INTERVIEWS WITH SOME OF THE TOP PROGRAMMERS OF OUR TIMES

Jamie Zawinski
Brad Fitzpatrick
Douglas Crockford
Guy Steele
Dan Ingalls
L Peter Deutsch

CODERS AT WORK

REFLECTIONS ON THE CRAFT OF PROGRAMMING

Brendan Eich
Joshua Bloch
Joe Armstrong
Simon Peyton Jones
Peter Norvig
Ken Thompson
Fran Allen
Bernie Cosell
Donald Knuth

"Peter Seibel asks the sort of questions only a fellow programmer would ask. Reading this book may be the next best thing to chatting with these illustrious programmers in person."

—Ehud Lamm, Founder of *Lambda the Ultimate - the programming languages weblog*

PETER SEIBEL

Coders at Work

ISBN-13 (pbk): 978-1-4302-1948-4
ISBN-13 (electronic): 978-1-4302-1949-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jeffrey Pepper
Technical Reviewer: John Vacca

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor: Anita Castro
Copy Editor: Candace English
Production Manager: Frank McGuckin
Cover Designer: Anna Ishschenko
Manufacturing Managers: Tom Debolsky; Michael Short

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact us by e-mail at info@apress.com, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

Contents

About the Author

Peter Seibel is either a writer turned programmer or programmer turned writer. After picking up an undergraduate degree in English and working briefly as a journalist, he was seduced by the web. In the early '90s he hacked Perl for *Mother Jones Magazine* and Organic Online. He participated in the Java revolution as an early employee at WebLogic and later taught Java programming at UC Berkeley Extension. In 2003 he quit his job as the architect of a Java-based transactional messaging system, planning to hack Lisp for a year. Instead he ended up spending two years writing the Jolt Productivity Award–winning *Practical Common Lisp*. Since then he's been working as chief monkey at Gigamonkeys Consulting, learning to train chickens, practicing Tai Chi, and being a dad. He lives in Berkeley, California, with his wife Lily, daughter Amelia, and dog Mahlanie.

Acknowledgments

First of all I want to thank my subjects who gave generously of their time and without whom this book would be nothing but a small pamphlet of unanswered questions. Additional thanks go to Joe Armstrong and Bernie Cosell, and their families, for giving me a place to stay in Stockholm and Virginia. Extra thanks also go to Peter Norvig and Jamie Zawinski who, in addition to taking their own turns speaking into my recorders, helped me get in touch with other folks who became my subjects.

As I traveled around the world conducting interviews several other families also welcomed me into their homes: thanks for their hospitality go to Dan Weinreb and Cheryl Moreau in Boston, to Gareth and Emma McCaughan in Cambridge, England, and to my own parents who provided a great base of operations in New York city. Christophe Rhodes helped me fill some free time between interviews with a tour of Cambridge University and he and Dave Fox rounded out the evening with dinner and a tour of Cantabrigian pubs.

Dan Weinreb, in addition to being my Boston host, has been my most diligent reviewer of all aspects of the the book since the days when I was still gathering names of potential subjects. Zach Beane, Luke Gorrie, Dave Walden and my mom also all read chapters and provided well-timed encouragement. Zach additionally—as is now traditional with my books—provided some words to go on the cover; this time the book's subtitle. Alan Kay made the excellent suggestion to include Dan Ingalls and L Peter Deutsch. Scott Fahlman gave me some useful background on Jamie Zawinski's early career and Dave Walden sent historical materials on Bolt Beranek and Newman to help me prepare for my interview with Bernie Cosell. To anyone I have forgotten, you still have my thanks and also my apologies.

Thanks to the folks at Apress, especially Gary Cornell who first suggested I do this book, John Vacca and Michael Banks for their suggestions, and my copy editor Candace English who fixed innumerable errors.

Finally, deepest thanks to my family, extended and nuclear. Both of my moms, biological and in-law, came on visits to watch the the kid and let me get some extra work done; my parents gave my wife and kid a place to escape for a week so I could make another big push. And most of all, thanks to the wife and kid themselves: Lily and Amelia, while I may occasionally need some time to myself to do the work, without you guys in my life, it wouldn't be worth doing. I love you.

Introduction

Leaving aside the work of Ada Lovelace—the 19th century countess who devised algorithms for Charles Babbage's never-completed Analytical Engine—computer programming has existed as a human endeavor for less than one human lifetime: it has been only 68 years since Konrad Zuse unveiled his Z3 electro-mechanical computer in 1941, the first working general-purpose computer. And it's been only 64 years since six women—Kay Antonelli, Jean Bartik, Betty Holberton, Marlyn Meltzer, Frances Spence, and Ruth Teitelbaum—were pulled from the ranks of the U.S. Army's "computer corps", the women who computed ballistics tables by hand, to become the first programmers of ENIAC, the first general-purpose electronic computer. There are many people alive today—the leading edge of the Baby Boom generation and all of the Boomers' parents—who were born into a world without computer programmers.

No more, of course. Now the world is awash in programmers. According to the Bureau of Labor Statistics, in the United States in 2008 approximately one in every 106 workers—over 1.25 million people—was a computer programmer or software engineer. And that doesn't count professional programmers outside the U.S. nor the many student and hobbyist programmers and people whose official job is something else but who spend some or even a lot of their time trying to bend a computer to their will. Yet despite the millions of people who have written code, and the billions, if not trillions of lines of code written since the field began, it still often feels like we're still making it up as we go along. People still argue about what programming is: mathematics or engineering? Craft, art, or science? We certainly argue—often with great vehemence—about the best way to do it: the Internet overflows with blog articles and forum postings about this or that way of writing code. And bookstores are chock-a-block with books about new programming languages, new methodologies, new ways of thinking about the task of programming.

This book takes a different approach to getting at what programming is, following in the tradition established when the literary journal *The Paris*

Review sent two professors to interview the novelist E.M. Forster, the first in a series of Q&A interviews later collected in the book *Writers at Work*.

I sat down with fifteen highly accomplished programmers with a wide range of experiences in the field—heads down systems hackers such as Ken Thompson, inventor of Unix, and Bernie Cosell, one of the original implementers of the ARPANET; programmers who combine strong academic credentials with hacker cred such as Donald Knuth, Guy Steele, and Simon Peyton Jones; industrial researchers such as Fran Allen of IBM, Joe Armstrong of Ericsson, and Peter Norvig at Google; Xerox PARC alumni Dan Ingalls and L Peter Deutsch; early Netscape implementers Jamie Zawinski and Brendan Eich; folks involved in the design and implementation of the languages the present-day web, Eich again as well as Douglas Crockford and Joshua Bloch; and Brad Fitzpatrick, inventor of Live Journal, and an able representative of the generation of programmers who came of age with the Web.

I asked these folks about programming: how they learned to do it, what they've discovered along the way, and what they think about its future. More particularly, I tried to get them to talk about the issues that programmers wrestle with all the time: How should we design software? What role do programming languages play in helping us be productive or avoid errors? Are there ways we can make it easier to track down hard-to-find bugs?

As these are far from settled questions, it's perhaps unsurprising that my subjects sometimes had quite varied opinions. Jamie Zawinski and Dan Ingalls emphasized the importance of getting code up and running right away while Joshua Bloch described how he designs APIs and tests whether they can support the code he wants to write against them before he does any implementation and Donald Knuth described how he wrote a complete version of his typesetting software TeX in pencil before he started typing in any code. And while Fran Allen lay much of the blame for the decline in interest in computer science in recent decades at the feet of C and Bernie Cosell called it the "biggest security problem to befall modern computers", Ken Thompson argued that security problems are caused by programmers, not their programming languages and Donald Knuth described C's use of pointers as one of the "most amazing improvements in notation" he's seen. Some of my subjects scoffed at the notion that formal proofs could be useful

In 1994 he finally left Lucid and the Lisp world to join Netscape, then a fledgling start-up, where he was one of the original developers of the Unix version of the Netscape browser and later of the Netscape mail reader.

In 1998 Zawinski was one of the prime movers, along with Brendan Eich, behind mozilla.org, the organization that took the Netscape browser open source. A year later, discouraged by the lack of progress toward a release, he quit the project and bought a San Francisco nightclub, the DNA Lounge, which he now runs. He is currently devoting his energies to battling the California Department of Alcoholic Beverage Control in an attempt to convert the club to an all-ages venue for live music.

In this interview we talked about, among other things, why C++ is an abomination, the joy of having millions of people use your software, and the importance of tinkering for budding programmers.

Seibel: How did you learn to program?

Zawinski: Wow, it was so long ago I can barely remember it. The first time I really used a computer in a programming context was probably like eighth grade, I think. We had some TRS-80s and we got to goof around with BASIC a little bit. I'm not sure there was even a class—I think it was just like an after-school thing. I remember there was no way to save programs so you'd just type them in from magazines and stuff like that. Then I guess I read a bunch of books. I remember reading books about languages that I had no way to run and writing programs on paper for languages that I'd only read about.

Seibel: What languages would that have been?

Zawinski: APL, I remember, was one of them. I read an article about it and thought it was really neat.

Seibel: Well, it saves having to have the fancy keyboard. When you were in high school did you have any classes on computers?

Zawinski: In high school I learned Fortran. That's about it.

Seibel: And somehow you got exposed to Lisp.

Zawinski: I read a lot of science fiction. I thought AI was really neat; the computers are going to take over the world. So I learned a little bit about that. I had a friend in high school, Dan Zigmond, and we were trading books, so we both learned Lisp. One day he went to the Apple Users Group meeting at Carnegie Mellon—which was really just a software-trading situation—because he wanted to get free stuff. And he's talking to some college student there who's like, "Oh, here's this 15-year-old who knows Lisp; that's novel; you should go ask Scott Fahlman for a job." So Dan did. And Fahlman gave him one. And then Dan said, "Oh, you should hire my friend too," and that was me. So Fahlman hired us. I think his motivation had to be something along the lines of, Wow, here are two high school kids who are actually interested in this stuff; it doesn't really do me much harm to let them hang out in the lab." So we had basic grunt work—this set of stuff needs to be recompiled because there's a new version of the compiler; go figure out how to do that. Which was pretty awesome. So there are the two of us—these two little kids—surrounded by all these grad students doing language and AI research.

Seibel: Was that the first chance you actually had to run Lisp, there at CMU.

Zawinski: I think so. I know at one point we were goofing around with XLISP, which ran on Macintoshes. But I think that was later. I learned how to program for real there using these PERQ workstations which were part of the Spice project, using Spice Lisp which became CMU Common Lisp. It was such an odd environment. We'd go to weekly meetings, learning how software development works just by listening in. But there were some really entertaining characters in that group. Like the guy who was sort of our manager—the one keeping an eye on us—Skef Wholey, was this giant blond-haired, barbarian-looking guy. Very intimidating-looking. And he didn't talk much. I remember a lot of times I'd be sitting there—it was kind of an open-plan cubicle kind of thing—working, doing something, writing some Lisp program. And he'd come shuffling in with his ceramic mug of beer, bare feet, and he'd just stand behind me. I'd say hi. And he'd grunt or say nothing. He'd just stand there watching me type. At some

point I'd do something and he'd go, "Ptthh, wrong!" and he'd walk away. So that was kind of getting thrown in the deep end. It was like the Zen approach—the master hit me with a stick, now I must meditate.

Seibel: I emailed Fahlman and he said that you were talented and learned very fast. But he also mentioned that you were kind of undisciplined. As he put it, "We tried gently to teach him about working in a group with others and about writing code that you, or someone else, could understand a month from now." Do you remember any of those lessons?

Zawinski: Not the learning of them, I guess. Certainly one of the most important things is writing code you can come back to later. But I'm about to be 39 and I was 15 at the time, so it's all a little fuzzy.

Seibel: What year did that start?

Zawinski: That must have been '84 or '85. I think I started in the summer between 10th and 11th grade. After high school, at 4:00 or so I'd head over there and stay until eight or nine. I don't think I did that every day but I was there a fair amount.

Seibel: And you very briefly went to CMU after you finished high school.

