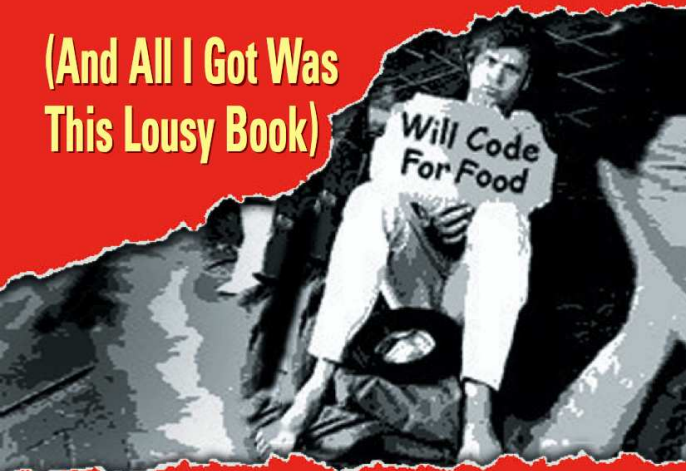


The
Pragmatic
Programmers

MY JOB WENT TO INDIA

(And All I Got Was
This Lousy Book)



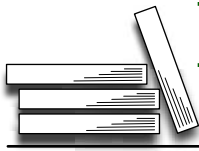
52 Ways To Save Your Job

Chad Fowler

My Job Went to India

And All I Got Was This Lousy Book

Chad Fowler



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking g device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragmaticprogrammer.com>

Copyright © 2005 Chad Fowler.

Allrightsreserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN 0-9766940-1-8

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

First printing, September 2005

Version: 2005-8-29

For Kelly Jeanne

Contents

Introduction	1
Part I—Choosing Your Market	10
1. Supply and Demand	12
2. Coding Don't Cut It Anymore	16
3. Lead or Bleed?	19
4. Invest in Your Intelligence	22
5. Be a Generalist	26
6. Be a Specialist	31
7. Don't Put All Your Eggs in Someone Else's Basket	34
8. Be the Worst	36
9. Love It or Leave It	39
Part II—Investing in Your Product	42
10. Learn to Fish	45
11. Understand Business Basics	48
12. Find a Mentor	50
13. Be a Mentor	54
14. Practice, Practice, Practice	56
15. The Way That You Do It	61
16. On the Shoulders of Giants	64
17. Automate Yourself into a Job	67
Part III—Executing	71
18. Right Now	73
19. Mind Reader	75
20. Daily Hit	78
21. Remember Who You Work For	81
22. Be Where You're At	83
23. How Good a Job Can I Do Today?	86
24. How Much Are You Worth?	89
25. A Pebble in a Bucket of Water	92
26. Learn to Love Maintenance	95

27. Eight-Hour Burn	99
28. Learn How to Fail	102
29. Say “No”	105
30. Say It, Do It, Show It	108
Part IV—Marketing...Not Just for Suits	112
31. Perceptions, Perschmeptions	115
32. Adventure Tour Guide	118
33. Me Rite Reel Nice	121
34. Being Present	123
35. Suit Speak	127
36. Change the World	129
37. Let Your Voice Be Heard	131
38. Build Your Brand	134
39. Release Your Code	136
40. Remarkability	138
41. Making the Hang	140
Part V—Maintaining Your Edge	143
42. Already Obsolete	145
43. You’ve Already Lost Your Job	147
44. Path with No Destination	149
45. Make Yourself a Map	151
46. Watch the Market	153
47. That Fat Man in the Mirror	155
48. The South Indian Monkey Trap	158
Part VI—If You Can’t Beat ‘Em	161
49. Lead ‘Em	163
50. Manage ‘Em	165
51. Learn from Open-Source	168
52. Think Global	170
What I Learned in India	173
Resources	176
A.1 Bibliography	176

It is true India has the advantage in software and China in hardware. If India and China cooperate in the IT industry, we will be able to lead the world...and it will signify the coming of the Asian century of the IT industry.

► Wen Jiabao, Chinese Premier, April 11, 2005

Introduction

I awoke to an odd smell. *Where am I?* I asked (aloud, I think).

I was in Bangalore, India's Garden City. That odd smell was the remarkably foreign combination of pollution, ultraspicy food from the hotel's kitchen, and something else that I could never quite put my finger on. It was my first morning there, and I was late for work. I didn't feel bad about that, considering the hellish 32-hour journey I had suffered to get there. And that my India-savvy co-worker tricked me into an all-night cultural immersion in the world's scariest hotel, after getting off the plane in Bombay the night before.

After coming to, I felt a panic, realizing that my driver must have been waiting downstairs for an *hour*. *Oh God, he'll be angry*, I thought, as I scrambled to get ready for the first in a series of all-day interviewing sessions. *That's just what I need on my first morning in a place like this...an angry taxi driver.*

I rushed downstairs, resisting the fabulous aroma of a South Indian breakfast, and ran out to ask the doorman to page my driver. I asked if the driver had been there long. He had. Two hours. *Ugh*.

I spent the first five minutes in the car with Joseph apologizing profusely over being late and making him wait. He laughed dismissively. *This is my job. I wait all day*. And as I found out later, he really did. He didn't just drop me off at work and come back at a fixed time. He waited at the office until the very minute I was ready to go. Without warning, at any time, I could come down from the office and expect to hop into the car and be driven away.

My first exposure to India in daylight was that drive across town from the northwestern corner of Bangalore to the southwestern corner. The culture shock started to hit me.

Bangalore is known as India's Silicon Valley. Being from a small city back home, it was exciting to realize that I had come to a technical mecca.

More surprising, though, were the extreme contrasts between high tech and low tech. I saw half-naked children playing in the dirt in front of a huge Yahoo! sign. I saw a rickshaw with a Novell advertisement on the back and another with what looked like a Sun Solaris CD dangling from the rearview mirror as an ornament.

We drove past beautiful, state-of-the-art office buildings, filled with the employees of some of the Western world's most innovative companies. We dodged buffalo in the street and begrudgingly yielded to rickety bicycles and full families on single mopeds.

We passed by fields containing huts made from twigs, mud, tarps, and assorted garbage. We drove through crowds of well-dressed young people, drinking coffee outside their office buildings before the start of the day, only to drive a little farther to be propositioned by lepers begging at a traffic light.

So before I even reached the office on my first day, my perspective had changed. This was a world of great extremes. These foreign voices I had heard through scratchy, unreliable phone connections, attached to the brains whose code I'd been ruthlessly reviewing, lived *here*? *These* are the people who are allegedly stealing our jobs?

I had come to India in the first stage of the setting up of a new software development center for my company. My job was to interview and select about 25 people who would form the "seed team" of a development shop that would eventually house 250 people. More precisely, my job was to *reject* more than 200 people. We had advertised our open jobs and received nearly 30,000 applications. That's four zeroes. You are reading it correctly. We hired outside firms to help us whittle the 30,000 down to a more manageable number and then used our own U.S.-based employees to further work that number down to a short list of a couple hundred that we could interview in person.

I was to be our interview panel's executioner, sniffing out the weak and finishing them off quickly and (I hoped) painlessly. While in India, I visited the hotel conference rooms of three different cities and met hundreds of people. I probably took a secret pleasure from the fact that I was going to go over and *stop* all these people from getting through the system and "stealing" our jobs.

It was post-boom time. By that, I mean the DotCom bubble had burst. The IT sector's lifestyle had gone from rock 'n' roll to Holiday Inn lounge act, and it was showing in India as well.

In fact, what I found was not an army of people, plotting to steal our comforts for themselves. Unlike their counterparts in the West, these people weren't angry that they had to get a small television set or even that they might not be able to afford this month's cable TV bill. These were sons and daughters who were scraping by, trying to raise money to support their parents and their spouses' parents. These were mothers and fathers whose IT jobs meant the difference between *really* educating their children or sending them to a school from which the further educational options have a *hard limit*. They weren't trying to steal the American dream. They were trying to squeeze a once-dry economy for a few drops of life-giving cash flow.

Ultimately, I was an executioner very much fit for the task. No physical injuries resulted, but many interviewees left with bruised egos. What I left with was a changed perspective. Things had changed. A vibrant society of *highly motivated* and intelligent people existed here. And they weren't playing for amenities; they were competing for the survival of their families.

You can't underestimate—or blame—someone with that kind of motivation.

Things Ain't What They Used to Be

According to the U.S. government, IT unemployment has doubled since 2000. The Bureau of Labor Statistics reports that between 2000 and 2004, the number of programmers in the American IT industry dropped by 17%. In just the first three months of 2005, U.S. technology companies cut 60,000 jobs—twice the number cut in the same period of the previous year.¹ The numbers are sobering. In this world in which every device seems to contain a computer, could software development be a doomed profession?

Matters are made more confusing by the bipolar temperament of the IT job market. Had you left and went on retreat in a cave in 2000 for several months, you would have emerged into an IT employment landscape that was as unrecognizable as Java to a COBOL programmer. In the mid- to late-nineties, a gold rush took place in the IT industry. I remember reading about employers giving BMWs as signing bonuses. A team from another company actually auctioned itself off on eBay for a huge signing bonus.

¹“Challenger Tech Sector Job Cuts Report,” <http://challengergray.com/>

IT employment was a seller's market, and people were jumping ship from other vocations in droves.

The market was suddenly being flooded with new talent (or, at least, people who considered themselves talented). At the time, demand was still outpacing supply. I saw Java programmers being shipped in from India who seemed to have passed the time during their flights reading their first Java manual. The flood surely gave passage to some great software people. But it also introduced a large population of people who wouldn't have ever considered a career in software—and probably shouldn't have.

We're all painfully aware that the boom has ended. When the bubble burst, it was as if an unruly bunch of children had been interrupted jumping on the bed and suddenly realized how much of a mess they had made of their room. It was a mess they now had to clean up. Our industry was filled with piles of unneeded software, hollow business models, and increasingly irrelevant people.

The turn of the century saw IT being demoted from knight to squire. Organizationally, CIOs had bubbled to the top, often reporting to their companies' CEOs. They were now being reorg'd back down under the COOs and CFOs where they started. And with this demotion came the budget cuts.

Where IT departments were previously under pressure to scoop up the best talent before the competition did, they were now under pressure to shed the excess baggage they had collected. In many cases, the reduction in force *didn't* come with a reduction in workload. Bubble or not, the technology boom made our businesses more reliant on IT than ever before. Business processes from the sales floor to the call center were now resting on the backs of IT's systems.

So, here we were with way too much work to do and way too few jobs to support all the work. What's a poor CIO to do?

"Offshoring." This silly-sounding made-up word now strikes fear into the hearts of IT professionals throughout the Western world. Too much work mixed with budget reductions leaves little choice for the nation's CIOs. Programmers in India can be hired for as low as a tenth of the salary of a programmer in the United States. And without a standardized, objective way to compare and contrast the talent difference, that's a bargain difficult for a smart business person to turn down. Even with the time zone and cultural differences, it's hard for a finance manager to imagine *not* saving real money with the right offshoring setup.

So, jobs have been shipped overseas by the boatload. Many American programmers have found themselves either unemployed or supporting the skeleton crew as one of the last of a dying breed. Early-morning or late-night teleconferences through fuzzy telephone connections with people who “talk funny” are becoming a common occurrence in the software development world.

And it looks like the burst of the bubble didn’t make a temporary trough. This is the new IT landscape. Over the years since the boom, offshoring has been growing at a steady rate. In 2004, IT outsourcing grew by 37%.² And according to Gartner, a research and advisory firm, worldwide off-shore spending on application development will more than double, reaching \$50 billion dollars by 2010.

It’s not just grunt work that’s going, either. While we’re already spending \$1.2 billion on R&D outsourcing, that number is expected to shoot up to \$12 billion by 2010.³

The IT offshoring boom has been historically associated with India. India started with a marked advantage over many other low-cost countries, largely because of its excellent educational institutions and, more important, the prominence of English as a first or second language. But even for India, competition is heating up. More and more business is being shipped to Eastern Europe (where it’s easier to find multilingual employees to support European language-speaking nations), Russia, Malaysia, and the Philippines, to name just a few.

Most recently, China has begun to figure into the equation. You know, China. They’re the ones who manufactured almost everything in your house. Go to Wal-Mart, and try to buy a clock or a phone that wasn’t made in China. It’s a real challenge. And now, they’ve got some forward thinking Indians wondering how long the “offshoring bubble” has left in India. Leading management consultancy McKinsey & Company reports that although it will be some time before China could eclipse India in IT offshoring, progress is being made. Chinese offshoring revenues are increasing by 42% each year on average, and the number of English-speaking college graduates in the Chinese workforce has more than doubled since 2000.⁴

²<http://management.silicon.com/itdirector/0,39024673,39127146,00.htm>

³<http://informationweek.com/story/showArticle.jhtml?articleID=160400498>

⁴http://www.mckinseyquarterly.com/article_page.aspx?ar=1556&L2=4&L3=115&srid=21&gp=1

The bottom line is that things have changed for us professional software developers here in the Western world. All signs indicate that the change is not temporary. We can expect our DotCom bubble glory days to become a more and more distant memory as the world continues to turn to lower-cost sources of software development labor.

It's All Our Fault

It's easy to demonize Big Money America or criticize the government for not protecting us. Or, for the truly adventurous of imagination, it's easy to believe that Indians have developed some sinister plot to maliciously rob us of our comforts. However, even if there is an ounce of truth in these sentiments, it is outweighed by the pounds of mediocrity under which our Western industry has languished for the last several years.

It's understandable that forlorn programmers would dump their personal tragedies at the feet of anonymous companies and governing bodies. It's somehow comforting to drown one's fear or despair in a healthy helping of anger and strategically directed blame. And to make matters worse, media sensationalists such as Lou Dobbs prey on our fears, hyping up the problem and sounding a rallying cry whose primary purpose is to get better ratings. But ultimately, blaming corporations is a dead-end road. We can't change corporate America. And though we have democracy on our side, none of us can single-handedly steer this massive ship of a country.

So though comforting in times of fear and uncertainty, this game of blame-the-big-guy is fruitless. We have no one to blame but ourselves.

This self-blaming attitude isn't defeatist, though. In fact, blaming the government is the defeatist choice here. Forming labor unions and picketing would be defeatist. Sitting on the couch flipping news channels and cursing in a fit of nationalist rage would be defeatist. All these courses of action lay the blame—and the imperative for action—at someone else's feet.

If we can calm down enough to look at the situation rationally, we see that it is our own fault that we're in this mess. We live and work in an economic ecosystem. In ecosystems, it's the strong who survive. During a period of staggering success, we've allowed ourselves to get fat, lazy, and slow. The state of our craft has been marred by years of mediocrity.

The fact that we can take responsibility for and see the path that led to our predicament is a good thing. It means we can start to take (and own) corrective action.

It's Up to Us

It's time to face the music. We are where we are, and waiting for things to change by themselves isn't going to lead us anywhere different. The trends aren't reversing, and the government has no incentive to bail us out. The good news is that each of us has the power to do something about it individually. We can each take control of our own piece of the situation, bringing sanity to the collective whole.

Of course, if you can't stand the heat, the most obvious action is to get out of the proverbial kitchen. Western IT has its share of post-boom dead-weight still lingering around, nervously drawing a paycheck. For some not-insignificant percentage of IT workers, the safest bet is to start looking for an alternate line of work. Choosing when you leave and where you go next is a lot less difficult than being thrown out. If you don't have passion and a drive that would force you to create software whether you were being paid for it or not, you're not going to be able to continue to compete with those who do.

For those who remain, here is the key to survival: Software is a business. We're going to have to be businesspeople. Our companies don't employ us because they love us. They never have, and they never will. That's not the job of a business. Businesses don't exist so we can have a place to go every day. The purpose of a business is to make money. To stay employed, you're going to have to understand how you fit into the business's plan to make money.

As we'll explore later, keeping you employed costs your company a significant amount of money. Your company is *investing* in you. Your challenge is to become an obviously good investment. If the *business value* you bring is clear, you are far less likely to end up on the offshoring chopping block.

Think of your career as if it is the life cycle of a product that you are creating. That product is made up of you and your skills. In this book, we'll look at four facets that a business must focus on when designing, manufacturing, and selling a product. And we'll see how these four facets can be applied to our careers:

1. **Choose your market.** Pick the technologies and business domains you focus on consciously and deliberately. How do you balance risk and reward? How do supply and demand factor into the decision?
2. **Invest in your product.** Your knowledge and skills are the cornerstone of your product. Properly investing in them is a critical part

of making yourself marketable. Simply knowing how to program in Visual Basic isn't good enough anymore. What other skills might you need in the new economy? How can you compete with both your offshore and onshore rivals?

3. **Execute.** Simply having employees with a strong set of skills doesn't pay off for a company. The employees have to *deliver*. How do you keep up the delivery pace without driving yourself into the dirt? How do you know you're delivering the *right* value for the company?
4. **Market!** The best product in history won't get purchased if nobody knows it exists. How do you get find recognition in both your company and the industry as a whole without "sucking up"?

The goal of this book is to give you a systematic way of approaching the challenges that lie ahead of you in the new world of IT. We will walk through specific examples and present a set of actions that you can take *right now* that will have both short-term and long-term positive effects.

Ultimately, the goal is not to bring our jobs back. These low-value jobs we've lost were *meant* to be sent offshore. Instead, we should be preparing for the new wave of higher value jobs that will be created in their places.

Acknowledgments

I would have never written a book if not for Dave Thomas and Andy Hunt. *The Pragmatic Programmer* [HT00] served as a catalyst for me, and I've been inspired by their work ever since. Without Dave's encouragement and guidance, I would have never believed I was qualified to write this.

Juliet Thomas served as an editor early in the process of writing this book. Her enthusiasm and perspective were invaluable. I received an amazing amount of feedback from first-draft reviewers: Carey Boaz, Karl Brophey, Brandon Campbell, Vik Chadha, Mauro Cicio, Mark Donoghue, Pat Eyler, Ben Goodwin, Jacob Harris, Adam Keys, Steve Morris, Bill Nall, Wesley Reiz, Avik Sengupta, Kent Spillner, Sandesh Tattitali, Craig Utley, Greg Vaughn, and Peter W. A. Wood. They truly made the book better, and I can't thank them enough for their time, energy, and insight.

The ideas in this book were inspired by the many great people I've had the opportunity to work with, both officially and unofficially, over the years. For listening, teaching, and talking, thanks to Donnie Webb, Ken Smith,

Walter Hoehn, James McMurry, Carey Boaz, David Alan Black, Avi Bryant, Rich Kilmer, Steve Akers, Ali Sareea, and Jim Weirich.

Thanks to the all extended family that adopted us in India, and especially to Suman Nag, Ramesh R., Bharath Kalyanram, Sheela Singh and Singh Ji, Rupali Wadhi, Fayaz Uddeen, A.K. Sreekanth, Brenda D'Souza, Vishal Kapoor, and Deepa Rajamani.

Thanks to my parents for their constant support. And most importantly, thanks go to my wife, Kelly, for making this all worthwhile.

Part I

Choosing Your Market

You're about to make a *big* investment. It may not be a lot of money, but it's your time—your life. Many of us just float down the stream of our careers, letting the current take us where it may. We just happen to get into Java or Visual Basic, and then our employers finally spring for a training class on one of the latest industry buzzwords. So, we float down that path for a while until something else is handed to us. Our career is one big series of undirected coincidences.

In *The Pragmatic Programmer* (HT00), Dave Thomas and Andy Hunt talk about *programming by coincidence*. Most programmers can relate to the idea: you start working on something, add a little code here, and add a little more there. Maybe you start with an example program that you copy and paste from a website. It seems to work, so you change it a little to be more like the program you really need. You don't really understand what you're doing, but you keep nudging the program around until it almost completely meets your needs. The trouble is, you don't understand how it works, and like a house of cards, each new feature you add increases the likelihood your program will fall apart.

As a software developer, it's pretty obvious that programming by coincidence is a bad thing. Yet so many of us allow important career choices to be, in effect, coincidences. Which technologies should we invest in? Which domain should we develop expertise in? Should we go broad or deep with our knowledge? These are questions we really should be asking ourselves.

Imagine you've started a company and you're developing what is destined to be the company's flagship product. Without a "hit" with this product, your company is going to go bankrupt. How much attention do you pay to who your target customers are? Before actually manufacturing the product, how much thought do you put into what the product actually *is*? None of us would let decisions like these be made for us. We'd be completely attentive to every detail of the decision-making process.

So, why is it that most of us don't pay this kind of attention to the choices we make in our careers? If you think of your career as a business (which it *is*), your "product" is made up of the services you have to offer. What are those services? Who are you going to sell them to? Is demand for your services going to grow or decline over the coming years? How big of a gamble are you willing to take on these choices?

This part will help you answer these important questions for yourself.

1

Supply and Demand

When the Web started to really take off, you could make a lot of money creating simple HTML pages for companies. Every company wanted a web page, and relatively few people knew how to make them. Companies were willing to pay top dollar for “experienced” web designers, which back then, meant that they knew the basics of HTML, hyperlinking, and how to structure a site.

Making HTML pages is pretty simple. It’s hard to make really nice-looking pages, but the basics are easy to grasp. As people observed the prices these web designers were demanding, more and more people started picking up books on HTML and teaching themselves. The market was hot, the salaries or hourly fees were attractive, and the supply of HTML experts started to rise as a response.

As the market flooded with web designers, the web people started to stratify between the truly artistic and the utilitarian. Furthermore, competition started to drive the prices down. As a result of lower prices, more companies were willing to take their first step into an internet presence. They might not have paid \$5,000 for their first website, but they would pay \$500.

Of course, some companies were still willing to give up the big bucks for a *fantastic* website. And, certain web designers could still command *fantastic* compensation.

Eventually, the web designer flood at the low-to-middle cost tiers receded. Less talented web designers were replaced by end users and other IT folk who didn’t necessarily specialize in HTML design. At this point, the supply, demand, and price of HTML creation reached an equilibrium.

This armchair history of the vocation of web design demonstrates an economic model that we’ve all heard of, called *supply and demand*. When most of us think of supply and demand, we think that it has to do largely with what price something can and will be sold at. If there are more of an item for sale than the number of people who want to buy that item, then the price of the item will decrease. If there are more people who want the item than there are items available to be purchased, the price of the item will increase as potential buyers compete.

In addition to predicting the prices of goods and services, the supply and demand model can predict how price changes will affect the number of

people willing to sell and purchase a product or service. There are usually more buyers for any given thing at a lower price than at a higher one.

Why is this important to us? The offshore software trend has just injected a large supply of low cost IT people into our economy. Though we're worried about losing jobs domestically, the lower cost per programmer has actually *increased* overall demand. At the same time, as demand increases, price decreases. Competition in high-demand products and services hinges on price. In the employment market, that means salary. You can't compete on price. You can't afford it. So, what do you do?

You can't compete on price. In fact, you can't afford to compete on price.

The offshore market has injected its low-cost programmers into a relatively narrow set of technologies. Java and .NET programmers are a dime a dozen in India. India has a lot of Oracle DBAs as well. Less mainstream technologies are very much underrepresented by the offshore development shops. When choosing a technology set to focus your career on, you should understand the effects of increased supply and lower prices on your career prospects.

As a .NET programmer, you may find yourself competing with tens of thousands of more people in the job market than you would if you were, for example, a Python programmer. This would result in the average cost of a .NET programmer decreasing significantly, possibly driving demand higher (i.e., creating more .NET jobs). So, you'd be likely to find jobs available, but the jobs wouldn't pay all that well. The supply of Python programmers might be much smaller than that of .NET programmers with a demand to match.

If the Python job market were to support noticeably higher prices per-programmer, additional people might be attracted to *supply* their services at this higher price range, resulting in competition that would drive the price back down.

The whole thing is a balancing act. But, one thing seems certain (for now). India caters to the already balanced IT services markets. You don't find mainstream Indian offshoring companies jumping on unconventional technologies. They aren't first-movers. They generally don't take chances. They wait for technology services markets to balance, and they disrupt those markets with significantly lower per-programmer costs.

Based on this observation, you might choose to compete in segments of the job market in which there is actually *lower* demand. As unintuitive as that may sound, if you're worried about losing employment to offshoring, one strategy would be to avoid the types of work that offshore companies are doing. Offshore companies are doing work that is in high demand. So, focusing on niche technologies is a strategy that, while not necessarily making the competition less fierce—there are fewer jobs to go around—might change the focus of competition from price to ability. That's what you need. You can't compete on price, but you *can* compete on ability.

Also, with the *average* price of these mainstream programmers decreasing, the demand will increase. An overall increase in demand for Java programmers, for example, might actually result in *more* jobs (of a certain type) at home—not fewer. An increase in the lower-priced offshore market could drive overall demand, including a higher bracket of developers.

This happens in practice. To make offshoring work well, many companies realize the need for a reserve of high-end, onshore developers who can set standards, ensure quality, and provide technical leadership. An increase in overall Java programming demand would naturally lead to an increase in this category of Java work. The low-end jobs might be going offshore, but there are more of the elite jobs to go around than there were pre-offshoring. As we saw in the niche job markets, in this tier of Java development work, the competition would shift from price to ability.

Exploit market imbalances.

The most important lesson we can learn from the supply and demand model is that with increased demand comes increased price competition. The tried-and-true, follow-the-jobs strategy will put you squarely in price competition with offshore developers as your skills fit into the offshore-friendly balanced markets. To compete in the mainstream technology market, you'll have to compete at a higher tier. Alternatively, one could exploit market *imbalances*—going where the offshore companies won't go. In either case, it pays to understand the forces at work and to be skilled and nimble enough to react to them.

Act on it!

1. Research current technical skill demand. Use job posting and career websites to find out which skills are in high demand and in low demand. Find the websites of some offshore outsourcing companies (or talk to employees of those companies if you work with them).

Compare the skills available via these companies with the high-demand list you compiled. Make note of which skills appear to be in high demand domestically with little penetration offshore.

Do a similar comparison between leading-edge technologies and the skills available via offshore outsourcing firms. Keep your eyes on both sets of technical skills that are underserved by the offshore companies. How long does it take for them to fill the holes (if ever)? This time gap is the window during which a market imbalance exists.

2

Coding Don't Cut It Anymore

It's not enough to think about what technologies you're going to invest in. After all, the technology part is a commodity, right? You're not going to be able to sit back and simply master a programming language or an operating system, letting the businesspeople take care of the business stuff. If all they needed was a code robot, it would be easy to hire someone in another country to do that kind of work. If you want to stay relevant, you're going to have to dive into the domain of the business you're in.

In fact, a software person should not only understand a business domain well enough to develop software for it but also become one of its authorities. At a previous company, I saw an excellent example of this. There was a database administration team consisted of people who really weren't interested in database technology. When I was first exposed to them, it was a bit of a shock. *Why are these people in Information Technology?*, I thought. In terms of technical skill, they just weren't very strong. But, this team had something special. Being the keepers and protectors of our enterprise data, they actually knew the business domain better than almost any business analyst we had. Their knowledge and understanding of the business made them hot commodities in the internal job posting market. While us geeks were looking at them disdainfully, the *business* for which they worked recognized a ton of value in them.

You should think of your business domain experience as an important part of your repertoire. If you're a musician, when you add something to your repertoire, it doesn't just mean you've played the song once. It means you truly *know* the song. You should apply the same theory to your business domain experience. For example, having worked on a project in the health insurance industry doesn't guarantee that you understand the difference between a HIPAA 835 and a HIPAA 837 EDI transaction. It's this kind of knowledge that differentiates two otherwise equivalent software developers in the right situation.

You might be "just a programmer," but being able to speak to your business clients in the language of their business domain is a critical skill. Imagine how much easier life would be if everyone you had to work with really understood how software development works. You wouldn't have to explain to them why it's a bad idea to return 30,000 records in a single page on a web application or why they shouldn't pass out links to your

development server. This is how your business clients feel about you: *Imagine how much easier it would be to work with these programmers if they just understood what I was asking them for without me having to dumb everything down and be so ridiculously specific!* And, guess what? It's the business that pays your salary.

Just like technologies that become hot, business domains can be selected in the same way. Java and .NET are the Big Things right now in software development. If you learn them, you can compete for a job in one of the many companies that will employ these technologies. The same is true of business domains. You should put the same level of care into selecting which industry to serve as you put into selecting which technologies to master.

In light of the importance that you should place on selecting a business domain when rounding out your portfolio, the company and industry you choose to work for becomes a significant investment on your part. If you haven't yet given real, intentional thought to which business domains you should be investing in, now is the time. Each passing day is a missed opportunity. Like leaving your savings in a low-yield savings account when higher interest rates are to be had, leaving your development on the business front in stasis is a bad investment choice.

Now is the time to think about business domains you invest your time in.

Act on it!

1. Schedule lunch with a businessperson. Talk to them about how they do their job. As you talk to them, ask yourself what you would have to change or learn if you aspired to have their job. Ask about the specifics of their daily work. Talk to them about how technology helps them (or slows them down) on the job. Think about *your* work from their perspective.

Do this regularly.

2. Pick up a trade magazine for your company's industry. You probably don't even have to buy one. Most companies have back issues of trade rags lying around somewhere. Start trying to work your way through a magazine. You may not understand everything you read, but be persistent. Make lists of questions you can ask your management or business clients. Even if your questions seem stupid to you, your business clients will appreciate that you are trying to learn.

Look for industry websites that you can monitor on a regular basis. In both the websites and the magazines, pay special attention to what

the big news items and the feature articles are about. What is your industry struggling with? What's the hot new issue right now? Whatever it is, bring it up with your business clients. Ask them to explain it and to give you their opinions. Think about how these current trends affect *your* company, your division, your team, and eventually your work.

3

Lead or Bleed?

If you're going to invest your money, a lot of options are available to you. You could put it in a savings account, but the interest it accrues probably wouldn't keep up with the pace of inflation. You could put it in government savings bonds. Again, you don't make much money as a result, but they're a safe bet.

Or, you could invest your money in a small startup company. You may, for example, put in several thousand dollars in exchange for a small portion of ownership in the company. If the company's idea is good and it's able to execute effectively on that idea, you could potentially make a *lot* of money. On the other hand, you have no guarantee that you'll even recoup your original investment.

This concept is nothing new. You start to learn it as a child playing games. *If I run straight down the middle, it might surprise everyone, and nobody will tag me.* You are reminded of it constantly throughout daily life. You make the risk-reward trade-off when you're late for a meeting and trying to decide on the right route to work. *If traffic isn't bad, I can get there 15 minutes quicker if I drive down 32nd Street. If traffic is bad, I'm toast.*

The risk-reward trade-off is an important part of making intentional choices about which technologies and domains to invest in. Ten years ago, a very low-risk choice would have been to learn how to program in COBOL. Of course, there were also so many COBOL programmers to compete with that the average salary of a COBOL programmer at the time was not phenomenal. You could easily have found work, but the work wouldn't have been especially lucrative. Low risk. Low reward.

On the other hand, if at the same time you had chosen to investigate the new Java language from Sun Microsystems, it might have been difficult to find employment at a company that was actually doing anything with Java for a while. Who knew if anyone would *eventually* do anything with Java? But, if you were looking at the state of the industry at that time, as Sun was, you may have seen something special in Java. You may have had a strong feeling that it was going to be big. Investing in it early would make you a leader in a big, upcoming technology trend.

