Upgrading to Geek 2008

I'D LIKE MY COLUMN TO BE MORE TECHNICAL BECAUSE IF YOU'RE GOING TO GET INTO THE NITTY-GRITTY OF BUILDING LIBRARIES IN COMPUTERS, YOU'VE GOT TO GET PRETTY TECHNICAL.

by DANIEL CHUDNOV

One of the reasons I'm grateful to be writing for *CIL* is that it's fun to find my fitful scribblings mixed in with such a diverse array of thoughtful and accomplished writers. What I write tends to be more technical than some, but I know that other folks will even include snippets of code in their pieces, something I haven't done much before. I'd like to be more technical in these pieces, though, because if you're going to get into the nitty-gritty of building libraries in computers, you've got to get pretty technical.

On the other hand, everywhere I've been during my career I've heard the following sorts of reactions from many colleagues in response to hearing news of various things I've worked on or read about: "Oh, that's too technical for me" or "What you're saying sounds great, but I'll just have to take your word for it" or "I'll never learn that techie stuff." Let me begin 2008 with a bang: This talk is nonsense. I've never liked being considered a "techie" by people who consider themselves to be "nontechies," and I can't stand it anymore. I'm a librarian, doggone it—what else matters? If you're a librarian, you "get" what information is about as deeply as anybody, and if you're willing and interested, you can develop great technical skills more easily than ever, right here, right now, in 2008. If you want to know how, keep reading. If you self-identify as "nontechie" and prefer to stay that way, well, you and I are just going to have to accept our differences. Though please keep reading if you have staff members under or around you who are more technical than you, and ask them if they've done the things I suggest. You might learn something useful anyway!

Where Do You Start?

There's a heck of a lot you can teach yourself. If you've ever "viewed source" from a browser menu, or edited or changed permissions on a file, or even automated some oft-used spreadsheet operations with a macro script, you're probably the kind of person who can learn a lot of this on your own. For the most part, that's what it was like for me when I started. I learned what I could by reading documentation, or looking for examples online, or just by poking around under the hood of stuff that looked cool. The problem with this approach is that it's limiting. Unless you're truly gifted, you still have a lot to learn to be able to contextualize what you can find just by thoughtfully observing cool stuff. When I started at my first job, I benefited a ton from working in a group where everybody else had better technical skills and knowledge than I did, and fortunately everyone freely shared what they knew. But if you're off on your own somewhere, and you want to learn this stuff for real, the best way to do it is to take some classes.

I went back to school myself a few years ago for this very reason. I'd learned a lot in

»

several years on the job, and I had a good sense of my technical skills, but I'd done the things I knew how to do for a while, and there were far more things I didn't know how to do and needed help learning. Fortunately, computer science is a well-understood discipline, so it's pretty likely that you will be able to find a local university or community college near you where you could take a few classes to get a leg up on all the tech stuff you have to do every day. I can't recommend it enough because it made me better at what I do, and I bet it can do the same for you. But which classes should you take? Having been through this, I can't fathom how I got by before I'd studied these three disciplines:

IF YOU WANT TO BE A GEEK, YOU'VE GOT TO BE ABLE TO HANDLE LOGICAL PROBLEM SOLVING.

1. Discrete mathematics: This is basic logic like building truth tables from Boolean-like statements; methods of proof; and a brief summary of set theory, number theory, and graph theory. All of this is usually done without calculators, with simple, small numbers, diagrams, and tables. The skills you'll come away with are a much clearer sense of how to break down any logical problem and how to know when you've considered all cases when solving anything. If you want to be a geek, you've got to be able to handle logical problem solving. Fake it all you want, but you'll get burned badly someday. If you've had at least a term of calculus, most schools should let you take this; if you haven't, you might still be able to talk your way into it. And, honestly, I don't understand why we don't teach number theory in high school, or even earlier. It's like learning the basics of music—deeply mysterious in its advanced levels, but approachable, useful, and enjoyable throughout.

2. Finite automata: What computers do, at their most basic level, is recognize input patterns and operate on them according to some rules. Learning about regular languages and automata will teach you what your computer is doing when you ask it to run a script, to match a regular expression, and to compute the most basic answers to simple operations like addition and multiplication. If you get deep enough, you learn about how those simple computations can build up into systems that look complicated but really are just big piles of simple computations. You can also learn how to think about the possibility that another of those big piles is instead something much more complicated and, possibly, impossible to compute. The best part about studying this topic, believe it or not, is that you get to draw fun diagrams of how parsers work, step by step, at the most minute level of detail—and yes, I really did call them "fun" diagrams because they are!

3. Database theory, design, and implementation: The previous two topics are both pretty abstract. If you find yourself moving deeper into tech stuff in your career in part because you just really enjoy the logic-puzzle aspect of it all, then you'll benefit from and enjoy studying discrete math and automata theory both. But if that doesn't float your boat, and what you mainly need to do is just work with code that operates on data in databases, you'll want to take a closer look at what databases are all about. There's a lot to know about how most database systems manage and index data under the hood and how they interpret and implement queries you send them. Maybe all of your data models are simple and all of your databases are small. But if not, the more you know about how these systems really work, the better off you're going to be when your models get more complex and your databases get bigger.