Zawinski: Yeah. What happened was, I hated high school. It was the worst time of my life. And when I was about to graduate I asked Fahlman if he'd hire me full-time and he said, "No, but I've got these friends who've got a startup; go talk to them." Which was Expert Technologies—ETI. I guess he was on their board. They were making this expert system to automatically paginate the yellow pages. They were using Lisp and I knew a couple of the people already who had been in Fahlman's group. They hired me and that was all going fine, and then about a year later I panicked: Oh my god, I completely lucked into both of these jobs; this is never going to happen again. Once I no longer work here I'm going to be flipping burgers if I don't have a college degree, so what I ought to do is go get one of those.

The plan was that I'd be working part-time at ETI and then I'd be going to school part time. That turned into working full-time and going to school full-time and that lasted, I think, six weeks. Maybe it was nine weeks. I know it lasted long enough that I'd missed the add/drop period, so I didn't get any of my money back. But not long enough that I actually got any grades. So it's questionable whether I actually went.

It was just awful. When you're in high school, everyone tells you, "There's a lot of repetitive bullshit and standardized tests; it'll all be better once you're in college." And then you get to your first year of college and they're like, "Oh, no—it gets better when you're in grad school." So it's just same shit, different day—I couldn't take it. Getting up at eight in the morning, memorizing things. They wouldn't let me opt out of this class called Introduction to Facilities where they teach you how to use a mouse. I was like, "I've been working at this university for a year and a half—I know how to use a mouse." No way out of it—"It's policy." All kinds of stuff like that. I couldn't take it. So I dropped out. And I'm glad I did.

Then I worked at ETI for four years or so until the company started evaporating. We were using TI Explorer Lisp machines at ETI so I spent a lot of my time, besides actually working on the expert system, just sort of messing around with user-interface stuff and learning how those machines worked from the bottom up. I loved them—I loved digging around in the operating system and just figuring out how it all fit together.

I'd written a bunch of code and there was some newsgroup where I posted that I was looking for a job and, oh, by the way, here's a bunch of code. Peter Norvig saw it and scheduled an interview. My girlfriend at the time had moved out here to go to UC Berkeley, so I followed her out.

Seibel: Norvig was at Berkeley then?

Zawinski: Yeah. That was a very strange job. They had a whole bunch of grad students who'd been doing research on natural language understanding; they were basically linguists who did some programming. So they wanted someone to take these bits and pieces

of code they'd left behind and integrate them into one thing that actually worked.

That was incredibly difficult because I didn't have the background to understand what in the world they were doing. So this would happen a lot: I'd be looking at something; I'd be completely stuck. I have no idea what this means, where do I go from here, what do I have to read to understand this. So I'd ask Peter. He'd be nice about it—he'd say, "It totally makes sense that you don't understand that yet. I'll sit down and explain it to you Tuesday." So now I've got nothing to do. So I spent a lot of time working on windows system stuff and poking around with screen savers and just the kind of UI stuff that I'd been doing for fun before.

After six or eight months of that it just felt like, wow, I'm really just wasting my time. I'm not doing anything for them, and I just felt like I was on vacation. There have been times when I was working really a lot when I'd look back at that and I'm like, "Why did you quit the vacation job? What is wrong with you? They were paying you to write screen savers!"

So I ended up going to work for Lucid, which was one of the two remaining Lisp-environment developers. The thing that really made me decide to leave was just this feeling that I wasn't accomplishing anything. And I was surrounded by people who weren't programmers. I'm still friends with some of them; they're good folks, but they were linguists. They were much more interested in abstract things than solving problems. I wanted to be doing something that I could point to and say, "Look, I made this neat thing."

Seibel: Your work at Lucid eventually gave rise to XEmacs, but when you went there originally were you working on Lisp stuff?

Zawinski: Yeah, one of the first projects I worked on was—I can't even remember what the machine was, but it was this 16-processor parallel computer and we had this variant of Lucid Common Lisp with some control structures that would let you fork things out to different processors.

understand. So GDB was pretty much an assembly stepper at that point. So you wanted to get out of the GDB world just as quickly as you could.

Seibel: And then you'd have a Lisp debugger and you'd be all set.

Zawinski: Right, yeah.

Seibel: So somewhere in there Lucid switched directions and said, "We're going to make a C++ IDE".

Zawinski: That had been begun before I started working there—it was in progress. And people started shifting over from the Lisp side to the Energize side, which is what the development environment was called. It was a really good product but it was two or three years too early. Nobody, at least on the Unix side, had any idea they wanted it yet. Everyone uses them now but we had to spend a lot of time explaining to people why this was better than vi and GCC. Anyway, I'd done a bit of Emacs stuff. I guess by that point I'd already rewritten the Emacs byte compiler because—why did I do that? Right, I'd written this Rolodex phone/address-book thing.

Seibel: Big Brother Database?

Zawinski: Yeah. And it was slow so I started digging into why it was slow and I realized, oh, it's slow because the compiler sucks. So I rewrote the compiler, which was my first run-in with the intransigence of Stallman. So I knew a lot about Emacs.

Seibel: So the change to the byte compiler, did it change the byte-code format or did it just change the compiler?

Zawinski: It actually had a few options—I made some changes at the C layer, the byte-code interpreter, added a few new instructions that sped things up. But the compiler could be configured to emit either old-style byte-code or ones that took advantage of the new codes.

So I write a new compiler and Stallman's response is, "I see no need for this change." And I'm like, "What are you talking about? It generates way faster code." Then his next response is, "Okay, uh,

send me a diff and explain each line you changed." "Well, I didn't do that—I rewrote it because the old one was crap." That was not OK. The only reason it ever got folded in was because I released it and thousands of people started using it and they loved it and they nagged him for two years and finally he put it in because he was tired of being nagged about it.

Seibel: Did you sign the papers assigning the copyright to the Free Software Foundation?

Zawinski: Oh yeah, I did that right away. I think that was probably the first thing in the email. It was like, send me a diff for each line and sign this. So I signed and said, "I can't do the rest; can't send you a diff; that's ridiculous. It's well documented; go take a look." I don't think he ever did.

There's this myth that there was some legal issue between Lucid and FSF and that's absolutely not true—we assigned copyrights for everything we did to them. It was convenient for them to pretend we hadn't at certain times. Like, we actually submitted the paperwork multiple times because they'd be like, "Oh, oh, we seem to have lost it." I think there was some kind of brouhaha with assignments and XEmacs much later, but that was way after my time.

Seibel: So you started with Lisp. But you obviously didn't stick with it for your whole career. What came next?

Zawinski: Well, the next language I did any serious programming in after Lisp was C, which was kind of like going back to the assembly I programmed on an Apple II. It's the PDP-11 assembler that thinks it's a language. Which was, you know, unpleasant. I'd tried to avoid it for as long as possible. And C++ is just an abomination. Everything is wrong with it in every way. So I really tried to avoid using that as much as I could and do everything in C at Netscape. Which was pretty easy because we were targeting pretty small machines that didn't run C++ programs well because C++ tends to bloat like crazy as soon as you start using any libraries. Plus the C++ compilers were all in flux—there were lots of incompatibility problems. So we just settled on ANSI C from the beginning and that served us pretty well. After that

Java felt like going back to Lisp a bit in that there were concepts that the language wasn't bending over backwards trying to make you avoid—that were comfortable again.

Seibel: Like what?

Zawinski: Memory management. That functions felt more like functions than subroutines. There was much more enforced modularity to it. It's always tempting to throw in a goto in C code just because it's easy.

Seibel: So these days it seems like you're mostly doing C and Perl.

Zawinski: Well, I don't really program very much anymore. Mostly I write stupid little Perl scripts to keep my servers running. I end up writing a lot of goofy things for getting album art for MP3s I have—that kind of thing. Just tiny brute-force throw-away programs.

Seibel: Do you like Perl or is it just handy?

Zawinski: Oh, I despise it. It's a horrible language. But it is installed absolutely everywhere. Any computer you sit down on, you're never going to have to talk someone through installing Perl to run your script. Perl is there already. That's really the one and only thing that recommends it.

It has an OK collection of libraries. There's often a library for doing the thing you want. And often it doesn't work very well, but at least there's something. The experience of writing something in Java and then trying to figure out—I myself have trouble installing Java on my computer—it's horrible. I think Perl is a despicable language. If you use little enough of it, you can make it kind of look like C—or I guess more like JavaScript than like C. Its syntax is crazy, if you use it. Its data structures are a mess. There's not a lot good about it.

Seibel: But not as bad as C++.

Zawinski: No, absolutely not. It's for different things. There's stuff that would be so much easier to write in Perl or any language like Perl than in C just because they're text-oriented—all these so-called

"scripting languages". Which is a distinction I don't really buy—"programming" versus "scripting". I think that's nonsense. But if what you're doing is fundamentally manipulating text or launching programs, like running wget and pulling some HTML out and pattern-matching it, it's going to be easier to do that in Perl than even Emacs Lisp.

Seibel: To say nothing of, Emacs Lisp is not going to be very suitable for command-line utilities.

Zawinski: Yeah, though I used to write just random little utilities in Emacs all the time. There was actually a point, early on in Netscape, where part of our build process involved running "emacs -batch" to manipulate some file. No one really appreciated that.

Seibel: No. I imagine they wouldn't. What about XScreenSaver—do you still work on that?

Zawinski: I still write new screen savers every now and then just for kicks, and that's all C.

Seibel: Do you use some kind of IDE for that?

Zawinski: I just use Emacs, mostly. Though recently, I ported XScreenSaver to OS X. The way I did that was I reimplemented Xlib in terms of Cocoa, the Mac graphics substrate, so I wouldn't have to change the source code of all the screen savers. They're still making X calls but I implemented the back end for each of those. And that was in Objective C, which actually is a pretty nice language. I enjoyed doing that. It definitely feels Java-like in the good ways but it also feels like C. Because it's essentially C, you can still link directly with C code and just call the functions and not have to bend over backwards.

Seibel: At Lucid, leaving aside the politics of Emacs development, what technical stuff did you learn?

Zawinski: I definitely became a better programmer while I was there. Largely because that was really the smartest group of people I've been around. Everyone who worked there was brilliant. And it was just nice to be in that kind of environment where when someone says, "That's

nonsense," or "We should do it this way," you can just take their word for it, believe that they know what they were talking about. That was really nice. Not that I hadn't been around smart people before. But it was just such a high-quality group of people there, consistently.

Seibel: And how big was the development team?

Zawinski: I think there were like 70 people at the company so probably; I don't know, 40 or so on the development team. The Energize team was maybe 25 people, 20. It was divided up into pretty distinct areas. There were the folks working on the compiler side of things and the back-end database side of things. The GUI stuff that wasn't Emacs. And then there was, at one point, me and two or three other people working on integrating Emacs with the environment. That eventually turned into mostly me working on mostly Emacs stuff, trying to make our Emacs 19 be usable, be an editor that doesn't crash all the time, and actually runs all the Emacs packages that you expect it to run.

Seibel: So you wanted the Emacs included in your product to be a fully capable version of Emacs.

Zawinski: The original plan was that we wouldn't include Emacs with our product. You have Emacs on your machine already and you have our product and they work together. And you had GCC on your machine already and our product, and they work together. I think one of the early code names for our product was something like Hitchhiker because the idea was that it would take all the tools that you already have and integrate them—make them talk to each other by providing this communication layer between them.

That didn't work out at all. We ended up shipping our version of GCC and GDB because we couldn't get the changes upstream fast enough, or at all in some cases. And same thing with Emacs. So we ended up shipping the whole thing. We ended up going down the path of, "Well, we're replacing Emacs. Shit. I guess we have to do that so we better make it work." One thing I spent a bunch of time on was making the vi emulation mode work.

Seibel: Six or seven being the whole Netscape development team or the Unix development team?

Zawinski: That was the whole client team. There were also the server folks who were implementing their fork of Apache, basically. We didn't talk to them much because we were busy. We had lunch with them, but that was it. So we figured out what we wanted to be in the thing and we divided up the work so that there were, I guess, no more than two people working on any part of the project. I was doing the Unix side and Lou Montulli did most of back-end network stuff. And Eric Bina was doing layout and Jon Mittelhauser and Chris Houck were doing the Windows front end and Aleks Totić and Mark Lanett were doing the Mac front end for the pre–version 1.0 team. Those teams grew a little bit after that. But we'd have our meetings and then go back to our cubicles and be heads-down for 16 hours trying to make something work.