Of course, in that instance, you would have been correct. And, if you played your cards right, your personal investment in Java may have been a very lucrative one. High risk. High reward.

Now imagine that, also in 1995, you saw a demonstration of the new BeOS from Be. It was incredible at the time. It was built from the ground up to take advantage of multiple processors. The multimedia capabilities were simply astounding. The platform created a definite buzz, and the pundits were giddy in anticipation of a solid new contender on the operating system block. With the new platform, of course, came new ways of programming, new APIs, and new user interface concepts. It was a lot to learn, but it may have really seemed worth it. You could have poured a lot of effort into becoming the first person to create, for example, an FTP client or a personal information manager for the BeOS. As Be released an Intel-compatible version of its OS, rumors circulated about Apple buying the company out to use its technology as the foundation for the next generation of the Macintosh OS.

Apple didn't buy Be. And, eventually, it became clear that Be wasn't going to capture even a niche market. The product just didn't stick. Many developers who had mastered programming for the BeOS environment became slowly and painfully aware that their investment wasn't going to pay off in the long-term. Eventually, Be was purchased by Palm, and the OS was discontinued. BeOS was a risky but attractive technology investment that didn't yield concrete long term returns for the developers who chose to invest in it. High risk. No reward.

So far, what I've been talking about is the difference between choosing technologies that are still on the bleeding edge and technologies that are firmly entrenched. Picking a stable technology that has already wedged itself into the production systems of businesses worldwide is a safer, but potentially less rewarding, choice than picking a flashy new technology that nobody has deployed yet. But, what about the technologies that have run their course? The ones that are just waiting for the last few nails to be driven into their coffins?

Who drives those nails? You might think of the last few RPG programmers, for example, as being gray-haired and counting the hours until retirement, while the new generation of youngsters hasn't even *heard of* RPG. They're all learning Java and .NET. It's easy to imagine that the careers of the last remaining stalwarts of an aged and dying technology are in the same death spiral as the technology itself.

But, the old systems don't just die. They are replaced. Furthermore, in most cases, homegrown systems are replaced in stages. In those stages, the old systems have to talk to the new systems. Someone has to know how to make the new speak to the old, and vice versa. Typically, the young tykes

don't know (or *want* to know) how to make the old systems listen. Nor do the crusty old pre-retirees know how to make the newfangled systems talk to their beloved creatures.

So, there's a role to be filled by a calculating technologist: *technology hospice*. Helping the old systems die comfortably and with dignity is a task that should not be underestimated. And, of course, most people will jump ship before it sinks, either via retirement or by sidestepping into another technology realm. Being the last one left to support still-critical systems, you can pretty much call the shots. It's risky, in that once the technology *really is* gone, you'll be an expert in something that doesn't exist. However, if you can move fast enough, you can look for the next dying generation of legacy systems and start again.

The adoption curve has edges at either end. How far out on the edges do you want to be?

Act on it!

1. Make a list of early, middle, and late adoption technologies based on today's market. Map them out on paper from left to right; the left is bleeding edge, and the right is filled by technologies that are in their sunsets. Push yourself to find as many technologies in each part of the spectrum as possible. Be as granular as possible about where in the curve they fall in relation to one another.

When you have as many technologies mapped out as you can think of, mark the ones that you consider yourself strong in. Then, perhaps in a different color, mark the ones that you have some experience with but aren't authoritative on. Where are most of your marks on the adoption curve? Do they clump? Are they spread evenly across? Are there any technologies around the far edges that you have some special interest in?

4

Invest in Your Intelligence

When choosing what to focus on, it can be tempting to simply look at the technologies that yield the most jobs and focus on those. Java is big. .NET is big. Learning Java has a simple, transitive effect: if I know Java, I can apply for, and possibly get, a job writing Java code.

Using this logic, it would be foolish to choose to invest in a niche technology, especially if you had no plans to try to exploit that niche.

TIOBE Software uses Internet search engines to indicate the relative popularity of programming languages, based on people talking about those languages on the Internet. According to TIOBE's website, "The ratings are based on the worldwide availability of skilled engineers, courses, and third-party vendors." It's definitely not a scientifically provable measure of popularity, but it's a pretty good indicator.

At the time of writing, the most popular language is C, followed by Java. C# is in a respectable ninth place, but with a slight downward trajectory. SAP's ABAP is in 16th place and is making strong progress upward. Ruby, my personal favorite programming language—the one I do pretty much all of my *serious* work in, and the one for which I co-organize an international conference every year—is not even in the top twenty.

Am I crazy to use Ruby or just stupid? I must be one of the two, right?

In his essay "Great Hackers",⁵ Paul Graham annoyed the industry with the assertion that Java programmers aren't as smart as Python programmers. He made a lot of stupid Java programmers mad (did I say that?), causing a lot of them to write counterarguments on their websites. The violent reaction indicates that he touched a nerve. I was in the audience when his essay was first presented, in the form of a speech. For me, it sparked a flashback.

When I was in India weeding through hundreds of candidates for only tens of jobs, the interview team was exhausting itself and running out of time because of a poor interview-to-hire hit rate. Heads hurting and eyes red, we held a late-night meeting to discuss a strategic change in the way we would go through the candidates. We had to either optimize the process so we could interview more people or somehow interview *better*

⁵<http://paulgraham.com/gh.html>

people (or both). With what little was left of my voice after twelve straight hours of trying to drag answers out of dumbstruck programmers, I argued for adding Smalltalk to the list of keywords our headhunters were using to search their résumé database. *But, nobody knows Smalltalk in India*, cried the human resources director. That was my point. Nobody knew it, and programming in Smalltalk was a fundamentally different experience than programming in Java. The varying experience would give candidates a different level of expectations, and the dynamic nature of the Smalltalk environment would reshape the way a Java programmer would approach a problem. My hope was that these factors would encourage a level of technical maturity that I hadn't been seeing from the candidates I'd met so far.

The addition of Smalltalk to the requirements list yielded a candidate pool that was tiny in contrast to our previous list. These people were diamonds in the rough. They really understood object-oriented programming. They were aware that Java isn't the idealistic panacea it's sometimes made out to be. Many of them *loved* to program! *Where have you been for the past two weeks?* we thought.

Unfortunately, our ability to attract these developers for the salaries we were able to pay was limited. They were calling the shots, and most of them chose to stay where they were or to keep looking for a new job. Though we failed to recruit many of them, we learned a valuable recruiting lesson: we were more likely to extend offers to candidates with diverse (and even unorthodox) experience than to those whose experiences were homogenous. My explanation is that either the good people seek out diversity, because they love to learn new things, or being forced into alien experiences and environments created more mature, well-rounded software developers. I suspect it's a little of both, but regardless of *why* it works, we learned that it works. I still use this technique when looking for developers.

So, other than trying to show up on *my* radar screen when I'm looking to hire someone, why else would you want to invest in fringe technologies that you may rarely or never have an opportunity to actually get paid to use?

For me, as a hiring manager, the first reason is that it shows that you're interested. If I *know* you learned something for the sake of self-development and (better) pure fun, I know you are excited and motivated about your profession. When I first went to India, it drove me crazy to ask people if they'd seen or used certain not-quite-mainstream technologies

only to hear, “I haven’t been given the opportunity to work on that” in return. *Given* the opportunity?! *Neither was I!* I thought. I *took* the opportunity to learn.

I haven’t been given the opportunity...? Seize the opportunity!

After having lived in India for a while, I have developed an armchair theory to explain why “never given the opportunity” came up so often. The citizens of this so-called low-cost country were only one or two generations from having been ruled by Great Britain. Their parents or grandparents had probably experienced extremes of poverty that I couldn’t have imagined before venturing out of the United States and Europe. I was talking to the first- or second-generation upper middle class, and priority number one was making sure they and their extended families *remained* upper middle class.

Whereas you and I here in the West may have the luxury of making job choices based on what excites us, the people of India are still a generation or two away from the same financial freedoms we enjoy. I was disappointed with their “never given the opportunity” response, but many of these people don’t even have computers at home. Software developers without their own computers! Perhaps they really *haven’t* been afforded the opportunity to learn some of these skills that I might take for granted. Hint: this is one of our key advantages—we have leisure time with which we can *choose* to invest in ourselves, creating a greater depth than the majority of these people could hope for. While you’re sitting on your couch, choosing between an episode of *Dharma and Greg*, a round of Xbox, or a little self-study, our Indian competitors—the captors of our jobs—are trying to work themselves up the management ladder so they can finally buy the house in which they will live with their parents and spouse’s parents.

More important than portraying the perception of being suitably motivated and engaged by your field is that exposure to these fringe technologies and methodologies actually makes you deeper, better, smarter, and more creative.

If that’s not good enough reason, you’re probably in the wrong profession.

Act on it!

1. Learn a new programming language. But, don't go from Java to C# or from C to C++. Learn a new language that makes you think in a new way. If you're a Java or C# programmer, try learning a language like Smalltalk or Ruby that doesn't employ strong, static typing. Or, if you've been doing object-oriented programming for a long time, try a functional language like Haskell or Scheme. You don't *have* to become an expert. Work through enough code that you truly feel the difference in the new programming environment. If it doesn't feel strange enough, either you've picked the wrong language or you're applying your old way of thinking to the new language. Go out of your way to learn the idioms of the new language. Ask old-timers to review your code and make suggestions that would make it more idiomatically correct.

5

Be a Generalist

For at least a couple of decades, desperate managers and business owners have been pretending that software development is a manufacturing process at heart. Requirements specifications are created, and architects turn these specifications into a high-level technical vision. Designers fill out the architecture with detailed design documentation, which is handed to robot-like coders, who hold pulp-fiction novels in one hand while sleepily typing in the design's implementation with the other. Finally, Inspector 12 receives the completed code, which doesn't receive her stamp of approval unless it meets the original specifications.

It's no surprise that managers want software development to be like manufacturing. Managers *understand* how to make manufacturing work. We have decades of experience in how to build physical objects efficiently and accurately. So, applying what we've learned from manufacturing, we should be able to optimize the software development process into the well-tuned engine that our manufacturing plants have become.

In the so-called software factory, the employees are specialists. They sit at their place in the assembly line, fastening Java components together or rounding the rough edges of a Visual Basic application on their software lathes. Inspector 12 is a tester by trade. Software components move down the line, and she tests and stamps them in the same way each day. J2EE designers design J2EE applications. C++ coders code in C++. The world is very clean and compartmentalized.

Unfortunately, the manufacturing analogy doesn't work. Software is at least as malleable as software requirements. Things change in business, and businesspeople know that software is *soft* and can be changed to meet those requirements. This means architecture, designs, code, and tests must all be created and revised in a fashion more agile than the leanest manufacturing processes can provide.

In this kind of rapidly changing environment, the flexible will survive. When the pressure is on, a smart businessperson will turn to a software professional can solve the problem at hand. So, how do you become that person whose name comes up when they're looking for a superhero to save the day? The key is to be able to solve the problems that may arise.

What are those problems? That's right: you don't know. Neither do I. What I *do* know is that those problems are as diverse as deployment issues,

critical design flaws that need to be solved and quickly reimplemented, heterogenous system integration, and rapid, ad hoc report generation. Faced with a problem set as diverse as this, poor Inspector 12 would be passed over pretty quickly.

The label *jack-of-all-trades—master of none* is normally meant to be derogatory, implying that the labelee lacks the focus to really dive into a subject and master it. But, when your online shopping application is on the fritz, and you're losing orders by the hundreds as each hour passes, it's the jack-of-all-trades who not only knows how the application's code works but can also do low-level UNIX debugging of your web server processes, analyze your RDBM's configuration for potential performance bottlenecks, and check your network's router configuration for hard-to-find problems. And, more important, after finding the problem, the jack-of-all-trades can quickly make architecture and design decisions, implement code fixes, and deploy a new fixed system to production. In this scenario, the manufacturing scenario seems quaint at best and critically flawed at worst.

Another way in which the software factory breaks down is in that, although in an assembly line the work keeps coming in a steady flow, software projects are usually very cyclical. Not only is the actual flow of projects cyclical, but the work inside a project is cyclical. A coder sits on the bench while requirements are being specified, architected, and designed, or the coder multitasks across many projects. The problem with multitasking coders is that, despite the software factory's intentions, when the rubber meets the road, the coders rely a great deal on context and experience to get their jobs done. Requirements, architecture, and design documents can be a great head start, but ultimately if the programmers don't understand what the system is supposed to do, they won't be able to create a good implementation of the system.

Of course, I'm not just picking on coders here. The same is true at nearly every spot on the software assembly line. Context matters, and multitasking doesn't quite work. As a result, we have an inefficient manufacturing system. There have been various attempts to solve this problem of inefficiency without departing from the manufacturing-inspired system, but we have not yet figured out how to optimize our software factories to an acceptable level.

If you are *just* a coder or a tester or a designer or an architect, you're going to find yourself sitting idle or doing busywork during the ebbs of your business's project flow. If you are *just* a J2EE programmer or a .NET programmer or a UNIX systems programmer, you're not going to have much

to contribute when the focus of a project or a company shifts, even temporarily, out of your focus area. It's not about where you sit on the perceived value chain of project work (where the architect holds the highest spot of royalty). It's about how generally useful you make yourself.

If your goal is to be the last person standing amid rounds of layoffs and the shipment of jobs overseas, you better make yourself *generally* useful. If you're afraid that your once-crowded development office will become home to an onshore skeleton crew, it would serve you well to realize that when the team has only a few slots, a *just-a-tester* or *just-a-coder* is not going to be in demand.

Generalists are rare...and, therefore, precious.

Something I have personally experienced in searching for employees in low-cost countries is that there aren't many generalists. The Indian IT industry, for example, was formed in the image of its cultural heritage—one that places great emphasis on rank and title. I interviewed people calling themselves *team leaders* who led teams of two (self-inclusive) and reported to managers of two such teams. In many cases it can get so ridiculous, that the organizations' structures are parodies of themselves.

The software factory system of development is a perfect fit for the Indian IT sector, because it naturally supports the hierarchy that the companies and their employees desire. Testers are the bottom rung of the ladder, and nearly everyone you meet there wants to become an architect and then a high-level manager. The culture breeds specialists. Architects don't stoop to design. Designers don't stoop to code, and so on.

The way to become a generalist is to not label yourself with a specific role or technology. We can become typecast in our careers in many ways. To visualize what it means to be a generalist, it can help to dissect the IT career landscape into its various independent aspects. I can think of five, but an infinite number exists (it's all in how you personally divide topics):

- Rung on the career ladder
- Platform/OS
- Code vs. data
- Systems vs. applications
- Business vs. IT

These are different dimensions on which you can approach the problem of becoming a generalist. This is just a way to think about the whole pic-

ture of your career, and you can probably come up with a better list for yourself. For now, we'll discuss these.

First, you can choose to either be a leader or manager type or be a technical person. Or, you might pigeon hole yourself into architect as opposed to being a programmer or tester. The ability to be flexible in the roles you can and will fill is an attribute that many people don't understand the value of. For example, while a strong leader should avoid pinch hitting as often as possible, the new world of onshore skeleton crews can benefit from a person who knows how to lead people and projects but can also roll up their sleeves and fix some last minute critical bugs while the Offshore team is sleeping. The same is true of a software architect who could perhaps dramatically speed up progress on a project if he or she would only *write some code* to get things moving. When it comes to hierarchical boundary crossing, it's most often not reluctance that stops people from doing it. It's ability. Programmer geeks can't lead and leaders can't hack. It's rare to find someone who's even decent at both.

If companies need generalists, they're going to have to get them in the West.

Another artificial (and inexcusable) line gets drawn around platforms or operating systems. Being a UNIX Guy who refuses to do Windows is increasingly more impractical as the jobs flow away. The same goes for .NET versus J2EE or any other such infrastructure platforms. Longevity is going to require that you are platform neutral in the work place. We all have our preferences, but you're going to have to leave your ideals at home. Master one and get good at the other. Your skills should transcend technology platform. It's just a tool. If we want a Windows person, we can hire them in the Phillipines. If we want someone who really understands Windows and UNIX development and can help us integrate them together, we're probably going to be looking Onshore. Don't get passed up over what is essentially team spirit.

Your skills should transcend technology platforms.

The dividing line between database administrator (a role that has solidified out of nothingness over the past decade) and software developer should also be fuzzy. Being a database administrator, or DBA, has in many organization come to mean that you know how to use some GUI admin tool and you know how to setup a specific database product. You don't necessarily know much of anything about how to *use* the database. On the flip side, software developers are growing increasingly lazy and ignorant about how to work with databases. Each side feeds the other.

What first amazed me most when I entered the information technology field was that many well-educated programmers (maybe *most*) didn't know the first thing about how to set up the systems they used for development and deployment. I worked with developers who couldn't even install an operating system on a PC if you asked them to, much less set up an application server on which to deploy their applications. It's rare, and refreshing, to find a developer who truly understands the platform on which he or she is working. Applications are better and work gets done faster as a result.

Finally, as we discussed in *Coding Don't Cut It Anymore*, on page 16, the wall between The Business and IT should be torn down right now. Start learning how your business operates.

Act on it!

1. On a piece of paper or a whiteboard, list the dimensions on which you may or may not be generalizing your knowledge and abilities. For each dimension, write your specialty. For example, if *Platform and Operating System* is one of your dimensions, you might write *Windows/.NET* next to it. Now, to the right of your specialty, write one or more topics you should put into your TO-LEARN list. Continuing with the same example, you might write *Linux* and *Java* (or even *Ruby* or *Perl*).

As soon as possible (some time this week at the latest!), find thirty minutes of time to start addressing at least one of the TO-LEARN items on your list. Don't just read about it. If possible, get some hands-on experience. If it's web technology, then download a web server package and set it up yourself. If it's a business topic, find one of your customers at work and ask them to go out for lunch for a chat.

6

Be a Specialist

How would you write a program, in pure Java, that would make the Java Virtual Machine crash? Dead silence. Hello?

I'm sorry. I'm not getting you. Could you repeat the question, please? The voice sounded desperate. I knew from experience that repeating the question wasn't going to help. So, I repeated the question, slowly and more loudly. *How would you write a program, in pure Java, that would cause the Java Virtual Machine to crash?*

Uh...I'm sorry. I've never done this before.

I'm sure you haven't. How about this question: how would you write a program that would NOT cause the JVM to crash? I was looking for really good Java programmers. To start the interview, I asked this person (and all the others I had interviewed that week) to rate himself on a scale of one to ten. He said *nine*. I'm expecting a star here. *If this guy rates himself so high, why can't he think of a single abusive programming trick that would cause a JVM to crash?*

Lack of technical depth.

This was someone who claimed to specialize in Java. If you met him at a party and asked what he did for a living, he would say, "I am a Java developer." Yet, he couldn't answer this simple question. He couldn't even come up with a *wrong* answer. Over two-and-a-half intense weeks of interviewing in Hyderabad, Bangalore, and Chennai, this was the rule—not the exception. Thousands of Java specialists had applied for open positions, nearly none of whom could explain how a Java class loader works or give a high-level overview of how memory management is typically handled by a Java Virtual Machine.

Granted, you don't have to know these things to hack out basic code under the supervision of others. But, these were supposed to be *experts*.

Too many of us seem to believe that specializing in something simply means you don't know about other things. I could, for example, call my mother a Windows specialist, because she has never used Linux or OS X. Or, I could say that my relatives out in the countryside in Arkansas are country music specialists, because they've never heard anything else.

Imagine you visit your family doctor, complaining about a strange lump under the skin of your right arm. Your doctor refers you to a specialist

Too many of us seem to believe that specializing in something simply means not knowing about other things.

Imagine you visit your family doctor, complaining about a strange lump under the skin of your right arm. Your doctor refers you to a specialist

to have a biopsy performed. What if that specialist was a person whose only credentials as a specialist were that they didn't attend any classes in medical school or have any experience in residencies that weren't *directly* relevant to the act of performing the specific procedure that they were going to perform on you today? I don't mean that they went *deeper* into the topics related to today's procedure. What if they had just skimmed the surface of these topics, but they didn't know anything else? *What if that machine over there starts beeping during the operation?* you might ask. *Oh, that's never happened before. It won't happen this time. I don't know what that machine does, but it never beeps.*

Thankfully, *most* software developers aren't responsible for life or death situations. If they mess up, it typically results in project overruns or production bugs that simply cost their employers money—not lives.

Unfortunately, the software industry has churned out a whole lot of these shallow specialists, who use the term *specialist* as an excuse for knowing only one thing. In the medical industry, a specialist is someone with a *deep* understanding of some specific area of the field. Doctors refer their patients to specialists, because in certain specific circumstances, the specialist can give them better care than a general practitioner.

So, what should a specialist be in the software field? I can tell you what I was searching for in every nook and cranny in South India. I was searching for people who deeply understood the Java programming and deployment environment. I wanted folks who could say “been there—done that” in 80% of the situations we might encounter and whose depth of knowledge could make the remaining 20% more livable. I wanted someone who, when dealing with high-level abstractions, would understand the low-level details of what went into the implementation of those abstractions. I wanted someone who could solve any deployment issue we might encounter or would at least know who to call for help if they couldn't.

This is the kind of specialist who will survive in the changing computer industry. If you're a .NET specialist, it's not just an excuse for not knowing anything *except* .NET. It means that if it has to do with .NET, you are the authority. IIS servers hanging and needing to be rebooted? *No problem.* Source control integration with Visual Studio .NET? *I'll show you how.* Customers threatening to pull the plug because of obscure performance issues? *Give me thirty minutes.*

If this isn't what *specialist* means to you, then I hope you don't claim to be one.

Act on it!

1. Do you use a programming language that compiles and runs on a virtual machine? If so, take some time to learn about the internals of how your VM works. For Java, .NET, and Smalltalk, many books and websites are devoted to the topic. It's easier to learn about than you think.

Whether your language relies on a VM or not, take some time to study just what happens when you compile a source file. How does the code you type go from being text that you can read to instructions that a computer can execute? What would it mean to write your own compiler?

When you import or use external libraries, where do they come from? What does it *actually* mean to import an external library? How does your compiler, operating system, or virtual machine link multiple pieces of code together to form a coherent system?

Learning these facts will take you several steps closer to being an expert specialist in your technology of choice.

2. Find an opportunity—at work or outside—to teach a class on some aspect of a technology that you would like to develop some depth in. As you'll see in *Be a Mentor*, teaching is one of the best ways to learn.

7

Don't Put All Your Eggs in Someone Else's Basket

While managing an application development group, I once asked one of my employees, "What do you want to do with your career? What do you want to be?" I was terribly disappointed by his answer: "I want to be a J2EE architect." I asked why not a "Microsoft Word designer" or a "RealPlayer installer?"

This guy wanted to build his *career* around a specific technology created by a specific company of which he *was not an employee*. What if the company goes out of business? What if it let its now-sexy technology become obsolete? Why would you want to trust a technology company with your career?

Somehow, as an industry, we fool ourselves into thinking *market leader* is the same thing as *standard*. So, to some people, it seems rational to make another company's product a part of their identities. Even worse, some base their careers around non-market-leading products—at least until their careers fail so miserably that they have no choice but to rethink this losing strategy.

Let's take a moment again to remember that we should think of our career as a business. Though it's possible to build a business that exists as a parasite of another (such as companies who build spyware removal products to make up for inadequacies in Microsoft's browser security model), as an individual, it's an incredibly risky thing to do. A company, such as the spyware example I just mentioned, can usually react to changing forces in the market such as an unexpected improvement in Microsoft's browser security (or Microsoft deciding to enter the spyware removal market), whereas an individual doesn't have the bandwidth or the surplus cash to suddenly change career direction or focus.

Vendor-centric views are typically myopic.

The sad thing about a vendor-centric view of the world is that, usually, the details of a vendor's software implementation are a secret.

You can really learn only so much about a piece of proprietary software until you reach the *professional services barrier*. The professional services barrier is the artificial barrier that a company erects between you and the solution to a problem you may have, so that it can

profit from selling you support services. Sometimes this barrier is intentionally erected, and sometimes it's erected as a side effect of the attempt the company makes to protect its intellectual property (by not sharing its source code).

So, while a single-minded investment in one particular technology is almost always a *bad idea*, if you *must* do so, consider focusing on an open-source option, as opposed to a commercial one. Even if you can't or don't want to make the case for using the open-source solution in your workplace, use the open-source option as the platform from which you can take a deep dive into a technology. For example, you may want to become an expert in how J2EE application servers work. Instead of focusing your efforts on the details of how to configure and deploy a commercial application server (after all, *anybody* can figure out how to tweak settings in a config file, right?), download the open-source JBoss or Geronimo servers and set aside time for yourself to not only learn how to operate the servers but to study their internals.

Before long, you'll realize you're naturally changing your view. This J2EE thing (or whatever you chose to get into) really isn't all that special. Now that you see the details of the implementation, you see that there are high-level conceptual patterns at work. And, you start to realize that, whether with Java or some other language or platform, distributed enterprise architecture is distributed enterprise architecture. Your view changes from narrow to wide, and your mind starts to open. You start to realize that these concepts and patterns that your brain is sorting through and making sense of are much more scalable and universal than any specific vendor's technology. "Let the vendors come and go—I know how to design a system!"

Act on it!

1. Try a small project, twice. Try it once in your home base technology and once, as idiomatically as possible, in a competing technology.

8

Be the Worst

Legendary jazz guitarist Pat Metheny has a stock piece of advice for young musicians: “always be the worst guy in every band you’re in.”⁶

Be the worst guy in every band you’re in.

Before starting my career in information technology, I was a professional jazz and blues saxophonist. As a musician, I had the good fortune of learning this lesson early on and sticking

to it. Being the worst guy in the band means always playing with people who are better than you.

Now, why would you always choose to be the worst person in a band? “Isn’t it unnerving,” you ask? Yes, it’s extremely unnerving at first. As a young musician, I would find myself in situations where I was so obviously the worst guy in the band that I was *sure* I would stick out like a sore thumb. I’d show up to a gig and not even want to unpack my saxophone for fear I’d be forcefully ejected from the bandstand. I’d find myself standing next to people I looked up to, expected to perform at their level—sometimes as the lead instrument!

Without fail (thankfully!), something magical would happen in these situations: I would fit in. I wouldn’t stand out among the other musicians as a star. On the other hand, I wouldn’t be obviously outclassed, either. This would happen for two reasons. The first reason is that I really wasn’t as bad as I thought. We’ll come back to this one later.

The more interesting reason that I would fit in with these superior musicians—my heroes, in some cases—is that my playing would transform itself to be more like theirs. I’d like to think I had some kind of superhuman ability to morph into a genius simply by standing next to one, but in retrospect I think it’s a lot less glamorous than that. It was more like some kind of instinctual herd behavior, programmed into me. It’s the same phenomenon that makes me adopt new vocabulary or grammatical habits when I’m around people who speak differently than me. When we returned from a year and a half of living in India, my wife would sometimes listen to me speaking and burst into laughter, “Did you *hear* what you just said?” I was speaking Indian English.

⁶Originally spotted by Chris Morris at <http://clabs.org/blogki>

Being the worst guy in the band brought out the same behavior in me as a saxophonist. I would naturally just play like everyone else. What makes this phenomenon really unglamorous is that when I played in casinos and hole-in-the-wall bars with those not-so-good bands, I played like *those* guys. Also, like an alcoholic who slurs his speech even when he's not drunk, I'd find the bad habits of the bar bands carrying over to my non-bar-band nights.

So, I learned from this that people can significantly improve or regress in skill, purely based on who they are performing with. And, prolonged experience with a group can have a lasting impact on one's ability to perform.

Later, as I moved into the computer industry, I found that this learned habit of seeking out the best musicians came naturally to me as a programmer. Perhaps unconsciously, I sought out the best IT people to work with. And, not surprisingly, the lesson holds true. Being the worst guy (or gal, of course) on the team has the same effect as being the worst guy in the band. You find that you're unexplainably *smarter*. You even speak and write more intelligently. Your code and designs get more elegant, and you find that you're able to solve hard problems with increasingly creative solutions.

The people around you affect your own performance. Choose your crowd wisely.

Let's go back to the first reason that I was able to blend into those bands better than I expected. I really wasn't as bad as I thought. In music, it's pretty easy to measure whether other musicians think you're good. If you're good, they invite you to play with them again. If you're not, they avoid you. It's a much more reliable measurement than just asking them what they think, because good musicians don't like playing with bad ones. Much to my surprise, I found that in many of these cases, I would get called by one or more of these superior musicians for additional work or to even start bands with them.

Attempting to be the worst actually stops you from selling yourself short. You might belong in the A band but always put yourself in the B band, because you're afraid. Acknowledging outright that you're not the best wipes away the fear of being discovered for the not-best person you are. In reality, even when you *try* to be the worst, you won't actually be.

Act on it!

1. *Find a “be the worst” situation for yourself*—You may not have the luxury of immediately switching teams or companies just because you want to work with better people. Instead, find a volunteer project on which you can work with other developers who will make you better via osmosis. Check for developer group meetings in your city, and attend those meetings. Developers are often looking for spare-time projects on which to practice new techniques and hone their skills.

If you don't have an active developer community nearby, use the Internet. Pick an open-source project that you admire and whose developers appear to be at that “next level” you're looking to reach. Go through the project's TO-DO list or mailing list archives, pick a feature or a major bug fix, and code away! Emulate the style of the project's surrounding code. Turn it into a game. Make your design and code so indistinguishable from the rest of the project that even the original developers eventually won't remember who wrote it. Then, when you're satisfied with your work, submit it as a patch. If it's good, it will be accepted into the project. Start over and do it again. If you've made decisions that the project's developers disagree with, either incorporate their feedback and resubmit or take note of the changes they make. On your next patch, try to get it in with less rework. Eventually, you'll find yourself to be a trusted member of the project team. You'll be amazed at what you can learn from a remote set of senior developers, even if you never get a chance to hear their voices.

9

Love It or Leave It

It may sound like some kind of rah-rah cheerleader crap, aimed at whipping you into an idealistic frenzy, but it's too important not to mention. You have to be passionate about your work if you want to be *great* at your work. If you don't care, it will show.

When my wife and I moved to Bangalore, I was expecting to find like-minded technologists with a passion for learning. I was expecting a vibrant after-work life of user group meetings and deep, philosophical discussions on software development methodologies and techniques. I was expecting to find India's Silicon Valley bursting at its seams with an overflow of artsians, enthusiastic in the pursuit of the great craft of software development.

What I found were *a whole lot* of people who were picking up a paycheck and *a few* incredibly passionate craftspeople.

Just like back home.

Of course, I didn't realize it was just like back home at the time. I had a few data points from the United States, but I always assumed I was just working in bad cities or bad company environments. I counted situations like my first experiences with IT employment as outliers. *Most software developers must get it*, I thought. *I just haven't found the right environment yet.*

I started work at my university's IT department on a blind recommendation from my friend Walter, who had seen me work with computers enough to know I could probably make them do things better than most of the people who needed help at the university. I didn't believe I could, having had no formal training. I was just a saxophone player who liked to play video games. But, Walter actually filled out an application for me and set up an interview. I was hired without so much as a single technical question being asked, and I was to start immediately.

