables, as well as decisions about what information to display. Using Ruby on Rails' naming convention, production quality code can be sketched up easily without much duplicate effort.

Many of the production-grade Ruby websites run on JRuby, a version written in Java that sits squarely on the JVM. JRuby users get all of the JVM's prowess in juggling threads, a very valuable asset in production-level deployments with many concurrent users.

PROGRAMMING LANGUAGES ON THE RISE: MATLAB

Built for mathematicians to solve systems of linear equations, MATLAB has found rising interest in the enterprise, thanks to the large volumes of data today's organizations need to analyze. Many of the more sophisticated statistical techniques that match people with advertisements, songs, or Web pages depend upon the power of algorithms like those solved by MATLAB.

Expect MATLAB use to grow as log files grow fatter. It's one thing for a human to look at the list of top pages viewed, but it takes a statistical powerhouse to squeeze ideas from a complex set of paths. Are people more likely to shop for clothes on Monday or Friday? Is there any correlation between product failures and the line that produced them?

MathWorks, the company behind MATLAB, offers a diverse set of whitepapers showing how engineers are searching for statistical answers. Toyota Racing, for instance, plans its NASCAR entries by analyzing tests in wind tunnels and other labs. Canada's Institute for Biodiagnostics is searching for the best treatment for burns.

There are also a number of open source alternatives, including Octave, Scilab, Sage, and PySci, one of the aforementioned Python libraries. All of these tools help with the complicated statistical analysis that is now becoming common for firms trying to understand what the customer did and what the customer may want to do in the future.

PROGRAMMING LANGUAGES ON THE RISE: JAVASCRIPT

JavaScript is not an obscure language by any means. If anything, it may be the most compiled language on Earth, if only because every browser downloads the code and recompiles it every time someone loads a Web page. Despite this fact and the increasing dominance of AJAX-savvy Web pages, JavaScript is rarely thought of as a language that runs on the big iron.

This isn't for lack of trying. Netscape tried to make JavaScript the common language on its server platform back in 1996, but ended up establishing it only in the browser. Aptana, one of the latest devotees, throttled its development of Jaxer when it never caught on. AppJet, a small experimental company, used the Rhino JavaScript library written in Java to make it simpler to code server-side. That company was acquired by Google in 2009 and now seems to be devoted to other projects.

Still, new applications for JavaScript abound. CouchDB, for instance, doesn't use SQL for queries, instead taking two JavaScript functions, one for selection (Map) and the other for bundling everything together (Reduce). Node.js is one of the more exciting server-side JavaScript frameworks to appear as of late, revitalizing the ancient dream of bringing harmony to both client and server-side programming. The package takes Google's V8 JavaScript engine created for the browser and lets it make the decisions about formatting outgoing data.

Everywhere people need a small amount of scripting power, JavaScript finds new uses. One of the simplest ways for developers of large applications to offer users the ability to create subapplications, JavaScript continues to grow in the enterprise, one small chunk of code at a time.

PROGRAMMING LANGUAGES ON THE RISE: R

Statistical analysis is being increasingly done in R these days, although some purists call the language S, its original name. Tibco sells a commercial version called S-Plus.

There probably won't be an S++ because the language is more a version of LISP or Scheme with additional features for computing statistical functions and then displaying the results in pretty pictures. If the boss wants the computer to churn through billions of lines of log files looking for patterns, clusters, and predictive variables, R or S is a well-loved solution.

R is another Swiss Army Knife of numerical and statistical routines for hacking through the big data sets -- collections big enough that it might be better called a Swiss Army

Machete. Lou Bajuk-Yorgan, senior director of product management for Tibco's Spotfire S-Plus, says its software is used by a number of clients who are studying how business or engineering projects might work or why they fail to work. Analyzing weather patterns to find the best places to build wind-powered generators is one example.

PROGRAMMING LANGUAGES ON THE RISE: ERLANG

Does your server need to respond to many different independent messages concurrently? Do you need to parcel these requests out to different cores or servers in various parts of the world? That's practically the definition of the hardest part of enterprise computing. Erlang, an open source language first created by scientists at Ericsson Computing Laboratory, excels at these tasks.

The language mixes traditional facets of functional programming (no side effects) with a modern virtual machine that compiles down to machine code. The structure of the language forces the programmer to build something that's easier to spread across multiple cores and multiple machines. There are a number of practical implementations of Web servers and the CouchDB. That's right: The database that asks to receive queries written in JavaScript instead of SQL is itself written in Erlang.