It was really a great environment. I really enjoyed it. Because everyone was so sure they were right, we fought constantly but it allowed us to communicate fast. Someone would lean over your cubicle and say, "What the fuck did you check in; that's complete bullshit—you can't do it that way. You're an idiot." And you'd say, "Fuck off!" and go look at it and fix it and check it in. We were very abrasive but we communicated fast because you didn't have to go blow sunshine up someone's ass and explain to them what you thought was wrong—you could say, "Hey, that's a load of shit! I can't use that." And you'd hash it out very quickly. It was stressful but we got it done pretty quickly.

Seibel: Are the long hours and the intensity required to produce software quickly?

Zawinski: It's certainly not healthy. I know we did it that way and it worked. So the way to answer that question is, is there another example of someone delivering a big piece of software that fast that's of reasonable quality where they actually had dinner at home and slept during the night? Has that ever happened? I don't actually know. Maybe it has.

But it's not always about getting it done as quickly as possible. It also would be nice to not burn out after two years and be able to continue doing your job for ten. Which is not going to happen if you're working 80-plus hours a week.

Seibel: What is the thing that you worked on that you were most proud of.

Zawinski: Really just the fact that we shipped it. The whole thing. I was very focused on my part, which was the user interface of the Unix front end. But really just that we shipped the thing at all and that people liked it. People converted immediately from NCSA Mosaic and were like, "Wow, this is the greatest thing ever." We had the button for the What's Cool page up in the toolbar and got to show the world these crazy web sites people had put up already. I mean, there were probably almost 200 of them! It's not so much that I was proud of the code; just that it was done. In a lot of ways the code wasn't very good because it was done very fast. But it got the job done. We shipped—that was the bottom line.

That first night when we put up the .96 beta, we were all sitting around the room watching the downloads with sound triggers hooked up to it—that was amazing. A month later two million people were running software I'd written. It was unbelievable. That definitely made it all worthwhile—that we'd had an impact on people's lives; that their day was more fun or more pleasant or easier because of the work we'd done.

Seibel: After this relentless pace, at some point that has to start to catch up with you in terms of the quality of the code. How did you guys deal with that?

Zawinski: Well, the way we dealt with that was badly. There's never a time to start over and rewrite it. And it's never a good idea to start over and rewrite it.

Seibel: At some point you also worked on the mail reader, right?

Zawinski: In 2.0 Marc comes into my cubicle and says, "We need a mail reader." And I'm like, "OK, that sounds cool. I've worked on mail readers before." I was living in Berkeley and basically I didn't come into the office for a couple weeks. I was spending the whole time sitting in cafes doodling, trying to figure out what I wanted in a mail reader. Making lists, crossing it off, trying to decide how long it would take me. What should the UI look like?

Then I came back and started coding. And then Marc comes in again and says, "Oh, so we hired this other guy who's done mail stuff before. You guys should work together." It's this guy Terry Weissman, who was just fantastic—we worked together so well. And it was a completely different dynamic than it had been in the early days with the rest of the browser team.

We didn't yell at each other at all. And the way we divided up labor, I can't imagine how it possibly worked or could ever work for anyone. I had the basic design done and I'd started doing a little coding and every day or every couple of days we'd look at the list of features and I'd go, "Uhhh, maybe I'll work on that," and he'd go, "OK, I'll work on that," and then we'd go away.

Check-ins would happen and then we'd come back and he'd say, "Alright, I'm done with that, what are you doing?" "Uh, I'm working on this." "OK, well, I'll start on that then." And we just sort of divided up the pieces. It worked out really well.

We had disagreements—I thought we had to toss filtering into folders because we just didn't have time to do it right. And he was like, "No, no, I really think we ought to do that." And I was like, "We don't have time!" So he wrote it that night.

The other thing was, Terry and I rarely saw each other because he lived in Santa Cruz and I lived in Berkeley. We were about the same distance from work in opposite directions and because the two of us were the only two who ever needed to communicate, we were just like, "I won't make you come in if you don't make me come in." "Deal!"

Seibel: Did you guys email a lot?

Zawinski: Yeah, constant email. This was before instant messaging—these days it probably all would have been IM because we were sending one-liner emails constantly. And we talked on the phone.

So we shipped 2.0 with the mail reader and it was well-received. Then we're working on 2.1, which is the version of the mail reader that I'm starting to consider done—this is the one with all the stuff that we couldn't ship the first time around. Terry and I are halfway through doing that and Marc comes in and says, "So we're buying this company. And they make a mail-reader thing that's kind of like what you guys did." I'm like, "Oh. OK. Well, we have one of those." And he says, "Well, yeah, but we're growing really fast and it's really hard to hire good people and sometimes the way you hire good people is you just acquire another company because then they've already been vetted for you." "OK. What are these people going to be working on?" "They're going to be working on your project." "OK, that kind of sucks—I'm going to go work on something else."

So basically they acquired this company, Collabra, and hired this whole management structure above me and Terry. Collabra has a product that they had shipped that was similar to what we had done in a lot of ways except it was Windows-only and it had utterly failed in the marketplace.

Then they won the start-up lottery and they got acquired by Netscape. And, basically, Netscape turned over the reins of the company to this company. So rather than just taking over the mail reader they ended up taking over the entire client division. Terry and I had been working on Netscape 2.1 when the Collabra acquisition happened and then the rewrite started. Then clearly their Netscape 3.0 was going to be extremely late and our 2.1 turned into 3.0 because it was time to ship something and we needed it to be a major version.

So the 3.0 that they had begun working on became 4.0 which, as you know, is one of the biggest software disasters there has ever been. It basically killed the company. It took a long time to die, but that was it: the rewrite helmed by this company we'd acquired, who'd never

accomplished much of anything, who disregarded all of our work and all of our success, went straight into second-system syndrome and brought us down.

They thought just by virtue of being here, they were bound for glory doing it their way. But when they were doing it their way, at their company, they failed. So when the people who had been successful said to them, "Look, really, don't use C++; don't use threads," they said, "What are you talking about? You don't know anything."

Well, it was decisions like not using C++ and not using threads that made us ship the product on time. The other big thing was we always shipped all platforms simultaneously; that was another thing they thought was just stupid. "Oh, 90 percent of people are using Windows, so we'll focus on the Windows side of things and then we'll port it later." Which is what many other failed companies have done. If you're trying to ship a cross-platform product, history really shows that's how you don't do it. If you want it to really be cross-platform, you have to do them simultaneously. The porting thing results in a crappy product on the second platform.

Seibel: Was the 4.0 rewrite from scratch?

Zawinski: They didn't start from scratch with a blank disk but they eventually replaced every line of code. And they used C++ from the beginning. Which I fought against so hard and, dammit, I was right. It bloated everything; it introduced all these compatibility problems because when you're programming C++ no one can ever agree on which ten percent of the language is safe to use. There's going to be one guy who decides, "I have to used templates." And then you discover that there are no two compilers that implement templates the same way.

And when your background, your entire background, is writing code where multiplatform means both Windows 3.1 and Windows 95, you have no concept how big a deal that is. So it made the Unix side of things—which thankfully was no longer my problem—a disaster. It made the Mac side of things a disaster. It meant it was no longer possible to ship on low-end Windows boxes like Win16. We had to

of the game. You never shipped your 1.0 because someone else ate your lunch.

Your competitor's six-month 1.0 has crap code and they're going to have to rewrite it in two years but, guess what: they can rewrite it because you don't have a job anymore.

Seibel: There must have been times, perhaps on a shorter time frame, where you've ripped out a big chunk of code because you thought it would be faster to start over.

Zawinski: Yes, there are definitely times when you have to cut your losses. And this always feels wrong to me, but when you inherit code from someone else, sometimes it's faster to write your own than to reuse theirs. Because it's going to take a certain amount of time to understand their code and learn how to use it and understand it well enough to be able to debug it. Where if you started from scratch it would take less time. And it might only do 80 percent of what you need, but maybe that's the 80 percent you actually need.

Seibel: Isn't it exactly this thing—someone comes along and says, "I can't understand this stuff. I'll just rewrite it"—that leads to the endless rewriting you bemoan in open-source development?

Zawinski: Yeah. But there's also another aspect of that which is, efficiency aside, it's just more fun to write your own code than to figure out someone else's. So it's easy to understand why that happens. But the whole Linux/GNOME side of things is straddling this line between someone's hobby and a product. Is this a research project where we're deciding what desktops should look like and we're experimenting? Or are we competing with Macintosh? Which is it? Hard to do both.

But even phrasing it that way makes it sounds like there's someone who's actually in charge making that decision, which isn't true at all. All of this stuff just sort of happens. And one of the things that happens is everything get rewritten all the time and nothing's ever finished. If you're one of those developers, that's fine because there's always something to play around with if your hobby is messing around with

your computer rather than it being a means to an end—being a tool you use to get whatever you're actually interested in done.

Seibel: Speaking of messing around with a computer for its own sake, do you still enjoy programming?

Zawinski: Sometimes. I end up doing all the sysadmin crap, which I can't stand—I've never liked it. I enjoy working on XScreenSaver because in some ways screen savers—the actual display modes rather than the XScreenSaver framework—are the perfect program because they almost always start from scratch and they do something pretty and there's never a version 2.0. There's very rarely a bug in a screen saver. It crashes—oh, there's a divide-by-zero and you fix that.

But no one is ever going to ask for a new feature in a screen saver. "I wish it was more yellow." You're not going to get a bug report like that. It is what it is. So that's why I've always written those for fun. They make this neat result and you don't have to think about them too much. They don't haunt you.

Seibel: And do you enjoy the puzzle of doing some math and figuring out geometry and graphics?

Zawinski: Yeah. What's this abstract little equation going to look like if I display it this way? Or, how can I make these blobs move around that looks more organic and less rigid, like a computer normally moves things? Stuff like that. What do I do to these sine waves to make it look more like something bouncing?

And then I end up writing all these stupid little shell scripts—self-defense stuff. I know I could do this by clicking on 30,000 web pages and doing it by hand, but why don't I write this script—little time-saver things. Which barely feels like programming to me. I know to people who aren't programmers, that seems like a black art.

I really enjoyed doing the Mac port of the XScreenSaver framework. That was actually writing a lot of new code that required thinking about APIs and the structure of the thing.

Seibel: Was that your APIs—how you were structuring your code?

Zawinski: Both. Both figuring out the existing APIs and figuring out the best way to build a layer between the X11 world and the Mac world. How should I structure that? Which of the Mac APIs is most appropriate? It was the first time in a long time that I'd done something like that and it was just like, "Wow, this is kind of fun. I think I might be kind of good at this."

It had been forever because I got completely burned out on the software industry. That part of it I just couldn't take anymore—the politics of it both in the corporate world and in the free-software world. I'd just had too much. I wanted to do something that didn't involve arguing online about trivia. Or having my product destroyed by bureaucratic decisions that I had no input in.

Seibel: Are you ever tempted to go back and hack on Mozilla?

Zawinski: Nah. I just don't want to be arguing with people and having pissing matches in Bugzilla anymore. That's not fun. That kind of thing is necessary to build big products. If it's something that requires more than one person working on it, which obviously Mozilla does, that's the way you have to do it. But I don't look forward to that kind of fight anymore. That's been beaten out of me by too many years of it. And the other alternative, as a programmer, is go work for someone else. And I don't have to do that, so I can't. My first bad day I'd just leave. And were I to start my own company I couldn't be a programmer—I'd have to run the company.

Seibel: Other than having two million people using your software, what about programming do you enjoy?

Zawinski: It's a hard question. The problem-solving aspect of it, I guess. It's not quite like it's a puzzle—I don't really play many puzzle-type games. Just figuring out how to get from point A to point B—how to make the machine do what you want. That's the basic element that the satisfaction of programming comes from.

Seibel: Do you find code beautiful? Is there an aesthetic beyond maintainability?

Zawinski: Yeah, definitely. Anything expressed just right, whether it's being really concise or just capturing it—like anything, a really well-put-together sentence or a little doodle, a caricature that looks just like someone but only used four lines, that kind of thing—it's the same sort of thing.

Seibel: Do you find that programming and writing are similar intellectual exercises?

Zawinski: In some ways, yeah. Programming is obviously much more rigid. But as far as the overall ability to express a thought, they're very similar. Not rambling, having an idea in your head of what you're trying to say, and then being concise about it. I think that kind of thinking is the overlap between programming and writing prose.

