Alex Groce (agroce@gmail.com), Northern Arizona University

Richard P. Gabriel's *Patterns of Software: Tales from the Software Community* is the kind of book that *Passages*, as a column, aspires to bring back to the attention of the software community. I am fairly sure this book, from 1996, is largely forgotten, but I think it deserves a wide audience, to this day, because it has things to say that we still need to hear, and it says them well. It's not as obscure as some "lost" software books: it remains available in Kindle format, and is easily obtained (though it seems to be technically out of print) in hardback or softcover formats. However, the most recent review on Amazon is from April 2016, more than ten years ago, and I had never heard of the book until recently, when I was probing more deeply into the influence of Christopher Alexander's views on architecture on software engineering, trying to go beyond just "everyone has read *Design Patterns* and knows about the visitor pattern, even people who don't care that much specifically about OO, like myself."

I started reading it perhaps a month ago, just after classes ended for the semester, a time of reflection and backing away from immediate concerns. I expected it to be interesting, perhaps provocative, based on my recollections of his famous "Worse is Better" argument; I was surprised to find that it was one of the more exciting things I'd read in years in the field. For one thing, the book completely changed my idea of what "Worse is Better" meant: in my recollection, the argument was likely ironic, made by a "LISP guy" who saw that UNIX (or even DOS/Windows) had conquered the world, an elegiac eulogy for a better world we lost. That was attractive, because I've always had a soft spot for LISP and LISP-centric thinking (I am perhaps the only person around who has read *both* of Paul Graham's books on LISP, when I was an undergraduate, but has not read *Hackers and Painters)*. Sure, I write code in Python and maybe C/C++, but by golly the functional folks were *right*, and something was a bit wrong with me that I'd left a graduate school and early-JPL career world of SML/NJ functors and signatures for the street-savvy but compromised life of Python/C. Life ought to be Haskell, even if the only time I was employed writing significant Haskell code I found the actual work of developing a relatively small piece of functionality painful due to some cutting-edge, beautiful-but-opaque approaches used in the code. It was admirable; it was deep; it was done by very very good engineers I knew personally and thought were superb, done by, frankly, much better programmers than myself. And I was very glad I was being paid by the hour on that consulting gig. The final code I delivered was very compact, and very elegant, and did what it was supposed to do nicely, but getting there was hard, even for someone like myself, who'd written many thousands of lines of real-world highly-abstract functional code in the past.

Gabriel didn't mean that; though he's most famous indeed as the driving force behind Common LISP, he meant that the UNIX/C people had a point. Software must be habitable, and successful languages will be ones that do not require an extremely high degree of mathematical sophistication. C is such a language, for all its faults. C++ is only somewhat such a language, but is more so than LISP. Building the "right thing" takes forever, will likely never reach market, will probably be focused too much on the "right thing" in terms of developer priorities, not user needs, and produces a brittle design that may be hard to "fix up." Punting on features that

vastly complicate the implementation but hide an obscure ugly case from users is often a winning proposition: don't pay too much for rare weird cases no one may really care about. Houses and programs are to be lived in, and changed by inhabitants, not constructed from abstract first principles to be monuments to the glory of the architect.

These are profound, Montaigne-like, Burke-like, ideas that surface now and then in software, but usually in a more ideological, programmatic, form. Agile argues against too much planning, but in a rational, intense way. Gabriel's tone throughout this book is nuanced, complex, reflective, in some sense unsure (but unsure in an informed, confident sense). He takes his cue, as he admits, from Samuel Johnson's *Rambler* essays, not from a manifesto. Johnson, the great cham of English literature, famously direct and sure in conversation, at times starts with one thesis, and by the end of an essay has abandoned it, or even half-contradicted it. As form, the true essay itself is a "lived in" thing, where the author explores an idea, rather than delivers a judgment from on top of the mountain.

Most software writing is impersonal. It talks about the code, not about life. This book is not like that at all, and the complexity and ambiguity of the essays is, in my opinion, more true to real-world software development than any manifesto or mathematical proof. The richness begins with the epigraph, from Dante, "so tangled and rough" the woods in which Dante (and by implication, Gabriel and the software world) finds himself. Then the complexity hinted at arrives immediately, in the form of a Foreword by none other than Christopher Alexander himself. The great architect states his surprise to find that he was something of a cult figure in the world of software, and states that in Gabriel's essay "The Bead Game, Rugs, and Beauty" (from this collection) he, Alexander discovered that "a computer scientist, not known to me, and whom I had never met, seemed to understand more about what I had done and was trying to do in my own field than my own colleagues who are architects." Wow! But Alexander is not done; this book begins with Alexander, after this praise, questioning whether it is in fact at all the case that what he has done in architecture "has a true parallel in the field of software engineering." In other words, the foreword says that 1) Alexander thinks Gabriel grasps what he is doing better than most architects, and is a very insightful writer worth reading and 2) Alexander is not sure if all this "Gang of Four" patterns stuff isn't a bit of a misapplication of irrelevant ideas poorly read, and that for all his careful thought and hard work Gabriel has failed to overcome this limitation. Alexander isn't sure it's bogus, he's just not sure.