When I showed up on the job, I was paranoid I would be discovered as the charlatan I really was. *What is this saxophone player doing here with us trained professionals?* After all, I was working with people who had advanced computer science degrees. And, here I was with only part of a music degree trying to fit in as if I knew something.

Within a few days of work, the truth started to sink in. *These people don't know what the hell they're doing!* In fact, some people were watching me work and *taking notes!* People with *master's degrees in computer science!*

My first reaction was to assume I was surrounded by idiots. After all, I didn't have any formal training. I spent my nights playing in bar bands and my days playing computer games. I had learned how to work with computers only because I was interested in them. In fact, I really learned how to write programs because I wanted to make my own computer games. I would come home late after a deafening evening at a bar and browse Gopher⁷ sites with tutorials on programming until the sun came up. Then I'd sleep, wake up, and continue my learning until I had to go out and perform again. I'd break up the study with my beloved computer games, eat, and then go back to goofing around with Gopher and whatever compilers I could get working.

Work because you
couldn't *not* work.

Looking back on it, I was addicted, but in a good way. My drive to create had been ignited in much the same way that it had when I started writing classical music or playing improvisational jazz. I was obsessed with learning anything and everything I could. I wasn't in this for a new career. In fact, many of my musician friends thought of it as an irresponsible distraction from my actual career. I was in it because I couldn't *not be*.

This was the difference between me and my overeducated, underperforming colleagues at work. Passion. These people had no idea *why* they were in the IT field. They had stumbled into their careers, because they thought computer programming might pay well, because their parents encouraged them, or because they couldn't think of a better major in college. Unfortunately, their performance on the job reflected it.

If you think about the biographies you read or the documentaries you watch about the greats in various fields, this same pattern of addictive, passionate behavior surfaces. Jazz saxophone great John Coltrane reportedly practiced so much that his lips would bleed.

Of course, natural talent plays a big role in ability. We can't all be Mozart or Coltrane. But, we can all take a big step away from mediocrity by finding work we are passionate about.

It might be a technology or business domain that gets you excited. Or, on the other hand, it might be a specific technology or business domain that drags you down. Or a type of organization. Maybe you're meant for small

⁷Gopher is a document-sharing system similar in intent to the World Wide Web. Its popularity declined dramatically with the rise of the Web.

teams or big teams. Or rigid processes. Or agile processes. Whatever the mix, take some time to find yours.

You can fake it for a while, but a lack of passion will catch up with you and your work.

Act on it!

1. Go find a job you're actually passionate about.

Part II

Investing in Your Product

While living in India, we had a car and a driver. It's not that we were trying to be big shots. Driving in India is scary. We didn't want to take the risk of trying to navigate Bangalore traffic under our own volition. The rules are so different there, at best we'd have ended up in a fender bender or three and at worst someone could have gotten badly hurt.

So, we were assigned a driver. His name was Ramesh. Like most Indians in his line of work, Ramesh was undereducated, having dropped out of school to help support his family business as a young teenager. He had spent a brief stint in the military as a soldier and had lucked into the driving gig a few years later. He lived in a small house, and his salary—hovering somewhere around what I probably spend on coffee in a month—supported himself, his wife, his baby girl, and his mother.

Our relationship with Ramesh started quietly. He was trained to respect his customers by not speaking to them unless spoken to. He would come to our place early in the morning and just wait in the car until we were ready to go somewhere. When we got to our destination, he would wait in the car until we were ready to go again. This would go on until sometimes late at night, with brief bits of driving surrounded by long periods of sitting in the car waiting.

My wife Kelly and I felt bad that he just *sat there*, probably bored out of his mind while we took our sweet time eating dinner or shopping. I asked Ramesh about this early on. *It's no problem, sir. I read while I wait*, he said, pointing to a pile of unintelligible pulp paper covered in Kannada script.

As I dug deeper, asking what he was reading about, it turned out that he did nothing but read educational material. In fact, Ramesh spoke six languages fluently and was a bit of an expert on Indian history and culture. We were hungry for Indian cultural lessons, and Ramesh quickly became our cultural ambassador. Every ride in the car was a language, culture, or history lesson. While we lived in India, he started learning both how to speak and read Sanskrit and how to play South Indian classical music on the veena (an instrument that was the precursor to the more well-known sitar).

Other than his own hunger for knowledge, he spent all of this time studying so that he could teach the subjects he learned to his young daughter, Likhitha. He was determined to prepare her with the skills and knowledge she would need to lead a better life than he had been born into. And, though his formal education was lacking, he had made himself into somewhat of a Renaissance man in his spare time.

Ramesh, sitting there in the lower middle class of a developing country, spoke *six* languages fluently and devoted himself to study so that he could change his family's life. In principle, he was not unique in this respect.

If you work with Indian people now, it's tempting to think that their sometimes unusual use of English or the accent they use somehow makes them unintelligent. However, the people you're talking to probably speak at minimum two or three languages and very possibly four or five! How many languages do you speak? If you're like most Americans, the answer is *one*.

I'm not telling you all this just to make you feel bad. But, you better not think that you're up against a bunch of people who, though cheaper to hire, are not as smart as you. That could be a career-limiting mistake.

Nope. The people in India (and China, Hungary, the Ukraine, and other low-cost outsourcing destinations) are not any less intelligent than you. Those you deal with at work are probably better educated than you. I met people with master's degrees in the sciences working as call takers in call centers—not even programming jobs. And, these people are *hungry*. I don't mean hungry for food, though many of them are only a generation away from such circumstances, but they are hungry for a better and different life.

So, even if you think you have *proof* that they can't be better than us, don't put your guard down and arrogantly assume that the cost of labor is the only real point of competition. You've got to keep up, or you may find yourself looking for a job as a taxi driver.

If you want to have a great *product* to sell on the job market—a product that stands out, and that lets you really compete—you're going to have to invest in that product. In business, ideas are a dime a dozen. It's the blood, sweat, tears, and money you pour into a product that make it really *worth something*.

In this part, we'll look at investment strategies for your career. We'll explore how to choose which skills and technologies to invest in as well as look at different *ways* of investing in ourselves. This part is where the real work starts.

Lao Tzu said, “Give a man a fish; feed him for a day. Teach a man to fish; feed him for a lifetime.” That’s all well and good. But Lao Tzu left out the part where the man doesn’t want to learn how to fish and he asks you for another fish tomorrow. Education requires both a teacher and a student. Many of us are too often reluctant to be a student.

Just what is a *fish* in the software industry? It’s the process of using a tool, or some facet of a technology, or a specific piece of information from a business domain you’re working in. It’s

Don’t wait to be told.

Ask!

how to check out a specific branch from your team’s source control system, or it’s getting an application server up and running for development. Too many of us take these details for granted. *Someone else can take care of this for me*, you may think. The build guy knows about the source control system. You just ask him to set things up for you when you need them. The infrastructure team knows how the firewalls between you and your customers are set up, so if you have an application need, you just send an email and the team will take care of it.

Who wants to be at the mercy of someone else? Or, worse: if you were looking to hire someone to do a job for you, would you want that person to be at the mercy of *the experts*? I wouldn’t. I’d want to hire someone who is self-sufficient.

The most obvious place to start is in learning the tools of your trade. Source control, for example, is a powerful tool. An important part of its job is focused on making developers more productive. It’s not just the place where you put your code when you’re done with it, and you shouldn’t treat it as such. It’s an integral part of your development process. Don’t let such an important thing—the authoritative repository of your work—be like voodoo to you. A self-sufficient developer can easily check differences between the version of a project that he or she has checked out and the last known good one in the repository. Or perhaps you need to pull out the last released code and make a bug fix. If your code has a critical bug in the middle of the night, you don’t want to have to call someone else to ask them to get you the right version so you can start troubleshooting. This goes for IDEs, operating systems, and pretty much every piece of infrastructure your code or process rides on top of.

Equally important is the technology platform you are employing. For example, you may be developing applications using J2EE. You know you have to create various classes, interfaces, and deployment descriptors. Do you know *why*? Do you know how these things are used? When you start up a J2EE container, what actually happens? You may not be an application server developer, but knowing how this stuff works enables you to develop solid code for a platform and to troubleshoot when something goes wrong.

A particularly easy way to get lazy is to use a lot of wizards that generate code for you. This is particularly prevalent in the world of Windows development where, to Microsoft's credit, the development tools make a lot of tasks really easy. The downside is that many Windows developers have no idea how their code really works. The work of the wizards remains a magical mystery. Don't get me wrong—code generation used correctly can be a useful tool. For example, code generators are what translate high-level C# code to byte codes that can run on the .NET runtime. You obviously wouldn't want to have to write all those byte codes yourself. But, especially at the higher levels, letting the wizards have their way leaves your knowledge shallow and leaves you limited to what the wizards can already do for you.

We may easily overlook the fish in our business domain. If you're working for a mortgage company, either you could ask an expert for the calculation of an interest rate for each scenario that you need during testing or you could learn how to calculate it yourself. While interactions with your customer are good, and it's good to clarify business requirements with them (as opposed to half-understanding and filling in the details yourself), imagine how much faster you could go if you actually knew the ins and outs of the business domain you're working in. You probably won't know every single business rule—that's not your job. But, you can at least learn the basics. Many of the best software people I've worked with over the years have become more expert in their domains than even some of their business clients. This results in better products. Someone who is domain-ignorant will let silly mistakes slip through—mistakes that a basic knowledge of the business domain would have avoided. Furthermore, they'll go slower (and ultimately cost the company more) than the equivalent developer who understands the business.

For us software developers, Lao Tzu's intent might be equally well served with "Ask for a fish; eat for a day. Ask someone to teach you to fish; eat for a lifetime." Better yet, don't *ask* to be taught—go learn for yourself.

Act on it!

1. *How and why?*—Either as you sit here reading or the next time you're at work, think about the facets of your job that you may not fully understand. You can ask yourself two extremely useful questions about any given area to drill down into the murky layers: *How does it work?* and *Why does this (have to) happen?*

You may not even be able to answer the questions, but the very act of asking them will put you into a new frame of mind and will generate a higher level of awareness about your work environment. *How does the IIS server end up passing requests to my ASP.NET pages? Why do I have to generate these interfaces and deployment descriptors for my EJB applications? How does my compiler deal with dynamic versus static linking? Why do we calculate tax differently if a shopper lives in Montana?*

Of course, the answer to any of these questions will lead to another potential opportunity to ask the question again. When you can't go any further down the *how and why* tree, you've probably gone far enough.

2. *Tip time*—Pick one of the most critical but neglected tools in your toolbox to focus on. Perhaps it's your version control system, a library that you use extensively but you've looked into only superficially, or it may be the editor you use when programming.

When you've picked the tool, allot yourself a small period of time each day to learn *one new thing* about the tool that will make you more productive or put you in better control over your development environment. You may, for example, choose to master the GNU Bourne Again Shell (*bash*). During one of those times when your mind starts to wander from the task at hand, instead of loading up *Slashdot* you could search the Internet for *bash tips*. Within a minute or two, you should find *something* useful that you didn't know about how to use the shell. Of course, now that you have a new trick, you can dive into its guts with a series of *Hows* and *Whys*.

11

Understand Business Basics

In the previous chapter, we discussed the importance of making an intentional choice about the business domain in which you work. Domain knowledge, being at best an employment differentiator for a job and at worst a showstopper, isn't something you should take lightly. Before making an investment in learning the ins and outs of a business domain, you should make sure you're investing in the right one for you and for the state of the market.

But, one body of knowledge is neither technical nor domain-specific and won't be outdated at any time soon: the basics of business finance. Regardless of your line of business, whether it be manufacturing, health-care, nonprofit, or an educational institution, it is still a *business*. And, *business* is itself a domain of knowledge that one can—indeed, must—learn.

I remember as a young programmer going to staff meetings, my eyes glazing over as some big-shot leader with whom I would never directly work showed chart after chart of numbers that I believed to be completely irrelevant to me. *I just want to go back and finish the application feature I'm working on*, I would whine to myself. My teammates sat together, looking like a row of squirming children on a long car ride. None of us understood what was being presented, and none of us cared. We blamed what we felt was a complete waste of time on the incompetent managers who had called the meeting.

You can't creatively help a business, until you know how it works.

Looking back on it, I realize how foolish we were. We worked for a business and our job was to contribute to either making or saving money for that business. Yet we didn't understand the basics of how the business came to profitability. Worse, we didn't think it was our job to know. We were programmers and system administrators. We thought our jobs were strictly about those topics that we had devoted ourselves to. However, how were we supposed to *creatively* help the business be profitable if we didn't even understand how the business *worked*?

The use of the word *creatively* in the previous paragraph is the key. It's plausible to have the view that we are indeed IT specialists and that is what we are paid to be. Given the right projects and leadership, we should

be putting effort into tasks that help the business. We don't need to fully understand how a business runs to provide value to it.

But, to *creatively* add value takes a more thorough understanding of the business environment in which you work. In the business world, we hear the phrase *bottom line* all the time. How many of us truly understand what the bottom line is and what contributes to it? More important, how many of us really understand how *we* contribute to the bottom line? Is your organization a cost center or a profit center (do you add to or take away from the bottom line)?

Understanding the financial drivers—and language—of your company will give you the ability to make meaningful changes, rather than stabbing in the dark at things that seem intuitively right to you.

Act on it!

1. Go get a book on basic business, and work through it. A trick for finding a good overview book is to look for books about getting an MBA (Master's of Business Administration) degree. One such book that I found particularly useful (and pleasantly short) is *The Ten-Day MBA* (Sil99). You can actually get through it in ten days. That's not a very big investment.
2. Ask someone to walk you through the financials of your company or division and explain them to you (if this is information your company doesn't mind sharing with its employees).
3. Explain them back.
4. Find out why the bottom line is called "the bottom line."

12

Find a Mentor

One thing that I found in India that the people have really gotten right is the practice of finding a mentor. Even in less craft-oriented lines of work, it is common for younger Indian professionals to have a somewhat formal mentoring relationship with a more experienced person they can trust. Mentors provide advice on real issues that come up in the careers of those who trust them. They help bail them out of tough situations. They help them find the right jobs to grow their careers with. In exchange, the people receiving the mentoring reciprocate in any ways they can.

Connections are made and people are hired every day via these relationships. Indian society has created a self-organizing culture and set of customs around the mentor/mentee relationship. It's a system that works so well that you would suspect it was guided by some kind of organizing body.

It's OK to depend on someone. Just make sure it's the right person.

In the Western world, we're less likely to ask each other for help. Depending on others is often seen as a sign of weakness. We're afraid to admit that we're not perfect. Everything is competition. Only the strong survive, and all that. Unfortunately, this leads to an extremely underdeveloped system of mentoring. In a country like India, if I were to ask a handful of programmers, "who is your mentor?" most of them would have an answer. In the United States, they'd probably respond with "What?"

It hasn't always been like this here. The history of the West includes a thriving system of mentoring, extending back into the Middle Ages. The craftsmanship approach to professional training was even stronger and more formalized than the system that has evolved in current Indian society. Young people would start their professional careers as apprentices to respected master craftsmen. They would work in exchange for a nominal salary and the privilege of learning from the master. The master's obligation was to train the apprentices to create things in the tradition (and of the quality) of the master himself.

The first and most important purpose that a mentor serves is that of a role model. It's hard to know what's possible until you see someone who can stretch the limits you're familiar with. A role model sets the standard for what "good" means. If you thought of yourself as a chess player, for

example, just being able to beat the people in your immediate family might feel pretty good. But, if you played with a tournament player, you would find that chess is a much deeper game than you ever knew. If you were to play with a grand master, you'd have another such revelation. If you keep playing with, and beating, your immediate family members, you might get the idea that you're *really good* at chess. Without a role model, there's no incentive to get better.

A mentor can also give structure to your learning process. As you saw in the previous chapter, you have an overwhelming number of choices to make about which technologies and domains to invest in. Sometimes, too many choices can get you stuck. Within reason, it's better to be moving in one direction than to be sitting still. A mentor can help take some of the choice out of what to focus your energies on.

When I started my career as a system support person, I latched onto a saint named Ken who was one of our university's network administrators. He came in to bail me out of a big problem with our campus NetWare network that was crippling the students who were trying to use our computer labs, and after that point he was unable to shake me (nor did he try). When I prodded him to give me direction on how to become more knowledgeable and self-sufficient, he gave me a simple recipe: dive into directory services, get comfortable with a UNIX variant, and master a scripting language.

He picked three skills for me to learn from the infinite number available. And, with the confidence that this person, who I considered to be a master, had prescribed them, I set out to learn those three skills. My career since has been built on the foundation of those pieces of knowledge, all three of which are still completely relevant in everything I do. It's not that Ken's direction was the absolute right answer—there are no absolute right answers. The important thing is that he narrowed down the long list of skills *I could* be learning into the short list of skills *I learned*.

A mentor also serves as a trusted party who can observe and judge your decisions and your progress. If you're a programmer, you can show them your code and get pointers. If you're planning to give a presentation at the office or a local user group meeting, you can run it by your mentor beforehand for feedback. A mentor is someone you can trust enough to ask, "What should be different about me as a professional?" because you know that they'll not only criticize you but they'll help you improve.

Finally, just as in India, not only do you create a personal attachment and responsibility to your mentor, but the reverse happens as well. If my role

in a relationship is to help someone, I become invested in that person's success. I'm nudging someone along their career on a path that I believe is the right one. So, if that path leads to success, it's my success as well.

This creates incentive on the part of the mentor for his or her mentees to succeed. Typically, being more experienced and already successful, a person in such a role would have the respect of a significant network of people. The mentor becomes a positively reinforced connection from you to his or her network. The importance of this kind of connection can't be underestimated. After all, the phrase "It's not what you know. It's who you know" isn't a cliché for nothing.

The degree of formality in a mentor relationship is not important. Nobody has to explicitly ask someone to be their mentor (though it's definitely not a bad thing if you do). In fact, your mentor may not even *know* they are serving that role for you. What's important is that you have someone you trust and admire that can help give you career guidance and help you hone your craft.

Act on it!

1. *Mentoring yourself*—We'd all ideally have someone to actively mentor us, but the reality is that we won't always be able to find someone in the same location that we can place in this role. Here's a way to proxy-mentor yourself.

Think of the person in your field whom you admire most. Most of us have a short list already formulated from some stage in our careers. It may be someone we've worked with, or it may be someone whose work we admire. List the ten most important attributes of this role model. Choose the attributes that are the reason why you have chosen this person to be your role model. These attributes might be specific areas of skill, such as technology breadth, or the depth of their knowledge in some particular domain. Or, they might be more personal traits like the ability to make team members comfortable or that they are an engaging speaker.

Now, rank those qualities in order of importance, with 1 being the least important and 10 being the most important. You have now created and distilled a list of attributes that you find admirable and important. These are the ways in which you should strive to emulate your chosen role model. But, how do you choose which to focus on first?

Add a column to the list, and for each item on the list, imagine how your role model would rate *you* on a scale of 1 to 10 (10 being the

best). Try to really put yourself into the mind of your role model and to observe yourself as if a third person.

When you have the attributes, ranking, and your own ratings, in a final column, subtract your rating in each row from the importance level you gave it in the preceding column. If you ranked something as 10, the most important attribute of your role model, and your rating was 3, that gives you a final priority score of 7. Having filled this column in completely, sorting in descending order will give you a prioritized top-ten list of areas in which you need to improve.

Start with the top two or three items, and put together a concrete list of tasks you can *start doing now* to improve yourself.

13 Be a Mentor

If you want to really learn something, try teaching it to someone else. There's no better way to crystalize your understanding of something than to force yourself to express it to someone else so that *they* can understand it. The simple act of speaking is a known elixir for treating an unclear mind. Speaking to puppets and other inanimate objects as a method of problem solving is a fairly well-known element of software development folklore.

To find out if you really know something, try teaching it to someone else.

I saw Martin Fowler⁸ give a talk to a room of developers in Bangalore, in which he said that whenever he wants to really learn about something, he writes about it. Martin Fowler is a well-known software developer and author. It could be said that he is one of the best-known

and influential *teachers* this industry has to offer if we consider his role as author to be that of a remote teacher and mentor.

We learn by teaching. It's ironic, because we expect a teacher to already know things. Of course, I don't mean we can learn new facts altogether by teaching them to someone—where would they come from? But, knowing facts is not the same as understanding their causes and ramifications. It's this kind of deeper understanding that we develop by teaching others. We look for analogies to express complex concepts, and we internally work through the reasons why one analogy seems to work but doesn't and another analogy would seem not to work but does. When you teach, you have to answer questions that may have never occurred to you. Through teaching, we clean the dusty corners of our knowledge as they are exposed to us.

So, just as you can benefit from finding a mentor, you can benefit from *being* a mentor to someone else.

Mentoring has positive social effects as well. An overlapping group of mentors and their mentees creates a tight and powerful social network. The mentor-to-mentee bond is a strong one, so the links in this kind of professional network are stronger than those of more passive acquaintances.

⁸No, we're not related.

When you are in a mentoring relationship with someone, you form an allegiance with each other. A network of this kind is a great place to circulate difficult problems or look for work.

You also shouldn't underestimate that it just *feels good* to help people. If you can hold the interests of others in mind, you will have actually done something altruistic with your skills. **Mentors tend not to get laid off.**

In the uncertainty of today's economic environment, actually *helping* someone is a job you can't be laid off from. And, it pays in a currency that doesn't depreciate with inflation.

You find a mentee not by going out and declaring yourself a guru but by being knowledgeable and willing to patiently share that knowledge. Don't be alarmed if you're not an absolute expert on a topic. Chances are that there is *something* that you have experience with that would qualify you to help someone less experienced. Find that thing, and start being helpful.

You might, for example, have done a sizable amount of PHP work. You could go to your local PHP user group meeting and offer to help less experienced users with their specific problems. Or, if you don't have an immediately available forum for providing face-to-face mentoring, you could simply start answering questions in an online message board or IRC channel or help people debug application problems. Keep in mind, though, that mentoring is about people. An online mentoring relationship can never compare to one that happens between two humans in the same place.

You don't have to set up a formal mentoring relationship to get these benefits. Just start helping people, and the rest will come naturally.

Act on it!

1. Look for someone to take under your wing. You might find someone younger and less experienced at your company. Perhaps an intern. Or, you could talk to the computer science or information systems department at your local university and volunteer to mentor a college student.
2. Find an online forum, and pick a topic. Start helping. Become known for your desire and ability to patiently help people learn.

14

Practice, Practice, Practice

When I was a music student, I spent long nights in my university's music building. Through the thin walls of the university's practice rooms, I was constantly immersed in some of the ugliest musical sounds imaginable. It's not that the musicians at my school weren't any good. Quite the contrary. But they were practicing.

When you practice music, it *shouldn't* sound good. If you always sound good during practice sessions, it means you're not stretching your limits. That's what practice is for. The same is true in sports. Athletes push themselves to the limit during workouts, so they can *expand* those limits for the real performances. They let the ugliness happen behind closed doors—not when they're actually working.

In the computer industry, it's common to find developers stretched to their limits. Unfortunately, this is usually a case of a developer being under-qualified for the tasks that he or she has undertaken. Our industry tends to practice on the job. Can you imagine a professional musician getting onstage and replicating the jibberish from my university's practice rooms? It wouldn't be tolerated. Musicians are paid to *perform* in public—not to practice. Similarly, a martial artist or boxer stressing himself or herself to fatigue during matches wouldn't go very far in the sport.

As an industry, we need to make time for practice. We in the West often make the case for domestic programmers based on the relatively high quality of the code they produce versus that of offshore teams. If we're going to try to compete based on quality, we have to stop treating our jobs as a practice session. We have to *invest the time* in our craft.

Several years back, I started experimenting with programming exercises modeled after my musical practice sessions. Rule number one was that the software I was developing couldn't be something I wanted to use. I didn't want to cut corners, rushing to an end goal. So I wrote software that wasn't useful.

I cut no corners but was frustrated to find that a lot of the ideas I had while practicing weren't working. Though I was trying to do as good a job as possible, the designs and code I was creating weren't as elegant as I had hoped they'd be.

Looking back on it now, I see that the awkward feeling I got from these

experiences was a *good sign*. My code wasn't completely devoid of brilliant moments. But I was stretching my mental muscles and building my coding chops. Just like playing the saxophone, if I sat down to practice and nothing but pretty sounds came out, I'd know I wasn't practicing. Likewise, if I sit down to practice coding and nothing but elegant code comes out, I'm probably sitting somewhere near the *center* of my current capabilities instead of the edges, where a good practice session should place me.

So, how do you know what to practice? What stretches your limits? The subject of how to practice as a software developer could easily fill a book of its own. As a start, I'll borrow again from my experience as a jazz musician. I'd break jazz practice down into the following categories (simplified for the nonmusicians among us):

Practice at your limits.

- Physical/coordination
- Sight reading
- Improvisation

These might serve as a framework for *one way* to think about practice as a software developer.

Physical/coordination: Musicians have to practice the technical aspects of their instruments: sound production, physical coordination (making your fingers move nimbly, for example), speed, and accuracy are all important to practice.

What equivalent do we software developers have of these musical fundamentals? What about the dusty corners of your primary programming language that you rarely visit? For example, does your programming language of choice support regular expressions? Regular expressions are an extremely powerful and tragically underutilized feature of many programming environments. Most developers don't use them when they could, because they don't have the level of skill that it would take to be productive with them. As a result, a lot of needless string parsing code gets created and then has to be maintained.

The same rules apply to your language's APIs or function libraries. If you don't get the environment's many tools under your fingers (as musicians say), it's less likely you'll pull them out when they could really help you. Try truly digging into, for example, the way multithreaded programming works in your chosen programming environment. Or, how about stream libraries, network programming APIs, or even the set of utilities avail-

able for dealing with collections or lists? Most modern programming languages offer rich and powerful libraries in all of these areas, but software developers tend to learn a small subset, with which they can less efficiently write the same code they could have written if they had mastered the full set of tools available to them.

Sight reading: Especially as a studio musician, the ability to read and play music near-perfectly the first time is paramount for a professional. I once played saxophone on a jingle for Blockbuster (the video rental company). I played both the lead and second alto parts on an up-tempo big-band song. I saw the music for the first time literally as the tape started rolling. We played through once on the lead part and once on the second part. Any mistakes, and the whole band had to do it again—and the cost of the studio time had to be accounted for.

As software developers, what would it mean to be able to sight read code? Or requirements specifications or designs? An excellent place to find new code with which to practice is the open-source community. Do you have any favorite pieces of open-source software? How about trying to add a feature? Go look at the TO-DO list for a piece of software you'd like to practice with, and give yourself a constrained amount of time to implement the new feature (or at least to determine what it would take to implement it).

After choosing a feature, download the source code for the software, and start exploring. How do you know where to look? What tricks do you use to find your way around a significant body of code? What's your starting place?

This is an exercise you can practice often and in short periods of time. You don't actually *have* to implement the feature. Just use it as a starting point. The real goal is to understand what you're looking at as quickly as possible. Be sure to vary the software you work with. Try different types of software in different styles and languages. Take note of issues that make it easier or harder for you to find your way around. What patterns are you developing that help you work through the code? What virtual bread crumbs do you leave for yourself to help you navigate as you move up and down the call stack of a complex piece of functionality?

Improvisation: Improvisation is taking some structure or constraint and creating something new, on the fly, on top of that structure. As a programmer, I've found myself doing the most improvisation in times of stress. *Oh no! The wireless network app server is down, and we're losing orders!* That's when some

of the most creative, impromptu programming happens. That's when you do crazy stuff like recovering lost data by manually replaying packets over a wireless network from a binary log file. Nobody meant for you to do these things, especially not in the heat of the moment. That kind of sharp and quick programming ability can be like a magical power when wielded at the right time.

A great way to sharpen the mind and improve your improvisational coding skills is to practice with self-imposed constraints. Pick a simple program, and try to write it with these constraints. My favorite exercise is to write a program that prints the lyrics to the tired old song *99 Bottles of Beer on the Wall*. How could you write such a program without doing any variable assignments? Or, how small of a program can you write that will still print the lyrics correctly? For an additional constraint, how *fast* can you code this program? How about practicing small, difficult problems with a timer?

This is just one limited perspective on how to practice. You can mine examples from any discipline, from visual arts to monastic religious practice. The important thing is to find *your* practice needs and to make sure you're not practicing during your performances (on the job). *You* have to make time for practice. It's *your* responsibility.

Act on it!

1. *TopCoder*—TopCoder.com is a long-standing programming competition site. You can register for an account and compete online for prizes. Even if you're not interested in competing with others, TopCoder offers a practice room with a large collection of practice problems that you can use as excellent fodder for your practice sessions. Go sign up and give it a try.
2. *Code Kata*—Dave Thomas, one of the Pragmatic Programmers (our beloved publisher), took the idea of coding practice and made something...well, pragmatic out of it. He created a series of what he calls *Code Kata*, which are small, thought-provoking exercises that programmers can do in the language of their choice. Each kata emphasizes a specific technique or thought process, providing a concrete flexing of one's mental muscles.

At the time of this printing, Dave has created 21 such kata and has made them available for free on his weblog (<http://blogs.pragprog.com/cgi-bin/pragdave.cgi/Practices/Kata>).

On the weblog, you'll also find links to a mailing list and to others'

solutions to the exercises along with discussion about how the problems were solved.

Your challenge: work through all 21 kata. Keep a diary (perhaps a weblog?) of your experiences with the kata. When you've finished working through all 21 exercises, write your *own* kata and share it with others.

The Way That You Do It

“Developing software” is not a thing, a noun. Instead, “developing software” is a *verb*; it’s the *process* of creating a thing. When we’re coding away, it’s as important to focus on the process we’re using as it is to focus on the product being developed. Take your eye off the process, and you risk delivering late, delivering the wrong product, or not delivering at all. These outcomes tend to be frowned on by our customers.

Fortunately, a lot of thought has been put into the process of making good software (and products in general). Much of this prior art has been codified into a group of *methodologies*. These methodologies are the subject of numerous books that can be found online or in your local bookstore.

Unfortunately, most developers don’t get to benefit from all this good information. For the majority of teams, the process is an afterthought or something imposed from above. The word *methodology* has, in their minds, become synonymous with paperwork and long, meaningless meetings. All too often, a methodology is something that their managers impose.

Managers intuitively know that they need to follow *some kind of* process, but they often don’t know about the options that are now available. As a result, they dust off the same processes that were imposed on them in the 1980s, wrap them up in buzzword-compliant ribbons (the pastel-colored Agile ribbon is a good choice at the moment), and pass the practices on to their teams. And unless someone breaks the cycle by actually doing research on what works and what doesn’t, the same process will happen again as the developers on the team become managers themselves.

You’d think that there must be a better way to develop software. And for most teams there is.

If you’re a programmer, tester, or software designer, you may not think the development process is your responsibility. As far as your company is concerned, you’re probably right. Unfortunately, it’s usually *nobody’s* responsibility. If it does get assigned to someone, it might fall into the hole of a “process group” or some other similarly disconnected organization. The truth is that for a software process to have any chance of being implemented successfully, it has to be embraced by the people who are using the process. People like you.