It feels like they use similar parts of my brain, but it's hard to express exactly what it is. A lot of times I'll read things that just look like bad code. Like most contracts. The really rigid style they use—it's so repetitive. I look at that and I'm like, why can't you break this out into a subroutine—which we call paragraphs. And the way they usually begin with definitions, like, so and so, referred to as blah blah blah.

Seibel: Lets talk a little bit about the nitty-gritty of programming. How do you design your code? How do you structure code? Maybe take your recent work on the OS X XScreenSaver as an example.

Zawinski: Well, first I messed around and made little demo programs that never ended up being used again. Just to figure out here's how you put a window on the screen, and so on. Since I'm implementing X11, the first thing to do is pick one of the screen savers and make the list of all the X11 calls it makes.

Then I create stubs for each of those and then I slowly start filling them in, figuring out how am I going to implement this one, how am I going to implement this one.

At another level, on the Mac side of things, there's the start-up code. How is the window getting on the screen? And at some point that's going to call out to the X code. One of the trickier parts of that was really figuring out how to set up the build system to make that work in any kind of sane way. So a bunch of experimentation. Moving things around. At some point, maybe I'd had this piece of code on top and this piece of code being called by it. And then maybe those need to be turned inside out. So there's a lot of cut-and-pasting until I kind of wrapped my head around a flow of control that seemed sensible. Then I went in and cleaned things up and put things in more appropriate files so this piece of code is together with this piece of code.

That was sort of the broad strokes, building the infrastructure. After that it was just a matter of moving on to the next screen saver; this one uses these three functions that the previous one hadn't used before, so I've got to implement those. And each of those tasks was fairly straightforward. There were some that ended up being really tricky because the X11 API has a ton of options for putting text on the screen and moving rectangles around. So I kept having to make that piece of code hairier and hairier. But most of them were fairly straightforward.

Seibel: So for each of these X11 calls you're writing an implementation. Did you ever find that you were accumulating lots of bits of very similar code?

Zawinski: Oh, yeah, definitely. Usually by the second or third time you've cut and pasted that piece of code it's like, alright, time to stop cutting and pasting and put it in a subroutine.

Seibel: If you were doing something on the scale of writing a mail reader again, you mentioned starting with a few paragraphs of text and a list of features. Is that the finest granularity that you would get to before you would just start writing code?

Zawinski: Yeah. Maybe there'd be a vague description of the division between library and front end. But probably not. If I was working alone I wouldn't bother with that because that part is just kind of obvious to me. And then the first thing I would do with something like

Zawinski: That's changed over the years a lot. When I was using the Lisp machines it was all about running the program and stopping it and exploring the data—there was an inspector tool that let you browse through memory and I changed it so basically the Lisp listener became an inspector. So anytime time it printed out an object there was a context menu on it so you could click on this thing here and have that value returned. Just to make it easier to follow around chains of objects and that sort of thing. So early on that was how I thought about things. Getting down in the middle of the code and chasing it around and experimenting.

Then when I started writing C and using GDB inside of Emacs I kind of tried to keep up that same way of doing things. That was the model we built Energize around. And that just never seemed like it worked very well. And as time went by I gradually stopped even trying to use tools like that and just stick in print statements and run the thing again. Repeat until done. Especially when you get to more and more primitive environments like JavaScript and Perl and stuff like that, it's the only choice you have—there aren't any debuggers.

People these days seem confused about the notion of what a debugger is. "Oh, why would you need that? What does it do—put print statements in for you? I don't understand. What are these strange words you use?" Mostly these days it's print statements.

Seibel: How much of that was due to the differences between Lisp and C, as opposed to the tools—one difference is that in Lisp you can test small parts—you can call a small function you're not sure is working right and then put a break in the middle of it and then inspect what's going on. Whereas C it's like, run the whole program in all of its complex glory and put a break point somewhere.

Zawinski: Lisp-like languages lend themselves more to that than C. Perl and Python and languages like that have a little more of the Lisp nature in that way but I still don't see people doing it that way very much.

Seibel: But GDB does give you the ability to inspect stuff. What about it makes it not usable for you?

Zawinski: I always found it unpleasant. Part of it is just intrinsic to being C. Poking around in an array and now I'm looking at a bunch of numbers and now I have to go in there and cast the thing to whatever it really is. It just never managed that right, the way a better language would.

Seibel: Whereas in Lisp, if you're looking at a Lisp array, they'll just be printed as those things because it knows what they are.

Zawinski: Exactly. It always just seemed in GDB like bouncing up and down, the stack things would just get messed up. You'd go up the stack and things have changed out from under you and often that's because GDB is malfunctioning in some way. Or, oh well, it was expecting this register to be here and you're in a different stack frame, so that doesn't count anymore.

It just always felt like I couldn't really trust what the debugger was actually telling me. It would print something and, look, there's a number. Is that true or not? I don't know. And a lot of times you'd end up with no debug info. So you're in a stack frame and it looks like it has no arguments and then I'd spend ten minutes trying to remember the register that argument zero goes in is. Then give up, relink and put in a print statement.

It seemed like as time went by the debugging facilities just kept getting worse and worse. But on the other hand now people are finally realizing that manual memory allocation is not the way to go; it kind of doesn't matter as much any more because the sorts of really complicated bugs where you'd have to dig deep into data structures don't really happen as often because those, often, in C anyway, were memory-corruption issues.

Seibel: Do you use assertions or other more or less formal ways of documenting or actually checking invariants?

Zawinski: We went back and forth about what to do about assertions in the Netscape code base. Obviously putting in assert statements is always a good idea for debugging and like you said, for documentation purposes. It expressed the intent. We did that a lot.

But then the question is, well, what happens when the assertion fails and you're in production code; what do you do? And we sort of decided that what you do is return zero and hope it keeps going. Because having the browser go down is really bad—it's worse than returning to the idle loop and maybe having leaked a bunch of memory or something. That's going to upset people less than the alternative.

A lot of programmers have the instinct of, "You've got to present the error message!" No you don't. No one cares about that. That sort of stuff is a lot easier to manage in languages like Java that actually have an exception system. Where, at the top loop of your idle state, you just catch everything and you're done. No need to bother the user with telling them that some value was zero.

Seibel: Did you ever just step through a program—either to debug it or, as some people recommend, to just step through it once you've written it as a way of checking it?

Zawinski: No, not really. I only really do stepping when I'm debugging something. I guess sometimes to make sure I wrote it right. But not that often.

Seibel: So how do you go about debugging?

Zawinski: I just eyeball the code first. Read through it until I think, well, this can't happen because that's going on right there. And then I put in something to try and resolve that contradiction. Or if I read it and it looks fine then I'll stop in the middle or something and see where it is. It depends. It's hard to generalize about that.

Seibel: As far as the assertions—how formally do you think? Some people just use ad hoc assertions—here's something that I think should be true here. And some people think very formally—functions have preconditions and postconditions and there are global invariants. Where are you on that scale?

Zawinski: I definitely don't think about things in a mathematically provable way. I'm definitely more ad hoc than that. You know, it's always helpful when you've got inputs to a function to at least have an

idea in your head what their bounds are. Can this be an empty string? That sort of thing.

Seibel: Related to debugging, there's testing. At Netscape, did you guys have a separate QA group or did you guys test everything yourselves?

Zawinski: We did both. We were all running it all the time, which is always your best front-line QA. Then we had a QA group and they had formal tests they went through. And every time there was a new release they'd go down the list and try this thing. Go to this page, click on this. You should see this. Or you shouldn't see this.

Seibel: What about developer-level tests like unit tests?

Zawinski: Nah. We never did any of that. I did occasionally for some things. The date parser for mail headers had a gigantic set of test cases. Back then, at least, no one really paid a whole lot of attention to the standards. So you got all kinds of crap in the headers. And whatever you're throwing at us, people are going to be annoyed if their mail sorts wrong. So I collected a whole bunch of examples online and just made stuff up and had this giant list of crappily formatted dates and the number I thought that should turn into. And every time I'd change the code I'd run through the tests and some of them would flip. Well, do I agree with that or not?

Seibel: Did that kind of thing get folded into any kind of automated testing?

Zawinski: No, when I was writing unit tests like that for my code they would basically only run when I ran them. We did a little bit of that later with Grendel, the Java rewrite, because it was just so much easier to write a unit test when you write a new class.

Seibel: In retrospect, do you think you suffered at all because of that? Would development have been easier or faster if you guys had been more disciplined about testing?

Zawinski: I don't think so. I think it would have just slowed us down. There's a lot to be said for just getting it right the first time. In the early days we were so focused on speed. We had to ship the thing even if it wasn't perfect. We can ship it later and it would be higher quality but someone else might have eaten our lunch by then.

There's bound to be stuff where this would have gone faster if we'd had unit tests or smaller modules or whatever. That all sounds great in principle. Given a leisurely development pace, that's certainly the way to go. But when you're looking at, "We've got to go from zero to done in six weeks," well, I can't do that unless I cut something out. And what I'm going to cut out is the stuff that's not absolutely critical. And unit tests are not critical. If there's no unit test the customer isn't going to complain about that. That's an upstream issue.

I hope I don't sound like I'm saying, "Testing is for chumps." It's not. It's a matter of priorities. Are you trying to write good software or are you trying to be done by next week? You can't do both. One of the jokes we made at Netscape a lot was, "We're absolutely 100 percent committed to quality. We're going to ship the highest-quality product we can on March 31st."

Seibel: That leads to another topic, maintaining software. How do you tackle understanding a piece of code that you didn't write?

Zawinski: I just dive in and start reading the code.

Seibel: So where do you start? Do you start at page one and read linearly?

Zawinski: Sometimes. The more common thing is learning how to use some new library or toolkit. If you're lucky there's some documentation. There's an API. So you figure out the piece of it you might be interested in using. Or work out how that was implemented. Thread your way through. Or with something like Emacs, maybe start at the bottom. What are cons cells made of? How's that look? And then skip around from there. Sometimes starting with the build system can give you an idea how things fit together. I always find that a good

Zawinski: I usually end up putting the leaf nodes up at the top of the file—try to keep it basically structured that way. And then usually up at the top, document the API. What are the top-level entry points of this file, this module, whatever? With an object-y language, that's done by the language for you. With C you've got to be a little more explicit about that. In C I do tend to try to have a .h file for every .c file that has all the externs for it. And anything that's not exported in the .h file is static. And then I'll go back and say, "Wait, I need to call that," and I change it. But you're doing that explicitly rather than just by accident.

Seibel: You put the leaves first in the file, but is that how you write? Do you build up from leaves?

Zawinski: Not always. Sometimes I start at the top and sometimes I start at the bottom. It depends. One way is, I know I'm going to need these building blocks and I'll put those together first. Or another way of thinking about it is, you've sort of got an outline of it in your head and you dig down. I do it both ways.

Seibel: So suppose for the sake of argument that you were going to come out of retirement and build a development team. How would you organize it?

Zawinski: Well, I think you want to arrange for there to be no more than three or four people working really closely together on a day-to-day basis. Then that can scale up a lot. Say you've got a project where you can divide it up into twenty-five really distinct modules. Well, you can have twenty-five tiny teams—maybe that's a little much. Say ten. And as long as they can coordinate with each other, I don't think there's a whole lot of limit to how big you can scale that. Eventually it just starts looking like multiple projects instead of like one project.

Seibel: So you've got multiple teams of up to four people. How do you coordinate the teams? Do you have one grand architect who's managing the dependencies and mediating between those teams?

Zawinski: Well, there's got to be agreement about what the interface between modules is. For that very modular approach to work at all, the interface between modules has to be clear and simple. Which,

hopefully, means it won't take too much screaming for everyone to agree on it and it won't be too difficult to follow the module contract. I guess what I'm getting at is the best way to make interaction between modules be easy is to just make it be really simple. Make there be fewer ways for that to go wrong.

And what lines you divide on depends entirely on a project. With some kind of web app you've probably got the UI and you've got your database and the part that runs on the server and the part that runs on the machine behind the server. And if it's a desktop application it's similar division of labor. There's file formats and GUI and basic command structure.

Seibel: How do you recognize talent?

