Book Reviews

ELIZABETH ZWICKY, WITH MARK LAMOURINE

Perl One-Liners

Peteris Krumins
No Starch Press, 2013. 138 pages.
ISBN 978-1-59327-520-4
Reviewed by Elizabeth Zwicky

Perl One-Liners is what anything named a cookbook should be. This book is not an attempt to cover every Perl one-liner ever, or even every useful Perl one-liner ever; it's not an attempt to cover Perl from end to end. Instead, it's a tour through the landscape of interesting and/or useful Perl one-liners. You can learn a lot about Perl from it, and if you spend much time throwing files around on UNIX, you'll also learn a few interesting tricks. Plus this book is just plain fun.

The introductory chapter is the weakest because it reads like it was added afterwards to provide some background. If you know something about Perl, this chapter isn't necessary. If you don't know much about Perl, this chapter probably isn't sufficient. Skip ahead to the good stuff, and if you get stuck, refer to a better introduction to Perl source.

As a crufty old UNIX type, I would have liked more of the one-liners to mention the existing commands they duplicate. Some of them do, but others don't. This book does not explain that "nl" exists and is happy to number lines for you, or that "grep" knows how to give you context lines.

And a final cranky complaint: for goodness' sake, eight random letters does not a password make. No, not even if you add numbers. And not really if you go to 14 characters. Take your one liner to generate eight random letters and call it a cat-naming scheme or something, but do something less pitiful for passwords. Punctuation. Uppercase characters. Start with 14 characters, not as a possible extension. Otherwise, you are living in the past century, and even then, not at the cutting edge.

Agile Data Science

Russell Jurney
O'Reilly, October 2013. 158 pages.
ISBN 978-1-449-32626-5

Doing Data Science

Cathy O'Neil and Rachel Schutt
O'Reilly, October 2013. 360 pages.
ISBN 978-1-449-35865-5
Reviewed by Elizabeth Zwicky

These two books are both interesting, somewhat eccentric takes on data science, and they make a nice complementary pair.

Agile Data Science walks the reader through one agile approach to data science, end-to-end, covering everything but the actual data science—the tools, the processes, the agile mindset. If I were doing data science for a startup, I would devour this book, and then almost certainly do something else. But that something else would be easier and more intelligent because I'd read the book. And I would have at least some idea about the tools in every piece of the chain.

If you have an existing agile organization and want to add in data science, this book will probably help. If you have an existing data science organization and want to become agile, look elsewhere. This book offers a perfectly nice description of one possible solution, but it's not a discussion of converting to agile.

Doing Data Science is a book version of a course about data science. It does a minimal job of covering tools, and concentrates on algorithms and the idea of data science. The book is a fascinating read and covers a lot of territory. I enjoyed the classroom feel and the introspection, for the most part, but I felt that other parts didn't work as well. The data visualization section, for instance, raised interesting questions, but didn't provide the sorts of insight I found in the algorithm discussions. In the end, I felt like the intriguing data art could have been left out without any great loss.

Doing Data Science provides a good introduction for people getting interested in data science.

Designing for Behavior Change

Stephen Wendel
O'Reilly, October 2013. 346 pages.
ISBN 976-1-449-36762-6
Reviewed by Elizabeth Zwicky

I'm always interested in behavior change, because it's one of the big problems in security: How do you convince people to be more secure? This is a book about building products that change behavior, and now I'm kind of sad that I'm not in a position to try to build a product that would convince people to be more secure. That never actually seemed like a vaguely plausible idea to me before, but now I have read a whole book dedicated to the idea that you could actually build software to change behavior, without even engaging in unfounded optimism.

The author talks believably about both behavior change and product development. In fact, if you're going to build a product that's not about behavior change, you could do a lot worse than reading this and just ignoring the parts about behavior; you'd be left with good advice on defining, developing, and evaluating a product. He advocates an approach based in science, right down to actually figuring out whether you're getting anywhere. He's

open about the difficulties involved, and totally practical about issues like statistical knowledge. How low can you go on statistics and still have reason to believe your numbers? What should you do if you're pushing that boundary and need to know how many test subjects are enough? If a professor would be really handy, how would you get one involved?

Linux Utilities Cookbook

James Kent Lewis
Packt Publishing, 2013. 198 pages.
ISBN 978-1-78216-300-8
Reviewed by Elizabeth Zwicky

You can't cover all of Linux in 198 pages. In order to do something useful in that little space, you need to define an audience and scope tightly. Sadly, this book does not do that. Instead, it veers back and forth between system administration topics and introductory topics, resulting in a chapter on file systems that talks about what "ls" and "cd" do, but starts with a discussion of inodes and superblocks.