The best way to feel ownership of these processes is to help implement them. If your organization has no process, research methodologies that

might work for you. Have brown-bag lunches with your team and discuss current development problems and ways that adopting a standard process might mitigate them. Put together a plan for rolling the chosen process into your organization and get everyone's buy-in. Then start to implement your plan.

If you want to feel you own a process, help implement it.

Alternatively, you might work in an environment where a process is passed down from on high. By the time the tablets arrive at the development team, the practices have often been watered down and reinterpreted to the point where they're unrecognizable from the originals. The process has suffered the same fate as the secret phrase in a game of Chinese Whispers.⁹ Again, this is an opportunity to take the initiative. Research the methodology you've been given and help interpret what it really means, both to your team and to your management. You're not going to be able to fight that a process has been imposed, so you may as well make it work by doing it right.

The methodology world can quickly begin to sound like a hollow shell of buzzwords. But, as buzzword compliant as some may be, you can always learn *something* from the study of a software process—even if that something is what *not* to do. If you're well versed in the software process landscape, you can make a more credible argument for how your team should be working.

Even with the abundance of prescriptive methodologies to choose from, it's not likely you'll ever work for a company that fully implements any of them. That's OK. The best process to follow is the one that makes your team most productive and results in the best products. The only way to find that hybrid (short of revelatory epiphany) is to study the available options, pick out the pieces that make sense to you and your team, and continuously refine them based on real experience.

Ultimately, if you can't do the process, you can't do the product. It's much easier to find someone who can make software work than it is to find someone who can make the *making of* software work. So, adding knowledge of the software development process to your arsenal can only help you.

⁹http://en.wikipedia.org/wiki/Chinese_whispers

Methodologies: Not Just for Geeks

Other process disciplines that can set the course for a software project. Though project management is not necessarily bound to software development methodology, you may find yourself running face first into your company's project management techniques. Numerous project management methodologies are in use throughout the industry. Probably most notable is the Project Management Institute's *Project Management Book of Knowledge*,* (with its widely recognized certification program).

Six Sigma† is another non-software-specific quality methodology. Driven by companies such as General Electric and Motorola, the Six Sigma approach emphasizes the measurement and analysis of processes and products to drive customer satisfaction and efficiency. While not specific to software development, Six Sigma's rigorous and methodical approach offers many lessons that are directly applicable to your job as a programmer.

* <http://www.pmi.org/>

† <http://www.isixsigma.com/>

Act on it!

1. Pick a software development methodology, and pick up a book, start reading web sites, and join a mailing list. Look at the methodology with a critical eye. What do you think would be its strong and weak points? What would be the barriers to implementing it where you work? Next, do the same with another. Contrast their strengths and weaknesses. How could you combine their approaches?

16

On the Shoulders of Giants

Imitate. Assimilate. Innovate.

► Clark Terry, Jazz Trumpeter

As a jazz musician, I spent a lot of time listening to music. I didn't just play music in the background while I was reading or driving. I would really *listen* to the music. If jazz improvisation is all about playing what you hear over the chords of a song, then actually listening to music is a critical source of inspiration and knowledge of what works and what doesn't. What sounds great and what just sits there.

The vast history of jazz recordings serves as an incredible body of knowledge, there for the taking by anyone with the skill to hear it. Listening to music, therefore, is not a passive activity for a jazz musician. It is study. Furthermore, the ability to understand what you're hearing is a skill that you develop over time. My circle of musician friends actually did this kind of listening explicitly. We would have listening parties, where a bunch of jazz musician geeks would sit around listening to music and then discussing it. Sometimes we would play *name that improviser* where one of us would play a recording of an improvised solo and the rest of us would have to figure out, based on style, who the recorded improviser was.

We in the jazz world weren't special, of course. Classical composers do the same thing. So do novelists and poets. So do sculptors and painters. Studying the work of masters is an essential part of becoming a master.

When listening to jazz recordings, we would discuss the musical devices that improvisors would use to communicate their musical points. "Wow! Did you hear the way he started sidestepping at the end of the form?" or "That was really strange the way he was playing behind the beat on the bridge." These discussions would help us all distill and discover tricks that we could take with us to our next improvisation session and try.

Mine existing code for insights.

Software design and programming have a lot in common with the arts in this way. We can mine a huge body of existing code for patterns and tricks. The design patterns movement (see *Design Patterns* [GHJV95]) is focused on the discovery and documentation of reusable solutions to common software development problems. Design patterns have formalized the study of existing code, making the practice accessible to a great number of software professionals. Still, design pat-

terns address only a small subset of the kinds of learning we can enjoy through code reading.

How do other programmers solve particular problems algorithmically? How do others strategically use variable, function and structure naming? If I wanted to implement the Jabber instant messaging protocol in a new language, how might I do it? What creative ways can I find to handle inter-process communication between UNIX and Windows systems? These are the type of questions you can answer through the study of existing code.

Even more important than finding solutions to specific problems is the use of existing code as a magnifying mirror to inspect our own style and capabilities. Just as listening to a John Coltrane recording always reminded me of where I stood on the skill ladder as a saxophonist, reading the work of a *great* software developer has a similarly humbling effect. Nevertheless, it's not just about being humbled. As you're reading through code, you will find things that you would have never done. You will find things you might have never even thought of. Why? What was the developer thinking? What were his or her motivations? You can even learn from bad code with this kind of critical, self-aware exploration of an existing work.

Use existing code to reflect on your own capabilities.

The act of learning from the work that came before you works well in the arts world, because there is no hidden source code for a painting or a piece of music. If you can hear the music or see the piece of art, you can learn from it. Thankfully, as software developers we have access to a practically infinite array of existing software in the form of open-source software.

Enough open-source software is available that it would be impossible to ever actually read all of it. There are definitely some bad open-source projects, but there are also quite a few *great* ones. There is open-source code available implementing almost any task that can be done with software in almost every available programming language.

As you look at this code with a critical eye, you will start to develop your own tastes, just as you would for music, art, or literature. Various styles and devices will amuse you, amaze you, anger you, and (I hope) *challenge* you. If you're really looking for them, you'll find everything from tricks that make you more productive to design paradigms that completely change the way you approach a class of problems. Just as in the arts, by studying and learning from the habits of others, you will develop your own distinctive style of software development.

A positive side effect of reading code is that you will learn more about what already exists. When you have a new problem that needs solving, you might remember that “Oh, I saw a library that implements MIME type handling in such and such project.” If the licensing terms are right, you may save yourself a lot of time and your company a lot of money by becoming more aware of what’s already out there for the taking. You might be amazed to realize just how much money we waste in the software industry by reimplementing the wheel (*invention* would be too generous a word) over and over again.

Sir Isaac Newton said, “If I have seen further, it is by standing on the shoulders of giants.” Smart guys like Isaac know that there is much to be learned from those who came before us. Be like Isaac.

Act on it!

1. Pick a project, and read it like a book. Make notes. Outline the good and the bad. Write a critique, and publish it. Find at least one trick or pattern that you can use from it. Find at least one bad thing that you observed that you will add to your *What not to do* checklist when you’re developing software.
2. Find a group of like-minded people, and meet once a month. Each session have someone nominate some code to study—2 lines to 200 lines. Break it down. Discuss what’s behind it. Think of the decisions that went into it. Ponder the code that *isn’t* there.

Automate Yourself into a Job

For the first three months of our stay in India, Kelly and I lived in a hotel in Bangalore. One morning, we woke up and decided to go down to the hotel restaurant for breakfast, instead of the usual outdoor stand-and-eat South India fast-food fare. We took a small table near the window, with a picturesque view of the beautiful hotel garden.

As we were having our first cups of chai, I got the feeling you get when some kind of insect is crawling nearby, and you can see it out of the corner of your eye. Something is moving where you don't expect motion. In this case, it was on the top of the roof of the garden canopy outside. The garden was made up in oriental style, and a beautiful outdoor dining area was covered with a strong, permanent wooden canopy. The roof of the canopy was made of some kind of small, solid, stone-like shingles.

It wasn't an insect—it was a hotel employee. He was sliding on his belly across the roof of the dining area, cleaning each shingle by hand with a polishing cloth. We had a good chuckle over what seemed like an embarrassing inefficiency of the hotel. But, we soon discovered that this was a pattern. Not more than a week later, I walked out of our room to head to work, and I looked down and saw two hotel employees crouching in the lawn, very carefully cutting the grass—with scissors!¹⁰

And, it didn't stop at the hotel. I noticed that the street workers didn't use heavy equipment. In fact, in many cases they seemed to be using homemade equipment. For example, groups of people tearing up a road to install cabling were bunched together pounding the asphalt with bare metal rods. I saw the same swarm idea applied to building houses, painting billboards, and farming: India throws labor at problems.

It's tempting to look at the street workers or the roof polishers and think, *I sure could save them some money—or at least some time!—by recommending some machinery to them.* But, as my wife first pointed out to me (and many Indians validated later), introducing machinery would put a *lot* of people out of work. In one of those forehead-slapping moments, I realized that I was staring right into the heart of why I was in India in the first place: labor is *cheap* there.

¹⁰It should be noted that not everyone in India cuts their grass with scissors

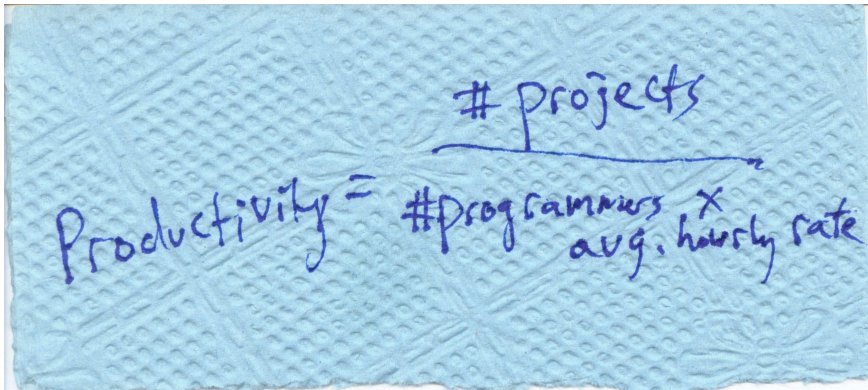
This is not just something to worry and complain about. These experiences offer an insight that can help us compete. I was in India for only a year and a half, but most of the Indian people who are the recipients of the U.S. IT industry's jobs *grew up there*. To them, it's perfectly normal to bash the street up with a metal pole or for 50 women to carry sand, one large bowl at a time, on their heads at a construction site.

Based on the positioning of the Indian IT labor force, it must also be perfectly normal to throw bodies at a software development project. *Need another application that looks like the one we're already making for you? No problem! We'll just add more programmers!*

There's more than one way to skin that cat. If the goal is to enhance software development throughput, you can either:

- get faster people to do the work,
- get *more* people to do the work, or
- automate the work.

Since we don't yet know how to truly measure software development productivity, it's hard to prove that one person is faster than another. So, as we discussed in the book's introduction, finance managers focus on per-hour costs. This leads to this simple (minded) formula, which assumes a fixed period of time:



$$\text{Productivity} = \frac{\# \text{ projects}}{\# \text{ programmers} \times \text{avg. hourly rate}}$$

In some environments, it's actually possible to calculate the true yield of a software investment. In most, you'll find squishy, amorphous measures such as *number of projects* or *number of requirements*, with no repeatable way of measuring one of those units.

So, the *faster programmer* approach is too hard to prove, and we *don't want to encourage the add more cheap programmers* approach. This leaves

us with automation. As we've seen, automation is not generally ingrained in Indian life. Going out on a limb, I'd assume that this applies in other developing countries. By contrast, we in the West have been automating ourselves out of work for years.

I remember the sensationalism surrounding job loss in the United States in the 1980s. Back then, not only were we blaming other countries, but we were blaming machines and, specifically, computers. Huge robotic arms were being installed in manufacturing plants. These robotic arms could outperform humans in both throughput and accuracy to a point that it was not even worth comparing them. Everyone was upset—everyone, that is, except for the people who *created* the robotic arms.

So, imagine your company is in the business of creating websites for small businesses. You basically need to create the same site over and over again, with contacts, surveys, shopping carts, the works. You could either hire a small number of really fast programmers to build the sites for you, hire an army of low-cost programmers to do the whole thing manually and repetitively, or create a system for *generating* the sites.

If we plug some (made-up) numbers into our finance manager's formula, we get the equations shown in Figure 1, on the next page.

You see, automation even makes sense to the finance manager, and he never seems to understand anything!

Automation is part of the DNA of the West. It's one true advantage that our pedigree affords. How can you *provably* make better software faster and cheaper than your offshore competition? Make the robots. Automate yourself into a job.

Act on it!

1. Pick a task you normally do repetitively, and write a code generator for it. Start simple. Don't worry about reusability. Just make sure your generator saves you time.
Think of a way to raise the level of abstraction of what you're generating.
2. Research Model Driven Architecture (MDA). Try some of the available tools. Look for somewhere in your work to apply the *concepts* of MDA if not the full toolset.

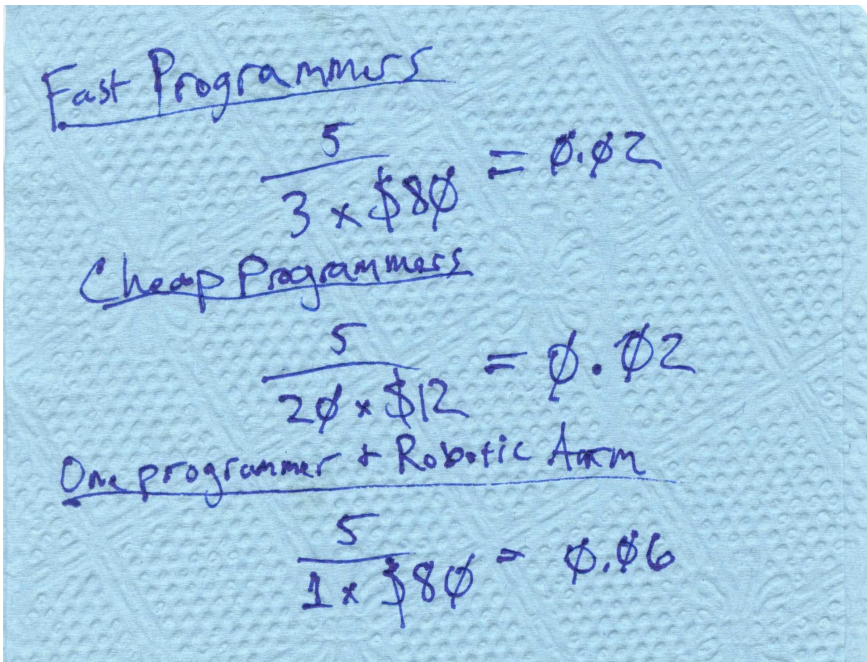


Figure 1: Productivity Comparisons

Part III

Executing

You've been making all the right investments and in the right market. You are becoming, for example, an expert in implementing Service Oriented Architectures for wirelessly enabled pizza delivery applications, and the pizza delivery industry is starting to boom like it has never boomed before. Before getting too wrapped up in amore with yourself, I should warn you that everything we've talked about so far is prep work. It all leads to this moment, where the sauce hits the dough: You have to actually *do something*.

Unless you're really lucky, you're probably not getting paid to be smart. And you aren't you getting paid to be a leading expert in the latest technologies. You work for an institution that is, most likely, trying to make money. Your job is to do *something* that helps the organization meet that goal. All of this careful thought and preparation has made you ready to show up at work and start kicking ass for your company.

Like the "I want to be a J2EE architect" guy from *Don't Put All Your Eggs in Someone Else's Basket*, on page 34, most of us don't find our identities in our associations with the companies we work for. I mean, I'm a programmer before I'm a person who helps a Fortune 500 company sell dishwashers, right? I'm an application architect—not a power company employee. From the perspective of viewing software as craft, this is not too surprising. The craft we've chosen isn't usually coupled with the domain in which we're applying that craft. An architect designing an office for a defense contractor is still an architect—not a defense contractor.

This identity observation creates some subtle problems, in that our macro-goals may conflict with our micro-responsibilities. If the architect creates an office that is dysfunctional for the defense contractor, he hasn't created something of value. Regardless of the aesthetic beauty of his creation, he's a bad architect.

We're being paid to create value. This means getting up out of our reading chairs and getting things done. To be successful, raw ability will get you only so far. The final stretch is populated by *closers*—people who finish things.

Getting things done feels *good*. It's often hard for people to get into a rhythm (try searching Amazon for *procrastination*), but once you've felt a fire under you, you won't want to stop. Let's start lighting the fire.

Imagine you are in a race with a \$100,000 cash prize. The first team that creates software to implement a new accounts receivable application wins the prize. You and your team at work have signed up to compete. The contest is to take place over a weekend. To win, your code has to be fully tested and implement a minimum set of specified features. You start on Saturday morning, and you have until Monday morning to complete the application. The first team to finish before Monday morning wins the race. If no team finishes before Monday, the team with the most features implemented wins.

You confidently peruse the application's feature requirements. Looking at the feature set, you realize that the system to be created is similar in size and scope to a lot of systems you've worked on in the past. While your team's agreed-upon goal is to finish some time mid-day on Sunday, for a fleeting moment, you start to question yourself. *How is it that an application of similar scope to those we spend weeks working on in the office is going to get finished in a single weekend?*

But as the opening bell sounds and you launch into coding, you realize that your team *is* going to be able to meet its goal. The team is collectively in a groove, churning out feature after feature, fixing each other's bugs, making split-second design decisions, and getting things done. It feels good. At design reviews and status meetings in the office, you've often daydreamed about taking a small team out of the bureaucratic environment and ripping through the creation of a new application in record time.

Many of us have this daydream. We *know* that our processes slow us down. Not only that, but we know that the speed of our environments cause *us* to slow down.

Parkinson's law states that "work expands so as to fill the time available for its completion." The sad thing is that even when you don't want it to be this way, you can fall into the trap, especially when there are tasks you don't really want to do.

What can we do? Right now?

In the case of a weekend coding race, you don't have time to put tasks off, so you don't. You can't delay making a decision, so you don't. You can't avoid the boring work, and you know that you have to do it so quickly that nothing can get *too* boring.

Parkinson's law is an empirical observation—not an unescapable human mandate. A sense of urgency, even if manufactured, is enough to easily double or triple your productivity. Try it, and you'll see. You can do it faster. You can do it now. You can get it done instead of talking about getting it done.

If you treat your projects like a race, you'll get to the end a lot faster than if you treat them like a prison cell. Create movement. Be the one who pushes. Don't get comfortable.

Always be the one to ask, "But what can we do *right now*?"

Act on it!

1. Look at your proverbial plate. Examine the tasks that have been sitting on it for a long time. The projects that are starting to grow mold. The ones you've been just a little bit *stuck* on—perhaps in bureaucracy, perhaps in analysis paralysis.

Find one that you could just *do* in between the cracks of your normal work, when you would normally be browsing the Web, checking your e-mail, or taking a long lunch. Turn a multimonh project into a less-than-one-week task.

I used to work with this guy named Rao. Rao was from a state in Southern India called Andhra Pradesh, but he was located in the United States and worked onsite with us. Rao was the kind of guy who could code anything you asked him to code. If you needed low-level system programming done, he was the guy you could ask. If you needed high-level application programming, he could also do pretty much anything you asked him to do.

However, what made Rao truly remarkable was what he did *before* you asked him to. He had this uncanny ability to predict what you were going to ask him to do and do it before you thought of it. It was like magic. I believe I even accused him of playing tricks on me at one point, but there's no way it could have been a trick. I would say, "Rao, I've been thinking about changing the way we're encapsulating the controller on our application framework. If we changed it just a tad, it could be used for applications other than web applications. What do you think?"

"I did that earlier this week," he would say. "It's checked into CVS. Have a look." This was *constantly* happening with Rao. It happened so often that the only way it could have been a coincidence is if Rao was literally doing *everything imaginable* with every piece of software that our team maintained.

At the time, I was leading my company's application architecture team. Among other things, we built and maintained frameworks for use in the company's applications. My teammates spent a lot of time talking about how we wanted to see the face of software development at the company improve. We also talked a lot about the role of our core infrastructure components in these improvements.

That's where Rao's magic trick came in. He didn't talk much in those conversations, but he was anything but disengaged. He was listening carefully. And, giving away his secret as no magician would, he later told me the trick was that he was only doing things that I had already said I wanted. I had just said them in ways that were subtle enough that *even I* didn't realize I had said them.

It's hard to read minds over a conference call.

We might be standing around waiting for a pot of coffee to brew, and I would talk about how great it would be if we had some new flexibility in

our code that didn't exist before. If I said it often enough or with enough conviction, even though I hadn't really put it on the team's TO-DO list, Rao might fill the gaps between "real work" looking at the feasibility of implementing one of these things. If it was easy (and cheap) to implement, he'd whip it out and check it in.

Stay off the mind
reading high-wire until
you have a safety net.

Mind reading not only applies to your managers but also to your customers. If they're giving you the right cues, you might be able to add features that they're either *going* to ask for or *would have* asked for if they had realized they were possible. If you always do what your customers ask for when they ask for them, you will satisfy your customers. However, if you do *more* than what they ask for or you have already done things before they ask, you will delight them. That is, unless your ability to read minds is defective.

This mind reading trick isn't entirely safe. It's a tight rope that you'll want to avoid walking unless you have left yourself a safety net. The major risks (with some suggested mitigations) are as follows:

- You spend the company's money doing work that nobody asked you to do. What if you were wrong? Start small. Only do the guesswork that you can fit in between the cracks of your normal job so the impact is little to none. If you're so inclined, take on these extra jobs in your free time.
- Anytime you add code to a system, you stand the very strong chance of making it less resilient to change. Avoid mind reading work that may force the system down a particular architectural path or limit what the system can do in some way. When the impact of change is great enough, a business decision needs to be made. And, it's seldom just the developers who need to weigh in on such a decision.
- You might take it upon yourself to change a feature your customers *did* ask for in a way that, unexpectedly to you, makes it less functional or desirable to the customer. Beware of guessing when it comes to user interfaces especially.

Managing people and projects is challenging work. People who can keep a project moving in the right direction without being given much guidance are highly valued and appreciated by their often overworked managers and customers. The mind reading trick, if done well, leads to people

depending on you—an excellent recipe for continued employment. It's a skill worth exploring and developing.

Act on it!

1. Karl Brophey suggests that: for your next project or a system you maintain, start making some notes about what you *think* your users and managers are going to ask for. Be creative. Try to see the system from their points of view. After you have a list of the not-so-obvious features that might come up, think about how you could most effectively implement each feature. Think about edge cases that your users might not immediately have in mind.

As you get into the project or enhancement requests, track your hit rate. How many of your guesses turned out to be features you were actually asked to implement? When your guessed features *did* come up, were you able to use the ideas you came up with in your brainstorming session?

We all like to believe, by virtue of our knowledge and that we're good software people, that we are going to naturally nail execution and be top performers. For a lucky few (and I *do* mean to use the word *luck* here), a strategy like this will work.

But, all of us can benefit from scheduling and tracking our accomplishments. Common sense tells us that if we exceed our managers' expectations, we'll be on the A list. Given that exceeding expectations is a worthy goal, surprisingly few of us have mechanisms of tracking how and when we exceed the expectations of our employers.

As with most tasks that are worth doing, becoming a standout performer is more likely with some specific and intentional work. When was the last time you went above and beyond the call of duty? Did your manager know about it? How can you increase the *visible* instances of this behavior?

Have an accomplishment to report every day.

James McMurry, a co-worker who's also a good friend,¹¹ told me very early in both of our careers about a system he had worked up to make sure he was doing a good job. It struck me as being remarkably insightful given his level of experience (maybe it's a hint he got from his parents), and I use it to this day. Without warning his manager, he started tracking *daily hits*. His goal was to, each day, have some kind of outstanding accomplishment to report to his manager—some idea he had thought of or implemented that would make his department better.

Simply setting a goal (daily, weekly, or whatever you're capable of) and tracking this type of accomplishment can radically change your behavior. When you start to search for outstanding accomplishments, you naturally go through the process of evaluating and prioritizing your activities based on the business value of what you might work on.

Tracking hits at a reasonably high frequency will ensure that you don't get stuck: if you're supposed to produce a hit per day, you can't spend two weeks crafting the *perfect* task. This type of thinking and work becomes a habit rather than a major production. And, like a developer addicted

¹¹<http://www.semanticnoise.com>

Monday - Automate the Build! ✓

Tuesday - Write tests for feed parsing code ✓

Wednesday - Look into Object/Relational Mapping tools
So we can stop writing all that SQL! ✓

Thursday - Script the web app deployment process

Friday - Clean up the project's compilation warnings

Figure 2: One Week of Hits

to the green bar of a unit test suite, you start to get itchy if you haven't knocked out today's hit. You don't have to worry so much about tracking your progress, because performing at this level becomes more akin to a nervous tic than a set of tasks that need to be planned out in Microsoft Project.

Act on it!

1. Block off half an hour on your calendar, and sit down with a pencil and paper in a quiet place where you won't be interrupted. Think about the little nitpicky problems your team puts up with on a daily basis. Write them down. What are the annoying tasks that waste a few minutes of the team's time each day but nobody has had the time or energy to do anything about?

Where in your current project are you doing something manually that could be automated? Write it down. How about your build or deployment process? Anything you could clean up? How might you reduce failures in your build? Write all of these ideas down.

Give yourself a solid twenty minutes of this. Write down all of your

ideas—good or bad. Don't allow yourself to quit until the twenty minutes are up. After you've made your list, on a new sheet of paper, write out your five favorite (most annoying) items. Next week, on Monday, take the first item from the list, and do something about it. On Tuesday, take the second item, Wednesday the third, and so on.

21

Remember Who You Work For

It's really easy to say, "Make sure your goals and your work align with the goals of your business." Really easy to say. Really hard to do, especially when you're a programmer, buried under so many organizational layers that you hardly know what your business is. Early in my career, I worked for a major package delivery company in a software development architecture team supporting the company's revenue systems. This company was so encumbered with hierarchy, I never saw anything in my daily work that gave me even a glimpse into the business of package delivery. I can remember my team attending quarterly all-hands meetings and feeling completely disjoint and alienated. "What is this achievement we're celebrating? What do all of these metrics mean?"

Granted, at that point in my career, I was more interested in building elegant systems and hacking open-source software than digging into the guts of a package delivery business. (OK, I admit it—I'm *still* more more interested in those things.) But, had I really wanted to align my work with the major goals of the organization, I'm not sure I would have known where to begin.

So, it's all fine and dandy to say we need to align our work with the goals of the company—to try to make sure we're impacting the bottom line and all that. However, truth be told, many of us just don't have visibility into how we can do this at the level from which we're grasping. We can't see the forest for the trees.

Maybe this one isn't our fault. We may be asking too much of ourselves. Maybe the idea of trying to directly impact the company's bottom line feels a bit like trying to boil the ocean. So, we need to take a more compartmentalized view, breaking the business into boilable puddles.

The most obvious puddle to start with is your own team. It's probably small and focused enough that you can conceptually wrap yourself around it. You most likely understand the problems the team faces. You know what your team is focused on improving, be it productivity, revenue, error reduction, or anything else. If you're not sure, you have one obvious place to go to find out: your manager.

Ultimately, in a well-structured environment, the goals of your manager are the goals of your team. Solve your manager's problem, and you've solved a problem for the team. Additionally, if your manager is taking

the same approach you are, the problems you're solving for him or her are really his or her boss's problems. And so on, and so on, until it rolls up to the highest level of your company or organization—the CEO, the shareholders, or even your customers.

By doing your small part, you're contributing to the fulfillment of the goals of your company. This can give you a sense of purpose. It gives your work meaning.

Some may resist this strategy. "I'm not going to do his work for him." Or, "She's just going to take credit for my work!"

Well, yea. Sort of. That's the way it works. The role of a good manager is not to, as Lister and Demarco say in *Peopleware* [DL99], "play pinch hitter," knowing how to do his or her whole team's job and filling in when things get difficult. The role of a good manager is to set priorities for the team, make sure the team has what it needs to get the job done, and do what it takes to keep the team motivated and productive, ultimately getting done what needs to get done. A job well done by the team is a job well done by the manager.

Your managers' successes are *your* successes.

If the manager's job is to know and set priorities but not to personally *do* all the work, then your job *is* to do all the work. You are not doing the manager's job for him or her. You're doing your job.

If you're really worried about who gets the credit, remember that it's your *manager* who holds the keys to your career (in your present company, at least). In most organizations, it's the direct manager who influences performance appraisals, salary actions, bonuses, and promotions. So, the credit you seek is ultimately cashed in with your manager.

Remember who you work for. You'll not only align yourself with the needs of the business, but you'll align the business with *your* needs.

Act on it!

1. Schedule a meeting with your manager. The agenda is for you to understand your manager's goals for the team over the coming month, quarter, and year. Ask how you can make a difference. After the meeting, examine how your daily work aligns to the goals of your team. Let them be a filter for everything you do. Prioritize your work based on these goals.

As a manager, I can tell you that the most frustrating thing to deal with is an employee who's always aiming for the next rung on the ladder. You know the guy: you can't sit with him for lunch without him bringing up who got what promotion. He always has some kind of office gossip to spread, and he seems to cling to corporate politics as if clinging to the story line of a soap opera to which he has some sick and sinful addiction. He complains about the incompetence of The Management and bitterly completes his tasks, knowing full well that he could do the job of management better than they can. They're just too incompetent to understand his potential.

He thinks many tasks are beneath him. He avoids them when possible and does them begrudgingly (and slowly) when not. He cherry-picks work that he thinks, even if subconsciously, is in tune with his level and might get him closer to his goal of the next promotion.

The sad thing about this guy is that, because he's living in the next job, he's usually doing a mediocre job in his current role. It's like mowing the lawn for me. I hate mowing the lawn.

Be ambitious, but don't wear it on your sleeve.

It makes me itch, and it makes me sweat. Worst of all, it keeps me from doing something I'd *rather* be doing. You can hire people to mow the lawn. I've been one of those people. That was a long time ago, and now I've graduated. So, when I *have* to mow the lawn, what do I do? I rush. I do a sloppy job. I spend the whole time thinking about how to get it finished so I can get on to the stuff I'd rather be doing. In a nutshell, I do a terrible job at mowing the lawn.

Thankfully, in my lawn mowing example, nobody is watching what I'm doing and grading me on it (though, my wife has become sufficiently annoyed that I'm not responsible for the lawn at our house anymore). It's my own problem if the lawn doesn't look great when I've finished. Nobody is holding me back to being "just a lawn mower" because of my performance in the yard. In the case of an IT job, that very same behavior could bring on a career catastrophe. Going back to our friend from the previous paragraphs, how do you think his management is going to view him? Will they see that they've been wrong to overlook his brilliance and decide to promote him? Will they give him big raises to try to make him happy?