CouchDB is just the beginning. A number of project managers dealing with "big data" are building systems for storing large volumes of data in a scalable way. Hibari, an open source project from Gemini Mobile, offers consistent, scalable clusters to store key-value pairs that repair themselves after failure. The functional structure makes it easier to create big applications that juggle multiple connections efficiently.

PROGRAMMING LANGUAGES ON THE RISE: COBOL

It may not be fair to call Cobol a niche language as it was once the dominant language in the enterprise. Grace Murray Hopper, famous for finding the first bug in the early mainframes, helped create the language in 1959 and it's been enhanced hundreds of times since. Cobol jockeys today get to play with object-oriented extensions, self-modifying code, and practically every other gimmick.

That never earned it much respect in some circles. Or as famous academic Edsger Dijkstra put it: "The use of Cobol cripples the mind; its teaching should, therefore, be regarded as a criminal offense." The folks in mainframe shops everywhere ignored this note and soldiered on. IBM calls one of the latest releases "Enterprise Cobol 4.2," but it could as easily be numbered 147.2 or maybe even 588.3. Cobol programmers like the syntax that's more like a natural language with actual nouns and verbs that form clauses and sentences -- a technique that might call Ruby to mind.

While fewer schools are teaching new programmers Cobol, the language is far from dying, with many corporations continuing to invest in their Cobol stacks. A recent search of Dice.com showed 580 jobs mentioning Cobol and 1,070 mentioning Ruby.

Versions of the languages run on JVMs and .Net virtual machines making it possible to migrate code stacks away from mainframes to Linux boxes. Programmers who want to use a more modern IDE can search for plug-ins to Eclipse, a project that is gaining new support.

PROGRAMMING LANGUAGES ON THE RISE: CUDA EXTENSIONS

As libraries for programming video cards to do massively parallel jobs, CUDA extensions are not technically a language; they're just extensions to C.

Still, some enterprise programmers are beginning to unlock the massively parallel architectures normally devoted to rendering realistic blood splattering in alternative game worlds. Moreover, recoding loops for massive parallelism means rethinking many of the idioms from basic C or C++ programming, making CUDA extensions all the more valuable.

Opportunities to tape CUDA extensions include machine vision, massive simulations, and huge statistical computations. Many problems of data analysis are naturally massively parallel, making GPU processors worth a look. One of Nvidia's recent conferences devoted to CUDA applications included separate tracks devoted to computational fluid dynamics, computer vision, databases and data mining, finance, and molecular dynamics. That list alone is long enough to explain why big enterprise coders are curious.

"It's clear that the GPU reached escape velocity," Dan Vivoli, senior vice president at Nvidia.

"The processor is now reaching all different disciplines of science and industry."

Peter Wayner is contributing editor of the InfoWorld Test Center.

BEST PRACTICES

Don’t be afraid to rewrite code

When your app’s overloaded, sometimes you need to start over from scratch

By Neil McAllister

IN MOST FIELDS, THERE’S A SPECIAL KIND OF SHAME ASSOCIATED with having to start a project over from scratch. As an architect, for example, the last thing you want to hear is that one of your buildings will be torn down and rebuilt from the ground up because it can no longer support the weight of its tenants.

According to computer scientist and entrepreneur Michael Stonebraker, however, that’s more or less the situation confronting Facebook right now. Only in Facebook’s case, the “building” is a Web application, and the problem isn’t concrete or steel girders; it’s MySQL.

In 2008, Facebook famously disclosed that it had deployed a whopping 1,800 production MySQL servers, and the social networking giant’s growth has only accelerated since then. As of now, Stonebraker says, Facebook has split its MySQL data store into some 4,000 shards, with 9,000 caching servers running 24/7 just to keep up with the load.

Facebook’s struggles with MySQL are far from secret. In fact, the company maintains a MySQL at Facebook profile page with updates on its continuous quest to keep the open source database running efficiently at such a massive scale.

But to hear Stonebraker tell it, that quixotic journey should have ended long ago. He describes being saddled with Facebook’s complex MySQL installation as “a fate worse than death.” The only way out of this purgatory, he says, is for Facebook to “bite the bullet and rewrite everything.” In other words: Tear this building down.