Worse yet, that discussion is neither technically nor grammatically accurate. "Things that are not available in the inode are the full path and name of the file itself. This is stored in the /proc filesystem under the PID (process ID) of the process that owns the file." This is stunningly incorrect. (Hint: files still have names when the system isn't running. The names are in the file system.)

I could go on at length, but the bottom line is that this is not an adequate resource for new Linux users or new Linux administrators. People interested in being power-users would be better off with something like *The Linux Command Line* (William Shotts, No Starch), and new system administrators should tackle the intimidating bulk of *UNIX and LINUX System Administration Handbook* (Evi Nemeth et al., Prentice Hall) or *Essential System Administration* (Aeleen Frisch, O'Reilly).

Programming Distributed Computing Systems: A Foundational Approach

Carlos A. Varela
MIT Press, 2013. 271 pages.
ISBN: 978-0-262-01898-2
Reviewed by Mark Lamourine

I'm not sure I'm qualified to review this book, but I'm going to do it anyway. *Programming Distributed Computer Systems* is the first real computer science book I've read in a long time.

Programming Distributed Computer Systems is a teaching textbook aimed at graduate or high-level undergraduate students in computer science. The author's goal is to teach four recent formal models of distributed programming with both theoretical and practical examples.

Varela divides the book generally into two sections. In the first half of the book he treats the models and the formal languages that have been created to express them. In the second half he presents a set of classic programming problems and demonstrates their solutions in languages that support the new models.

In the introductory chapters, Varela first introduces and details the lambda calculus, the basis for sequential programming theory (and implemented as functional programming). I actually have a little experience with functional programming and I found this an excellent refresher course. The next three chapters introduce newer formal languages which add features or concepts to express different aspects of the newer models.

Each model and the corresponding formal language has a name that doesn't necessarily have any additional meaning. They are known as the pi calculus, actors (an extension of the lambda calculus), the join calculus, and the ambient calculus. The programming chapters use Pict, SALSA, and JoCaml to demonstrate the pi calculus, the actor model, and the join calculus, respectively. (There's currently no language that implements ambients.)

Varela explicitly states in the introduction that he intends to present all of the theory before beginning any of the programming examples. He claims that he's found that to be the most effective way to consume and digest the concepts. I'm not sure I agree. In the early chapters, he details the syntax and semantics of each new calculus, but the "meaning" that he offers is almost purely in terms of the formal language itself. He teaches the rules, but not really what they mean and why they're useful or make sense. It's not until he's done with all of the theory that he gets to showing how the formal model applies to real problems. I tried reading each chapter in sequence as he recommends.

Although I wouldn't want code interleaved at the presentation of each new concept, I found myself slogging and struggling to retain the rules because they didn't yet have a meaning for me. To his credit, Varela offers an alternate map for the book. Instead of reading the chapters sequentially, he indicates that you can follow each theory chapter in the first half of the book immediately with the matching programming chapter in the second half. I'd recommend that path, especially for the solo reader.

This book is meant to be taught. Although Varela's writing is clear and not dense or turgid, the topic is abstract (at least to a professional system administrator). A couple of hours of lecture and discussion a week would certainly help throw light in the shadows and fix the ideas in my memory. A set of well-crafted programming assignments would help even more. A solo reader will need a lot of discipline to get everything he or she should from this book.

I'm still trying to decide whether I would recommend this book to anyone I know. It is extremely well written, but I don't know

that many people who would be interested in this topic presented in such an academic way. Formal languages are used (mostly) to reason about programming rather than to produce working applications. The languages that are used aren't mainstream enough for large-scale applications. If I were teaching the analysis of distributed programming, I'd definitely use this book. I enjoyed reading it, and I'm still working my way through the examples again to see what I missed the first time. I guess you'll know best if that appeals to you, too.

Alternative DNS Servers

Jan-Piet Mens
UIT Cambridge , 2009. 695 pages.
ISBN: 978-0-9544529-9-5.
Reviewed by Mark Lamourine

Alternative DNS Servers gives several good first impressions, starting with its heft and with the quality of the paper and printing. Then Jan-Piet Mens had my cranky grammarian heart when he properly constructed the title of the introductory chapter of his book: "Introduction to the DNS." My happiness grew as I read the rest of the book.

This is a book I wish I'd known about when it was published almost five years ago. It's a fantastic addition to the canonical O'Reilly book by Cricket Liu and Paul Albitz [1]. Unlike Liu and Albitz, Mens discusses a wide range of alternatives to ISC BIND, eight in all. He also addresses something I've wondered about for years: how to create a database-backed DNS service.