Of course not. He's a mediocre performer with a bad attitude. *So what* if he has high potential? Right now, he's not showing it. The company doesn't make money because of potential. Shareholders don't hang onto investments if their potential isn't met. Furthermore, his attitude makes his managers want to stop investing in *him*.

So, that's a manager's view point. Now, of course, I'm not completely guilt-free here myself. I've been this guy myself to some extent. It really isn't a very good from this side of the street either. You spend all your time wanting something. Craving is the opposite of contentment. You wake up in the morning and have to go to "that bloody job" where nobody understands your potential. With resentment, you toil over your work, going over strategies for how to get ahead. You fantasize about what *you* would do in the latest situation that your manager screwed up—how you would handle it differently. You put off living while you're at work until you can do it *your* way in the position you deserve.

Here's a secret: that feeling will never end. If and when you finally land the big promotion you've been dreaming about, you'll quickly grow tired and realize that it's not *this* job you were meant for—it's the *next* one. The cycle begins again. I haven't reached the top yet, but I have a strong hunch that if there were such a position and I were to reach it, I would look ahead and realize I'd been chasing a ghost. What a frustrating waste of a professional life.

But, shouldn't we have ambitions? Would there be a Microsoft or a General Electric if the great entrepreneurs hadn't been ambitious and had goals?

Of course we should. I'm not advocating an apathetic outlook. It's good to have goals, and it's good to want to succeed. But, think of the negative, resentful guy I described at the beginning of this section. Do you think that guy is going to be the one who succeeds? It seems backward, but keeping your mind focused on the present will get you further toward your goals than keeping your mind focused on the goal itself.

It sounds difficult at first. Monk-like, even. Casting off the daily drive to succeed may sound like an ascetic, unattainable goal. You'll find that it's very pragmatic, though. Focusing on the present allows you to enjoy the small victories of daily work life: the feeling of a job well done, the feeling of being pulled in as an expert on a critical business problem, the feeling of being an integral member of a team that *gels*. These are what you'll miss if you're always walking around with your head in the clouds.

You'll always be waiting for *the big one* while ignoring the little things that happen every day that make your job worth showing up for.

Not only will *you* feel better, but those around you will feel it as well. Your co-workers, managers, and customers will feel it. It will show in your work. As unintuitive as it may be, letting go of your desire to succeed will result in an enhanced *ability* to succeed.

Your *presence* is a major advantage you have over your offshore brethren.

You are close to your clients. You are close to the leaders and decision makers who will shape your career in the short term and, possibly, the long term. Developers in India or the Phillipines don't have this advantage, but *you* do. So, *be* where you're *at*.

23

How Good a Job Can I Do Today?

It's rewarding to do a good job and to be appreciated. While most of us know this intuitively, we allow ourselves to be extremely selective about where and when we really go out of our way to excel. We dote over the design for the marketing department's *Next Big Thing* project, or we're quick to jump in to save the day in the face of some big, visible catastrophe, because our brains are wired to understand these moments as opportunities to show our proverbial stuff. We'll even do our work in the middle of the night with a level of focus and detail that would normally bore us to tears. A dire situation will often bring out the best in us.

I've let that intoxicating feeling of elation keep me awake and working effectively through some of the most grueling system outages and missed deadlines. Why is it that, without facing great pressure, we're often unable to work ourselves into this kind of altruistic, ultra-productive frenzy? How well would you perform if you could treat the most uninteresting and annoying tasks with the same feverish desire to do them right?

How much more *fun* could you make your job?

That last question may be better if we restate it. How much more *fun* would your job be if you could treat the most uninteresting and annoying tasks with the same feverish desire to do them right? When we have more fun, we do better work. So, when we have no interest in a task, we're bored and our work suffers as a result.

How can you make the boring work more fun? The answer to that question might be more apparent if you flip it around. Why is the boring work boring? Why isn't it *already* fun? What's the difference between the work you enjoy and the work you abhor?

For most of us techies, the boring work is boring for two primary reasons. The work we love lets us flex our creative muscles. Software development is a creative act, and many of us are drawn to it for this reason. The work we *don't* like is seldom work that we consider to be creative in nature. Think about it for a moment. Think about what you have on your TO-DO list for the next week at work. The tasks that you'd love to let slip are

probably not tasks that leave much to the imagination. They're just-do-'em tasks that you wish you could just get someone else to do.

The second reason that the boring tasks are boring, admittedly closely joined to the first, is that the boring tasks are not challenging. We love to dig in and solve a hard problem where others have failed. It's the same feeling that drives members of our species to recreationally risk their lives scaling mountains and bungee jumping off bridges. We love to do things to prove that we're able. The boring tasks are usually no-brainers. Doing them is about as challenging as taking out the trash.

So, how can we still use our creativity and challenge ourselves while tending to the mundane leftovers of our workday (which probably take up greater than 80% of the time for most of us)?

What if you tried to do the boring stuff *perfectly*? Say, for example, you hate unit testing. You love programming, but you get annoyed with having to write automated test code. What if you strove to make your tests perfect? How might that change your behavior? What does *perfect* even mean with regard to unit testing? It probably has something to do with test coverage. Perfect test coverage would mean that you had tested 100% of the functionality of your real code. Perfect unit tests are also clean and maintainable and don't depend on a lot of external factors that might be hard to replicate on another computer. They should be runnable directly after a fresh version control checkout on a new machine. And, of course, all of the tests should pass at 100%.

This is starting to sound difficult; 100% test coverage almost sounds impossible. And the business of decoupling the tests so that they can run without external dependencies presents a lot of challenges. In fact, you'll probably have to change your code to make this even possible. But, if you could do it, the tests would be incredible.

I don't know about you, but that sounds kind of fun to me. Granted, this is a manufactured example, but you can apply the same type of thinking to most of the tasks that cross your path. Try it tomorrow. Look at your workday and ask yourself, "How good a job can I do today?" You'll find that you'll like your job better, and your job will like you.¹²

¹²Thanks to Andy Hunt for this idea (<http://www.toolshed.com/blog/articles/2003/07/09/how-good>)

Act on it!

1. *Make it visible*—Turn those boring tasks into a competition with your co-workers. See who can do them better. Don't like writing unit tests? Print out the number of test assertions for the code you checked in every day, and hang it on your cubicle walls. Keep a scoreboard for the whole team. Compete for bragging rights (or even prizes). At the end of a project, arrange for the winner to have his or her grunt work done by the rest of the team for a whole week.

How Much Are You Worth?

Have you ever stopped to consider exactly how much you cost to the company you work for? I mean, you know your salary. That part is easy. What about benefits, management overhead, training, and all that other stuff that doesn't necessarily show up on your paycheck?

It's easy to get into the mode of just *wanting more*. It unfortunately seems to be a basic human tendency, in fact. You get a salary increase, and it feels good for a little while, but then you're thinking about the next one. "If I could make just 10% more, I could afford that new..." We've all done it. At some point, the actual number becomes abstract. It's not about \$5,000 more per year. It's about making whatever the baseline number is go up. If we don't get a satisfactory salary increase one year, we become dissatisfied with our work and our company. "Why don't they appreciate me?"

How much do you really cost? As I already mentioned, it's obviously more than your base salary. For the sake of discussion, let's estimate it at roughly two times your base salary. So, if you make \$60k per year, the company actually spends about \$120k keeping you employed.

That was easy. Now's the hard part: How much value did you produce last year? What was your *positive* impact on the company's bottom line? We already know that you cost the company (in our imaginary scenario) roughly \$120k. What did you give back? How much money did you cause the company to save? How much more in revenues did you contribute?

Is that number bigger than twice your salary?

It's a difficult exercise to go through, because it's often hard to relate every aspect of our work to the bottom line of the company. It may even seem like an unreasonable question to you. "How do I know? I'm just a coder!" That, of course, is the point. You work for a business, and unless you provide some kind of real value, you are a waste of money. It's easy to fall into the trap of thinking that salary increases are an entitlement. Analogously, a company has the right to charge more for its products every year. But, consumers have the right to *not purchase that product* if the price isn't attractive.

Now that you've started thinking about how much you cost versus how much you deliver, how much do you think you need to deliver to be considered a worthwhile investment to the company? We've talked about the

rough twice-your-salary figure, but is that enough? If you deliver value totaling twice your salary, the company has broken even. Is that a good way to spend money?

As a point of reference, think about the interest rate on a typical consumer savings account. It's not great, right? Still, it's definitely better than zero. Given the choice, would you put a year's worth of savings into a savings account that yields 0% or 3%? To deliver only twice your salary in value is as unappealing a prospect for a company as a 0% savings account is to you. They've tied up \$120k in cash for the year, and you're not even delivering enough value to keep up with the economy's inflation rate. Breaking even in this case is actually still a loss.

I can remember when I started thinking this way. It made me paranoid at first. A month would pass, and I would think, "What did I deliver this month?" Then, I started getting as granular as weeks and days. "Was I worth it today?"

Ask, "Was I worth it today?"

You can make this concrete. Just how much value *do* you add? Talk to your manager about how to best quantify it. The very fact that you *want* to quantify it will be taken as a good thing. How could you creatively save the company money? How could you make your development team more efficient? Or what about the end users of your software? You'll be surprised at how many opportunities you can spot if you start asking these questions. Now, start implementing some of them. Hold that figure in your head: *twice my salary*. Don't let yourself off the hook until you've surpassed that number for the year.

Act on it!

1. When companies make investments, they try to make sure they're using their money in the best possible way. Simply calculating a return on investment (I put in \$100 and get back \$120) isn't enough to make a smart decision. Among other factors, companies have to take inflation, opportunity cost, and risk into consideration. Specifically unintuitive to those of us who haven't been to business school is the concept of the time value of money. At risk of oversimplification, it goes something like this: a dollar today is worth more than a dollar next year, because a dollar today can be used to *generate more dollars*. Most companies set a *rate of return* bar, under which an investment will not be made. Investments have to yield an agreed-upon per-

centage in an agreed-upon period of time, or they aren't made. This number is called the *hurdle rate*.

Find out what your company's hurdle rate is and apply it to your salary. Are you a good investment?

A Pebble in a Bucket of Water

What would happen if you got up and walked out of your office, never to return? I know a lot of programmers who take comfort in imagining that scene. You just stand up, walk to your boss's office, and hand in your resignation. *I'll show them why they needed me!* This works as a daydream to get you through the really bad days, but it's obviously not a very productive attitude to carry with you all the time.

Beside that, it's not true. People leave companies every day. Many of them are let go. Many choose to leave. Some even live out your daydream and walk out with no notice. But in few cases do the companies they leave actually feel a significant impact as a result of their departure. In most cases, even in critical positions, the effect is surprisingly low. Your presence on the job is, to the company, like a pebble in a bucket of water. Sure, the water level is higher as a result. You get things done. You do your part. But, if you take the pebble out of the bucket and stand back to look at the water, you can't really see a difference.

I'm not trying to depress you. We all need to feel that our contributions mean something. And, they do. But, we spend so much time being *me* that we can easily forget that everyone else is a *me*, too. Everyone employed at your company walks around, a sentient and autonomous being, stuck in this thing called a *self*, which is the only window from which they see their jobs. Think of it this way: if you left tomorrow, the difference would be (on the average) no more or less impactful than if any of your co-workers left.

I once worked for a CIO who was one of the most powerful CIOs in one of the most powerful companies in the world. He and his team (of which I was a part) were winning every award and setting every IT standard in the company. This was a guy who had obviously figured out some kind of magic elixir and was sprinkling it into the free lunches and dinners that he had served during Y2K parties.

One of the few real pieces of advice that I ever got from this CIO—and I heard it over and over again—is that you should never get too comfortable. He professed to waking up every day and intentionally and explicitly reminding himself that he could be knocked off of his pedestal any day. *Today could be it*, he'd say.

His staff would look at him incredulously. No. Today couldn't be it. Things are going so well. You've got too much going for you.

That was his point. Humility is not just something we develop so we can claim to be more spiritual. It also allows you to see your own actions more clearly. What our CIO was teaching us was that the *more successful* you are, the

Beware of being
blinded by your own
success.

more likely you are to make a fatal mistake. When you've got everything going for you, you're less likely to question your own judgment. When the way you've always done it has always worked, you're less likely to recognize a new way that might work better. You become arrogant, and with arrogance you develop blind spots. The less replaceable you *think* you are, the more replaceable you are (and the less desirable you become).

Feeling irreplaceable is a bad sign, especially as a software developer. If you can't be replaced, it probably means you perform tasks in such a way that others can't also do them. While we'd all like to claim some kind of special genius, few software developers are so peerless that they in fact *should* be irreplaceable.

I've heard lots of programmers half-joking about creating "job security" with unmaintainable code. And, I've seen actual programmers attempt to do it. In every case, these people have become *targets*. Sure, it was scary for the company to finally let go of them. Ultimately, though, fear is the worst that ever came of it. Attempting to be irreplaceable is a defensive maneuver that creates a hostile relationship with your employer (and your co-workers) where one may not have already existed.

Using this same logic, attempting to be *replaceable* should create an unhostile working relationship. We're all replaceable. Those of us who embrace and even work toward this actually differentiate ourselves and, unintuitively, improve our own chances. And, of course, if you are replaceable nothing is stopping you from moving *up* to the next big job.

Act on it!

1. Inventory the code you have written or maintain and all the tasks you perform. Make a note of anything for which the team is completely dependent on you. Maybe you're the only one who fully understands your application's deployment process. Or there is a section of code you've written that is especially difficult for the rest of the team to understand.

Each of these items goes on your TO-DO list. Document, automate, or refactor each piece of code or task so that it could be easily understood by anyone on your team. Do this until you've depleted your original list. Proactively share these documents with your team and your leader. Make sure the documents are stored somewhere so that they will remain easily accessible to the team.

Repeat this exercise periodically.

Learn to Love Maintenance

When we set up our development center in Bangalore, we faced a challenge we never expected. Everyone wanted to make new systems. Nobody wanted to maintain old systems. And with the IT market booming in India for the past several years, we had to pay a lot of attention to what our people *wanted* to do with their careers.

Everyone likes creating. That's when we feel we are given the opportunity to really put our stamps on a piece of work. To feel like we own it. To express ourselves through our creation. We also tend to believe who project work is the most visible to our organizations. The people who build the new generation are the ones that must get the glory, right? I knew this attitude to be prevalent because of my experience back at home. But, in Bangalore, I saw this to an extreme that I never expected.

I think the motivation behind doing project work was almost caste-like in India. Though the caste system is officially dead in India, the hierarchical thinking it imposed lives on in the hearts and minds of its citizens. Programmers want to be designers want to be architects, and so on. And, the project-to-project mentality of the hired-gun offshore consultants emphasizes the insertion of a completed project in one's curriculum vitae. Maintenance work gives them neither a notch in their belts nor a clear, elevated role (such as *architect*) that they can tell their parents.

So, the motivating factors are the ability to be creative and the chance to make steps toward a promotion. The funny thing about it is that project work is *not* necessarily the best place to do either.

Maintenance work is typically littered with old, rotting systems and pushy end users. Since the software is thought of as being done, IT departments are usually focused on reducing the cost of maintaining these systems, so they look for the cheapest possible way to keep the systems running. That usually amounts to too few resources being assigned to look after the systems and no significant investment dollars being pumped into rejuvenating the systems.

Project work, on the other hand, is where you start with a nice, clean, green field. In a well-run company, every project contributes to either making or saving money, so the projects are usually funded sufficiently for the work to be done (though, experiences may vary here). There is no existing minefield of old code that the programmers have to tiptoe carefully through so

they can develop features “right” with fewer hinderances than if they were working on an existing system. In short, the circumstances in project land are usually much more ideal.

If I give you \$1,000 and ask you to go get me a cup of coffee, I’m going to be very unhappy if you return with 1,000 less dollars and no cup of coffee. I’m even going to be unhappy if you bring me plenty of really nice coffee, but it takes you two hours. If I give you \$0 and ask you to go get me a cup of coffee, I’ll be extremely appreciative if you actually return with the coffee, and I’ll be understanding if you don’t. Project work is like the first scenario. Maintenance is like the second.

When we don’t have the constraints of bad legacy code and lack of funding to deal with, our managers and customers rightfully expect more from us. And, in project work, there is an expected business improvement. If we don’t deliver it, we have failed. Since our companies are counting on these business improvements, they will often put tight reigns on what gets created, how, and by when. Suddenly, our creative playground starts to feel more like a military operation—our every move dictated from above.

Maintenance can be a place of freedom and creativity.

But in maintenance mode, all we’re expected to do is keep the software running smoothly and for as little money as possible. Nobody expects anything flashy from the maintenance crew. Typically, if all is going well, customers will stay pretty hands-off with the daily management of the maintainers and their work. Fix bugs, implement small feature requests, and keep it running. That’s all you have to do.

What if a bug turns up the need to redesign a subsystem in the application? That’s all part of bug fixing, right? The designs may be old and moldy, and broken windows¹³ may be scattered throughout the system. That’s an opportunity to put your refactoring chops to the test. How elegant can this system be? How much faster can you fix or enhance this section next time because of the refactoring you’re doing this time?

As long as you’re keeping it running and responding to user requests in a timely fashion, maintenance mode is a place of freedom and creativity. You are project leader, architect, designer, coder, and tester. You can flex your creative abilities all you like, and measurable success or failure of the system is yours to bear.

¹³For more on *broken windows*, see *The Pragmatic Programmer* [HT00].

When you're maintaining a system, you can also plan for more visible improvements. Your three-year-old web system might not take advantage of some of the snappy new user interface features available to modern web browsers. If you can work it in between keeping the system running and making bug fixes, you could visibly enhance the user experience with the system. Adding a few well-placed bells and whistles your customers weren't expecting is not too different from surprising your wife with flowers or, as a kid, cleaning the house while your parents were out shopping. And, without the bureaucracy of a full-blown project underway, you'll be surprised at just how much you can fit into those cracks. Your customers will be too.

A hidden advantage of doing maintenance work is that, unlike the contractual environment of many of today's project teams, the maintenance programmer often has the opportunity to interact directly with his or her customers. This means that more people will know who you are, and you'll have the chance to build a larger base of advocates in your business. It also puts you in a prime spot for truly learning the inner workings of your business. If you're responsible for a business application in its entirety, always working with its end users through problems and questions, chances are that even without much effort, you will come to understand what the application actually does as well as many of its business users. Business rules are encoded into application logic that businesspeople can't usually read. I've seen many situations where it was only the maintenance programmers who fully understood how a specific business process in a company worked. No one else had direct exposure to the authoritative encoding of that business logic.

The big irony surrounding the project versus maintenance split is that project work *is* maintenance. As soon as your project team has written its first line of code, each additional feature is being grafted onto a living code base. Sure, the code might be cleaner or there might be less of it than if you were working on a legacy application, but the basic act is the same. New features are being added to and bugs are being fixed in existing code. Who knows how to do this better and faster than someone who has truly embraced maintenance programming and made it a mission to learn how to do it well?

Act on it!

1. *Measure, improve, measure*—For the most critical application or code that you maintain, make a list of *measurable* factors that represent the quality of the application. This might be response time for the application, number of unhandled exceptions that get thrown during processing, or application uptime. Or, if you handle support directly, don't directly assess quality for the *application*, support request turn-around time (how fast do you respond to and solve problems) is an important part of your users' experience with the application.

Pick the most important of these measurable attributes, and start measuring it. After you have a good baseline measurement, set a realistic goal, and improve the application's (or your own) performance to meet that goal. After you've made an improvement, measure again to verify that you really made the improvement you wanted. If you have, share it with your team and your customers.

Pick another metric, and do it again. After the first one, you'll find that it becomes fun, like a game. Measurably improving things like this gets addictive.

Eight-Hour Burn

One of the many sources of controversy around the Extreme Programming movement is its initial assertion that team members should work no more than 40 hours per week. This kind of talk really upsets slave-driving managers who want to squeeze as much productivity as possible from their teams. It even kind of upset programmers themselves. The number of hours worked continuously becomes a part of the developer machismo, like how many beers a frat boy can chug at a kegger.

Bob Martin,¹⁴ one of the Extreme Programming community's luminaries, turned the phrase around in a way that made it much more tolerable for both parties while staying true to Kent Beck's original intent. Martin renamed *40-hour workweek* to "eight-hour burn." The idea is that you should work so relentlessly that there is no way that you could continue longer than eight hours.

Before we go too deeply into the burn, why the emphasis on keeping the number of hours down anyway? This chapter is about getting things done. Shouldn't we be talking about working *more* hours?

When it comes to work, less really can be more. The primary reason cited by the Extreme Programmers is that when we're tired, we can't think as effectively as when we're rested. When we're burnt out, we aren't as creative, and the quality of our work reduces dramatically. We start making stupid mistakes that end up costing us time and money.

Most projects last a long time. You can't keep up the pace of a sprint and finish a marathon. Though your short-term productivity will significantly increase as you start putting in the hours, in the long term you're going to crash so hard that the recovery time will be larger than the productivity gains you enjoyed during your 80-hour weeks.

Projects are marathons,
not sprints.

You can also think of your time in the same way you think of your money. When I was a teenager, working part-time jobs for minimum wage, I would have been happy to live off of the amount of money that I *waste* now. I have so much more money available to me now than I did when I

¹⁴<http://www.objectmentor.com>

was a teenager that I tend to be less aware as I spend each dollar. Somehow, I was able to survive back then. I had a place to live, a car to drive, and food to eat. I have the same things today. And, I don't lead a particularly extravagant lifestyle now. Apparently, when money was scarce, I found ways to be more efficient with my cash. And, the end result was essentially the same.

We treat scarce resources as being more valuable, and we make more efficient use of them. In addition to money matters, we can also apply this to our time. Think about day 4 of the last 70-hour week you worked. No doubt, you were putting in a valiant effort. But, by day 4, you start to get lax with your time. *It's 10:30 AM, and I know I'm going to be here for hours after everyone else goes home. I think I'll check out the latest technology news for a while.* When you have too much time to work, your work time reduces significantly in perceived value. If you have 70 hours available, each hour is less precious to you than when you have 40 hours available.

When the value of the dollar suffers from inflation, you need more dollars to buy the same stuff. When the value of the *hour* is deflated, you need more hours to *do* stuff. Bob Martin's eight-hour burn places a constraint on you and gives you a strategy for dealing with that constraint. You get to work and think, *I've only got eight hours! Go go go!* With strict barriers on start and end times, you naturally start to organize your time more effectively. You might start with a set of tasks that need to get done for the day, and you lay them out in prioritized order and start nailing them one at a time.

The eight-hour burn creates an environment that feels like that ultraproductive weekend you might have occasionally spent in college, cramming for a test in a course that you had been neglecting or jamming out a term paper that had fallen prey to procrastination. The difference is that this is constrained *cramming*. Times of cramming are usually extremely productive, because time becomes scarce and therefore extremely valuable. The eight-hour burn is a method of cramming early and often without having to stay up all night taking NoDoze and drinking Jolt Cola.

As thought workers, even if we're not in front of a computer or in the office, we can be working. You might be working while you're driving to dinner with your spouse or while you're watching a movie. Your work is following you around nagging you.

My work usually nags me when I haven't paid enough attention to it. I might be letting a specific task slip or letting tasks pile up and not taking

care of them. This is when the work follows me home and badgers me while I'm trying to relax. If you work intensely every day, you'll find that the work doesn't follow you home. Not only are you deliberately stopping yourself from working after-hours, but your mind will actually *allow* you to stop working after-hours.

Budget your work hours carefully. Work less, and you'll accomplish more. Work is always more welcome when you've given yourself time away from it.

As programmers, we know that the sooner in the development process that we can discover software failures, the more robust the software is going to be. Unit testing helps us ferret out the strange bugs as early as we can. If we discover bizarre errors in our own code, if they happen early, we are happy. Though they signify a small failure on our part as developers—we made a programming error—finding them early and often is a good sign of what the health of the software will become.

We are taught to allow our programming errors to be loud and messy early on. You want to know about them when they happen so you can put the correct fixes or defensive measures in place. When you're coding, you don't go out of your way to silence the little software failures that are destined to arise during development. That is the code's way of talking to you. Those little failures are part of the strengthening process. So, we add assertions that crash our programs when something goes wrong or unit tests that turn a bar red if we goof up.

The little failures we encounter also teach us what kind of failures to expect. If you've never walked through a minefield before, you might not know which lumps of dirt *not* to step on. If your software hasn't been complaining to you regularly, you might not know where the dangerous nooks and crannies are. You can program just so defensively when you're coding blind.

Furthermore, it's important to program defensively. Software quality is really put to the test when things go wrong. It's inevitable that *something* will happen for which you did not build a contingency case. Segfaults and blue screens in production code mean that the programmers didn't do a good job of dealing with the failures they couldn't foresee.

Every wrong note is but one step away from a right one.

The same principles apply on the job. A craftsman is really put to the test when the errors arise. Learning to deal with mistakes is a skill that is both highly valuable and difficult to teach. As a jazz improviser, I learned that every wrong note is at most one step away from a right one. What makes improvisations bad is when the improviser doesn't know what to do when the wrong note pops out. Standing in front of a band on one side and an audience on another, the sound of a real stinker of a note is enough to

freeze an amateur to the bone. Even the masters play wrong notes. But they recover in such a way that the listener can't tell that the whole thing wasn't intentional.

We're all going to make stupid mistakes on the job. It's part of being human. We make coding mistakes that lead to customers looking at stack traces. We paint ourselves into corners with critical design mistakes. Or, worse, we say the wrong things to our team members, managers, and customers. We make bad commitments or false claims about what we're capable of doing. Or we accidentally give our team members bad advice on how to solve a technical problem, wasting hours of their time.

Because we all make mistakes, we also know that everyone else makes mistakes. So, within reason, we don't judge each other on the mistakes we make. We judge each other on how we deal with those inevitable mistakes.

Whether it is a technical, communication, or project management mistake, the following rules apply:

- Raise the issue as early as you know about it. Don't try to hide it for any length of time. As in software development and testing, mistakes caught early are less of a problem than mistakes caught late. The earlier you suck it up and expose what you've done, the less the negative impact is likely to have.
- Take the blame. Don't try to look for a scapegoat even if you can find a good one. Even if you're not wholly to blame, take responsibility and move on. The goal is to move past this point as quickly as possible. A problem needs a resolution. Lingering on whose fault it is only prolongs the issue.
- Offer a solution. If you don't have one, offer a plan of attack for finding a solution. Speak in terms of concrete, predictable time frames. If you've designed your team into a corner, give time frames on when you will get back with an assessment of the effort required to reverse the issue. Concrete, attainable goals, even if small and immaterial, are important at this stage. Not only do they keep things moving from bad to good, but they help to rebuild credibility in the process.
- Ask for help. Even if you are solely to blame for a problem, don't let your pride make it worse by refusing help in a resolution. Your team members, management, and customers will look at you in a much more positive light if you can maintain a healthy attitude and set your ego aside while the team helps you dig your way out. Too often,

we feel a sense of responsibility that drives us to proudly shoulder a burden too large, and we end up thrashing unproductively until someone forcibly intervenes.

Think about the last time you experienced a customer service issue at a restaurant. Perhaps you waited way too long for the *wrong dish* to ultimately reach your table. Think about how the waiter reacted to your complaint.

Stressful times offer the best opportunities to build loyalty.

The wrong reaction is for the waiter to make excuses or to blame the cooks. The wrong reaction would be for the waiter to walk off to resubmit the order and stay out of sight while you sit there starving and wondering when the hell your food is finally going to arrive. Of the course, the *really* wrong reaction would be for the waiter to arrive with a dish that he already knows is wrong, hoping you would either not notice or not complain.

The difference between how a company treats us when they make a mistake can be the ultimate in loyalty building (or destroying). A mistake handled well might make us more loyal customers than we would have been had we never experienced a service problem. Remember this with *your* customers when you make mistakes on the job.

Say "No"

The quickest path to missing your commitments is to make commitments that you know you can't meet. I know that sounds patently obvious, but we do it every day. We are put on the spot and we don't want to disappoint our leaders, so we agree to impossible work being done in impossible timelines.

Saying "yes" is an addictive and destructive habit. It's a bad habit masquerading as a good one. But there's a big difference between a *can-do* attitude and the misrepresentation of one's capabilities. The latter causes problems not

Saying "yes" to avoid disappointment is just lying.

only for you but for the people to whom you are making your promises. If I am your manager and I ask you if you can rewrite the way we track shipments in our company's fulfillment system by the end of the month, chances are that I asked specifically about the end of the month for a reason. Someone probably asked *me* if it could be done by then. Or there might be another critical business change we're trying to make that is dependent on the fulfillment system. So, armed with your assurance that you can make the date, I run off and commit to my customers that it will be done.

Saying "yes" in this way is as good as lying. I'm not saying it's malicious. We lie to ourselves as much as we do to those we make the commitments to. After all, saying "no" feels bad. We are programmed to want to always succeed. And, saying we *can't* do something feels like we failed.

What we humans fail to internalize is that "yes" is not always the right answer. And, "no" is seldom the wrong answer. I say internalize, because I think we all *know* this to be true. After all, none of us wants to be the recipient of false commitments.

The inability to say "no" happens to be a common part of the Indian culture. Companies that are inexperienced with offshore outsourcing almost always run into it. You learn with time to sniff out uncertainty and ask the right questions. Enough "one more day until it's done" conversations naturally train you to probe deeper. And, it's not only a part of the IT culture. When I lived in Bangalore, I stayed home from work no less than five times waiting for a cable modem installation that never happened. It turned out that for the first three times the company didn't even have

the parts required to do the installation when they made the appointment. But, they didn't want to disappoint me. I told them I was hoping to have the cable modem installed next week, so they promised that it would be installed, knowing full well that the installation was not going to be possible next week.

Though the intent is positive, the ramifications are negative. I eventually got a little nasty with my cable modem installers and even made them come to my house on a holiday to do the installation. I didn't trust the promise that it would be installed “tomorrow, after the holiday.” Repeatedly missing commitments had destroyed any chance I had of trusting them. In fact, I'd developed a sense of hostility toward them.

On the other hand, what happens when you're asked to do a critical task and you say that you can't? As a manager of both onshore and offshore teams, I can tell you that “no” has become a source of relief to me. If I have a team member who has the strength to say “no” when that's the truth, then I know that when they say “yes,” they really mean it. A commitment from someone like this is going to be more credible and carry a lot of weight. If they actually hit the targets that they commit to, I'm not going to question them when they say they *can't* hit one.

If someone always says “yes”, they're either incredibly talented or lying. The latter is usually the case.

“I don't know” is also a great thing to say when it's appropriate. You might be responding to whether you can meet a date and need time to research the task before committing. Or you might be asked how a technology works or how some piece of your project's code is implemented. Just as in the case of commitments, not knowing the answer to something feels like a small failure. But, your co-workers and managers will have more faith in you when you claim to know something. You'll notice that when you meet a real guru in a subject area, they're never afraid to admit when they don't know something. “I don't know” is not a phrase for the insecure.

That same courage can also come in handy when dealing with decisions from above. How many times have you seen a technology decision dictated by a manager who caused the team members to sit around the table quietly looking at their shoes, waiting for the chance to escape the meeting room so they could complain to each other? Managers are often the target of the *Emperor's New Clothes* phenomenon. Everyone knows that a decision is bad, but they're all afraid to speak up. As a manager, I make decisions or strong suggestions all the time. However, I don't hire my employees

to be robots. It’s the ones who speak up and offer a better suggestion that become my trusted lieutenants.

