Book Reviews

MARK LAMOURINE AND RIK FARROW

Refactoring, 2nd ed.
Martin Fowler, with contributions by Kent Beck
Pearson Education, Inc., 2019, 418 pages
ISBN 978-0-13-475759-9

Reviewed by Mark Lamourine

Martin Fowler released the first edition of *Refactoring* in 1999. It would be nice to think that after 20 years it wasn't necessary to promote the idea of refactoring as a best practice for software development. Sadly, I still find that people blink at the idea of modifying code without changing its behavior, solely to make it easier to read and maintain.

Refactoring is a practice, a technique for improving code and, at least in part, a philosophy. The main idea is that, especially in the "fail fast" world of Agile methodologies, everyone is likely to write a lot of ugly, non-optimal code. That's not a problem by itself because this working-but-incomplete code is revisited over and over, eventually resulting in a polished product. Unfortunately, often the pressure to add features overwhelms the need to clean up the structure of the code, resulting in layers of hacks on hacks that, while strictly functional, become increasingly difficult to maintain and extend.

This (valid) criticism of Agile development styles leads Fowler to compile a set of change patterns (reminiscent to me of *Design Patterns* [1]) which "smell" code that can be improved, based on prior work by William Griswold [2], William Opdyke [3], and others. Following Opdyke's lead, Fowler called them "refactorings."

This second edition largely follows the format of the first, though there are numerous updates that justify revising the original. The examples are presented using JavaScript rather than the original Java. This makes the examples accessible to a larger audience and cuts the amount of boilerplate significantly. Technology has changed in 20 years, and the list of refactorings has been adapted, including some entries for functional programming and some for refactoring without classes. Fowler removed several chapters from the end of the first edition that discussed tools (which go obsolete) and "large refactorings," which are, in reality, redesigned from scratch.

In the first section, Fowler presents the concept and purpose of refactoring. The opening chapter walks through cleaning up a piece of sample code, now in JavaScript. The second chapter provides a definition and motivations for the formal practice. Chapter 3 introduces the idea of code "smells" as ways to recognize poor code. This may be a little subjective, but Fowler argues that these are only flags to look at code more closely. The final chapter of this opening section discusses testing, and how critical it is to be able to confirm that your refactoring actually *didn't* change your code behavior.

The main body of the book is really a catalog of refactoring patterns. This takes up three-fourths of the total space. Each refactoring (yes, it is a noun and a verb) is presented as a kind of encyclopedia entry, and each follows a specific format. An entry consists of five parts:

1. Name: a short description, 2-4 words
2. Sketch: a graphic depiction of the change in code
3. Motivation: a few paragraphs on when and when not to use this entry
4. Mechanics: how to apply the change (sometimes referring to other refactorings as intermediate steps)
5. Example: a sample of code to change, an explanation, and intermediate steps leading to a new result

The refactorings are grouped logically based on the type of change being made. All have an inverse, though not all of these are presented, because many have little value.

The hardcover second edition, published under the Pearson imprint of Addison-Wesley, has about the same page count as the first, but it is only half the width on the shelf and significantly lighter. This does not mean lower quality in any way. The bindings are solid. The pages are smooth without being glossy. Fowler is able, with modern printing, to use color in graphics and to highlight code changes. Both include page marker ribbons in the bindings. The removal of almost 50 pages of summary at the end of the first edition allows the addition of new refactorings and the expansion of the sketch graphics for all of them.

Fowler will be the first to tell you that nearly all of us do refactoring without thinking about it on a daily basis. His purpose

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* 1st ed. (Addison-Wesley, 1994).

[2] W. G. Griswold, "Program Restructuring as an Aid to Software Maintenance": http://cseweb.ucsd.edu/~wgg/Abstracts/gristhesis.pdf.

[3] W. F. Opdyke, "Refactoring Object-Oriented Framework" doctoral dissertation, 1992: http://dl.acm.org/citation.cfm?id=169783.

in writing about and promoting refactoring as a formal practice is to help us avoid the mess that can happen when we try to combine structural changes with feature additions. By keeping these separate in our minds (and in our code), we can both recognize what we are doing and focus on the immediate purpose of our work.

Concurrency in Go

Katherine Cox-Buday
O'Reilly and Associates, 2017, 224 pages
ISBN: 978-1-491-94119-5

Reviewed by Mark Lamourine

When Go was first released, one of the major selling points was the inclusion of concurrency primitives. Together, goroutines and channels were touted as the new solution to a major problem for many web services, or any service that must accept and process many communications at the same time. I read about goroutines and channels and, because that's not the kind of coding I tend to do, I shrugged and moved on. I have written a fair amount of Go, and, because I didn't really understand concurrency, I didn't try to use it.

One problem was that, while many sources showed mechanically how to use goroutines and channels, they all seemed to stop there, assuming that it would be evident that these were cool and useful and that the reader would Go Forth And Code. Another aspect that most examples neglect is how Go's primitives work under the hood. It's always helpful to know not just the overt consequences of some code feature, but the hidden ones as well.