The DNS is the most fundamental Layer 3 protocol of the Internet. Without it, little else will work. In fact, the DNS works so well that even most system administrators don't fully understand what happens when they issue a lookup query. The first three chapters of Mens' book are an excellent introduction to the workings of the DNS, from the process of resolution to the contents and meaning of the resource records as returned by `dig`. The third chapter itemizes the considerations when planning a DNS service deployment.

This is where Mens gets to the "alternate" in the title. The next 11 chapters are a whirlwind of DNS servers. Some are small, suitable for self-contained labs, whereas others are backed with enterprise-quality replicated databases. In each case Mens highlights the features that would make each one the right choice for an environment. He does include BIND backed by both relational and LDAP databases, but it's treated as just another possibility. This section should be an eye opener even to people who've been running DNS services for a long time.

The DNS servers range from tinydns and dnsmasq, which are suitable for small work groups and labs, to PowerDNS, Microsoft DNS, and BIND backed by MySQL or OpenLDAP, which are commonly used to provide enterprise-level services. This section also has chapters on configuration variations, such as dedicated caching nameservers, private DNS roots, and split DNS for NAT networks.

Mens doesn't leave the reader with a whirlwind tour of DNS servers. The final section addresses operational issues and discusses how they'll affect the choice of server and deployment options. There are chapters devoted to database update procedures, internationalization, DNSSEC, and relative performance (with *real* measurements and details of the testing environment).

The appendices provide details that would have been diversions if they had been placed in line. Many books provide a primer for setting up an LDAP server. Mens provides a collection of tips and tools for problems he's found through experience. His toolkit includes things such as a pattern to avoid updating a zone file without also incrementing the SOA serial number, tuning the database with MySQL-defined functions, and small DNS servers in Perl.

Now comes something I didn't expect. Within a year after publication with UIT, Mens made the complete text of *Alternative DNS Servers* available free in PDF form at the link [2] below.

The DNS is an overlooked element of general system administration. It doesn't need to be the domain of a priesthood or the cranky steampunk machine you don't touch out of fear that it will fail in mystical ways. *Alternative DNS Servers* makes it possible for ordinary system administrators to deploy and manage production-quality DNS services sized properly for any environment.

[1] *DNS and BIND*, 5th ed., ISBN 978-0-596-10057-5.

[2] Also available free as a PDF at http://jpmens.net/2010/10/29/alternative-dns-servers-the-book-as-pdf/.

In Search of Certainty

Mark Burgess
XtAxis Press, 2013. 445 pages.
ISBN: 978-14923891-6-3
Reviewed by Mark Lamourine

I could write a whole article about my thoughts on this book, but that wouldn't be a review; that would be an analysis. I had some pretty strong responses both to the writing style and the content. For the first time since I've been reviewing books, I actually sought out the author so that I could be sure I understood his intent before I responded based only on my assumptions.

Mark Burgess is well known in the sysadmin and configuration management (CM) communities. He has some fairly strong and somewhat controversial opinions about the state and direction of development of CM philosophy and tools. He's also one of only a

handful of people I know of who are even trying to study system administration and CM as proper academic disciplines. Lots of people are doing CM (such as it is), but not many are *thinking* about it in what I consider a rigorous way.

So, let's talk about the book.

In Search of Certainty is presented in three sections. The first two challenge our assumptions about how computers and computation work. In the third section, Burgess describes a way to reason about computer systems and applications that (he claims) has the ability to resolve the issues of stability and uncertainty that are inherent in any large complex distributed system.

In the first section Burgess highlights the flaws in our assumption that our computer systems are inherently stable and that we are in complete control. He begins by discussing the concepts of scale and emergent phenomena. The next sections examine the fact that measurements are necessarily discrete and only approximate the actual state of the observed system, even setting aside experimental error and communication uncertainty (more on those later). He touches on the idea that modern distributed computer systems are so complex (in the colloquial sense) that their behavior has become complex (in the mathematical sense). That is, their behavior has become non-linear and unpredictable in the same way that weather systems and financial market behavior are. We can't control them at a macro scale because we can't even understand them. They fail in spectacular and unexpected ways because they are both unpredictable and contain much more energy in temporary equilibrium than the system can withstand when that energy is released suddenly. The section completes with a chapter on the idea of stability as the process of finding the (local?) minimum energy state of a system (the "zero state") and the idea of engineering to make sure that the desired state of a system is its zero state.

In the second section Mark turns his attention to our view of the state of the systems we create and manage and the uncertainty inherent in any communication over distance. Here he explains the limits to what we can know about the state of a system. Based in information theory as first described by Claude Shannon in 1948, his argument is that even when we get an answer to a query about a system, even when you account for measurement error, there is still the possibility that the message was garbled in transmission or just plain misunderstood by the receiver. (These are not the same thing!) In the same way that quantum uncertainty is a fundamental principle rather than something we can overcome with finer measurements, information uncertainty is a fundamental feature of communication. In fact there is a sense in which they are actually facets of the same properties of physics.