Zawinski: That I don't know. I've never really been the person who had to hire people. And when I've been involved in interviews I've always just felt like I had no idea. I can tell from the interview whether I'd get along with this person, but I can't tell whether they're any good or not just by talking to them. I always found that difficult.

Seibel: How about if they're bad? Are there reliable clues then?

Zawinski: Sometimes. Normally I would think that someone who is a big fan of C++ templates—keep me away from that guy. But that might just be a snap judgment on my part. Maybe in the context they've used them, they actually work fine. Certainly with the folks I've worked with, ability to argue their point was important because we all ended up being a pretty argumentative bunch. With that environment, that helped a lot. That certainly doesn't have anything to do with programming ability. That's just interpersonal-dynamics stuff.

Seibel: And on a different team, that would actually be detrimental.

Zawinski: Yeah, absolutely.

Seibel: It sounds like at Netscape you guys divided things up so people owned different parts of the software. Some people think

that's really important. Other folks say it's better for a team to collectively own all the code. Do you have an opinion on that?

Zawinski: I've done it both ways. They both have their merits. The idea that everyone should own all the code, I don't think is really practical because there's going to be too much of it. People are going to have to specialize; you need an expert sometimes. It's just always going to work out that way. There's always going to be the code you're familiar with because you happened to write more of that module than some other guy did. Or there's just going to be parts that resonate with you more. It's certainly good for other people to have their hands in it, if only because you're not going to be maintaining it yourself forever. It's going to have to be handed off to someone else for one reason or another. And for that knowledge to be spread around is good. But it's also good to have someone to blame. If everybody is responsible for it then there's no one to put their foot down.

Seibel: Have you ever been a manager?

Zawinski: Not really. When I was doing the Emacs stuff at Lucid, there would be a lot of modules that were included in Lucid Emacs that were written by other people. Those people didn't really work for me but it was a little bit like management. And a lot of those people were definitely less experienced and the way that worked out well was they were doing their favorite thing and I was basically giving them feedback: "Well, I want to include this but first I need this, this, and this from it."

Seibel: And did you give them a free rein? You tell them you want X, Y, and Z and then they get to figure out how to do it?

Zawinski: Yeah. If I'm trying to decide whether to include this module in the thing that I'm going to ship, I'm going to have requirements about it. Does the damn thing work is really the bottom line there. So I would give them advice on, "I think you're going to have better luck if you try it this rather than this way." But I wanted it to work and I wanted to not have to be the one to write it. If they wanted to go do it in some crazy way but it worked, that was OK

because that gave me point two: I didn't have to write it. But mostly the feedback I was giving them was just, does it work and does it make sense.

Seibel: On the flip side, when you were the less-experienced programmer, what did your mentors do that was helpful?

Zawinski: I guess the most important thing is they'd recognize when it was time to level up. When I went to work for Fahlman I was given some silly little busy work. And eventually got given tasks that were a little more significant—not that they were significant at all really.

Seibel: I think you talked about Skef, who just hovered and said, "Wrong!" Was that balanced, perhaps, by someone else who was a little more nurturing?

Zawinski: Well, he wasn't completely a cave man. He would actually tell me things, too. I know I ended up doing a lot of reading of code and asking questions. I think one thing that's really important is to not be afraid of your ignorance. If you don't understand how something works, ask someone who does. A lot of people are skittish about that. And that doesn't help anybody. Not knowing something doesn't mean you're dumb—it just means you don't know it yet.

Seibel: Did you read code mostly because it was something you were working on, or was it just something you wanted to know how it worked?

Zawinski: Yeah. Just poking around—"I wonder how that works." The impulse to take things apart is a big part of what gets people into this line of work.

Seibel: Were you actually one of those kids who took toasters apart?

Zawinski: Yeah. I made a telephone and learned how to dial with a telegraph tapper that I made out of a tin can. When I was little I had these old books I got at a garage sale or something, like *Boy's Own Science Book* from the '30s, and I remember getting a really big kick out of those. That was really hacker culture in the '20s and '30s where

they're showing how to wire up a telegraph between your room and the barn and making Leyden jars.

Seibel: That brings me to another of my standard questions: do you, as a programmer, think of yourself as a scientist or an engineer or an artist or a craftsman or something else?

Zawinski: Well, definitely not a scientist or engineer because those have very formal connotations. I don't do a lot of math; I don't draw blueprints; I don't prove things. I guess somewhere between craftsman and artist, depending on what the project is. I write a lot of screen savers—that's not craftsman; that's making pretty pictures. Somewhere in that area.

Seibel: Do you feel like you taught yourself computer science or did you just learn to program?

Zawinski: Well, I certainly picked up a bunch of computer science over the years. But learning to program was the goal. Making the machine do something was the goal and the computer-science side of it was a means to an end.

Seibel: Did you ever feel that as a lack—did you ever wish you had been exposed to things in a more systematic way?

Zawinski: There were definitely times, especially at Lucid, where it'd be obvious that there's this whole big black hole that these guys are talking about that I just completely missed because I never needed to know it. And I'd then I'd pick up the terminology and have a basic idea what they're talking about and maybe do a little bit of reading on it if it was something I needed to know. So there were definitely times, especially early on, where I felt like, "Oh my god, I don't know anything." It would just be embarrassing—but that was just being insecure. Being the young kid around all these people with PhDs—"Aaah, I don't know anything! I'm an idiot! How did I bluff my way into this?"

Then there was another book that everybody thought was the greatest thing ever in that same period—*Design Patterns*—which I just thought was crap. It was just like, programming via cut and paste. Rather than thinking through your task you looked through the recipe book and found something that maybe, kinda, sorta felt like it, and then just aped it. That's not programming; that's a coloring book. But a lot of people seemed to love it. Then in meetings they'd be tossing around all this terminology they got out of that book. Like, the inverse, reverse, double-back-flip pattern—whatever. Oh, you mean a loop? OK.

Seibel: Is there a key skill programmers must have?

Zawinski: Well, curiosity—taking things apart. Wanting to know what's going on under the hood. I think that's really the basis of it. Without that I don't think you get very far. That's your primary way of acquiring knowledge. Taking something apart and looking at it is how you learn to build your own. At least for me. I've read very few books about computers. My experience has been digging through source code or reference manuals. I've got a goal and, alright, to do this I need to know what this thing does and what this thing does. And I'll just sort of random-walk through that until I find where I'm going.

Seibel: Have you read Knuth's, *The Art of Computer Programming*?

Zawinski: I haven't. And that's one of those things where, I really probably should have. But I never did.

Seibel: It's tough going—you need a lot of math to really grok it.

Zawinski: And I'm not a math person at all.

Seibel: That's interesting. Lots of programmers come out of mathematics and lots of computer-science theory is very mathematical. So you're an existence proof that it's not absolutely necessary. How much math or mathy kind of thinking is necessary to be a good programmer?

Zawinski: Well, it depends on where you draw the line as to what's mathy and what's not. Is being good at pattern matching mathy? Having an understanding of orders of magnitude and combinatorics is important at a gut level. But I'm sure I would completely flunk if I had to take a basic intro quiz on that kind of stuff. It's been so long since I've had to do anything formal like that.

Really the only math classes I had were in high school. I had algebra. A little bit of calculus. I wasn't terribly good at it. I got through it but it didn't really come naturally to me. I had a physics class in high school where we were doing mechanics and doing labs dragging blocks across sandpaper and stuff like that. I did terribly in that class and felt like an idiot because I actually enjoyed the class. I did the labs really well—the procedure was spot on—and then I just couldn't do the math.

I'd get an answer that I knew was three orders of magnitude off. I'd show my work—I don't know what I did wrong. I'd get half credit since the data was collected properly and I cleaned up afterwards. So math was never really my forte.

But I wouldn't go so far as to say you don't need that to be a programmer. There's obviously different kinds of programming. Without people who are not like me none of this would exist. But I've always seen much more in common with writing prose than math. It feels like you're writing a story and you're trying to express a concept to a very dumb person—the computer—who has a limited vocabulary. You've got this concept you want to express and limited tools to express it with. What words do you use and what does your introductory and summary statement look like? That sort of thing.

The issue of taste really fits in there. You can have a piece of text describing something that describes it correctly or it can describe it well, with some flair. And the same thing's true of programs. It can get the job done or it can also make sense, be put together well.

Seibel: And why does that matter? Is that just for the satisfaction of it or is tasteful code also better in some practical way?

Zawinski: To a large degree, tasteful and maintainable are similar. Or very closely related. One of the things that makes a piece of writing tasteful is if it's structured in a way that's easy to grasp. Are the facts loaded up at the front or are they scattered around? If you're referring back—if you're flipping through a book, can you figure out where in the book is the thing you kind of remember? "This was somewhere near the middle because that's where he talked about this thing." Or is it just scattered all through. And that's the same sort of thing that goes on with programming a lot.

Seibel: Do you think the kind of people who can be successful at programming has changed?

Zawinski: Certainly these days it's impossible to just write a program from scratch that doesn't have any dependencies. The explosion of toolkits and libraries and frameworks and that sort of thing—even the most basic piece of software needs those these days. It's just exploded. These days, everything's got to be a web app. And that's just a whole different way of going about it.

So, if anything, that makes the part of the skill set that is being able to dive into someone else's code and figure out how to make use of it even more important. "I don't understand this, so I'm going to write my own" worked better in the past. Whether it was ever a good idea or not, you could do it. It's much harder to get away with that now.

Seibel: I wonder if the inclination to take things apart and understand everything also needs to be a little more tempered these days. If you try to take apart every piece of code you work with, it'll never end—these days you've got to have a little capacity for saying, "I sort of understand how this works and I'm going to let it go at that until it becomes urgent that I understand it better."

Zawinski: Yeah. My first instinct, because things work that way, is you're breeding a generation of programmers who don't understand anything about efficiency or what's actually being allocated. When they realize, "Oh, my program's getting gigantic," what are they going to do? They're not going to know where to start. That's my first instinct

because I'm a caveman. Really that probably doesn't even matter because you'll just throw more memory at it and it'll be fine.

Seibel: Or perhaps people will actually learn a more sophisticated view of what all those things mean. Like, maybe it doesn't really matter whether we allocated six bytes or four bytes here—what matters is whether we've sized this thing so it fits in one node of the cluster versus having to spill over onto a second node.

Zawinski: Right, exactly. I think programming has definitely changed in that sense. The things you had to focus on before were different. Before you would focus on counting bytes, and "How big are my objects? Maybe I should do something different here because that array header is really going to add up." Things like that. No one is ever going to care about that stuff again. Tricks like XORing your forward and back pointers into the same word are voodoo—why would anyone do that; it's crazy. But there's this whole different set of skills now that were always around but come more to the front. People who can dig into an API and figure out which parts you need and which parts you don't, is, I think, one of those important things now.

Seibel: If you were 13 today, would you be drawn to programming the way programming is today?

Zawinski: So hard to say. I don't know any 13-year-olds. I don't know what the world looks like. Things are harder to take apart now. There's not going to be some 10-year-old who pops open his cell phone and figures out how the speaker works like I did with a phone when I was a little kid. There are no user-serviceable parts anymore.

I feel like that kind of tinkering, taking the back of the tape deck off and seeing how the gears fit together, that sort of exploration is what led to this for me. Aside from things like LEGO Mindstorms, I don't think there's a lot of opportunity for people to follow that path these days. But maybe I'm wrong—like I said, I don't know any 13-year-olds. I don't know what the toys are like. There's a lot of video games; there's a lot of things with remote controls. I haven't seen any really good construction kinds of toys. Which seems sad.

Seibel: On the flip side, programming itself is much more accessible. You don't have to master all the intricacies of assembly programming right off the bat just to making a computer do something neat.

Zawinski: Yeah. I imagine that today's kids who are getting into programming start off building some web app or writing a Facebook plugin or something. Brad Fitzpatrick, who wrote LiveJournal, is a friend of mine. When he wrote LiveJournal he was goofing around and wrote this Perl script where he and his friends could say, "I'm going to get lunch." The way he started out was he wrote a little Perl script and put it on a web server. Probably things will go more in that direction.

introduced me to C and gave me Turbo C. This was maybe when I was eight or ten. My dad moved to Intel in '84 and we moved to Portland. He helped design the 386 and 486. He's still at Intel. We always had new, fun computers.