Don’t go overboard with the “no” game. Can-do attitudes are still appreciated, and it’s good to have stretch goals. If you’re not sure you can do something but you want to give it a try, say that. “This is going to be a challenge, but I’d like to give it a try” is a wonderful answer. Sometimes, of course, the answer is simply “yes.”

Be courageous enough to be honest.

Act on it!

1. Karl Brophay, a reviewer, suggests keeping a list of every commitment you make.
 - What was asked of you for a due date?
 - What did you commit to?
 - If you were overridden, record both what you thought and what you were told to accept.
 - Record when you delivered.

Examine this daily. Communicate where you’ll fail as soon as you know. Examine this monthly—what is your hit rate? How often are you right on?

Say It, Do It, Show It

The easiest way to never get anything done is to never commit to anything. If you don't have a deadline, you don't have any pressure or much incentive to finish something. This is especially true when the *something* that needs to get done isn't 100% exciting.

Even a bad manager's instinct usually tells them that it's important to plan. For some developers, the invocation of the word *plan* is cause for alarm. Endless meetings with pointy-haired bosses creating reams of printed Microsoft Project plans that nobody understands or uses are a valid cause for alarm. So, techies often overcompensate in our rebellion against perceived overplanning by constantly flying by the seat of our pants.

Planning isn't such bad-tasting medicine that we should have to hold our breath to force it down. Planning can be a liberating experience. When you have too much to do, a plan can make the difference between confused ambiguity at the start of a workday and clear-headed confidence when attacking the tasks ahead.

Plans don't have to be big and drawn out. A list in a text document or e-mail is perfectly fine. Plans don't have to cover a large span of time. Being able to start the day and answer the question, "What are you going to do today?" is a great first step. I know many people whose days stay so hectic that they would almost always fail this test. A good first step would be to find time this afternoon and list everything you want to get done on the next workday and arrange them in priority order. Try to be realistic about what can fill a day, though you're likely to be wrong and specifically likely to overcommit yourself.

You can be as detailed or as loose as you want with your one-day plan. I had a roommate in college named Chris who would wake up every morning and, even at risk of being late for his first class, would meticulously plan out his day, specifically focusing on his piano practice schedule (he was a jazz piano major). His schedule was fairly rigid already with the selection of classes he had to attend. Still, Chris would actually plan down to how he was going to use the fifteen minutes *between* classes to fit in practice routines that could be done quickly. Many of his classes were in the same building, so it was common to have plenty of leisure time in between them for some quick socializing or grabbing a drink from the vending

machines. Chris would be cramming in scales or ear training while the rest of us were sitting around waiting for the next class to start. He would even divide his schedule into multiple three-to-five-minute segments, so he could fit more than one practice exercise into a given ten-minute period. Chris ended up becoming one of the most respected musicians in our city. Natural talent had something to do with it, of course, but I've since held the belief that he planned and executed his way into the musical elite.

So, you've made your plan. It may not be as detailed as Chris's, but it's enough to answer the question of what you're going to do with your day. When you get to work tomorrow, pull out the list and start on the first item. Work through the list until you go to lunch, and then pick up where you left off and try to finish the list.

As you complete each item on the list, mark it DONE. Use capital letters. Say the word, *done*. Be happy. At the end of the day, look at your list of DONE stuff and feel like you've accomplished something. Not only did you know what you were going to do today, but now you know what you've *done*.

If you didn't get everything done, don't worry about it. You knew you weren't going to be right about how much would fit in a day anyway. Just move the incomplete items from today (if they're still relevant) onto tomorrow's list, and start the process again. It's a stimulating process. It's rhythmic. It allows you to divide your days and weeks into a series of small victories, each one propelling you to the next. You'll find that not only does it give you visibility into what you're accomplishing but you'll actually get *more done* than if you weren't watching things so closely.

Having established a rhythm of plan and attack, you are ready to start thinking in terms of weeks and even months. Of course, the larger the time span you're planning for, the higher level your plan should get. Think of week and day plans as being tactical battle plans, with thirty, sixty and ninety-day plans focusing on more strategic goals that you want to accomplish.

The very act of thinking about what you want to have accomplished in ninety days is something unnatural for software soldiers on the field. We are tactical people. Forcing yourself to imagine an end state for your system, your team's processes, or your career after ninety days will cause things to surface that you never expected. The view from above the field shows us very different things than the view from the ground. It will be difficult at first, but stick with it. Like all good skills, it gets easier with

practice, and the benefits will be visible to both you and those who work with you (even if they don't know you're doing it).

Status reports can help you market yourself.

You should start communicating your plans to your management. The best time to start communicating the plans is after you have gotten through at least one cycle of the plan through execution. And—this is an important point—start doing it before they ask you to do it. No manager in his or her right mind would be unhappy to receive a *succinct* weekly e-mail from an employee stating what was accomplished in the past week and what they plan to do in the next. To receive this kind of regular message unsolicited is a manager's dream.

Start by communicating week by week. When you've gotten comfortable with this process, start working in your thirty, sixty, and ninety-day plans. On the longer views, stick to high-level, impactful progress you plan to make on projects or systems you maintain. Always state these long-term plans as proposals to your manager, and ask for feedback. Over time, these anticipation attempts will require less tweaking from your managers as you learn which items usually go unedited and which are the subject of more thrashing.

The most critical factor to keep in mind with everything that goes onto a plan is that it should always be accounted for later. Every item must be either visibly completed, delayed, removed, or replaced. No items should go unaccounted for. If items show up on a plan and are never mentioned again, people will stop trusting your plans, and the plans and you will counteract the effectiveness of planning. Even if the outcome is *bad*, you should communicate it as such. We all make mistakes. The way to differentiate yourself is to address your mistakes or inabilities publicly and ask for help resolving them. Consistently tracing tasks on a plan will create the deserved impression that no important work is getting lost in the mix.

Get this process going, and suddenly in the eyes of your management you have exposed your strategic side. Creating and executing plans shows that you are not just a robot typing code, but you are a *leader*. It's this kind of independent productivity that companies need as they reduce overhead.

A final benefit of communicating in terms of plans is that your commitments become more credible. If you say what you're going to do and then you do it and show that it's done, you develop a reputation for being a *doer*. With credibility comes influence. Imagine you want to introduce a

new process, such as an agile development practice,¹⁵ into an organization or you want to bring in a new technology. With the proven ability to make and meet commitments, you'll be granted more leeway to try new things.

In our Bangalore software center, we had a team that had been working night shifts for more than a year. Of the seven members on the team, two were always on the night shift. They rotated weekly, so every third or fourth week, each team member would switch to a 7 PM–3 AM schedule. The team was getting frustrated and burnt out, saying that they almost constantly felt jet-lagged. But, the team was playing a critical support role, and the team's U.S.-based internal customers were convinced they couldn't get by without live real-time help from the group in Bangalore.

So, the team put together a plan of attack. They looked at their various support processes and associated measurements and crafted a plan to both switch back to a single-day shift *and* to make significant improvements in their customer experience, simultaneously. As acting operations leader of the software center, I helped them fine-tune their plan and was present (as moral support) for the formal proposal they made to their manager in the United States.

They knew this was going to be a touchy subject for their manager, who had to answer to his U.S.-based customers in person. There was naturally much trepidation among the team members as the meeting started. However, the team's manager was so impressed that he immediately and happily signed off on the proposal, and the team put its plan into action. Within weeks, the jet lag was over, and everyone was back on day shifts.

The solidity of their plan for how to not only deal with the change in work hours but how they were going to strategically improve the performance of their team inspired great confidence in the leaders and, eventually their customers. The team's manager used the plan when communicating the change with his customers. And, the team followed through. Within months, the team was running at a new level of efficiency. They've since gained such credibility and confidence that they have taken more and more ownership and independence over the workings of their team.

The team used its plan as a concrete response to a problem. They came to their manager not with complaints but with proposed solutions.

Your leaders want you to have independence and ownership. Making, executing, and communicating plans will help you attain both.

¹⁵<http://www.agilemanifesto.org>

Part IV

Marketing...Not Just for Suits

You are the most talented software developer you know. Elegant designs flow out of the seemingly unending river of your creativity. Your architectural insightfulness is unmatched in your workplace. You can code faster and more accurately than anyone your company has ever employed.

So what?

Many software developers—especially the most conceited ones, it seems—live with the misconception that their skill should be self-evident to any clued-in manager or employer. They are able to comfortably veil this lie inside the cloud of a make-believe moral ethic: they're just too "humble" to market their own abilities. Going out of their way to make their abilities known would be *brownnosing*. No self-respecting programmer would be caught dead sucking up to The Man.

This is all just an excuse. Actually, they're afraid.

Most programmer types were the last kids picked for every team when they were in school. They probably avoided social situations where possible and failed miserably where not possible. It's no surprise that these people are afraid to stick their necks out by trying to show someone their capabilities.

Suspending disbelief for a minute, let's pretend the moral ethic nonsense isn't such a put-on after all. Regardless of one's intentions, it's stupid not to let people know what you're capable of doing. Think of it this way: you are employed to develop software that adds value to a company. The job of a leader is to develop teams that deliver the maximum amount of value to the company. How is a leader to do his or her job without knowing who in an organization is capable of what kind of work?

As one manager told me recently, if someone does something truly fantastic and nobody knows about it, in his eyes, it didn't happen. It may sound ruthless, but from a company's perspective it makes sense. Pragmatically speaking, managers don't have time to keep close tabs on what each employee is doing every day. And neither companies nor their employees would want managers spending their time this way. Companies want managers focusing on the big picture—not tracking daily tasks. And employees (especially programmers) hate to be micromanaged.

In short, you may have the best product in history, but if you don't do some kind of advertising, nobody is going to buy it. We all know—especially in the software world—that the best product doesn't always win. There's a lot more to success in the marketplace than having a great product. Let's not forget this truth in the job market.

Enough already...what should I do?

On the surface, marketing yourself is simple. You have only two goals: to let people know you exist and to let them know you are the person who can solve the tough problems that keep them up at night. This applies not only to the job market at large but also to the company at which you currently work. Don't assume that just because you're employed with a company, its management knows who you are. Furthermore, don't assume that just because a leader knows your name that he or she has even the faintest understanding of your capabilities.

This part will not only help make sure your current leaders understand what you're capable of but it will show you how to expand your scope to the industry at large. In the book so far, we've talked about how to be marketable. Now we're going to learn how to put that marketability into action.

Perceptions, Perschmeptions

It's comfortable to play the idealist and pretend you don't care what other people think about you. But, that's a game. You can't let yourself believe it. You *should* care what other people think about you. Perception is reality. Get over it.

You probably know the old clichéd philosophical question, "If a tree falls in the forest but nobody is there to hear it fall, did it make a sound?" The correct answer to the question is, "Who cares?"

I mean, the fall probably made a sound. That's not a very exciting answer on a metaphysical level, but it probably did. But, if nobody heard it fall, then the fact that it made a sound doesn't really matter.

The same goes for your work. If you kick ass and no one is there to see, did you really kick ass? Who cares? No one.

In the subculture of Indian IT bureaucracy, I was amazed at how people just didn't *get* this simple truth. Almost everyone I dealt with there didn't understand why it should matter that their managers, for example, knew what they were doing. If *you* knew you were better than so and so, then it should be reflected in your performance reviews, ratings, and salary. They had fooled themselves into thinking that how other people perceived them was somehow subservient to the *truth*, whatever that was.

This truth thing...what is it? Who defines it? What is good and what is bad in an absolute sense?

The answer is that there is no absolute good or bad, at least not in terms of judging who is better at a creative, knowledge worker job. How do you define what makes a good song? What about a good painting? You might have your own definitions, but I doubt I would agree with them. They're subjective.

Horrible risk-averse human resources departments in horrible risk-averse companies spin their wheels chasing objective measures of the people they employ. Sometimes they even implement "objective" appraisal systems. All of my team members in India thought *they* wanted to be measured this way. That's because they had never experienced it before.

Performance appraisals are never objective.

There is no way to objectively measure the quality of a knowledge worker, and there is no way to objectively measure the quality of their work. Go ahead. Disagree. Now think about your argument for a while. See the holes?

So, if the measure of your goodness at your company (or in the industry or the job market or wherever) is subjective, what does that mean? That means that you are always going to be measured based on someone else's *perception* of you. Your potential promotions or salary increases—even the decision of whether you should continue to be on the payroll at all—is completely dependent on the perceptions of others.

Subjectivity, being based on *personal* taste, implies that you can't count on any two opinions being the same. Different people are impressed with different factors. Some people might like rigid structure while others prefer loose, free creativity. Some may prefer to communicate via e-mail and others face to face or by phone. Some managers may like their employees to be aggressive while others prefer them to act like a subordinate. You say "Poh tay toh"—I say "poh tah toh."

It doesn't come down only to personal preference. People in different roles and relationships to you build their perceptions based on the qualities most important to making *that particular* relationship work well. If I'm a project manager, the quality of your source code is a lot less important to me than the quality of your communications. If I'm a fellow programmer, your raw ability and creativity drive my perception of you more than, for example, your follow-through. But, if I'm your manager, raw ability is ultimately meaningless to me unless you actually *do* something with it.

We've culturally trained ourselves to *perceive* that managing perception is somehow a dirty and shameful activity. But, as you can see, managing perception is just practical. When you explicitly take note of the factors that drive other people's perceptions of you, you more firmly discover how to make them happy customers. You're not going to impress your nontechnical business client with your object-oriented design skills. You might be a design genius, but if you can't communicate effectively and you don't manage to complete your work on time, your customers will think you stink. It's not their fault. You *do* stink.

Perceptions really do matter. They keep you employed (or unemployed). They get you promoted or get you stuck in the same job for years. They give you raises or lowball you on salary. The sooner you get over yourself and learn to manage perceptions, the sooner you'll be on the right track.

Act on it!

1. Perceptions are driven by different factors, depending on who the audience is. Your mother doesn't much care how well you can design object-oriented systems, but your teammates might.

Understanding what's important in each of your relationships is an important part of building credible perceptions with those you interact with. Think about the different classes of relationships you generally have with people in the office. For example, you probably have teammates who do the same type of job you do. You also probably have a direct manager, and you may have one or more customers, and a project manager.

Take these different groups (or whichever actually apply given the structure of your workplace), and list them. Next to each, write down which of your attributes is most likely to drive that group's perception of you. Here's an example:

Group	Perception Drivers
Teammates	Technical skills, social skills, teamwork.
Manager	Leadership ability, customer focus, communication skills, follow through, teamwork.
Customers	Customer focus, communication skills, follow through.
Project manager	Communication skills, follow through, productivity, technical skills.

Put some thought into your own list. How might you change your behavior as a result of this list? In what ways have you already been adjusting your focus as you interact with each group? In what ways have you *not* been appropriately adjusting your behavior?

At risk of stating the obvious, the most important, aspect of getting the word out in the workplace is your ability to communicate. Gone are the days of the disheveled hacker, crouching over his terminal, coding by the light of his monitor in the deepest bowels of the server room. The occasional monosyllabic grunt between feats of wizardry isn't gonna cut it.

As disturbing a proposition as it may be, put yourself into the mind of a manager or customer (I'll just use the word *customer* throughout this section to refer to both).

They're responsible for something gravely important which they ultimately have to entrust to some scary IT guys for implementation. They do what they can to help move things along, but ultimately they're at the mercy of these programmers. Moreover, they have no idea how to control them or even to communicate intelligently about what it is that they're doing. In this situation, what's the most important attribute they'll be looking for in a team member? I'll bet you the price of this book it's not whether they've memorized the latest design patterns or how many programming languages they know.

They're going to be looking for someone who can make them comfortable about the project they're working on.

Your customers are
afraid of you.

These managers and customers we're talking about have a dirty little secret: they are *afraid* of you. And for good reason. You're smart. You speak a cryptic language they don't understand. You make them feel stupid with your sometimes sarcastic comments (which you might not have even intended to be sarcastic). And, your work is often the last and most important toll gate between a project's conception and its birth.

Your job is to be your customer's tour guide through the unforgiving terrain of the information technology world. You will make your customers comfortable while guiding them through an unfamiliar place. You will show them the sights and take them where they want to go while avoiding the seedy parts of town you've encountered in the past.

Nonprogrammers are, on the average, as intelligent as programmers. (That is to say that most of them aren't very intelligent, but a few of them

really are.) Chances are high that your customer is just as smart as you but just doesn't happen to know how to program a computer. That's OK. You probably don't know how to do much of what he or she does on a daily basis. That's why there are two of you, and you're both being paid to come to work.

I mention the bit about intelligence because computer people all too often assume that anyone who doesn't know how to operate a computer is not intelligent. Saying it explicitly like this makes it sound idiotic, but that's true of all prejudices. However, this feeling is so ingrained in many of us that we don't even know when we're feeling it. Explicitly trying to control it doesn't work.

My advice is to reverse the relationship. Instead of feeling like you are the computer genius, descending from computer heaven to save your poor customer from purgatory, turn the tables around. If you're, for example, working in the insurance industry, think of your customer as a subject matter expert in insurance from which *you* have to learn in order to get *your* job done.

That being said, you need to be cognizant that your customers may need topics toned down a bit when you're discussing software-related matters. There's a delicate balance between too techy and too dumb.

"Why all this talk of how to treat your customers? I thought we were going to talk about how to market myself." If you work in a typical IT shop, much of the budget that keeps you gainfully employed comes from a business function—the same business function for which your customers work and influence decisions. When staffing decisions are being made, the best advocate you can have is a customer who doesn't want to live without you. On the flip side, imagine the impact of a customer who feels you are condescending. Your customer represents the needs of the business, and you are paid to provide for those needs. Don't forget this.

Act on it!

1. *Check yourself*—Are you the grumpy old code ogre everyone fears? Are you *sure*? Are they afraid to tell you?

Go through your mailbox, and find examples of e-mails that you have sent to less-technical co-workers, managers, and customers. As you read through, try to see things from the recipient's perspective. If some time has passed since sending the messages, you'll be able to see yourself as a third-party observer would.

Even better, show the e-mails to your mom. Tell her that someone you work with sent the messages to a customer, and ask her how the messages would make *her* feel.

2. *Hop the fence*—Find an opportunity to be flung into a situation in which you are *not* the expert and are dependent on others who are. If you have two left feet, imagine yourself on a soccer team. If you have two left thumbs, imagine you're part of the National Knitting Team. How would you like your teammates to communicate with you?

The days of the monosyllabic programmer grunt are over. If companies want to have difficulty communicating with their programmers, they'll sit the programmers on a different continent and in a different time zone and communicate with them only via e-mail and phone. That's what you and I are trying to avoid, right?

So, the communication issue is an important one. On the list of tasks you need to do to stay gainfully employed, it might sound a little contrived, silly, or trivial. You might feel a bit like you're back in high-school English class. That's OK. You can actually pay attention this time.

We'll get the most boring one over with first: grammar and spelling are important. You've probably got a degree in an advanced subject like engineering or computer science, and here I am telling you to learn how to spell. *The nerve!*

But, at least here in the United States, we have a problem.

According to a report by the National Commission on Writing, more than half of companies consider writing skills when making both hiring and promotion decisions. Forty percent of surveyed companies in the services sector said that a third or fewer of their new hires had the writing skills they desired.¹⁶

When you really step back and take a look at the big picture, writing skills are both necessary *and* are in short supply.

As you know, the world's workforce is distributing itself globally (that's why you're reading this book, right?). As this trend continues, there will come a time—for some, that time is now!—when *most* workplace communication will take place in written form via either instant messaging or e-mail.

You're going to be writing *a lot*. If so much of your job is going to involve writing, you better get good at it. More than ever, perceptions of you are going to be formed based on your writing ability. You may be a great coder, but if you can't express yourself in words, you won't be very effective on a distributed team.

¹⁶<http://www.writingcommission.org/report.html>

The ability to write creates both a superficial perception of you and a real insight into how your mind works. If you can't organize your thoughts in your mother tongue so that others can clearly understand them, how can we expect that you can do it in a programming language? The ability to shape an idea and lead a reader through a thought process to a logical conclusion is not much different from the ability to create a clear design and system implementation that future maintainers will be able to understand.

This isn't all about being judged, either. If you have team members in different time zones and distant locations, writing may be the only way you have to explain what you've done, how you've designed something, or what your team members need to work on.

You *are* what you
can *explain*.

Communication, especially through writing, is the bottleneck through which all your wonderful ideas must pass. You *are* what you can *explain*.

Act on it!

1. Start keeping a development diary. Write a little in it each day, explaining what you've been working on, justifying your design decisions, and vetting tough technical or professional decisions. Even though you are the primary (or only—it's up to you) audience, pay attention to the quality of your writing and to your ability to clearly express yourself. Occasionally re-read old entries, and critique them. Adjust your new entries based on what you liked and disliked about the old ones. Not only will your writing improve, but you can also use this diary as a way to strengthen your understanding of the decisions you make and a place to refer back to when you need to understand how or why you did something previously.
2. Learn to type. If you don't already "touch type," take a course or download some software that will teach you. You're more likely to be comfortable and natural in your writing if you are comfortable with the input method itself. Of course, with all this writing you'll be doing, you'll save yourself some time by learning to type quickly.

You have the advantage of being face to face with your leaders and your business customers. Don't squander the opportunity.

While I was living in Bangalore as CTO of our software development center, I had the unpleasant experience of reporting to a manager who I not only disliked (and who disliked me) but who was in the United States. We had strained, late-night or early-morning phone conversations, made increasingly frustrating by background noise or unintended disconnections. I would write long, descriptive e-mails in an attempt to help close the distance and time zone gap, only to be ignored. And, if I complained about being ignored, I would be criticized for writing long e-mails. It seemed like a no-win situation.

My company at the time had an annual performance review process in which managers would list their employees' strengths and (so-called) development needs. The top of my development needs list that year was something called *presence*.

Now, presence in this context is an ultracorporate word describing an ever-so-fuzzy leadership trait. It's the unmeasurable quality of having your presence felt—particularly in face-to-face situations. It also includes the equally unmeasurable quality of carrying yourself like a leader.

When I was sitting down talking about my performance review (over the phone) with my beloved manager, I muted my phone when she said "presence." I didn't want the laugh to be audible. I wondered if she could hear the half-grimace and half-smile that I couldn't wipe from my lips for the rest of our conversation. She and I both knew that the *real* problem was presence in the more common form of the word: I just wasn't there in the United States with everyone else.

Most of us who were willing to share our feelings disliked this manager. She did little to command respect, so it wasn't much of a surprise. The pattern that emerged was that the only employees who had really *negative* relationships with her were the ones who weren't in the same geographic place as her. Those in other countries such as India, Hungary, and Great Britain (in decreasing order) had strained relationships with her, since we were not only physically separated but we had time zone, infrastructure, culture, and language boundaries as well.

It seemed as though even for the people in the United States who were doing everything they could to avoid this manager, physical proximity and the occasional face-to-face conversation was all it took to make this manager comfortable. And, of course, the “out of sight—out of mind” phenomenon was very quickly validated when I hit the ground in India.

In addition to just telling a story about a bad manager, you can learn something from this experience. Physical proximity is an advantage in the workplace.

Think about the last time a relative or friend who was not computer savvy called you to ask for help with a computer problem. You try to walk them through the problem over the phone, and if they’re not getting it, you just get more and more agitated. *If I could only just show them...* In contrast, face-to-face communication is incredibly effective. You can hear the other party more clearly. You can explicitly use visual aids to get points across, by using hand motions or drawings on whiteboards. And, we all *implicitly* express a great deal of content in our facial expressions without even consciously realizing it.

Not only do we see greater productivity and enhanced communication from face-to-face interactions, but we also form tighter personal bonds. It takes a lot longer to create something you would call a friendship if you never meet someone in person. Fifteen years ago, it was unheard of. These days, with the ubiquity of the Internet, it’s just less common than traditional face-to-face friendships. For many of the same reasons that we work less effectively via phone, e-mail, and chat, we are also much less efficient in building relationships that way. Add to that the discomfort of the unnaturalness of e-mail and chat-based conversation (something that the next generation probably won’t remember), and in the majority of cases, the relationship built in a remote work environment will remain strictly centered around accomplishing tasks.

A strong team relationship with effective, high-bandwidth communication makes for better software delivered faster. In most environments, important project decisions are made in person, over coffee breaks and side conversations. These are fairly obvious observations, and the advantage one has by being a part of this is also fairly obvious. What’s not so obvious—especially to us geeks—is the importance of being *seen*.

I *never* go into a bank. I do any banking I have to do either online or via automated teller machines. My grandparents are different. They do virtually all their banking *in the bank* talking to *real people*. They don’t even

like to do business over the phone. It just doesn't make them comfortable. They also know the people at the grocery store they go to. They go back over and over again and chat with them as they're checking out. They wouldn't consider changing grocery stores (or banks), because the choice of bank or grocery store is more than a pragmatic weighing of cost and convenience. It's personal.

Until we have robots or computer programs to perform our performance appraisals, all business will continue to be personal. We people like to interact with other people in person. Some of us, anyway.

The natural work mode of a computer person is to hole up in a cubicle or office, put on a pair of headphones, and get into "the zone" until it's time to eat. Douglas Coupland, in his book *Microserfs* [Cou96] tells the entertaining story of a team having to buy flat food to slide under the office door of a programmer on a mission. This kind of focused isolation has become part of the culture and folklore of the software industry.

Unfortunately, speaking for your career, this is bad for business. If you're locked up in an office, accessible only by phone (if you answer) or e-mail, perhaps even working all hours of the night and sleeping late as a result, there's no difference between you being onsite with your bosses and your customers and being offshore. You are missing a huge opportunity to become a *sticky* fixture in your company. Remember, you need to make it *personal*, and to do that you have to remember the natural human tendency to want to work with other humans. Not voicemail, e-mail, or instant messaging but actual people.

In today's distributed environment, you may find that while you're in the same country as the people you're working with, you're not in the same city or state. Regular trips for face-to-

Learn about your
colleagues.

face meetings are great in these situations if they're practical for you and your company. But, the best thing you can do is pick up the phone and call your bosses and co-workers. Don't use speaker phones when you can help it, and don't rely on scheduled meetings. You need to try to simulate the kind of casual, coffee-break conversation that you might experience if you lived and worked in the same place, so budget time for (apparently) spontaneous conversations. On occasion, take the opportunity to make the conversation personal. Let "How are you today?" continue into "What do you generally do on the weekends?" Try to actually learn about the *people* you work with. Not only does it more firmly entrench you into your company, but it's a more enriching way to live.

Act on it!

1. One day in the next week, force yourself (within reason) not to send any e-mail. Every time you would normally send an e-mail, either call the person you would have sent it to on the phone or (better) walk to their office and speak to them in person.
2. Make a list of co-workers, bosses, and customers who you don't talk to enough. Put recurring appointments on your calendar to call and check in with them (either by phone or in person). Make the conversations short and meaningful. Use them to communicate something work related and also to simply make a human connection.

My young nephews all use computers regularly. They are, relatively speaking, quite computer savvy. They use computers to communicate with friends all over the world. They are completely comfortable with instant messaging, e-mail, web browsing, and of course personal publishing and the other stuff you might use if you were a high-school student working on assignments.

But, if I were to brag to them that my new computer had a 10,000 RPM Serial ATA hard drive, they might at best do a teenage-level job of feigning enthusiasm. They would probably be equally unimpressed if I told them it had one gigabyte of RAM and a GPU that was faster than the CPUs in the systems I used just five years ago.

However, if I told them they could run *Half Life 2* at full resolution without so much as a stutter in the game's visual appearance, they'd sit up and take notice.

Gigahertz and revolutions per minute aren't interesting to the average fourteen-year-old boy. Computer games are.

Businesspeople aren't that interested in gigahertz and RPMs either. They like it when their applications are fast, because they don't have to wait while on the phone with a customer or while trying to close out the books for the quarter. But, they don't care how many requests per second your new custom application server process can handle.

Businesses and those who run them are interested in business *results*. So, marketing your accomplishments in any language other than the language of the business is ineffective.

You wouldn't market a product to American audiences in German. A soft drink company wouldn't try to sell a drink to consumers based on the measured quantity of red dye #8 it contains. Common sense tells you that to sell a product to an audience, you have to speak to that audience in a language they can both understand and relate to.

As a software developer, that means framing your accomplishments in the context of the business you work for. Sure, you *got it done*, but what *was*

Market your accomplishments in the language of your business.

it? Why did it matter? How was this so-called accomplishment not just a waste of company time?

My guess is that if you were to think about the past month's accomplishments, you might not be able to articulate just *why* they were useful tasks to do in the first place. Sure, you might have been told to do them, but what benefit did they deliver to the business?

At General Electric, there is an urban legend that former Chief Executive Jack Welch used to enjoy getting on the elevator of one of the tall GE buildings with whatever random GE employee might have gotten on with him. He would then turn and ask the already-frightened underling, "What are you working on?" and then (here's where it might hurt) "What is the benefit of that?" The moral of the story was that you should always have your *elevator speech* ready, just in case.

What would you say if your CEO asked you the same question out of the blue? Even given a few minutes to prepare, would you be able to explain the business benefit of the tasks you are doing or the tasks you had recently done. Could you do it in words that a totally nontechnical senior executive could not only understand but also *appreciate*?

Act on it!

1. Make a list of your recent accomplishments. Write out the business benefit for each. If there are accomplishments on the list that you *can't* write a business benefit for, ask a manager or trusted acquaintance how they would frame the benefit.
2. Make your elevator speech, and memorize it.

Change the World

The worst thing anyone at work can ask about you is “What does he (or she) do?” Having to ask this question means that they don’t know what you’ve *done*.

It’s sad, but I don’t know what most of the people I’ve worked with in big-company IT have done. People just don’t think that way. They go to work, do their assigned thing, and go home. There’s no lasting impact, other than the trail of code, documents, and e-mail they leave behind them.

That’s what happens when you show up to work without a *mission*. You just sit around waiting to be told what to do. And, when you do what people tell you, the only people who know what you’ve done afterward are the ones who asked you to do it. That’s fine if you want to work in retail sales or maybe even if you want to be an offshore programmer.

But if you want to be a software developer in a high-cost country, you need to come to work with a mission. You need to effect change but not change within yourself or your own work (that’s a given). You need to effect visible change through your team, organization, or company.

Have a mission. Make sure people know it.

The change can be small. You might be carrying the torch for unit testing, driving test practices into the hearts of the unwashed masses of your company’s programmer pool. Or, it might be something bigger, like a radical new technology introduction that will lead to cheaper, better systems made faster.

You do these things because you are internally *driven* to do them. You can’t stand back and watch the people in your company do things wrong. You know things could be better, and you *have* to change them.

Of course, if you’re out to change the world, you’re bound to make some people angry. That’s OK as long as your intentions are right. Don’t be a jerk about it, but don’t tiptoe around, always playing it conservative, either.

If you *do* end up ticking a few people off, you can at least take comfort in the fact that they won’t ever ask, “What does *he* (or *she*) do?”