In a mere 200 pages, Cox-Buday explains what concurrency means, how it's been done in the past, and how Go offers a new way. She devotes a full 30 pages of that to detailing why concurrency is important and why it's hard. Long ago I did some hard real-time programming, and this section brought back memories. The realms of web services and flight control systems are very different, but they share a need to process masses of input from many sources in limited time. In this first section she provides examples of race conditions, deadlock, livelock, and resource starvation as well as illustrating the traditional ways of managing them with shared memory and mutex locks. She introduces communicating sequential processes [1], which inspired Go's concurrency model, but also dispels the myth that mutex and shared memory are dead, even in Go.

Cox-Buday introduces the Go concurrency primitives and constructs in a mere 30 pages more. The narrative is clear, concise, and complete but without any fluff. I've seen all of this elsewhere, but the author interlaces code samples, graphics, and program output so that each reinforces the others to make her points.

The remainder of the book is a list of usage patterns. Pretty much every example offered something I hadn't considered, and it's going to take me a few times through and a lot of practice to really get them. There's a lot here to digest, but it's well enough written that it doesn't feel arcane or obscure.

You don't want to start learning Go from this book. Concurrency is fundamental to the language, but you won't get the most out of it by starting there. This is a book for developers who are experienced and want to expand into areas like microservices that will require high volume, low latency communications.

I've found Go to be useful and intuitive even without the concurrency constructs. Now that I have a taste for what they can do, I'm going to be coming back to *Concurrency in Go* often over the next few months to make sure I get the most out of it.

Reference

[1] C. Anthony and R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 1, no. 8 (August 1978): http://bit.ly/HoareCSP.

Cloud Native Go: Building Web Applications and Microservices for the Cloud with Go and React

Kevin Hoffman and Dan Nemeth
Addison-Wesley, 2017, 244 pages, ebook code samples add 180 pages
ISBN: 978-0-672-33779-6

Reviewed by Mark Lamourine

Some books I find with the intent to read and review them. *Cloud Native Go* I picked up because I have a pet project that I use for learning new languages and techniques. I was browsing one day for a book that treats web apps built with Go, MongoDB, and React, and I really was only looking for that last section. As it happens I learned a lot all the way through.

The book is subtitled *Building Web Applications and Microservices for the Cloud with Go and React*. As you might surmise for a book that tries to cover so much, Hoffman and Nemeth don't speak in depth on any topic. For example, they introduce Git in seven pages and Go in eighteen.

Cloud Native Go is really a survey of the suite of technologies and specific tools that are needed to create a comprehensive cloud-based service. The chapters range between 10 and 25 pages, but the longer ones often contain a few smaller sections describing more than one sub-service or component. The authors do provide source code examples on GitHub, although these have not been updated since May 2016, at least six months before publication.

As a survey the book does well, outlining the anatomy of a completely cloud-based service. All of the technologies the authors use have good online references and tutorials. At the end of

that short primer on Go, they suggest that readers new to the language set their book aside for a while and get more familiar with the language before coming back. In each case, the authors explain their choice of a particular tool where choices exist (Werker for CI/CD, MongoDB for database, and RabbitMQ for messaging) but acknowledge that these are opinionated selections from a field of possibilities.

Hoffman and Nemeth are evangelists for cloud services and "the way of the cloud." In the first chapter they describe their philosophy, which is an evolution of Heroku's 12 factors [1] and a combination of other Agile tenets. This lightweight philosophy informs the rest of the book.

I'm not really sure who the audience is for this book. While the introductions to Git, Go, and Docker are clear and concise, they're not enough for a new learner and are unnecessary for the initiated. In nearly every chapter, Hoffman and Nemeth refer the reader to online resources to learn more. That's not bad, but sometimes it feels like that's all there is. I think this would make a good blog series.

The writing does work in one significant way, and perhaps that was the intent. For some readers it may be enough, but I found myself at the end of each chapter looking for more. Where there were links and references, I would go back and follow them. Where there was an example or challenge, I would look and sometimes try it. I have the names of some new tools and technologies that I didn't have before, and I have a sense of just how much more there is to a cloud application than a database, a web server, and some JavaScript.

Reference

[1] A. Wiggins, "The Twelve-Factor App": https://12factor.net/.

Designing an Internet

David D. Clark
The MIT Press, 2018, 419 pages
ISBN: 978-0-262-03680-7

Reviewed by Rik Farrow

David Clark was the head of the Internet Activities Board from 1980 to 1990. He has also served on committees investigating other designs for the Internet as well as writing papers about features of alternate designs. As you might expect, Clark is certainly an expert, if not *the* expert, in this area.

Clark wrote his book partially for policy-makers, those who might need to make decisions about future Internet designs. For that reason, he starts out with Internet basics, explains what he means when he writes about architecture, design, and requirements. While I found Clark's writing clear, I have been studying TCP/IP since 1989, and wonder just how well the book would work for a lawyer or someone working on a politician's staff. That said, Clark does use startlingly clear analogies that should help a determined reader understand at least the basics of internetworking.