Naturally, Stonebraker’s comments have ruffled a lot of feathers in the Facebook camp. But for the sake of argument, let’s assume he’s right. Let’s assume Facebook really is nearing the limits of what MySQL can possibly do, and that the most effective solution at this point would be a total rewrite.So what’s the big deal?

TRY AND TRY AGAIN

Facebook would hardly be the first high-traffic website to attempt a major technology upgrade late into its life. In fact, a two-stage rollout has become something of a tradition among Web startups. The list of sites that have undergone a major technical revamp after launch reads like a veritable who’s who of the Web’s biggest names.

Remember when Twitter sounded like a silly idea? Its founders must have been skeptical at first, too, which may have been why they chose to build the site using the Ruby on Rails framework. Rails is known for its fast development times; according to O’Reilly Media’s Tim O’Reilly, “Powerful Web applications that formerly might have taken weeks or months to develop can be produced in a matter of days.”

As Twitter’s user base grew, however, it must have soon become evident that a few days’ coding wasn’t going to cut it. In 2009, Twitter engineers announced that the company had begun migrating key systems from Ruby to Scala, a language that runs on the Java virtual machine (JVM), as a way around bottlenecks in the Ruby runtime environment. Today, Twitter still uses a mix of Ruby and Scala, but the effort to migrate performance-sensitive systems to the JVM continues (search being the most recent candidate).

Even before Ruby on Rails, developers were building sites using other Web frameworks designed for rapid application development. Remember ColdFusion? Now an Adobe product, the venerable platform doesn’t get much truck with developers these days, but in 2003 it allowed a small group of colleagues to develop a social networking competitor to Friendster in just 10 days. The name of their site: MySpace.

MySpace’s user base exploded, and in 2005 the social network and its parent company were acquired by Rupert Murdoch’s News Corp. for $580 million. That same year, with its back-end servers buckling under the weight of its newfound popularity, the company began transitioning its systems from ColdFusion to .Net, with help from New Atlanta Communications’ BlueDragon migration tool.

START SMALL BUT STAY AGILE

Imagine how much time, money, and effort could have been saved had Facebook hitched its fortunes to an enterprise-class database instead of MySQL, or if Twitter or MySpace had built their services using Java or .Net to begin with, rather than bumbling around with Ruby on Rails or ColdFusion. But of course, that's all hindsight. The truth is, there are plenty of good reasons to launch a site using the tools you have available at the moment, even if it means you'll have to rewrite most of your code later.

For starters, it's easy to criticize a popular site, but for every Web application that succeeds, countless more fail. It simply doesn't make sense to invest big dollars on the most robust, scalable tools possible when your idea has yet to be proven in the marketplace.

Second, at the early phases of a Web project, developer efficiency is often even more important than the efficiency of your infrastructure. The longer it takes to bring a site to market, the more opportunity competitors have to outflank you. When your budget is modest, it makes sense to choose tools that allow the smallest staff possible to get the most done in the least amount of time, which is exactly what tools such as Rails and ColdFusion offer.

Third, technology itself evolves. Who's to say your platform of choice won't outgrow today's performance issues, allowing your current design to scale as your site grows?

Fourth, no matter how meticulously you plan, not every contingency can be foreseen. Reengineering your code base gives you the opportunity to correct past mistakes, such as problems with your security model or your database schema. It's possible you might end up doing extensive rewrites even if you don't switch platforms.

Finally, a website simply is not like a building. Investing in Web infrastructure is not the same as investing in steel and concrete. Building Web applications is a business that's intrinsically more agile and flexible than building real-world objects, which is a big part of what makes it such an exciting business to be in. So why not act like it?

BEWARE AXE-GRINDERS

As for Michael Stonebraker, he has an axe to grind. As the co-founder and CTO of VoltDB, Stonebraker would like nothing better than to see Facebook rewrite its code to free itself of its dependence on MySQL.

That would only lend fuel to his arguments that "old SQL" products, such as MySQL, should be "sent to the home for retired software" and that new startups should choose products like VoltDB to avoid Facebook's "fate worse than death."

Personally, I take Stonebraker's arguments with a hefty grain of salt, even if Facebook does end up rewriting substantial portions of its software. Given how Twitter and MySpace both weathered their own growing pains and how successful all three sites have been (despite MySpace's recent turn in fortune), most startups can only dream of failing so spectacularly.

Neil McAllister is a freelance writer based in San Francisco. He also writes InfoWorld's Fatal Exception blog.