Philippe Kruchten ■ University of British Columbia ■ kruchten@ieee.org

Not on the Shelves

Greg Wilson

Every few years, I attempt a bit of sympathetic magic by reviewing books that don't yet exist, in the hope that someone will then feel compelled to write them. Doing this helps me figure out where our field is and (more importantly) where I'd like it to be. Here is my current list; I welcome your suggestions for additional entries (or pointers to current ones that actually do exist).

The Architecture of Open Source Applications

This survey, which is intended for use in a senior or graduate course, describes the architecture of a dozen pairs of open source applications: Apache and lighttpd, Firefox and Konqueror, Gnumeric and OpenOffice Calc, and so on. For each pair, the book lays out the key characteristics of the problem domain, sketches a reference architecture for a solution, then dissects the chosen applications in detail, describing where, how, and (most important) why they differ from, or elaborate on, the reference architecture. Each chapter is a useful introduction to the innards of the systems it discusses; together, they're compelling examples of how to design and explain large pieces of software.

Virtual Machines: Theory and Practice

With the exception of C and C++, most of today's programming languages run on top of some kind of virtual machine. However, while most undergraduate programs have a course on compilers, few if any teach students how VMs work. This book fills that gap: the first three chapters present a simple, direct interpreter, which each of the following chapters improves in some way. The resulting "Dragon Book"[1] for virtual machines also happens to be an excellent introduction to systems programming.

Debuggers: Theory and Practice

Written as a sequel to the *Virtual Machines* book, this book examines another crucial but often-neglected topic: programs that inspect and control other programs. It assumes at least some prior knowledge of computer architecture and, like John R. Levine's *Linkers and Loaders*,[2] delves into the tricky details that make real systems different from toys. The chapter on user interface design for program comprehension tools is as welcome as it was unexpected.

Agile Development for Scientists and Engineers

This book is an introduction to basic software development practices for graduate students in science and engineering who don't have any formal computer science background. Chapters alternate between tools (such as Subversion, Make, and the Eclipse development environment) and techniques (unit testing, processing XML, and design patterns). While it necessarily glosses over many fine points, it gives scientists and engineers a useful toolkit and a sense of where to go next.

Software Tools for the 21st Century

Windows PowerShell is as revolutionary in its own way as XML and HTTP: instead of flat text, PowerShell enables tools to send one another streams of objects, with all the flexibility and extensibility that implies. This book describes SuperShell, an open source equivalent that uses Javascript for scripting instead of PowerShell's custom Perl-like syntax. Like Brian Kernighan and P.J. Plauger's groundbreaking *Software Tools* books,[3,4] this book builds up SuperShell piece by piece in order to convey the "why" as well as the "what."

Programming Small Devices

This book's aim is to help students who have been spoiled by gigabytes of RAM and gigahertz processors deal with the harsher, leaner world of small devices. It assumes readers already know how to program and focuses on dispelling their misconceptions and curing their sloppy habits. What do you do when you don't have garbage collection? What do you do when you don't even have floating point, or when you have to worry about watts as well as bytes? Ranging over architecture, basic data structures, and neat programming tricks, this is an excellent introduction to programming in a world where nothing is free.

Quality Assurance: A Modern Synthesis

Quality assurance has gone through a not-so-quiet renaissance in the past 10 years, inspired by agile methodologies on the one hand and the rapid development of static and dynamic analysis tools on the other. This survey describes the principles that underpin those tools, ranging from mock objects and browser-based acceptance tools such as Selenium to more experimental ideas of the kind discussed in Andreas Zeller's *Why Programs Fail*.[5] It then goes on to look at the socioeconomics of testing, which may point the way to the next wave of innovation.

Difference Engines

Modern version control systems do a great job of managing text but are much clumsier when it comes to images, MP3s, spreadsheets, and other so-called "binary" files. The reason is simple: those formats are supported by tools for reading and writing, not for differencing and merging. This survey describes the science behind an open source library of tools (the "engines" of the title) that can handle many widely used formats, from structured text such as XML to common audio and video standards. Some readers will be put off by the math the authors use when proving bounds on the performance of the algorithms underlying these tools, but even they should appreciate this work's power and elegance.

Big, Fast, Cheap, or Good

Students' first software projects are different from "real" industrial projects in many ways: student teams usually don't have a more experienced project manager to show them the ropes, and team members are almost always time-slicing their work with other courses. This down-to-earth book, subtitled *A Student's Guide to Software Project Management*, is written with that audience in mind. It takes ideas from several books aimed at practitioners already in industry,[6–9] but it explains them to people who have never before had to find their way around a large body of code or coordinate with half a dozen other people.

Error Handling

Up to a third of the code in a large application exists to detect, handle, and report errors, but you'd be lucky to find more than a page or two on the subject in most programming texts (which routinely omit error-handling code "for the sake of clarity"). This book corrects that imbalance with a comprehensive bottom-up discussion of what programs (and programmers) can and should do when things go wrong. The discussion ranges from cyclic redundancy check (CRC) codes, through the zero/nonzero convention used for both Unix and Windows system calls, to exceptions (three chapters), database transactions, and the partial failures that bedevil distributed systems. Readers will need a solid grasp of computer architecture, compiler design, networking, and distributed algorithms to follow it all, but then, they'll need to know those things to build robust programs.

Performance Analysis and Tuning

People use computers because they're fast; unfortunately, a lot of modern software seems designed to make them slow. This book aims to correct that by showing readers how to find the bottlenecks in everything from simple string-sorting routines to multitier Web applications, and then how to get rid of them. Some of the material, such as the refresher on algorithmic complexity and the introduction to queueing theory, is theoretical; other parts are intensely practical. Taken together, it's a worthy successor to Jon Louis Bentley's *Writing Efficient Programs*.[10]

Great Ideas from Lesser-Known Languages

The overwhelming majority of programmers still use imperative techniques that would have been familiar to anyone working in the 1970s. Those techniques might be wrapped up in objects but still basically push bits around using assignment, indexing, loops, conditionals, and function calls. This book surveys several dozen techniques that are common in alternative languages: list and set comprehensions, recursive data structures, type inference, generators, unification, continuations, reflection, hygienic macros, transactional memory, and a bewildering variety of concurrency mechanisms. Only some of these can be emulated in traditional languages without heroic effort, but all are mind-expanding.

Writing reviews of books that don't yet exist helps me figure out where our field is and where I'd like it to be.

References

1. A.V. Aho et al., *Compilers: Principles, Techniques, and Tools*, 2nd ed., Addison-Wesley, 2006.
2. J.R. Levine, *Linkers and Loaders*, Morgan Kaufmann, 2000.
3. B.W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley Professional, 1976.
4. B.W. Kernighan and P.J. Plauger, *Software Tools in Pascal*, Addison-Wesley Professional, 1981.
5. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann, 2005.
6. M. Doar, *Practical Development Environments*, O'Reilly Media, 2005.
7. M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2005.
8. K. Fogel, *Producing Open Source Software*, O'Reilly Media, 2005.
9. R.L. Glass, *Facts and Fallacies of Software Engineering*, Addison-Wesley Professional, 2002.
10. J.L. Bentley, *Writing Efficient Programs*, Prentice Hall, 1982.

Greg Wilson is an assistant professor in the Department of Computer Science at the University of Toronto. Contact him at gvwilson@cs.utoronto.ca.