Patterns of Software is broken into five parts. The first part, "Patterns of Software" is the closest to your usual software book, and introduces Alexander's ideas, including the idea of habitable, incrementally designed and developed, software. Besides the solid outline of Alexander's concepts, struggles, and visionary approach here, two things stand out: first, Gabriel is quite willing to go against standard computer science (and specifically LISP-world) thinking and note that maximizing abstraction is not always a great idea. Gabriel is a writer of poetry and fiction, and from modern poetry takes the idea of "compression" as a related but different approach to some things computer scientists normally think of as related to "abstraction." T. S. Eliot is highly compressed and seldom abstract. In other words, Gabriel is implicitly arguing that, from the

point of view of programmers, not users, software should use terms and ideas from artistic thinking, not just ideas lifted from mathematics. Second, Gabriel talks about how software should be habitable not from the point of view of users (a point made by agile and many requirements-elicitation thinkers) but from the point of view of developers. This is, I think, a point of view very ahead of its time, a focus on software as a thing that lives a long time and is tinkered with and handed on that is everywhere in the GitHub world, and fairly rare in 1996, when this book was published: in 1996 the notion that software development was more about maintenance and long-term understanding than delivering a first working version was around, but it was very, very rare, and certainly not dominant.

Part II, “Languages” is similarly familiar, asking questions and making predictions about language success that are still relevant, including a prediction about C++, based on some very sensible empirical analysis, that turns out to be wrong (the reader will, I think, understand why, after sitting and pondering the argument a while). The criteria for a successful programming language presented here, originated by Gabriel and Guy Steele, still seem likely to be valid.

Part III, “What We Do” is short, very sympathetic to the approach taken by *Passages* (it cites John McPhee, endorses reading good poetry, and exhorts computer scientists to read Rick Bass, of all people), and probably the least interesting part of the book. I agree with it that computer scientists and developers should write good, readable, prose, but I think the arguments here are more familiar and less striking than anything else in the book. Perhaps this is just a case of preaching to the choir always being a bit dull. I add that this small part of the book is still worth reading, and the surprising nomination of Guy Steele, Jr., Don Knuth, and Stanley Lippman as among the best writers in computer science is important: Knuth would be on many such a list, but Steele (despite his fame) and Lippman (despite his relative obscurity compared to Knuth and Steele) are a surprise. That’s because Gabriel isn’t simply identifying CS writers who incline towards the metaphorical and “literary,” writers like Fred Brooks and Gerald Weinberg (obvious, though worthy, choices), but writers whose focus is intensely technical (Steele and Lippman are both most famous for works that are essentially extremely complex language reference manuals, for Common LISP and C++, respectively), but whose writing is still a joy to read. I suspect Gabriel is also nodding here to Knuth at his most technical and mathematical, rather than the “fun Knuth” who writes articles for MAD magazine and waxes poetic.

It is Parts IV and V, “The Life of the Critic” and “Into the Ground” that elevate *Patterns of Software* above most good software books. These sections are frankly autobiographical, describing how a vindictive high school teacher, based on a misunderstanding, derailed Gabriel’s entire path in life, ending his chances at scholarship-supported admission to Harvard; how Gabriel failed and moved schools repeatedly in graduate school; how Gabriel’s first marriage (seen as exemplary and solid in a time and culture of widespread rampant divorce among computer people) collapsed; and how the company he is most famous as founder and driving mind behind, Lucid, produced technically exciting products, and made employees very happy indeed, and ceased to exist in short order perhaps partly due to its strengths. This is not

American business-world “try try again” description of failure; it is real analysis of real failure, seen not as a stepping stone but as a thing that happened and had real consequences, including personal reactions inclining more to daydreaming and depression than “can-do” Ben Franklin bootstrapping.

It also offers an unusual point of view on capitalism: Gabriel makes two points about how companies really work that are quite striking. First, he says that academics tend to think of, and enter, the world of selling software with a vision of a dog-eat-dog purely adversarial, secretive, capitalism that has almost nothing to do with the real, practical, way things work. Second, he notes that capitalism (and it’s clear that Gabriel thinks capitalism is in some ways actually *good*, an aspect of “worse is better,” not just a useful evil) rewards not innovation per se but incremental, habitable, usable growth in software, where large changes are seldom the result of a company innovating for the sake of innovation, but the result of a small, modest change become extremely valuable and powerful and capable of extrapolation in a novel environment. “No one ever made money by typing,” is his conclusion. Gabriel, in the end, argues that software engineering should be more like the Army Corps of Engineers responding to the Mississippi, which tells the Corps how to proceed, rather than like a heroic efficient, elegant, profit-centered, Promethean architect making a skyscraper that touches the heavens. Gabriel thinks the skyscraper will usually fall, but the river will go on, and sometimes deliver surprising rewards.