If you don't know what your crusade is, you probably don't have one. If you're not already *actively* trying to make your mark, you're probably not making it.

Act on it!

1. Catalog the crusades you've personally witnessed in the workplace. Think of the people you've worked with who have behaved as if on a mission. Think of the most driven and effective people in the places where you've worked. What were *their* missions?

Can you think of any such missions that were inappropriate? Where is the line between drive and zealotry? Have you seen people cross it?

Let Your Voice Be Heard

The ideas we've explored so far have been fairly conservative and focused on being recognized for the work you do in your workplace. If you want to be noticed, move up, or even stay employed with your current company, the topics we've touched on will be critical.

But, how boring!

The world is changing. If you want to write your ticket, you've got to think bigger than the IT workers of yesteryear. While moving from level-23 Programmer to level-24 Programmer Analyst might *really be* your short-term career goal, as a programmer today, you need to think beyond the next promotion or even your current place of employment.

Set your sights higher. Don't think of yourself as a programmer at a specific company—after all, it's not likely that you'll be at the same place forever—but as a participating member of an industry. You are a craftsman or an artist. You have something to share beyond the expense reporting application you're developing for your human resources department or the bugs you've got stacked up in your company's issue tracking system.

Companies want to hire experts. While a résumé with a solid list of projects is a good way to demonstrate experience, nothing is better at a job interview than for the interviewer to have already heard of you. It's especially great if they've heard of you because they've read your articles or books or they've seen you speak at a conference. Wouldn't *you* want to hire the person who "wrote the book" on the technology or methodology you're attempting to deploy?

In my previous life as a professional saxophonist, I played a lot in the clubs in and around Memphis's Beale Street. As I began to adapt to the computer industry, I saw a lot of overlap between the way you have to get your name out in music and in the computer industry. As a musician trying to find work, the following properties were true:

- (This one's the most important.) The best saxophonist doesn't always get the gig.
- Who you've played with is at least as important as how well you play; musicians are *cool by association*.

- Sometimes, the better musicians are overlooked for work because everyone assumes they won't be available or because they are too intimidated to ask.
- Music works via a network effect. If your social/music network terminates before reaching someone, it's not likely you'll ever be asked to perform with that person until an intermediary connection is made.

The computer industry is the same way. No objective system exists for rating and employing software developers. Being good is important, but it doesn't get you all the way there. Our industry, like the music industry, is a big, complex web of people connecting each other. The more places you are attached to the network, the better your chances of connecting with that perfect job or career break. Limiting yourself to the company you work for places serious limits on the number of connections you can form.

What better ways than publishing and public speaking are there for your name to be propagated and your voice to be heard? So, how do you go from Joe Schmo programmer to published author and then to public speaker? Start on the Web.

First, read weblogs. Learn about weblog syndication and get yourself set up with an aggregator. If you don't know what to read, think of a few of your favorite technical book authors and search the Web. You will probably find that they have a weblog. Subscribe to their feed and to the feeds of the people they link to. Over time, your list of feeds will grow as you read and find links to the weblogs other people have been writing.

Now open your own weblog. Many free services are available for hosting and driving a weblog. It's dead simple to do. Start by writing about (and linking to) the stories in your aggregator that you find interesting. As you write and link, you'll discover that the weblog universe is itself a social network—a microcosm of the career network you are starting to build. Your thoughts will eventually show up in the feed aggregators of others like you, who will write about and spread the ideas you've created.

The weblog is a training ground. Write on the Web as if you were writing a feature column for your favorite magazine. Practice the craft of writing. Your skill will increase, and your confidence will grow.

Your writings on the Web will also provide work examples that you can use in the next step. Now that you're writing in your own forum, you

might as well take your writing to community websites, magazines, or even books. With a portfolio of your writing ability available on the Web, you'll have plenty of example material to include in an article or book proposal. Get yourself in print, and your network grows. More writing leads to more writing opportunities. And all of these lead to the opportunity to speak at conferences.

Just as we started easily with the Web in our writing endeavors, you can start your speaking career in your local developer group meetings. If you're a .NET person, prepare a presentation for your local Microsoft development group. If you're a Linux programmer, do a talk at your Linux users group. Practice makes perfect when it comes to speaking. Be sure to put a lot of thought into preparing for these talks. Don't take them lightly. Though you're speaking only to a small crowd in your home city, this is where you live and work. A job *really* well done will (eventually) not go unrewarded. You'll find that if you give it the right amount of attention locally, these small talks are no different from the big ones at major industry conferences. Those are obviously the next logical step.

With all these ways to get your name and your voice out there, the most critical tip of all is to start sooner than you think you're ready. Most people undersell themselves. You *have* something to teach. You will never feel 100 percent ready, so you might as well start now.

Brand building has two parts: actually *making* your mark so that people will recognize it and then making sure that mark is associated with positive traits. Recognition and respect.

Today, when we see a swastika, we think of Hitler and Nazi Germany. From a brand building perspective, that's very good for the Nazis. They accomplished the first half of brand building: awareness. But, those of us who are mentally healthy have an extremely negative association with all things related to the Holocaust. So, the Nazis ultimately failed miserably in the positive association department. In fact, Hitler stole the swastika from the Hindus, perpetrating the crime that all companies serious about their brands struggle to prevent. To the Hindus, who lay original claim to the swastika (or *swasti*), it is an auspicious symbol of well being. But, now throughout the West, this spiritual icon has been defamed. Lots of recognition and little respect.

On the flip side is Charlie Wood.¹⁷ Charlie is an incredible singer, songwriter, and Hammond B3 organ player in Memphis, Tennessee. He plays five nights a week in a club on Beale Street. Everyone who knows him or has heard him knows how amazing he is. They all look up to him. He is as talented as you can get when it comes to rhythm and blues music.

But relatively nobody knows who the hell he is. No recognition and lots of respect.

What *you* want is both recognition and respect. Your name is your brand.

Your name is your brand. This entire part of the book is all about how to get both recognition and respect. Right here in this paragraph, what you need to understand is that the combination of the two is an asset worth building and guarding. Unlike a big, scared, corporate marketing department suing college kids over websites that misappropriate a corporate image or phrase, you don't need to spend too much time guarding your brand against *other* people. The most potentially destructive force for Brand You is yourself.

Don't water down what you stand for. Be careful where you let your name show up. Don't do lousy projects or send lousy e-mails to large groups of

¹⁷<http://www.charliewood.us>

people (or make lousy weblog posts for the whole Internet to read). Don't be a jerk. Nobody likes a jerk, even if they somehow *deserve* to be a jerk.

Most important, remember that the things you choose to do and associate yourself with have a lasting impact on what your name means to people. And, now that so many of our interactions take place via the Internet on public forums, websites, and mailing lists, a lot of are actions are public record and are cached, indexed, and searchable—forever.

Google never forgets.

You might forget, but Google doesn't.

Guard your brand with all your might. Protect it from yourself. It's all you've got.

Act on it!

1. *Google yourself*—Search Google for your full name in quotes. Look through the first four pages of results (are there actually four pages of results?). What could someone surmise about you following only the links from the first four pages of Google results? Are you even represented in the first four pages of search results for your name? Is the picture that these first four pages paints a picture that you appreciate?

Do the same search again, but this time pay special attention to forum and mailing list conversations. Are you a jerk?

Imagine how much easier it would be to find a job if there were companies already relying on software you had written. You could say, “Oh, are you running Nifty++? I can help you with that—I wrote it.” That would be different. Interviewers and hiring managers would remember you. That’s what you want.

Just a decade ago, while sounding like a wonderful idea, there wouldn’t have been many opportunities for such a scenario to be played out. You would have had to work for a commercial software vendor first, and your credentials would have been tied to the products you helped develop while working for that software vendor. But things have changed. You don’t have to work for the Big Guys to develop popular, name-brand software anymore.

Now there’s another outlet: open-source. Open-source software has hit the mainstream. As IT shops start new projects, the age-old debate has shifted from *build vs. buy* to *build vs. buy vs. download*. If not entire applications, frameworks ranging from small libraries to full-blown application containers are being released under open-source licenses and are becoming de facto standards.

And the people who are developing this software, for the most part, are people like you. They are people sitting in their homes in the evenings and on the weekends, pounding out software as a labor of love. Sure, there are some corporate-funded efforts creating or supporting Open Source products. But, the majority of Open Source developments are done by independent developers as a hobby.

Anyone can use Struts.
Few can say
Struts *committer*.

While many of these developers are just having fun and expressing themselves, some real incentives exist there. They are moving their way up the social chain of a community. They are building a name for themselves. They are building a reputation in the industry. They may not be doing it on purpose, but they are *marketing* themselves in the process.

Aside from building a name for yourself, contributing to open-source software shows you are passionate about your field. Even if a company hasn’t used or heard of your software, the fact that you’ve created and released it is a differentiator in itself. Think about it this way; if you were looking

to hire someone as a software developer, would you prefer to pick someone who puts in their nine-to-five day and then goes home and watches TV? Or would you prefer someone who is so passionate about software development that they take it upon themselves to do software development after-hours and on weekends?

Open-source contributions *demonstrate* skill. If you're making real code and contributing to a real project, it's a lot better on your résumé than just *saying* you know a technology. Anyone can write Struts or Nant on their résumé. Very few can write *Struts committer* or *Nant committer*.

Leading an open-source project can demonstrate much more than technical ability. It takes skills in leadership, release management, documentation, and product and community support to foster a community around your efforts. And, if you do *these* things successfully—in your spare time as a hobby—you'll be amazingly different from most of the other people competing for the same jobs. Most companies can't *pay* their developers to do all these things and do them well. Most can't even pay their developers to do *some* of them well. Showing that you can not only do them, but you care enough to do them for free shows an incredible amount of initiative.

If you create something really useful, you might even end up being famous. You could be famous in a small technical field—maybe famous among Struts people, for example. Or if you're lucky, you could be *really* famous even outside the geek community like Linus Torvalds or...well, like Linus. Even if you're not quite famous, releasing your code will definitely make you *more* famous. If fame means that lots of people know who you are, then having one more person know about you makes you more famous. And the open-source community is a *worldwide* network of people who, searching the Web for code, may come across your software, install it, and use it. In doing so, they will come to know about you, and as your software spreads, so will your name and reputation. That's what marketing is all about. That's what you want.

Traditional marketing curricula refer to the four Ps of marketing: product, price, promotion, and placement. The idea is that if you cover all four of these categories, you'll have a complete marketing plan. Equal weighting is put on each of the four categories.

But, what is the goal of marketing? Its purpose is to create a connection between producers and consumers of a product or service. This link starts with awareness about a product. The traditional mechanism of building awareness is via promotions, including advertisements, mailings, and educational seminars.

Recently the marketing world has turned its attention to what is called *viral*—word of mouth—marketing. Viral marketing happens when an idea is remarkable enough that consumers spread it from one person to the next. It spreads like a virus, with each new infected consumer potentially infecting many others.

Viral marketing is preferred not simply because it's expensive to send out paper mailing and buy television ad space. It's preferred because consumers trust their friends more than they trust television commercials and junk mail. They are more likely to buy something they hear about from a colleague at work than something on a pamphlet they dig out from the middle of their Sunday newspaper.

In his book *Purple Cow* [God03], master marketer Seth Godin makes the somewhat obvious assertion that the best way to get a consumer to remark on a product is to make your product remarkable. Godin goes so far as to say that the traditional four Ps are obsolete and that consumers have become numb to the old spray-and-pray strategies of mass marketing. The only way to stand out in the crowd, he says, is truly to be outstanding.

So, here's where the cynical readers start to applaud. All the marketing mumbo jumbo you might try is nothing compared to the power of a remarkable set of capabilities. Before you start saying, "I told you so," we should probably talk about the definition of *remarkable*.

Remarkable definitely doesn't mean the same thing as good. Usually, products that are remarkable *are* good. But, products that are good are seldom remarkable. To be remarkable means that something is worthy of attention. You will not become a remarkable software developer by sim-

ply being better than all the other software developers you know. Being incrementally better than someone else isn't striking enough to result in the viral spread of your reputation. If someone were to ask, you might have a glowing report card, but *remarkability* means that people talk about you *before* they are asked.

To be remarkable, you have to be significantly different from those around you. Many of the self-marketing strategies discussed in this chapter are geared toward remarkability. Releasing successful open-source software, writing books and articles, and speaking at conferences may all increase your remarkability.

If you look back at that last sentence, though not an exhaustive list, you'll notice that each of the items I've included as being potentially remarkable involve *doing something*. You might be the smartest or the fastest, but just *being* isn't good enough. You have to be *doing*.

Demo or die!

Godin uses the phrase *purple cow* to remind us of what it takes to be remarkable. Not *best cow* or *most milk-giving cow* or *prettiest cow*. A *purple cow* would stand out in a crowd of best, most milk-giving, and prettiest cows. It would be the purple one that you would talk about if you saw that group.

What can you do that would make you and your accomplishments like the purple cow? Don't just master a subject—write the book on it. Write code generators that take what used to be a one-week process to a five-minute process. Instead of being respected among your co-workers, become your city's most recognized authority on whatever technologies you're focusing on by doing seminars and workshops. Do something previously unthinkable on your next project.

Don't let yourself *just* be the best in the bunch. Be the person and do the things that people can't *not* talk about.

Making the Hang

When I was a young jazz saxophonist in Arkansas, people often asked me, “Oh, do you know Chris?” I didn’t. Chris was apparently the *other* high-school teenager in Arkansas who was a serious aspiring jazz musician. So, when people met me they would make the obvious connection, expecting us to be comrades in our very unhighschoolish jazziness.

One summer, I had the opportunity to see the Count Basie Jazz Orchestra perform an outdoor concert at an amphitheater on the bank of the Arkansas River. Through some combination of good mood and uncharacteristic courage, I ended up backstage chatting with the musicians before they went on. I’ve never been a very chatty person, so this was a real twist of fate. I stood in the back talking to one of the saxophonists from the orchestra, and another young kid walked up and started chatting. After five or ten minutes, the band started, leaving the two of us standing unattended. *Are you Chris/Chad?* we said simultaneously.

In the years to come, I would spend a lot of my free time with Chris. Chris, I soon learned, had a strange knack for associating himself with the town’s best musicians. He was just a high-school kid. But, he was already playing gigs, substituting for Little Rock’s most respected jazz pianists. Chris was pretty good—especially for his age—but he wasn’t *that* good.

It didn’t take me long to understand what was happening. We went out, several nights per week sometimes, to clubs where jazz music was being performed. It was always a somewhat uncomfortable experience for introverted me, because like clockwork, when the band we were watching took a set break, Chris would break mid-sentence and just walk away from me to go talk to the band members. He was like a robot. I have to admit, it was a little sickening to watch him. He was so predictable. Wasn’t he annoying these poor musicians? They were taking a break, for God’s sake. They didn’t want to talk to this damned kid! Being left hanging, I had to either follow him or sit awkwardly by myself waiting. Occasionally, on the days when I just didn’t have the energy, I chose the latter. However, for the most part I would follow Chris and try to pretend I was fitting in.

Usually, much to my surprise, the musicians on break actually seemed to enjoy talking to Chris—and even to me. He was pushy as hell and would always ask if he could sit in with the band, no matter how inappropriate it seemed to me. He would also ask the musicians for lessons, which

meant that he would go to their houses, listen to music, and chat about jazz improvisation with them. I would occasionally be dragged along, with the same feeling I had on the set breaks—that I was imposing.

But, I was obviously the one who was confused about this relationship that Chris was developing with these musicians. He was getting real, paying gigs with really good musicians. I was just some guy that hung around with him. He was my conduit to the city’s best musicians. The only difference between us being that he was more outgoing.

Over the years, Chris’s “be the worst” strategy coupled with the ability to unabashedly force himself on people, led him to become an incredible pianist. In fact, he squeezed his way into playing with some really famous jazz musicians. I, on the other hand, was still the guy he knew. He pulled me into some of these more high-profile gigs, but it was always him doing the pulling—not the other way around.

Since then, I’ve seen the same pattern in people I’ve met in classical music, the American Tibetan Buddhist community, software development, and even the confines of a single office. Chris called it “making the hang,” which made it even more repulsive to me. But, the short story is this: the really good people won’t mind if you want to know them. People like to be appreciated, and they like to talk about the topics they are passionate about. The fact that they are the professional, or the guru, or the leader, or the renowned author doesn’t change that they’re human and like to interact with other humans.

Speaking for myself (and extrapolating from there), the most serious barrier between us mortals and the people we admire is our own fear. Associating with smart, well-connected people who can teach you things or help find you work is possibly the best way to improve yourself, but a lot of us are afraid to try. Being part of a tight-knit professional community is how musicians, artists, and other craftspeople have stayed strong and evolved their respective artforms for years. The gurus are the supernodes in the social and professional network. All it takes to make the connection is a little less humility.

Fear gets between us
and the pros.

Of course, you don’t want to just randomly start babbling at these people. You’ll obviously want to seek out the ones with which you have something in common. Perhaps you read an article that someone wrote that was influential. You could show them work you’ve done as a result and get their input. Or, maybe you’ve created a software interface to a sys-

tem that someone created. That's a great and legitimate way to make the connection with someone.

Of course, you can make the hang online as well as in person. A lasting connection is a lasting connection. The heroes of the software world are globally distributed. The same is true in the music industry, though you can't take for granted that all musicians are connectable via e-mail. So, the music world tends to form local professional clumps, whereas software developers have the advantage of being able to easily reach each other no matter where we may be. This makes it easy to not only reach out to the gurus in your own city but to reach out to the gurus, period. Some of the most influential minds in software development are readily accessible via e-mail or even real-time chat.

The story that leads to me writing *this book* actually started with an e-mail about a Ruby library to one of its publishers followed by many conversations via online chat. Though I was timid about sending that original e-mail, apparently it didn't annoy Dave too much, and here you are reading my words. Thanks, Chris.

Part V

Maintaining Your Edge

Do you remember a pop star named Tiffany (no last name) from the 1980s? She was in the top of the top forty, and a constant sound on the radio back then. She enjoyed immense success, becoming for a short time a household name.

When was the last time (if ever) you heard anything about her? My guess is that you can't remember. I can't.¹⁸

Tiffany had what it took to be a hit in the 80s—at least for a short time. Then the 90s came along, and Tiffany was way out of style. Apparently, if she tried, she didn't move fast enough to hold the affection—or even the attention—of her fans. When the tastes of the nation turned from bubble gum to grunge, Tiffany suddenly became obsolete.

The same thing can happen to you in your career. The process in this book is a loop that repeats until you retire. Research, invest, execute, market, repeat. Spending too much time inside any iteration of the loop puts you at risk of becoming suddenly obsolete.

It can creep up on you if you're not explicitly watching for it. And when it catches you off guard, it's too late. Tiffany probably had no idea the grunge thing was going to take off. She was putting all of her efforts into being a teenage, bubble-gum pop star, and by the time grunge music took over the top forty, she was irreversibly out of style.

This part will show you how to avoid becoming a one-hit wonder.

¹⁸Apparently, Tiffany has staged a comeback within the last year, so I might be wrong.

Already Obsolete

Many of us are drawn to the IT industry because things are always changing. It's an exciting and fresh work environment. There's always something new to learn. On the flip side, though, is the disheartening fact that our hard-earned investments in technology-related knowledge depreciate faster than a new Chevy. Today's hot new item is tomorrow's obsolete junk with a limited shelf life.

In *Leading the Revolution* [Ham02], Gary Hamel talks about how the incumbent industry leaders in any given industry become complacent and, through their complacency, develop blind

Your shiny new skills are already obsolete.

spots. The more successful your business, the more likely you are to grow comfortable with your business model, making you incredibly vulnerable to those who come along behind you with a radical idea—even a stupid one—that might make your wonderful, winning business model look like an old, worn-out sweater at a disco. The same can be said of technology choices. If you've mastered the Big One of any given time period, such as J2EE or .NET at the time this book was published, you may feel extremely comfortable. It's the profitable place to be, right? Every job website and newspaper classified section serves as an affirmation of your decision.

Beware. Success breeds hubris, which breeds complacency. A wave like J2EE might feel like it will never end. But, all waves either dissipate or meet the shore eventually. Too much comfort for too long might leave you defenseless, wondering what you'd do in a non-J2EE world.

That being said, folks have been pronouncing COBOL's death for decades. Every new incumbent is called "the COBOL of the 21st century," or some variation thereof. These days, the label's applied to Java. As much as I hate to touch, see, or be near COBOL code, to call Java the COBOL of the 21st century is quite a compliment. As much as some of us would love to see it go away, COBOL is here, and it has been working for a *long* time. COBOL programmers have been working with COBOL for an entire career. That's really saying something in this roller coaster of an industry. It's hard to say if the same kind of investment would work in today's economy.

COBOL's story is the exception—not the rule. Few technologies provide such a lasting platform for employment. The message here isn't to run out and shed yourself of your mainstream knowledge. That would be

irresponsible. I *will* say that the more mainstream your knowledge, the greater risk you have of being left in the technology stone age.

We've all heard the extrapolations of Moore's law, which say that computing power doubles every eighteen months. Whether the numbers are exactly correct, it's easy to see that technology is still advancing at roughly the same rate as it was in 1965 when Intel's Gordon Moore made this assertion. And, with these advances in hardware horsepower come advances in what is possible to do with software.

Computing power *doubles*. With technology progressing so quickly, there is too much happening for any given person to keep up. Even if your skills are completely current, if you're not almost through the process of learning the Next Big Thing, it's almost too late. You can be ahead of the curve on the current wave and behind on the next. Timing becomes very important in an environment like this.

You have to start by realizing that even if you're on the bleeding edge of today's wave, you're already probably behind on the next one. Timing being everything, start thinking *ahead* with your study. What will be possible in two years that isn't possible now? What if disk space was so cheap it was practically free? What if processors were two times faster? What would we not have to worry about optimizing for? How might these advances change what's going to hit?

Yes, it's a bit of a gamble. But, it's a game that you will *definitely* lose if you don't play. The worst case is that you've learned something enriching that isn't directly applicable to your job in two years. So, you're still better off looking ahead and taking a gamble like this. The best case is that you remain ahead of the curve and can continue to be an expert in leading-edge technologies.

Looking ahead and being explicit about your skill development can mean the difference between being blind or visionary.

Act on it!

1. Carve out weekly time to investigate the bleeding edge. Make room for at least two hours each week to research new technologies and to start to develop skills in them. Do hands-on work with these new technologies. Build simple applications. Prototype new-tech versions of the hard bits of your current-tech projects to understand what the differences are and what the new technologies enable. Put this time on your schedule. Don't let yourself miss it.

You've Already Lost Your Job

The job you were hired to do no longer exists. You might still be drawing a paycheck. You might be adding value. You might even be making your employer ecstatically happy. But, you've already lost your job.

The one certain thing is that everything is changing. The economy is shifting. Jobs are moving offshore and back on. Businesses are trying to figure out how to adapt. Things have not reached a steady point in our industry. Our industry is like the awkward adolescent going through puberty. Awkward, ugly, and different year after year—day after day.

So, if you were hired to be a programmer, don't think of yourself as a programmer. Think of yourself as maybe not a programmer anymore. Keep doing your job, but don't get too comfortable. Don't try to settle into the *identity* of a programmer. Or a designer. Or a tester.

In fact, it's no longer safe (as if it ever was) to identify yourself too closely with the job you were hired to do. If your surroundings are changing and the context of your work is constantly moving, clinging to your job creates an unhealthy dissonance that infects your work. You may find yourself as the would-be programmer doing the job of a should-be project manager. And doing it poorly.

Back before you lost your job, you might have had plans. You might have imagined your progression through the company's ranks. You would do your time as a designer and take the architect role when your just reward is due. You could see the entire progression from architect to analyst to team leader up the management chain.

You are not your job.

But, you've already lost your job, and your plans have changed. They're going to keep changing. Every day. It's good to have ambition, but don't buy too heavily into a long, imagined future. You can't afford to have tunnel vision with something too far off in the future. If you want to hit a moving target, you don't aim for the target itself. You aim for where the target is likely to go. The path from here to there is no longer a straight line. It's an arc at best but most probably a squiggle.

Act on it!

1. If you're a programmer, try a day or two of doing your job as if you were a tester or a project manager. What are the many roles that you might play from day to day that you have never explicitly considered? Make a list, and try them on for size. Spend a day on each. You might not even change your actual work output, but you'll see your work differently.

Path with No Destination

One of America's biggest problems is that it is a goal-oriented society. We're a nation of people who are always focused on the *outcome* of a process, whether it is the process of learning, one's career, or even a drive in the car. We're so centered on the outcome that we forget to look at the scenery.

If you think about it, the focus on outcomes is logically the reverse of what we should be spending our time on. You typically spend all your time *doing* things and little of your time actually reaching goals. For example, when you're developing software, the development process is where you spend all your time, not on the actual event of the finished software popping out of the end of the process.

This is true of your career as well. The real meat of your career is not the promotions and salary advances. It's the time you spend working toward those advances. Or, more important, it's the time you spend working *regardless* of the advances.

If this is the core of your work life—the actual work—then you've already arrived at your destination. The goal-oriented, destination-focused thinking that you usually do leads only from one goal to the next. It has no logical end. What most of us fail to realize is that *the path* is the end.

Back to the software development example, it's easy to get wrapped up in the delivery of the code you are creating. Your customer needs a web application up, and you focus on finishing that application. But, a living application is never "done." One release leads to the next. Too much focus on the end product distracts us from the real deliverable: the sustainable development of a new entity.

Focusing on the ending makes you forget to make the process good. And, bad processes create bad products. The product might meet its minimum requirements, but its insides will be ugly. You've optimized for the short-term end goal—not for the inevitable, ongoing, future of the product's development.

Focus on doing, not on being done.

Not only do bad processes make bad products, but bad products make bad processes. Once you have one of these products that is messy inside, your

processes adapt around it. Your product's *broken windows* lead to broken windows in your process. It's a vicious cycle.

So instead of constantly asking, "Are we there yet? Are we there yet?" realize that the only healthy answer is "yes." It's how you traverse the path that's important—not the destination.

Make Yourself a Map

When you're in maintenance mode, it's easy to snap into a groove and just keep on being like you are. As a software developer, you know this is true from your experience with systems. If you maintain an application or a library that other developers use, it will sit stagnant in bug-fix mode (or worse) unless you have a solid feature road map. You might make the occasional enhancement because of user requests or your own needs, but the code will usually reach a steady state and change at an exponentially slower rate as you consider it done.

But a living application is never done unless it's on the road to retirement. The same is true of you and your career. Unless you're looking to exit the industry, you need a road map. If Microsoft had considered Windows 3.1 done, we'd all be using Macintoshes right now. If the Apache developers had considered their web server done when they reached 1.0, they might not be overwhelmingly leading the market right now.

Your personal product road map is what you use to tell whether you've moved. When you're going to the same office day in and day out, working on a lot of the same things, the scenery around you doesn't change. You need to throw out some markers that you can see in the distance, so you'll know that you've actually moved when you get to them. Your product "features" are these markers.

Unless you really lay it out and make a plan, you won't be able to see beyond the next blip on the horizon. In Chapters 2 and 3, you discovered how to be intentional about your choice of career path and how to invest in our professional selves. Though I focused on what seemed like a one-time choice of what to invest in, each choice should be part of a greater whole. Thinking of each new set of knowledge or capability as equivalent to a single feature in an application puts it in context really well. An application with one feature isn't much of an application.

What's more, an application with a bunch of features that aren't cohesive is going to confuse its users. *Is this an address book or a chat application? Is it a game or a web browser?* A personal product road map can not only help you stay on track, constantly evolving, but it can also show you the bigger picture of what you have to offer. It can show you that no single feature stands alone. Each new investment is part of a larger whole. Some work fabulously well together. Others require too much of a mental leap

for potential employers. *Is he a system administrator or a graphic designer? Is she an application architect or a QA automation guru?*

While it's definitely OK to learn diverse skills—it expands your thinking—it's also a good idea to think about the story your skillset tells. Without a road map, your story might look more like a Jack Kerouac novel than a cohesive set of logically related capabilities. Without a road map, you might even *actually* get lost.

Act on it!

1. Before mapping out where you want to go, it can be encouraging and informative to map out where you've *been*. Take some time to explicitly lay out the timeline of your career. Show where you started and what your skills and jobs were at each step. Notice where you made incremental improvements and where you seemed to make big leaps. Notice the average length of time it took to make a major advancement. Use this map as input as you look forward in your career. You can set more realistic goals for yourself if you have a clear image of your historical progress. Once you've created this historical map, keep it updated. It's a great way to reflect on your progress as you move toward your newly defined goals.

You'd be a fool to invest your money in a volatile stock and then ignore it. Even if you've done a great deal of research and made an intentional choice about *what* to invest in, the market is uncertain. You can't just fire-and-forget when it comes to investments. Stock value might be increasing now, but that doesn't mean a stock isn't going to start tanking tomorrow.

You might also be missing an opportunity. You may find a really safe bet, yielding a ten percent annual return. That sounds like a pretty good deal as long as the rest of the market isn't suddenly doing much better than ten percent. Your workhorse investment of today, even if it continues to perform, may not be very impressive compared to what's possible tomorrow.

As the conditions of the market change, not paying attention could result in money lost or money that *could* have been earned, missed.

The same holds true for your knowledge investments. Java is the conservative choice of today. What might change to make that not true anymore? How might you know if it changed?

What if, for example, Sun Microsystems started showing signs of going under? Java isn't an open standard. It is dictated and developed by Sun. At any point, a dying Sun might attempt to suddenly make its language and virtual machine into a last-minute profit center. They might introduce incompatible changes or suddenly change the license restrictions of Java, causing an industry panic followed by a mass exodus.

With your head in your monitor coding, you might not even hear about something like this until it was too late. You might find yourself on the job market with a suddenly less valuable skill. This is an unlikely hypothetical situation, but something like this could happen.

Even more likely is that, comfortable in your current job with your current set of skills, you might remain blissfully ignorant of the Next Big Thing as it rolls in. Ten years ago, you would have been surprised to find out just how big object-oriented languages with garbage collection would become. But, there were definitely signs if you were watching. Ten years from now, who knows what the Next Big Thing will be?

You've got to keep your eyes and ears open. Watch the technology news, both the business side and the purely technical side, for developments that

might cause a ripple. As Tim O'Reilly¹⁹ of O'Reilly and Associates says, watch the *alpha geeks*. Alpha geeks are those supernerds who are always on the bloodiest tip of the bleeding edge, at least in their hobby activities. Tim's assertion, which I have since observed in the wild, is that if you can find these people and see what they're into, you can get a glimpse of what's going to be big one or two years down the road. It's uncanny how well this works.

Watch the alpha geeks.

However you choose to do it, you need to be aware that in the technology sector, what's a good investment today will eventually *not* be a good investment. And, in case you pay attention to the mood of the market, it might catch you by surprise. You don't want this kind of surprise.

Act on it!