The last section is where Burgess finally starts to restore our hope that we can ever build real, useful systems and have any chance that they will perform as we expect. Given the triple problems of complexity, entropy, and uncertainty, Burgess argues, it is foolish to expect to build and control large, complex, stable and robust information systems by imposing structure and state rigidly from outside.

His response is also based in information theory. Rather than trying to model large systems from above, Burgess proposes modeling them from the bottom. Over several decades, he's formulated something he calls promise theory. A promise is a purely local expression of some desired state. It is more than a simple assertion, which can only have two states, true or false. A promise can be fulfilled or not, but it can also indicate whether the state required repair at the last state check. Relationships between parts of a computer system are expressed as mutual promises between the parts. Promises can be grouped into collections that express more complex structures and behaviors and can also express how the parts will respond if some promise is not (or not yet) fulfilled.

Burgess claims that promise theory and its expression in the software system provides a means to describe the large systems we are deploying in a way that will allow us to avoid the catastrophic failures that are inevitable using traditional configuration methods. He's developed and evolved a software service called CFEngine to embody the theory.

Back to my impressions of the book.

I think I came to *In Search of Certainty* with expectations that Burgess didn't ever plan to meet. I was hoping for a tight technical exposition leading to a conclusion. What he wrote was a twisty personal journey through elements of classical and quantum physics, information theory, and quite a few personal anecdotes and inspirations. For me, the really interesting material doesn't begin until the end of the final section.

I was left hungry to learn more of the underpinnings of promise theory. He spends a lot of time explaining how our current top down software and service creation methods are doomed to ultimate failure but doesn't give much time to explaining how promise theory can be used for engineering.

I found some of the analogies between physics and computation a bit thin. Gas pressure and Boltzmann's constant don't really have that much to do with the tendency of hardware and software to fail. Hardware failure aside, general system configurations don't degrade over time without humans making uncontrolled changes. Although the statistical similarities between one specific computer system behavior and some properties of quantum mechanical systems may be interesting, the differences limit any comparison to a curious coincidence.

Although I agree with his assessment of the current state and trends in creating computational systems and find the mecha-

nism of promise theory interesting, I find Burgess's arguments that promise theory is the way to solve the problems a bit weak. It's not that I think he's wrong, just that his arguments don't make the case for him. In fact, I think there are elements that really deserve more attention than they have been given up to now.

I really want to see more formal research into how the use of more or less purely declarative languages (of which CFEngine is only one) can be used to describe complex systems in a way that is significantly different from any other method. I want to see real demonstrations that the stability and uncertainty issues inherent in large complex distributed information systems are actually mitigated in practice when compared to other methods. I want to see research into how to apply promise theory (or anything else) to the problem of engineering emergent behaviors from localized concrete state definitions.

I came to this book hoping for some of those answers. What Burgess provided was a meandering travelogue of how we've reached where we are now as seen through the lens of his own personal journey. That's not without merit. I think the science and mathematics could have been expressed much more briefly and clearly, but in the end, I think I got the message. I will certainly read whatever comes next, because I want to know, too.

The core of any perl one-liner is the -e switch, which lets you pass a snippet of code on the command-line: perl -e 'print "hi\n"' prints "hi" to the console. The second standard trick to perl one-liners are the -n and -p flags. Both of these make perl put an implicit loop around your program, running it once for each line of input, with the line in the $_ variable. -p also adds an implicit print at the end of each iteration. Both of these use perl's special "ARGV" magic file handle internally. Perl One Liner Basics. enclose program in single quotes to avoid shell expansion. you donâ€™t need the final semicolon. no strict, no warnings, no tests (unless specified). Perl One Liner Flags. -e. Allows it to be a one-liner. -l. print a newline in addition to whatever you print. -n. automatically wrap the one-liner in a loop instead of saying perl -e â€˜while (<>) { print $_ }â€™ *.txt , you can say: perl -ne â€˜print $_â€™ *.txt. -i or -i.bak. make changes directly to the file isntead of STDOUT or saves to .bak instead of clobbering your good file. Perl One-Liner Recipes, not Regex Recipes. Other pages about Perl regex one-liners focus on showing you the regular expressions to accomplish certain tasks. In contrast, this page assumes you know regex, as teaching you regex is the focus of the rest of the site. What this page shows you is, given a certain regex, the Perl syntax to write one-liners to accomplish various tasks. The idea is to get you up-and-running with Perl one-linersâ€”not to get you up-and-running with regex.