Seibel: Did you get into assembly programming at all?

Fitzpatrick: I did assembly a little on calculators. Like Z80 on the TI calculators, but that was about it.

Seibel: Do you remember what it was that drew you to programming?

Fitzpatrick: I don't know. It was just always fun. My mom had to cut me off and give me computer coupons to make me go outside and play with friends. My friends would come over: "Brad's on the computer again. He's so boring." My mom's like, "Go outside and play."

Seibel: Do you remember the first interesting program that you wrote?

Fitzpatrick: We had this Epson printer and it came with big, thick manuals with a programmers' reference at the end. So I wrote something—this was back on an Apple—where I could draw something in the high graphics mode, and then, once my program finished drawing whatever it was drawing—lines or patterns or something—I'd hit control C and be typing in the background, in a frame buffer that's not showing, and load my other program, which read the screen off and printed it.

Before that I remember writing something that every time I hit a key, it moved the head and I had wired backspace up to go backwards so as I typed it felt like a typewriter.

This was one of my first programs—it was something like K equals grab the next char. Then I said if K equals "a", print "a"; if K is "b", print "b". I pretty much did every letter, number, and some punctuation. Then at one point I was like, "Wait, I could just say, 'Print the variable!'" and I replaced 40 lines of code with one. I was like, "Holy shit, that was awesome!" That was some major abstraction for a six-year-old.

Those are the notable early ones. Then in middle school I would make games and I would make the graphics editors and the level editors for my

friends, and my friends would make the graphics into levels, and then we would sell it to our classmates. I remember having to make games that detected EGA versus VGA. If one of 'em failed on VGA, it would fault back to EGA and use a different set of tiles that fit on the screen, so we'd have to have two sets of graphics for everything. People from school would buy it for like five bucks and they would go to install it and it wouldn't work, and their parents would call my parents and yell, "Your son stole five dollars from my kid for this crap that doesn't work." My mom would drive me over there and sit in the cul-de-sac while I went in and debugged it and fixed it.

Seibel: During that time did you take any classes on programming?

Fitzpatrick: Not really. It was all one or two books from the library, and then just playing around. There weren't really forums or the Internet. At one point I got on a BBS, but the BBS didn't really have anything on it. It wasn't connected to the Net, so it was people playing board games.

Seibel: Did your school have AP computer science or anything?

Fitzpatrick: Well, we didn't have AP C.S., but we had a computer-programming class. There was a guy teaching it but then I would teach sort of an advanced class in the back. They still use the graphics editor and the graphics library I wrote—their final project is to make a game. I still occasionally run into that C.S. teacher—he's a friend of my family's and I'll see him at my brother's soccer games—he'll be like, "Yep, we still use your libraries."

I did take the AP C.S. test. It was the last year it was in Pascal before they switched to C, which was one year before they switched to Java or something like that. I didn't know Pascal so I went to a neighboring high school that had AP C.S. and I went to some night classes, like three or four of them. Then I found a book and learned the language, and I spent most of my time building asteroids in Pascal because I had just learned trig. I was like, "Oooh, sin and cosin; these are fun. I can get thrust and stuff like that."

Seibel: How'd you do?

Fitzpatrick: Oh, I got a five. I had to write bigint classes. Now that's one of the interview questions I give people. "Write a class to do arbitrary,

bigint manipulation with multiplication and division." If I did it in high school on an AP test, they should be able to do it here.

Seibel: Your freshman year in college you worked at Intel during the summer. Did you also work as a programmer during high school?

Fitzpatrick: Yeah, I worked at Tektronix for a while. Before I had any official job, I got some hosting account. I got kicked off of AOL for writing bots, flooding their chat rooms, and just being annoying. I was scripting the AOL client from another Windows program. I also wrote a bot to flood their online form to send you a CD. I used every variation of my name, because I didn't want their duplicate suppression to only send me one CD, because they had those 100 free hours, or 5,000 free hours. I submitted this form a couple thousand times and for a week or so the postman would be coming with bundles of CDs wrapped up.

My mom was like, "Damn it, Brad, you're going to get in trouble." I was like, "Eh—their fucking fault, right?" Then one day I get a phone call and I actually picked up the phone, which I normally didn't, and it was someone from AOL. They were just screaming at me. "Stop sending us all these form submissions!" I'm not normally this quick and clever, but I just yelled back, "Why are you sending me all this crap? Every day the postman comes! He's dropping off all these CDs!" They're like, "We're so sorry, sir. It won't happen again." Then I used all those and I decorated my dorm room in college with them. I actually still have them in a box in the garage. I can't get rid of them because I just remember them being such a good decoration at one point.

After I got kicked off of AOL, I got a shell account on some local ISP. That's basically where I learned Unix. I couldn't run CGI scripts, but I could FTP up, so I would run Perl stuff on my desktop at home to generate my whole website and then upload it. Then I got a job at Tektronix, like a summer intern job. I knew Perl really well and I knew web stuff really well, but I had never done dynamic web stuff. This was probably '95, '94—the web was pretty damn new.

Then I go to work at Tektronix and on my first day they're introducing me to stuff, and they're like, "Here's your computer." It's this big SPARCstation or something running X and Motif. And, "Here's your browser." It's

Netscape 2 or something—I don't remember. And, "If you have some CGIs, they go in this directory." I remember I got a basic hello-world CGI, like three lines working that night and I was like, "Holy shit, this is so fun." I was at work the next day at six in the morning and just going crazy with CGI stuff.

Then I started doing dynamic web-programming stuff on my own. Maybe at that point I had found a web server for Windows that supported CGI. I finally convinced my ISP—I'd made friends with them enough, or sent enough intelligent things that they trusted me—so they said, "OK, we'll run your CGIs but we're going to audit them all first." They'd skim them and toss them in their directory. So I started running this Voting Booth script where you created a topic like, "What's your favorite movie?" and you could add things to it and vote them up. That got more and more popular. That was going on in the background for a couple of years.

Seibel: That was FreeVote?

Fitzpatrick: Yeah, that turned into FreeVote after it flooded my host. Banner ads were really popular then, or they were just getting really popular, and I kept getting more and more money from that, better contracts, more cost per click. At the height I was getting 27 cents per click of banner ads, which I think is pretty ridiculous even by today's standards. So at the height, I was making like 25, 27 grand per month on fucking clicks on banner ads.

This was all through high school—I did this in the background all of high school. And I worked at Intel two summers, and then started doing LiveJournal my last summer, right before college. So then my first year of college, I was just selling FreeVote, which I basically sold for nothing to a friend, for like 11 grand just because I wanted to get rid of it and get rid of legal responsibility for it.

Seibel: When you got on your ISP and got to use Unix, did that change your programming much?

Fitzpatrick: Yeah. It didn't drive me crazy. I couldn't understand what was going on with Windows. You've probably seen the Windows API—there are like twenty parameters to every function and they're all flags and half of

them are zero. No clue what's going on. And you can't go peek underneath the covers when something's magically not working.

Seibel: Are there big differences you can identify between your early approach to programming or programming style to the way you think about programming now?

Fitzpatrick: I went through lots of styles, object-oriented stuff, and then functional stuff, and then this weird, hybrid mix of object-oriented and functional programming. This is why I really love Perl. As ugly as the syntax is and as much historical baggage and warts as it has, it never fucks with me and tells me what style to write in. Any style you want is fine. You can make your code pretty and consistent, but there's no language-specified style. It's only since I've been at Google that I've stopped writing much Perl.

I've also done a lot of testing since LiveJournal. Once I started working with other people especially. And once I realized that code I write never fucking goes away and I'm going to be a maintainer for life. I get comments about blog posts that are almost 10 years old. "Hey, I found this code. I found a bug," and I'm suddenly maintaining code.

I now maintain so much code, and there's other people working with it, if there's anything halfway clever at all, I just assume that somebody else is going to not understand some invariants I have. So basically anytime I do something clever, I make sure I have a test in there to break really loudly and to tell them that they messed up. I had to force a lot of people to write tests, mostly people who were working for me. I would write tests to guard against my own code breaking, and then once they wrote code, I was like, "Are you even sure that works? Write a test. Prove it to me." At a certain point, people realize, "Holy crap, it does pay off," especially maintenance costs later.

Seibel: When did you start working with other people?

Fitzpatrick: It was pretty much towards the end of college when I started hiring other people, and especially once I moved back to Portland after college.

$10.00 a month. Why isn't it working?" So he would say, "Oh, do this." Pretty soon I was learning Unix and learning what was actually going on.

Then I converted to FastGCI. Then I tuned Apache and turned off reverse DNS lookups. All these steps you go through. Finally, I was I/O-bound or CPU-bound. Then I got my own dedicated server, but it was still just one and it was dying and I was out of capacity. I had originally opened it up for my friends and I just left the signup page alive. Then they invited their friends who invited their friends—it was never really supposed to be a public site. It just had an open signup page on accident. So then I put something up on the LiveJournal news page and I said, "Help. We need to buy servers."

I think that raised maybe six or seven thousand dollars or something to buy these two big Dells and put them in Speakeasy in downtown Seattle. Somebody recommend some servers, Dells, these huge 6U things, like ninety pounds each. The logical split was the database server and the web server. That was the only division I knew because I was running a MySQL process and an Apache process.

That worked well for a while. The web servers spoke directly to the world and had two network cards and had a little crossover cable to the database server. Then the web server got overloaded, but that was still fairly easy. At this point I got 1U servers. Then we had three web servers and one database server. At that point, I started playing with three or four HTTP load balancers—mod_backhand and mod_proxy and Squid and hated them all. That started my hate for HTTP load balancers.

The next thing to fall over was the database, and that's when I was like, "Oh, shit." The web servers scale out so nicely. They're all stateless. You just throw more of them and spread load. So that was a long stressful time. "Well, I can optimize queries for a while," but that only gives you another week until it's loaded again. So at some point, I started thinking about what does an individual request need.

That's when—I thought I was the first person in the world to think of this—I was like, we'll shard it out—partition it. So I wrote up design doc with pictures saying how our code would work. "We'll have our master database just for metadata about global things that are low traffic and all the per-blog

and per-comment stuff will be partitioned onto a per-user database cluster. These user IDs are on this database partition." Obvious in retrospect—it's what everyone does. Then there was a big effort to port the code while the service was still running.

Seibel: Was there a red-flag day where you just flipped everything over?

Fitzpatrick: No. Every user had a flag basically saying what cluster number they were on. If it was zero, they were on the master; if it was nonzero, they were partitioned out. Then there was a "Your Account Is Locked" version number. So it would lock and try to migrate the data and then retry if you'd done some mutation in the meantime—basically, wait 'til we've done a migration where you hadn't done any write on the master, and then pivot and say, "OK, now you're over there."

This migration took months to run in the background. We calculated that if we just did a straight data dump and wrote something to split out the SQL files and reload it, it would have taken a week or something. We could have a week of downtime or two months of slow migration. And as we migrated, say, 10 percent of the users, the site became bearable again for the other ones, so then we could turn up the rate of migration off the loaded cluster.

Seibel: That was all pre-memcached and pre-Perlbal.

Fitzpatrick: Yeah, pre-Perlbal for sure. Memcached might have come after that. I don't think I did memcached until like right after college, right when I moved out. I remember coming up with the idea. I was in my shower one day. The site was melting down and I was showering and then I realized we had all this free memory all over the place. I whipped up a prototype that night, wrote the server in Perl and the client in Perl, and the server just fell over because it was just way too much CPU for a Perl server. So we started rewriting it in C.

Seibel: So that saved you from having to buy more database servers.

Fitzpatrick: Yeah, because they were expensive and slow to migrate. Web servers were cheap and we could add them and they would take effect immediately. You buy a new database and it's like a week of setup and validation: test its disks, and set it all up and tune it.

Seibel: So all the pieces of infrastructure you built, like memcached and Perlbal, were written in response to the actual scaling needs of LiveJournal?

Fitzpatrick: Oh, yeah. Everything we built was because the site was falling over and we were working all night to build a new infrastructure thing. We bought one NetApp ever. We asked, "How much does it cost?" and they're like, "Tell us about your business model." "We have paid accounts." "How many customers do you have? What do you charge?" You just see them multiplying. "The price is: all the disposable income you have without going broke." We're like, "Fuck you." But we needed it, so we bought one. We weren't too impressed with the I/O on it and it was way too expensive and there was still a single point of failure. They were trying to sell us a configuration that would be high availability and we were like, "Fuck it. We're not buying any more of these things."