This trio is really just a bare-bones starting point, but if everybody graduating from library school had some exposure to the concepts covered in classes like these, libraries and librarians would be much better prepared to handle the "insurmountable opportunities" facing us today (to quote Walt Kelly's Pogo, who knew a thing or two about complicated logic). There's a lot more you can study. If you administer systems, for instance, you should take an operating systems course. If you run production systems that get a lot of use, you should study statistics and look into a performance analysis course. If your work on simple scripts starts to grow into maintaining or supporting complicated software for a lot of users, you'll need to know what introductory classes in data structures and algorithms can teach you. A course in information retrieval will also do wonders for your ability to put together a good search interface. The list goes on, but for my money, these topics should all be at the core of any library science education.

No Time for Homework?

I'm not joking when I say that, for many of us, going back to school could help a lot. But everybody's busy. I had to quit school when I moved last year to start a new job and haven't been able to get back to it yet. If you're stressed for time and just need some basic tips on what to do next, or if you supervise one or more up-and-coming geeks who need some more guidance, here are a few ways to move things forward with little to no formal classroom training.

THERE'S A LOT TO KNOW ABOUT HOW MOST DATABASE SYSTEMS MANAGE AND INDEX DATA UNDER THE HOOD AND HOW THEY INTERPRET AND IMPLEMENT QUERIES YOU SEND THEM.

Learn and use the Python or Ruby programming languages. Hackers the world over will argue day and night about which languages are best, and why, but that doesn't really help anybody. Today there are many good languages to learn and use, all of which can help you get your job done. To me, though, Python and Ruby stand out because they're both easy to learn, they both come with useful built-in components, and there are good books in stores and tutorials online for both. Most important, though, are two even more practical matters: Both Python and Ruby are great languages for web development (Django for Python, among others, and Rails for Ruby), and both Python and Ruby are widely used by library geek types. So if you're developing code for libraries, or putting your stuff on the web, or both, you can find people like you working with systems like yours who can help save you time, money, and frustration.

Use version control, and start with Subversion. If you develop, support, integrate, or even just customize code at your job, and you're not using version control, then stop whatever you're doing immediately and learn to use Subversion for your stuff. (I mean it. Put down this magazine and go do this now.) Version control is an absolute must for any professional geek. It lets you record all the minute changes you make to your code over time, and it'll save your behind when you realize you've messed up and need to go back in time to an older copy. It's also absolutely required if you're not the only person working on something. Without version control, you're always going to be stepping on each others' toes, with no way to clean up after mistakes.

Why Subversion? It's easy to learn, it's free software, it runs on all kinds of machines, its documentation is plentiful, and it's good enough for most everything most people do. There are other, newer version control toolkits that offer interesting improvements, but start with Subversion (http://subversion.tigris.org) and you'll probably stick with it a long time.

Use a ticketing system, like Trac. Just like with version control, if you're working with code, things get complicated quickly. If you've never used a system that helps you keep track of change requests and bug reports (either of which can be called a "ticket" in a "ticketing system"), then you owe it to yourself to get one, and Trac's a great one to start with (http://trac.edgewall.org). Trac offers a ticket-tracking system integrated with a wiki and version control source code browser optimized for use with Subversion. What this means is that you can record tickets for changes you need to make, and then when you make those changes and save them in a new version, all the messages about the closed ticket and the changed code link back and forth between each other. Trac's wiki system knows about all of this, too, so you can write up project documentation right there in the same place that can point right at pieces of code or tickets.

We use it at my current job to manage active development projects with many users in different locations. Before this job, I used it just by myself on other projects. Either way, I don't know how I'd manage without it.

Read *Joe Celko's SQL for Smarties* book. Most systems people touch SQL code sometime. Reading this book is the best way to learn how to accomplish database tasks better, faster, and more intelligently than most of us do most of the time. Celko's writing is direct, his pedigree is impeccable, and his books are best-sellers for a reason. Keep this one close to your keyboard.

Practice Your Reading

Finally, a great way to learn to be a better coder is to read other people's code. I mean particularly good code, from systems you admire or use yourself. Just like reading good prose teaches you vocabulary, or history, or engages you in a novel's storylines, sometimes all at once, reading code is an efficient way to engage directly with other people's problem-solving skills. Exposure to their language usage will help you learn idiomatic patterns, engage in the subtleties of particular issues, and build up your own tool set. Fortunately, there's a ton of great free software out there that gives you the freedom to do this and lots within the library domain. You don't need to tell a librarian that reading and freedom go hand in hand, but it always bears repeating. ■

Daniel Chudnov is a librarian working as an information technology specialist in the Office of Strategic Initiatives at the Library of Congress and a frequent speaker, writer, and consultant in the area of software and service innovation in libraries. Previously, he worked on the DSpace project at MIT Libraries and the jake metadata service at the Yale Medical Library. His email address is daniel.chudnov@gmail.com, and his blog is at http://onebiglibrary.net.