Clark has been preparing the ground for Chapter 7, where he discusses alternative internetwork architectures. These vary widely, with some designs being closer to sketches and others much more detailed. The only one I recognized was Name Data Networking (NDN), which Clark often used as an example of applying a label instead of a global address in packet headers. When considering designs, a lot of what Clark has written focuses on just two areas: the expressive power in headers and the per-hop behavior of routers.

Expressive power has to do with the expressiveness used as a means for determining per-hop behavior. PHB is what we expect routers to do, to either forward packets or drop packets in the current Internet, but in future designs could cause other actions, such as requesting information from one or more central servers about how to set up a flow. The expressive power of IPv4 addressing has changed over time, from the initial flat address space, to the classful, then classless inter-domain routing (CIDR) architecture we see today.

Clark also discusses security in Chapter 10, something I was certainly looking forward to reading. For the most part, Clark provides persuasive arguments about why security is really a function of the application layer, with a few exceptions, DDoS being one and inter-domain routing being another. BGP has long been a problem, as any router can send updates about routing, a technique that has been used by nation-states and some attackers to subvert "normal" routing behavior. Clark deftly explains that we could be signing updates today, but the problem is political/social, as in who gets to run the public key server. Clark's suggestion is that regions could start by running regional servers, so that they can at least govern routing updates in their own regions.

Clark covers naming, addresses, availability, economics, and management and control. Most of these areas were not part of the design of the Internet, and Clark discusses why these weren't considered important or relevant in the early days, while providing suggestions for the future.

Clark provides an extensive glossary, pages covering three-letter-acronyms, 15 pages of references, a large index, and a 20-page appendix. The appendix seemed more like a summary of the history of internetworking and is excellent reading by itself, as long as you are willing to use the glossary to find explanations of terms that seemed to me to be unique to this field, like PHB.

As someone who learned about TCP/IP in the field, as it were, I occasionally found myself wondering if Clark and I lived in different realities. For example, UDP does not appear in the glossary and is mentioned just once in the book. Clark writes that he expects that core routers, because they have a very restricted set of responsibilities, shouldn't be as easily exploitable as desktop systems. He uses the notion that there haven't been public announcements of these exploits as proof. Yet a search of Cisco CVEs turns up dozens of router exploits, with few allowing complete control of the router, and most able to cause the router to crash. I asked my security-geek friends, and they pointed out that other core routers, like Juniper, are based on FreeBSD while still others are based on Linux. Huawei is rumored to have copied Cisco IOS firmware, and so shares the same buggy code that Cisco has been building since the late '80s.

But I make those comments because I would be enjoying reading Clark's writing, subtle humor, and careful explanations, only to be jolted by rare assertions that seemed to come from another realm of being. Other than that, I highly recommend his book to those interested in how we wound up with the Internet architecture we did, but not the corporate/political side of things, why certain decisions were made, and about the many ways that people in the US and Europe have been working to create new designs for a future Internet. Clark is certainly a master in the realm of Internet design.

Timefulness

Marcia Bjornerud
Princeton University Press, 2018, 208 pages
ISBN 978-0-691-18120-2

Reviewed by Rik Farrow

A geology professor writes beautifully about her passion, instilling in the reader the immensity of time as seen in the geological records. Bjornerud explains how geologists began by mapping time based on the fossil record, then added depth and precision by learning how to use natural radioactivity. She covers the terrifically slow changes in earth's crust, the faster changes at the surface, the balance that has protected atmospheric components, and how these aspects have worked over the aeons.

Bjornerud concludes with a chapter clearly demolishing hopes of using the current notions of geoengineering to solve the climate crisis. Appendix II provides wonderful tables demonstrating the scales of various geologic changes, from mountain growth and erosion (50-100 million years) to cycles and reoccurance intervals (supercontinent cycle, 500 million years). An easy read, one not requiring any background in geology or science—just an interested reader seeking to expand herself.

Empress of Forever

Max Gladstone
Tor Books, 2019, 480 pages
ISBN 978-0-765-39581-8

Reviewed by Rik Farrow

I read a preview of this book, after being attracted by comparisons to Iain M. Banks and William Gibson. While I don't agree with that, that's largely because the writing style is different and the pace much quicker than either of those authors.

The heroine, Viv Liao, is a Silicon Valley entrepreneur on the run, having attracted the attention of an increasingly authoritarian government. She seeks revenge by installing her nearly completed work, a conscious, aware intelligence requiring a distributed system of a trillion cores to run. But something goes awry during the break-in and installation at a datacenter, throwing her into what appears to be the end of time.

The Empress of Forever rules over all, destroying worlds if they threaten to attract the "Bleed," a rapacious force magnetized by high-technological development. What made this book relevant for me was the insight into how an imagined innovator and controller of vast companies thinks and how this allowed Viv to succeed in an unfamiliar reality, assemble a team of disparate allies, and take the fight to the all-powerful Empress's Citadel. A fun read, fast-paced, when what you need is a break from reading papers.