So then we just started working on a file system. I'm not even sure the GFS paper had published at this point—I think I'd heard about it from somebody. At this point I was always spraying memory all over just by taking a hash of the key and picking the shard. Why can't we do this with files? Well, files are permanent. So, we should record actually where it is because configuration will change over time as we add a more storage nodes. That's not much I/O, just keeping track of where stuff is, but how do we make that high availability? So we figured that part out, and I came up with a scheme: "Here's all the reads and writes we'll do to find where stuff is." And I wrote the MySQL schema first for the master and the tracker for where the files are. Then I was like, "Holy shit! Then this part could just be HTTP. This isn't hard at all!"

I remember coming into work after I'd been up all night thinking about this. We had a conference room downstairs in the shared office building—a really dingy, gross conference room. "All right, everyone, stop. We're going downstairs. We're drawing." Which is pretty much what I said every time we had a design—we'd go find the whiteboards to draw.

I explained the schema and who talks to who, and who does what with the request. Then we went upstairs and I think I first ordered all the hardware because it takes two weeks or something to get it. Then we started writing the code, hoping we'd have the code done by the time the machines arrived.

Everything was always under fire. Something was always breaking so we were always writing new infrastructure components.

Seibel: Are there things that if someone had just sat you down at the very beginning and told you, "You need to know X, Y, and Z," that your life would have been much easier?

Fitzpatrick: It's always easier to do something right the first time than to do a migration with a live service. That's the biggest pain in the ass ever. Everything I've described, you could do on a single machine. Design it like this to begin with. You no longer make assumptions about being able to join this user data with this user data or something like that. Assume that you're going to want to load these 20 assets—your implementation can be to load them all from the same table but your higher-level code that just says, "I want these 20 objects" can have an implementation that scatter-gathers over a whole bunch of machines. If I would have done that from the beginning, I'd have saved a lot of migration pain.

Seibel: So basically the lesson is, "You have to plan for the day when your data doesn't all fit into one database."

Fitzpatrick: Which I think is common knowledge nowadays in the web community. And people can go overkill on assuming that their site is going to be huge. But at the time, the common knowledge was, Apache is all you need and MySQL is all you need.

Seibel: It does seem that while you were writing all this stuff because you needed it, you also enjoyed doing it.

Fitzpatrick: Oh, yeah. I definitely try to find an excuse to use anything, to learn it. Because you never learn something until you have to write something in it, until you have to live and breathe it. It's one thing to go learn a language for fun, but until you write some big, complex system in it, you don't really learn it.

Seibel: So what languages would you say you've really lived and breathed with enough to claim as your own?

Fitzpatrick: Perl. C. Back in the day, BASIC, but I'm not even sure BASIC counts. I wrote a lot of Logo too. In our Logo class in elementary school, people were doing pen up, pen down and I would be not in graphics mode—there's some key to get out of graphics mode—writing functions. My teacher would come over and say, "What are you doing? You're doing the wrong thing. You're supposed to be drawing houses." "No, I'm writing Logo. Look," "No, you're not." Then at the end of the class I'd do something—I had a library that drew every letter of the alphabet, but at arbitrary scales and rotations. So I could print entire messages on wavy banners going into the distance and stuff, and everyone was like, "What the fuck?" I don't know if that one counts either.

But a lot of Perl and C, and then a lot of C++ in college for work and for Windows stuff. Then I forgot C++, or it atrophied, and now at Google, in the last year, it's a lot of C++, Python, and Java. I also wrote a lot of Java back in the day when it first came out, but then I got sick of it. Now I'm writing a lot of Java again, and I'm kinda sick of it.

Seibel: Does it matter much to you what language you use?

Fitzpatrick: I'm still not happy with any of them. I don't know what exactly would make me totally happy. I hate that for a given project you have to jump around all the time. I want something that lets me have static types and checks all that stuff at compile time, when I want. Perl gets me pretty close in that it lets me write in any style I want. It doesn't let me do enough static checking at compile time but I can make it blow up pretty hard when I want to a runtime. But it's still not good enough.

I want optional static typing. In Perlbal, there's no reason for half the things to be performant except for the core, copying bytes around. I would like to give the runtime hints in certain parts of the code and declare types. But if I want to be lazy and mock something out, I want to write in that style.

Seibel: So you want types mostly so the compiler can optimize better?

Fitzpatrick: No. I also want it to blow up at compile time to tell me like, "You're doing something stupid." Then sometimes I don't care and I want it to coerce for me at runtime and do whatever. I don't want to be too

Fitzpatrick: I don't know. I see people that are really smart—I would say they're good programmers—but say they only know Java. The way they think about solving things is always within the space they know. They don't think end-to-end as much. I think it's really important to know the whole stack even if you don't operate within the whole stack.

When I was doing stuff on LiveJournal, I was thinking about things from JavaScript to how things were interacting in the kernel. I was reading Linux kernel code about epoll and I was like, "Well, what if we have all these long TCP connections and JavaScript is polling with these open TCP connections that are going to this load balancer?" I was trying to think of how much memory is in each structure here. That's still somewhat high-level, but then we were thinking about things like, we're getting so many interrupts on the Ethernet card—do we switch to this NAPI thing in the kernel where rather than the NIC sending an interrupt on every incoming packet it coalesces them to boundaries that were equivalent to 100 megabits speed even though it was a gigabit NIC. We were collecting numbers to see at what point this made sense and freed up the processor.

We were getting a lot of wins for really low-level stuff. I had somebody recently tell me about something: "Java takes care of that; we don't have to deal with that." I was like, "No, Java can't take care of this because I know what kernel version you're using and the kernel doesn't support it. Your virtual machine may be hiding that from you and giving you some abstraction that makes it look like that's efficient, but it's only efficient when you're running it on this kernel." I get frustrated if people don't understand at least the surface of the whole stack.

In practice, nothing works. There are all these beautiful abstractions that are backed by shit. The implementation of libraries that look like they could be beautiful are shit. And so if you're the one responsible for the cost of buying servers, or reliability—if you're on call for pages—it helps to actually know what's going on under the covers and not trust everyone else's libraries, and code, and interfaces.

I almost don't think I would be a programmer today if I was starting off. It's just too ugly. This is why I'm so excited about things like App Engine. Someone described Google's App Engine as this generation's BASIC. Because this generation, everything is networked. When I was

programming, it was one language, and it was on my own machine, and the deploy was up enter, or RUN enter. Kids today don't want to write something stupid like a "bounce a ball" app on their own machine. They want a web site to interact with.

I still have people mailing me who are like, "Hey, I have this idea—I want to make Wikipedia meets YouTube, meets—" Everyone wants to do a web site where their favorite four web sites aren't quite right and they want to make one that looks kind of like that.

The fact that App Engine gives you one button, "Put this on the Web," and you write in one language, arguably a pretty easy-to-learn one, Python, is perfect. It's a great intro to programming—there are so many layers and layers of bullshit that it gets rid of.

Seibel: How does that fit with your dismay at the Java guys who tell you, "Oh, Java takes care of that for you." Isn't that the same? "Well, App Engine will take care of that for you."

Fitzpatrick: I don't know. Maybe it's because I know what's going on. Actually the JVM isn't that bad. I guess it's when people have blind faith in their abstractions without understanding what's going on.

Seibel: You had a lot of programming experience by the time you got to college and studied computer science. How did that work out?

Fitzpatrick: I skipped a lot of my early C.S. classes, because they were just really boring. I would go and take the tests. Then towards the end they got kind of fun, once you get to the 300- and 400-level classes. But right when it got interesting, I graduated. And they wouldn't let me take the fun grad-level classes, because I wasn't a grad student.

I remember in the compiler class, the final project was we had to take this existing language that we had been playing with and add a whole bunch of features, including one feature of our own choosing as the bonus part of the project. So I chose to implement run-time array bounds checking. Anyway, the professor took our compiled binary and ran his test suite against it, and it failed a couple of his tests. He was like, "Sorry, you get a C because you failed my unit test," When I went to look at it, I was like, "You have off-by-

ones in your test suite." So he gave me the grade back and I got an A, but I never got the bonus points for adding a feature to the language. I was angry at school at that point.

And I remember our database class was taught by someone who, it seemed, had no real-world experience with databases. At this point I'd worked with Oracle, Microsoft Server, and tons of MySQL. So I was asking all these real-world questions I actually wanted answers to—things that were melting right now—they would just give me some textbook answer. I'm like, "No, no. That doesn't work."

Seibel: You graduated in 2002. Do you have any greater appreciation now of what they were trying to teach you?

Fitzpatrick: Half the classes I totally loved, and either I learned something totally new that I wouldn't have learned at the time, or I learned the proper background material and the proper terminology. Prior to that, I knew programming pretty well but I didn't have the vocabulary to describe what it was I was doing. Or I would make up my own terminology for it and people would think I didn't know what I was talking about. Formal C.S. education helped me be able to talk about it.

Seibel: Do you have any regrets about combining running a business with school? Would you rather have just done one or the other?

Fitzpatrick: No, I think that was the best way. I had friends who went to college and just did college, but I knew so much of it already, I would've been bored. I had one friend who also knew a whole bunch of it but he was of this school of thought that he's at college to learn, not for grades, so he was, on the side, studying Arabic and Chinese and Japanese. And all the crazy programming languages. Every week it was like, "I have a new favorite language. This week I'm only going to write in OCaml." So he kept himself busy that way. I kept myself busy and not bored other ways.

Then I had friends who dropped out after their freshman year just to do web stuff. A couple were doing a porn web site or something. They were like, "Oh, we're making all this money." But they just worked a whole bunch; they were always in their basement working. College was awesome

for meeting people and partying. If I *just* did LiveJournal, I would've killed myself stresswise.

Seibel: Are you glad you studied computer science?

Fitzpatrick: I probably could have done without it. I did a lot of things I wouldn't have done normally, so I guess it was good. I wish maybe I would have like done something else as well, maybe stayed another year and double-majored in something totally unrelated. Did linguistics more. I'm kind of sad I left college and I felt I only did half studying because so much of it I already knew. My early C.S. classes I barely attended and it was only towards the end where things just started to get interesting when it was like, "OK, you're done."

Seibel: Did you ever think about going to grad school?

Fitzpatrick: Yeah. It would have been fun, but I was busy.

Seibel: Do you try to keep up with the C.S. literature?

Fitzpatrick: Me and my friends still forward each other papers around, neat papers. I read something the other day about some new technique for resizing Bloom filters at runtime. It was pretty awesome. The papers that come out of the storage conferences, some out of industry and some from academics, about different cool systems—I try to read those. There are different reading groups at Google—systems reading groups or storage reading groups. I'll see something on Reddit or a friend will forward a paper or something like that or link it on a blog.

Seibel: You just mentioned papers from the academy and from industry. Do you have any sense of whether those two meet in the right place these days?

Fitzpatrick: They kind of feel about the same to me. But it's more interesting, a lot of times, to read the industry ones because you know they did it to solve a problem and their solution works as opposed to, "We'd think it would be cool if—" There's a lot of crazier stuff that comes out of academia and it doesn't actually work, so it's just a crazy idea. Maybe they turn it into commercial things later.

Seibel: How do you design software?

Fitzpatrick: I start with interfaces between things. What are the common methods, or the common RPCs, or the common queries. If it's storage, I try to think, what are the common queries? What indexes do we need? How are the data going to be laid out on disk? Then I write dummy mocks for different parts and flesh it out over time.

Seibel: Do you write mocks in the test-first sense so you can test it as you go?

Fitzpatrick: More and more. I always designed software this way, even before testing. I would just design interfaces and storage first, and then work up to an actual implementation later.

Seibel: What form would the design take? Pseudocode? Actual code? Whiteboard scribbles?

Fitzpatrick: Generally I would bring up an editor and just write notes with pseudocode for the schema. After it got good, I would make up a real schema and then I would copy-paste it in just to make sure that "create table" works. Once I got that all going, I'd actually go implement it. I always start with a spec.txt first.

Seibel: After you write a bunch of code do you ever discover that you really need to reconsider your original plan?

Fitzpatrick: Sometimes. But I've started with the hard bits or the parts I was unsure of, and tried to implement those parts first. I try not to put off anything hard or surprising to the end; I enjoy doing the hard things first. The projects that I never finish—my friends give me shit that it's a whole bunch—it's because I did the hard part and I learned what I wanted to learn and I never got around to doing the boring stuff.

Seibel: Do you have any advice for self-taught programmers?

Fitzpatrick: Always try to do something a little harder, that's outside your reach. Read code. I heard this a lot, but it didn't really sink in until later. There were a number of years when I wrote a lot of code and never read anyone else's. Then I get on the Internet and there's all this open source

Seibel: Programs like that, the code base is pretty huge. When you look at something like that for fun, how deeply do you get into it?

Fitzpatrick: Generally, I'll just pipe `find` into `less` and try to understand the directory structure. Then either something grabs my eye or I don't understand what something is. So I pick a random file and get a feel for it. Then I bounce around and wander aimlessly until I'm bored and then pick a new random spot to jump in.

A lot of times, I'll work on building it in parallel with reading it because they're very parallelizable tasks, especially if it's hard to build. By the time it's finally built, then I can start tweaking it if I want to.

Seibel: So when you read good code it either fits into patterns that you already understand, or you'd discover new patterns. But not all code is good. What are the first warning signs of bad code?

Fitzpatrick: Well, I'm particularly snooty now, having worked at Google with really strict style guidelines in all languages. On our top six or seven languages, there's a really strict style guide that says, "This is how we lay out our code. This is how we name variables. This is how we do spacing and indentation, and these patterns and conventions you use, and this is how you declare a static field."

We've started putting these online too, just as a reference for external contributors contributing to our projects. We wanted to have a documented policy so we don't just say, "We don't like your style."

Now when I work on projects in C, the first thing I do is add a style guide. Once a project is mature and has a lot of people hacking on it, they'll have a style guide. It's not even always written, but the programmer just respect the style of code written already. Maybe they don't like the brace style, but fuck it, it's more important to have it consistent within a file, within a project, than to do it your favorite way.

Seibel: Do you ever do any pair programming?

Fitzpatrick: I think it's pretty fun. It's good for lots of things. Sometimes you just need to think and want to be left alone. I don't subscribe to it all the time, but it's definitely fun.

I start too many projects. I finish them because I have guilt if I don't finish them, but I definitely context-switch way too often and I'm spread too thin. This is why I really need pair programming—it forces me to sit down for three solid hours, or even two or one solid hour, and work on one thing with somebody else, and they force me to not be bored. If I hit a bored patch, they're like, "Come on. We've got to do it," and we finish.

I like working alone but I just bounce all over the place when I do. On a plane I'll bring extra laptop batteries and I have a whole development environment with local web servers and I'll be in a web browser, testing stuff. But I'll still be hitting new tabs, and typing "reddit" or "lwn"—sites I read. Autocomplete and hit Enter, and then—error message. I'll do this multiple times within a minute. Holy fuck! Do I do this at work? Am I reading web sites this often that I don't even think about it? It's scary. I had a friend, who had some iptables rule, that on connection to certain IP addresses between certain hours of the day would redirect to a "You should be working," page. I haven't got around to doing that, but I need to do something like it, probably.

Seibel: What about code ownership? Is it important for people to own code individually or is it better for a team to share ownership?

Fitzpatrick: I don't think code should be owned. I don't think anyone really thinks that. The way it works within Google is that it's one massive source tree, one root, and one unified build system across all of it. And so anyone can go and change anything. But there are code reviews, and directories have owners, always at least two people, just in case some quits or is on vacation.

To check in you need three conditions met: You need someone to review it and say it looks good. You need to be certified in the language—basically you've proven you know the style of this language—called "readability." And then you also need the approval above from somebody in the owner's file in that directory. So in the case that you already are an owner of that directory and you have readability in that language, you just need someone

to say, "Yeah, it looks good." And it's a pretty good system, because there tends to be a minimum of two, up to twenty, thirty owners. Once you work on a code base for a while, someone just adds you to owners. I think it's a great system.

Seibel: So let's go back in time a bit—how did LiveJournal start?

Fitzpatrick: It was just fucking around with my friends—what I wanted and what we thought would be funny. Commenting on LiveJournal was a practical joke. I was checking my LiveJournal right before I ran into class. We had just introduced friend pages and I saw something my friend wrote and it was really stupid and I wanted to make fun of him. "Oh, but I can't reply." So I went to class and all throughout class I was thinking, "How can I add a reply system?" I was thinking of the existing schema and how we could render it. I had a two-hour break between classes, so I add commenting and I reply something smartass and sarcastic and go to my other class. When I came back from my second class, and he's like, "What the fuck? We can comment now?"

Everything on LiveJournal was pretty much a joke. The whole security thing, like friends-only posts and private posts, was because a friend wrote that he went to a party and woke up drunk in a ditch the next day. His parents read it and were like, "What? You're drinking?" He was like, "Brad, we need a way to lock this shit down!" I was like, "On it!" We already had friends, so we just made it so some posts are friends only and then your parents—just don't be friends with them.

Seibel: In the early days of LiveJournal it seems your life was an endless series of late nights, sleeping late, and overall working long hours. How much of that is a necessary part of programming?

Fitzpatrick: I just thought it was the least stressful time. During the day, there's always something coming up, like another meal is coming up, or a class, or maybe you get a phone call. There's always some interruption. I can't relax. If I go into work two hours before some meeting, that two hours is less productive than if I didn't have that meeting that day or if the meeting was the first thing in the morning. Knowing that I have nothing coming up to bug me, I'm so much more relaxed.

At night I feel like this is my time and I'm stealing this time because everyone else is sleeping. There's no noise and no interruptions, and I can do whatever. I still stay up late sometimes. I did it this weekend; I was up quite a bit working on different things. But that screws me up for days sleepwise. I did that mostly when I had to in college, because I had some project, and I was also doing LiveJournal on the side. The only time to do it was at night and also all our server maintenance had to be at night. And then in the summer, just because why not? There's no reason to wake up early in the morning to go to a class or anything, so might as well work at night.

Seibel: What about the length and intensity? I'm sure you've done the 80-, 100-, 120-hour weeks. Is that necessary? Under what circumstances is that really necessary and when is it just a macho thing that we do?

Fitzpatrick: In my case, I'm not sure it was either necessary or a macho thing. I was having fun and it was what I wanted to be doing. Sometimes things were breaking, but even when they weren't breaking, I was still doing it just because I was working on a new feature that I really wanted to see happen.

Seibel: Have you ever been in a situation where you really had to estimate how long things were going to take?

Fitzpatrick: Once I got to Six Apart. I guess that was my first experience, three and a half years ago. We had started doing migration—we'd have a customer and they'd say, "Can you move this data?" That requires adding this support for this code and testing, and pushing it out. I was terrible at it. I probably still am terrible at it, because I always forget a factor, like the bullshit multiplier of having to deal with interruptions and the fact that I'm never going to get away from maintaining a dozen projects on the side.

I think I'm getting better, but fortunately they don't ask for that too often. And now when I actually do get a deadline for something, I'm like, "Yay! A deadline!" and I get so excited that the adrenaline kicks in, and I work, and I finish the damn thing. Nothing with Google is really a deadline. With Google it's like, "What do you think about launching this? How does that feel?" It's rare that there's some real deadline. Most of them, we think it'd be nice to

launch on this date and so everyone tries really hard. But you're only letting down other people that want to see it launch by that day if you don't finish something. And most of the things I work on are very "When it's done, it's done."

Seibel: When you were hiring programmers at LiveJournal, did you manage them?

Fitzpatrick: Well, I kind of assumed that none of them would need managing; that they would just all be self-driven like me. That was a learning experience in HR, that some people just do what they're told and don't really have a passion for excellence. They're just like, "Done. Next assignment," Or they don't tell you and just browse the Web. So I had a couple of painful experiences. But I think after a year or two of that, I learned that people are different.

Some are purists. They would just do abstraction on abstraction on abstractions. They would go really slowly and are very religious about their style. They're like, "I'm an artisan programmer." And I was like, "Your code doesn't run. It's not efficient and it doesn't look like any of the other code that you're interacting with."

Seibel: Did you figure out how to make good use of people like that?

Fitzpatrick: One person, I tried dozens of different things. I think he might've been ten years older than me. I don't know how much, because I never ask that—I was afraid of legal hiring questions. But I got the feeling that he didn't want to work for some young punk. I was like 22. That one eventually didn't work out. That was the only person I let go.

Other people I eventually figured out what motivated them. One guy was really good at tinkering and getting a prototype working. He wrote sysadmin Perl. He could wire stuff together, write shell scripts, and write really bad Perl and really bad C, but kind of get it working. Then we would be like, "Holy crap, you researched all this stuff, you got all these components talking to each other?"

We were setting up a voice bridge to LiveJournal so you record something and post it to LiveJournal. There were just so many moving parts involved. I

and I was like, "Holy crap." I don't know why I spent 90 minutes on it; I was so obsessed that I didn't step back and check, is my command line correct?

There's a lot of that. We always had some good stuff with Perl like the $_ isn't lexically scoped. So if you fuck with $_ in a sort, you can mess with somebody else's far away. So we had this bug that took us forever and we had a bunch of corruption going on. We finally figured that out. Then I audited all our code we had a new policy of "never do this."

Seibel: What are your debugging tools? Debuggers? Println? Something else?

Fitzpatrick: Println if I'm in an environment where I can do that. Debugger, if I'm in an environment that has good debuggers. GDB is really well maintained at Google and is kind of irreplaceable when you need it. I try not to need it too often. I'm not that great at it, but I can look around and kind of figure things out generally. If I have to go in there, I generally can find my way out. I love strace. Strace, I don't think I could live without. If I don't know what some program is doing, or what my program is doing, I run it under strace and see exactly what's happening. If I could only have one tool, it would probably be that. All the Valgrind tools, Callgrind and all that, those are good.

But a lot of times lately, if there's something weird going on, I'm like, "OK, that function is too big; let's break that up into smaller parts and unit-test each one of them separately to figure out where my assumptions are wrong, rather than just sticking in random println."

Then maybe in the process of refactoring, I have to think about the code more, and then it becomes obvious. I could, at that point, go back to the big, ugly state where it was one big function and fix it but I'm already halfway there; I might as well continue making it simpler for the next maintainer.

Seibel: How do you use invariants in your code? Some people throw in ad hoc asserts and some people put in invariants at every step so they can prove formal properties of their programs, and there's a big range in the middle.

Fitzpatrick: I don't go all the way to formal. My basic rules is, if it could possibly come from the end user, it's not a run-time crash. But if it is my code to my code, I crash it as hard as possible—fail as early as possible.

I try to think mostly in terms of preconditions, and checking things in the constructor and the beginning of a function. Debug checks, if possible, so it compiles away. There are probably a lot of schools of thought and I'm probably not educated about what the proper way to do it is. There are languages where all this stuff is actually a formal part of the language. Pretty much all the languages I write in, it's up to you.

Seibel: You wrote once that optimization is your favorite part of programming. Is that still true?

Fitzpatrick: Optimization is fun because it's not necessary. If you're doing that, you've got your thing working and nothing else is more important and you're either saving money or doing it because it's like a Perl golf contest—how short can I make this or how much faster. We would identify hotspots in LiveJournal, and I would send out some contests. "Here's some code. Here's the benchmark. Make it fast." I sent our load balancer's header parsing. We were all writing crazy regexps that didn't backtrack and tried to capture things with the most efficient capture groups. And we were all competing, getting faster and faster and faster. Then one guy comes over the next day. He had written it all in C++ with XS, and so he was like, "I win."

Seibel: The flip side of that these days is . . .

Fitzpatrick: Programmers' time is worth more and all that crap? Which can be true. This is true for a small number of machines. Once you get to a lot of machines, all of a sudden the programmers' time is worth less than the number of machines that this will be deployed against, so now write it in C and profile the hell out of it, and fix the compiler, and pay people to work on GCC to make this compile faster.

Seibel: But even Google uses C++ rather than assembly, so there's some point at which trying to squeeze the maximum performance isn't worth it. Or is the theory that a good C++ compiler generates better code than all but the most freakishly rare assembly coders?