1. Spend the next year trying to become one of the alpha geeks. Or at least *make the hang* with one.

¹⁹<http://tim.oreilly.com/>

That Fat Man in the Mirror

I am, unfortunately, overweight. I have been for a long time. While living in India, though, I lost a *lot* of weight. Part of it was due to diet. Part due to exercise. But, mostly it was from getting sick. After I came back to the U.S., I slowly gained the weight back. It was a disappointing thing, which I reacted to by signing up for a gym and a fitness instructor. The weight started coming back off.

I've gone through several such fluctuations. What's fascinating about them is that I can't really tell when I'm gaining or losing weight. The only way I know is if someone tells me or my clothes suddenly stop fitting the same. Or if someone tells me. My wife sees me every day, so she can't tell either, and, in the U.S., people generally don't mention it when you *gain* weight. In India, they do.

I can't tell, because I see me too often. If you're constantly exposed to something, it's hard to see it changing unless change happens rapidly. If you sit and watch a flower bloom, it will take a long time to notice that anything has happened. However, if you leave and come back in two days, you'll see something very noticeably different from when you left.

You'll notice the same phenomenon with your career. Actually, you *won't* notice it. That's the problem. You might look at yourself in the metaphorical mirror each day and not see an ounce of change. You seem as well adjusted as before. You seem as competitive as before. Your skills seem to be as up-to-date as before.

Then suddenly, one day your job (or your industry) doesn't fit you anymore. It's just uncomfortable at first, but you've already reached a critical point at which you have to either act quickly or go buy a new pair of (metaphorical) pants.

When it comes to fluctuations in body weight, you have a scale, so it's fairly easy to measure your progress (or lack thereof, in my case). There is unfortunately no such scale for measuring your marketability or your skill as a software developer. If there were, we could sit you on a scale and autogenerate your paychecks. Since we don't have that scale, you'll have to develop your own.

An easy way to measure your progress is to use a trusted third party. A mentor or a close colleague doesn't live in your head with you and can

help give you a more objective look at where you stand. You might discuss your abilities as a software developer, project leader, communicator, team member, or any other facet of the total package that makes you who you are. At GE, there is a process called a 360-degree review, which formalizes this idea and encourages employees to seek feedback from peers, managers, and internal customers. Despite the corporate doublespeak nature of its name, the process is a great way to get a number of different perspectives of yourself as an employee.

Developer, review thyourself.

The most important thing to ferret out as you go through a process like this (either alone or with help) is where your blind spots are. You don't *have* to fix all of them. You just have to know where they are. Without being explicit about it, you'll be blind to your blind spots. That's when the bad things happen and take you by surprise. Bad things will happen, so it's best to know they're coming.

Even if you had a magic *value scale* that you could weigh yourself on, it would do you no good unless you used it. Schedule your reviews. You won't reflect unless you make the reflection time explicit. Saying, *don't forget to ask for feedback* isn't a strong enough message. If you have a calendar program that pops up reminders, make appointments for yourself for self-evaluation. First determine your measurement system, and then put it on the schedule. If it's not a built-in part of your work life, you won't do it.

If your company has such processes in place already, don't write them off as HR nonsense. Take them seriously and *make good* come out of them. They may be implemented poorly where you work, but the motivation (at least what *used to be* the motivation) for them is right on.

Finally, when you've got your system in place and you've scheduled time to make sure it gets fit in, *capture the results in writing*. Keep your evaluation somewhere handy. Review and revise it often. Tying the self-evaluation process to a physical artifact will make it concrete.

Don't let obsolescence creep up on you like a pair of tight-fitting pants.

Act on it!

1. Do a 360 review:
 - Make a list of trusted people who you feel comfortable asking for feedback. The list should preferably contain representatives

from your peers, customers, and managers (and subordinates if you have any).

- Make another list of about ten characteristics you believe are important measurements of you as a professional.
- Convert this list to a questionnaire. On the questionnaire, ask for participants to rate you in terms of each characteristic. Also include the question, “What should I have asked?”
- Distribute the questionnaire to the list of people from the first step. Ask that your reviewers be constructively critical. What you need is honest feedback—not sugarcoating.

When you get the completed answers back, read through all of them and compile a list of actions you are going to take as a result. If you’ve asked the right questions of the right people, you *are* going to get some actionable items. Share your the outcome of your questionnaire with your reviewers—not the answers but the resultant changes you plan to make. Be sure to thank them.

Repeat this process occasionally.

2. Start keeping a journal. It could be a weblog, as we discussed in *Let Your Voice Be Heard*, on page 131, or a personal diary. Write about what you’re working on, what you’re learning, and your opinions about the industry.

After you’ve been keeping the journal for some time, re-read old entries. Do you still agree? Do they sound naive? How much have you changed?

The South Indian Monkey Trap

In *Zen and the Art of Motorcycle Maintenance* [Pir00], Robert Pirsig tells an enlightening story about how people in South India used to catch monkeys. I don't know if it's true, but it teaches a useful lesson, so I'll paraphrase it.

The people of South India, having been pestered by monkeys over the years, developed an ingenious way of trapping them. They would dig a long, narrow hole in the ground and then use an equally long, slender object to widen the bottom of the hole. Then they would pour rice down into the wider portion at the bottom of the hole.

Monkeys like to eat. In fact, that's a large part of what makes them such pests. They'll jump onto cars or risk running through large groups of people to snatch food right out of your hand. People in South India are painfully aware of this. (Believe me, it's surprisingly unsettling to be standing serenely in a park and have a macaque come suddenly barreling through to snatch something from you.)

So, according to Pirsig, the monkeys would come along, discover the rice, and stretch their arms deep into the hole. Their hands would be at the bottom. They would greedily clutch as much of the rice as possible into their hands, making a fist in the process. Their fists would fit into the larger portion of the hole, but the rest of the narrow opening was too small for the monkeys to pull their fists through. They'd be stuck.

Of course, they could just let go of the food, and they'd be free.

But, monkeys place a high value on food. In fact, they place such a high value on food that they cannot force themselves to let go of it. They'll grip that rice until either it comes out of the ground or they die trying to pull it out. It was typically the latter that happened first.

Pirsig tells this story to illustrate a concept he calls *value rigidity*. Value rigidity is what happens when you believe in the value of something so strongly that you can no longer objectively question it. The monkeys valued the rice so highly that when forced to make the choice between the rice and captivity or death, they couldn't see that losing the rice was the right thing to do at the time. The story makes the monkeys seem really stupid, but most of us have our own equivalents to the rice.

If you were asked whether it was a good idea to help feed starving children in developing countries, you would probably say “yes” without even thinking about it. If someone tried to argue the point with you, you might think they were crazy. *This* is an example of value rigidity. You believe in this one thing so strongly that you can’t imagine *not* believing it. Clearly, not all values that we hold rigidly are bad. For most people, religion (or lack thereof) is also a set of personal beliefs and values that are unfaltering.

But not all rigidly held values are good ones. Also, many times something that is good in one set of circumstances is not good in another.

For example, it’s easy to get hung up on technology choices. This is especially true when our technology of choice is the underdog. We love the technology so much and place such

Rigid values make you fragile.

a high value on defending it as a choice for adoption that we see every opportunity as a battle worth fighting—even when we’re advocating what is clearly the wrong choice. An example I encounter (and have probably been guilty of myself) is the overzealous Linux fan base. Many Linux users would put Linux on the desktop of every receptionist, office assistant, and corporate vice president with no regard for the fact that, in terms of usability, the toolset just doesn’t compare to much of the commercial software that’s available for a commercial operating system. You look foolish and make your customers unhappy when you give the right software to the wrong people.

It’s hard to tell you’re losing weight because you see yourself every day. Value rigidity works the same way. Since we live every day in our careers, it’s easy to develop value rigidity in our career choices. We know what has worked, and we keep doing it. Or, maybe you’ve always wanted to be promoted into management, so you keep striving toward that goal, regardless of how much you like *just programming*.

It’s also possible for your technology of choice to become obsolete, leaving you suddenly without a foundation to stand on. Like a frog in a slowly heating pot of water, you can suddenly find yourself in a bad situation. Many of us in the mid-1990s swore by Novell’s NetWare platform when it came to providing file and print services in the enterprise. Novell was way ahead of its time with its directory services product, and those of us “in the know” were almost cocky in our criticism of competing technologies. Novell’s product was enjoying a healthy majority in market share, and it was hard to imagine the tide turning.

No single event made it obvious that Novell was losing to Microsoft. Microsoft never made that magic Active Directory release that made us all say, “Wow! Drop NetWare!” But, Netware has slowly gone from bleeding-edge innovator to legacy technology. For many NetWare administrators, the water was boiling before they even realized the pot was warm.

Whether it is the direction your career is taking or the technologies you advocate and invest in, beware of monkey traps. Those originally intentional choices may become the last handful of rice you find yourself gripping prior to your career being clubbed to death.

Act on it!

1. *Find your monkey traps*—What are *your* rigid assumptions? What are those values that guide your daily actions without you even consciously knowing it?

Make a table with two columns, *Career* and *Technology*. Under each heading list the values that you hold unfalteringly true. For example, under *Career*, what have you *always* known to be one of your strengths? Or your weaknesses? What is your career *goal* (“I want to be a CEO!”)? What are the right ways to achieve your goal?

In the *Technology* column, list what you most value about the technologies you choose to invest in. What are the most important attributes of a technology that should be considered when making a choice? How do you make a scalable system? What’s the most productive environment in which to develop software? What are the best and worst platforms in general?

When you’ve got your list down and you feel like it’s fairly complete, go one at a time through the list and mentally reverse each statement. What if the opposite of each assertion were true? Allow yourself to honestly challenge each assertion.

This is a list of your vulnerabilities.

2. *Know your enemy*—Pick the technology you hate most, and do a project in it. Developers tend to stratify themselves into competing camps. The .NET people hate J2EE, and the J2EE people hate .NET. The UNIX people hate Windows, and the Windows people hate UNIX. Pick an easy project, and try to do a *great* application in the technology you hate. If you’re a Java person, show those .NET folks how a *real* developer uses their platform! Best case, you’ll learn that the technology you hate isn’t all that bad and that it is in fact possible to develop good code with it. You’ll also have a (granted, undeveloped) new skill that you might need to take advantage of in the future. Worst case, the exercise will be a practice session for you, and you’ll have better fodder for your arguments.

Part VI

If You Can't Beat 'Em

You *can't* beat 'em. Nor should you try.

When I got into software development, what thrilled me was the chance to use my creativity, along with skills that other people didn't have, to solve tough problems for people. For many people in companies around the world, technology is a scary thing. It's exciting to be able to play tour guide in such an ostensibly hostile environment, making it look easy along the way. It's rewarding to watch someone's fear turn into confidence with you at the helm.

As much as we'd like to blame these evil businesspeople for incubating the threat of offshoring, most of our customers are no more comfortable than we are about it. If you were a businessperson dependent on an IT group to deliver software that was a critical part of your personal success, how comfortable would you be replacing your onsite team, members of which you might have shared a common hometown or college, with a group of distant people whose names you have trouble pronouncing? Add to this the possibility that you are already somewhat afraid of the technology, and it makes for a chilling proposition.

So, we have a new kind of challenge that we can master. It requires at least as much creativity and skill to overcome it. How do we make offshoring work?

It's not going away. Even offshoring's biggest proponents readily admit that it's not easy to do. The ability to make offshoring work is a skill that is just as important as any programming language or operating system you might have put on your résumé in the past.

Companies are going to keep moving work offshore. They're going to need people to help them do it. People who aren't afraid. People who are open-minded enough to see that their jobs are *changing*—not going away.

While I was living in India, I played a technical leadership role in our software business. Part of my job was to increase the overall skill level of the team in India, driving independence into the India team. The more self-sufficient the team in India was, the more productive they would be. The less they would drag the U.S. team members into daily meetings, explanatory e-mails, and extended instant messaging conversations.

Being there on the other side of the ocean myself, I finally saw the problem from the Indian perspective. The Americans (myself included) constantly complained that the team members in India didn't "get it" or wouldn't drive to their own technical conclusions. The Indian team just wasn't "as good" as the American team, so they said.

However, to a team member in India, every suggestion they made to their American peers fell on deaf ears. E-mails were ignored. New ideas, many of which were authentic improvements, were stonewalled. The team in India was being ignored.

I left behind in the U.S. a team of talented software developers and architects. They were well known in the company for their collective ability both to dream up the innovative technical solutions that ran our business and to confidently walk their customers through the minefields of technology change.

If they were so good, and the Indian team was so "green," why the hell couldn't they make the Indian team better? Why was it that, even with me in India helping, the U.S.-based software architects weren't making a dent in the collective skill level of the software developers in Bangalore?

The answer was obvious. They didn't want to. As much as they professed to want our software development practices to be sound, our code to be great, and our people to be stars, they didn't lift a finger to make it so.

These people's jobs weren't at risk. They were just resentful. They were holding out, waiting for the day they could say "I told you so," then come in and pick up after management's mess-making offshore excursions.

But that day didn't come. And it won't. The offshoring effort was a major cost-saving initiative for the company. It was one of the few strategic focuses for the IT group. By sitting back waiting for it to fail, the team in the U.S. was nothing but a barrier to progress. If offshoring was about

saving money for the business, then the U.S. team was *wasting* money. If application quality was a concern, the U.S. team was serving only to reduce quality by not doing what was in its power to help.

By fighting against the perceived threat, the U.S. team members were in fact putting themselves at risk.

Offshore outsourcing is big business, and it's going to keep getting bigger. To companies that venture into it, it's a strategic direction—not an afterthought. But, offshore outsourcing is difficult to do. To do it well, companies absolutely *require* the help of skilled developers to coach and guide new teams through the process.

Offshoring isn't going away. Given the social and economic dynamics in the popular offshore destinations, chances are that companies doing work offshore will be working with a much lower average experience level than they are accustomed to with domestic employees. The people in these countries are just as smart as and probably *more* motivated than many of us here. But, being (on the average) young, they need guidance from highly skilled developer mentors.

If you refuse to help, you're useless.

You might be one of those skilled software developers. You may have chosen the right technologies and continuously made the right investments of time and effort to bring your skill level up above the crowd. But, even if you're smart enough and knowledgeable enough, if you refuse to help, you're *useless*.

On the other hand, if you can take a handful of green recruits who are working for a lower wage, half a world away, and put your mark on them, what a great accomplishment! You can give the company both higher quality *and* lower cost. You can take a scary situation and make it feel safe. You can be the hero.

And, you won't just be the hero to the people in your time zone. You will be genuinely *helping* your offshore colleagues. It's rewarding work, if you can talk yourself into doing it. You'll be amazed at how much *you* learn in the process.

Don't try to sink the offshoring effort. Do your part to lead it to victory. The ability to lead offshore teams—even if only in terms of technical direction—is a skill you *need* on your résumé. It's a skill that's likely to remain relevant longer than many of the technologies you currently work with.

When you split a project team across geographical, time zone, and cultural barriers, it doesn't matter how good the people are—you're going to run into some problems.

Think of a software system you've worked on that you would say is of medium complexity. Now, in your head or to a friend who will tolerate it, try to explain the system without using any visual aids. No whiteboard. No hand gestures. No ability to see the face of the person you're explaining it to, so you can't *see* if they're getting it. Now imagine the person you're explaining it to can't hear you very well and speaks English as a second language.

Now imagine asking someone a touchy, personal question over the phone. And, imagine that with this person you can half-expect them to fib in order to skirt the issue. How can you tell if they're lying over the phone? How much easier is it for *you* to tell a white lie over the phone than face to face? I'm not implying that offshore developers are liars, but it can be hard to pin people down on due dates for tasks when they aren't colocated with you.

These imagined scenarios pointedly reflect a portion of the sort of trouble you will run into when you do geographically distributed work. Communication can be a killer. How do you clearly express application requirements or designs to someone you might never see in person and with whom you have, at best, only a couple of hours of time zone overlap? How, as they start working, do you make sure that what you've asked for is what is actually being developed?

Working with offshore teams is a new and harder problem when it comes to project management. Many companies have very relaxed project management processes, relying heavily on high-bandwidth face-to-face communication and constant interaction between the participants involved in each project. These work really well when everyone is together. But, try using your same old processes with an offshore team, and believe me, you're in for a disappointment.

Offshore teams need someone who can keep a *close* eye on what's going on. Requirements have to be explained in enough detail so that the team members can work productively without the opportunity to ask clarifying questions until the end of their workday. Even if you *could* spend hours on

the phone, most projects will require visual aids in order for the offshore teams to understand application functionality, flow, or software design.

...a new kind of project manager with a new set of skills.

Additionally, because of a combination of the relative inexperience of the average offshore developer and the history of how offshore teams are typically used, the teams are going to need much more concrete work direction than

the typical onsite development team. Someone is going to have to decompose application requirements into tasks that can be added to a checklist and crossed off when finished. The task definitions will have to be more verbose than you would imagine. There's a fine line between specifying tasks so much that you might as well have just done the task yourself and specifying them too sparsely so that the team can't make any progress. You'll learn where that line begins via trial and error. Just be aware that it exists and that you're *going* to get close to it.

Now that a team has all of these little well-specified tasks underway, someone has to start tracking which ones are getting done and which ones need to get done next. And, "done" needs to be explicitly defined. With so much room for misinterpretation and communication breakdown, the only way to really measure completeness is to divide tasks into *working pieces of functionality* and then actually *run* the code that is created.

Being removed from the onsite fray of activity, it's easy for an offshore team to get stuck. Given an unanswered question coupled with the usual cultural aversion to admitting defeat, I've seen teams sit idle for days on end waiting for a resolution that isn't even in the pipeline. The most important role that needs to be played on the onsite side of an offshore development team is the "roadblock breaker." This person will spend a portion of *each day* doing the onsite legwork to get the offshore team's questions answered so they can get back to business when they get in the next day.

What I've just described is a project manager. But it's a new kind of project manager with a new set of skills. It's a project manager who must act at a different level of intensity than the project managers of the past. This project manager needs to have strong organizational, functional, *and* technical skills to be successful. This project manager, unlike those on most onsite projects, is an *absolutely essential* role for the success of an offshore-developed project.

This project manager possesses a set of skills and innate abilities that are hard to come by and are in increasingly high demand.

It could be *you*.

Act on it!

1. The ability to write clear, complete functional and technical specifications is critical in the world of offshore development. Read a book on writing use cases. If you don't already know the Unified Modeling Language (UML), learn it. Practice modeling and writing specifications on your current project. Imagine you had to delegate some of your work to someone with whom the only ways you could communicate were via specification documents and UML diagrams.

What makes offshoring so hard? This should be pretty easy to answer, right? Let's try a list:

- It's hard to communicate without being face to face. Most communication is done via e-mail.
- People being in different time zones doesn't allow enough overlapping hours of work.
- Language and cultural barriers inhibit communication.
- It's difficult to keep track of who is working on what task with a geographically distributed team.

Pulling something like this off sounds more like a magic trick than something that big, conservative, corporate America is likely to take part in.

But, we actually have a successful existing model that we can look to for examples of how to make it work: Open-source software development.

Open-source projects are done by geographically distributed teams. In most Open-source efforts, the team members never get a chance to meet each other, much less work face to face. It's not even common for Open-source developers to speak to each other on the phone. They typically use e-mail or IRC (internet relay chat) for their meetings. They tend to use text (especially code) to communicate requirements and design. Open-source teams are often made up of people from all over the globe, speaking many different languages.

Open-source projects don't usually have "project managers." They are self-organizing. There is usually a leader, but the role of the leader is not at all focused around project management. Tasks get done because people need them to get done. Software and products evolve in the same way. Open-source project leaders will sometimes create high-level product road maps, but they usually don't break these down into workable tasks. That happens naturally and in a self-directed way, driven by members of the distributed team.

They manage all of this complexity, creating high-quality software that we are *all* using in some capacity somewhere.

They do it for free.

If I were a software developer looking to add *Distributed Software Development (offshoring) Expert* to my list of credentials, I'd go underground in the open-source world like a journalist going undercover with a street gang. I'd study their habits. I'd try to become one of them. See how they manage to pull this stuff together. See how they *fail*. What makes a successful project succeed? What makes a failure fail? (Hint: there are a *lot* of failures to draw on.)

Unlike a street gang, open-source projects are, well, "open." Thousands of case studies are wide open and ready for studying. In fact, with mailing lists and IRC channels logged and archived, you can follow the evolution of an idea from conception to implementation.

Go underground in the open-source world.

Also unlike a street gang, it's not likely you'll be killed trying to study the inner workings of an open-source project. Give it a try.

Act on it!

1. Observe successful open-source projects. make notes of how they do: communication, design, tracking work, tracking defects, source control.
2. Get involved in an open-source project. Fix bugs, add new features, respond to support requests, and try to get included in the release process.
3. Think about how offshore programmers are different from open-source developers. How does that change the way you deal with them? Which open-source processes are unlikely to work as well with the average offshore programmer? Which will work just as well?

Namaste. I could see eyes in the room light up. *Mera naam Chad hai.* Those still looking sleepy were suddenly at the edges of their seats. Those who had already taken notice looked as if they were on the verge of bursting with joy. I suddenly had the attention of this team of employees in a way that I was rarely able to achieve back home. All for saying, *Hello. My name is Chad* in India's official national language, Hindi.

As I got to know our team members in India, I often heard them say that I wasn't like the typical American manager. When I asked what they meant, those who felt comfortable enough would say, *You actually take an interest in us. Most of you are just angry and short with us.*

It became quickly apparent, as I interacted with the team in India, that this difference they had noted was a big advantage for me. Most of my colleagues in the United States, viewing offshoring as a necessary evil at best, treat offshore teams accordingly. They keep the teams at arm's length. And, it's not just managers. In this environment everyone has to interact with offshore teams, from managers to project leads, to businesspeople, to programmers, so we all need to be good at crossing the cultural boundaries, and most of us are bad at it.

I'm not just getting touchy-feely here about bringing the world together by having us all become global thinkers. There are pragmatic reasons to get involved and learn about the cultures of the people you work with. The most obvious reason is that, without putting forth an effort, it's very likely that you and the people you have to work with on a daily basis won't understand what each other is saying. Sure, you both speak English, but that vague thing we call *English* isn't enough to allow two diverse groups of people to really understand each other. There are obviously issues of accent and colloquialisms to deal with. For example, if I say, "I went down to the military hotel for a fag and saw Ramesh getting down from the auto," would you know what actually happened? The American English translation would read, "I went down to the non-vegetarian restaurant for a cigarette and saw Ramesh get out of a rickshaw."

One of my roles at the software center in India was *translator*. I do speak Hindi, but my translator job had nothing to do with that. Our business was headquartered in Kentucky, so many of the people there speak with something of a drawl. And, of course, the team members in Bangalore,

hailing from all parts of India, spoke with all manners of Indian accent, when speaking English. I was pulled into meetings that were completely irrelevant to me simply so I could translate English to English.

I remember the first time it happened. We were on a speakerphone talking to a network administrator in Kentucky. We were planning the upgrade of a file server in Bangalore, and the Kentuckian was instructing our Indian team member on how to perform the upgrade, since he had just completed the same operation at headquarters. The Kentuckian spoke for a minute or two, ending with a question. I was disinterested and zoned out for a moment until I realized there was an abnormally long pause. The Indian was looking at me with a worried look. *Did you understand?* I asked. He shook his head. So, I repeated what the Kentuckian said, pretty much word for word. In English. After I finished, the Indian network administrator responded (in English). The Kentuckian paused, and then asked if I could repeat what was said. And so on, and so on.

The problem as I saw it was that neither of these people was trying to either understand the other or help make themselves understood. They were both either insensitive or lazy (or both). Because of their cultural ignorance, we had to have a translator with no special qualifications bridge the gap between them. How horribly inefficient! Thankfully, over time, we campaigned enough that both sides of the ocean made great strides toward understanding each other both verbally and culturally.

There are other tangible reasons to explore the culture of your offshore colleagues. As I illustrated at the beginning of this section, a basic attempt to understand the culture of a colleague shows you are interested in this colleague as both a co-worker *and* a human. Americans are particularly bad about taking for granted that everyone they interact with is America savvy. If they aren't, they must be stupid. So, it's no surprise to us that our colleagues in India should know about our pop culture and our history. In fact, if they *don't* understand us already, we get irritated.

However, what do you know about India (or any other country you might be working with)? In my experience, the average American's answer would be "zilch." People in India, for example, don't speak Hindu (Hindu is a religion). And, there's no such thing as a Hindi temple (Hindi is a language). What's worse is that these mistakes were made by people who had *been* to India.

Nope. Most of us just don't care about other cultures. It's sad but true. But that leaves openings for those of us who do to be special. I am gen-

uinely interested in India, its culture, and its people. Whenever I work with Indian offshore teams, I delight in asking each team member where they're from, what languages they speak, and what their favorite foods are. It brings them closer, and strong working relationships make happier, more productive teams. If I have to depend on someone to get something done for me or to deliver a piece of software that I have to successfully integrate with, I'm going to have much better luck if that person feels I respect them and if they respect me. Would you respect someone who wouldn't even bother to learn how to pronounce your *name*?

If I have to depend on someone...I'm going to have better luck if that person feels that I respect them.

As with most of the topics I talk about in this book, none of this is limited to India. I had the same experience in Hungary. I went over from Bangalore for a week to lay the foundation for a new development center my company was going to set up. A week before I went, I got a book and started learning some Hungarian phrases. When I met people, I would attempt to greet them in my broken Hungarian. Within a couple of days, the news had spread, and I was greeted by every new person I met with a boisterous "Jo Napot Kivanok!" and a smile. I was now known as the American who speaks Hungarian, which was a shameful overstatement. But, the resultant attitudes and cooperation of the people I had to work with were greatly appreciated, and my week in Hungary was a very successful one.

What I've noticed since coming back from India is that in America we are so focused on ourselves that we don't even take the time to learn about our teammates from other parts of the United States. What's the *special food* in Minnesota? What do Arizonans do on the weekends in their nonexistent winters? The United States is a diverse place, and we don't even bother to learn about our own diverse culture, much less the cultures of people on the outside. The same principles apply: if you show your teammates that you are interested in them as people, you will form tighter bonds and, on the whole, do better work.

Act on it!

- Learn to say "hello" in every language anyone you know speaks. Use those words on a regular basis when you speak with the appropriate people. Next, branch out, and learn more words, such as "thank you." Or learn short, fun colloquial phrases that will make people smile. Don't be afraid to sound stupid. It's inevitable. That makes it *better*.

But I say to you that when you work you fulfill a part of earth's furthest dream, assigned to you when that dream was born, and in keeping yourself with labour, you are in truth loving life, and to love life through labour is to be intimate with life's inmost secret.

► Kahlil Gibran, *The Prophet*

What I Learned in India

Naturally, after living in Bangalore for a year and a half, I felt a little anxiety in the days leading up to our return from India. We had immersed ourselves quite fully in what it means to be Indian, and now we were going back to the incredibly foreign world of Big Macs and basketball. I regarded it as a visit to the dentist. I know going to the dentist is good for me, but it always makes me nervous. And the worst part of a dentist visit is not the visit itself but the time leading up to it.

But, late one balmy May night, we packed up the last of our scant few belongings and took the long drive across Bangalore to the airport. We looked out the car windows at the pothole-filled streets with a sense of belonging. We passed row after row of shops with their garage-door fronts closed. What was once exotic was now familiar. This alien world was now home to us.

Change is like that. The new quickly becomes old. The things you desire now will become at best familiar and at worst junk after you've attained them. The exotic and the mundane had traded places for us. We were now faced with the adventure of returning home. As we were sad to say goodbye to home when we left it, we were now sad to say goodbye to our new home. It was fitting that we were driving out at night. Everything was shut down. The party had ended, and everyone was at home.

A trip to or from India is so long and so exhausting that it serves as a kind of mental reset. This mental reset was amplified for us, because we chose to stop in Europe for a day to break up the trip. By the time we reached the United States, we were tired and confused. Walking out of the airport in Chicago, I felt like an immigrant arriving for the first time, wide-eyed and awe-struck by the sterile, mechanical engine of capitalism that unfolded in front of us.

Our first experience back in the motherland involved hailing a cab to take us to a hotel from which we would recover enough to drive back home to Louisville. In the United States, people drive really, really fast. It's freakish

the first time you get in a car and glide onto the ice-smooth roadways. It's scary. You probably can't imagine that it could be scary to ride in a car here, but it is.

In America, you get farther faster than is possible in India. In fact, you can generally do everything faster. Life is more convenient that way.

When you're thirsty, you can go to the tap and pour a glass of water and drink it. You don't have to wait the eternity it takes for water to drip indignantly from a filter. You just pour and drink with no fear for your health. And, when you turn on the tap, water almost *always* comes out.

You don't have to buy alarm clocks with batteries, because, unlike in Bangalore, on most days the power doesn't go out.

And the Internet! Being the junkie that I am, I was amazed at how fast it was. Downloads were so quick that they were unnoticeable. I was irrationally convinced that the Internet had actually somehow sped up since I'd left. And with the electricity always being available and our cable modem service reliable as it is in the United States, I once again felt reconnected to the thriving community of technology thinkers that I had missed so much since I'd moved. I could pop down to the local bookstore and get anything I wanted. In America, you can turn on a firehose of information whenever you like. I thirstily drowned myself in books, articles, and open-source software as soon as I returned.

Having been immersed in the Indian IT perspective for well over a year, part of me worried how the IT people of the United States could keep up. I'd seen the IT sector growing so fast in India that I wondered how my compatriots could possibly maintain their hold on the job market. However, as soon as I returned, I saw it. Americans are blessed with the infrastructure, the wealth, and the freedom to *choose* their destinies. This is a damned good place for a software developer to live. The best place, in fact.

So, why did it feel so grim back in the office?

If there's one thing I took away from India, it's that people there who live in what we Americans would consider to be destitute poverty are on average happier than us. I met people who were dirt poor, living in tiny little houses, but had somehow managed to develop an outlook that was decidedly much healthier than mine.

It's from this exposure that I really learned that it's not what you do for a living or what you *have* that's important. It's how you choose to accept

these things. It's internal. Satisfaction, like our career choices, is something that should be sought after and *decided upon with intention*.

Resources

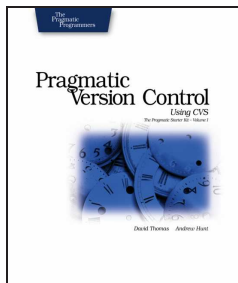
A.1 Bibliography

- [Cou96] Douglas Coupland. *Microserfs*. Regan Books, New York, 1996.
- [DL99] Tom Demarco and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House, New York, NY, second edition, 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [God03] Seth Godin. *Purple Cow: Transform Your Business by Being Remarkable*. Portfolio, 2003.
- [Ham02] Gary Hamel. *Leading the Revolution: How to Thrive in Turbulent Times by Making Innovation a Way of Life*. 2002.
- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
- [Pir00] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. Perennial Classics, reprint edition edition, 2000.
- [Sil99] Steven A Silbiger. *The Ten-Day MBA: A Step-By-step Guide To Mastering The Skills Taught In America's Top Business Schools*. Quill, 1999.

Pragmatic Starter Kit Series

Version Control. Unit Testing. Project Automation. Three great titles, one objective. To get you up to speed with the essentials for successful project development. Keep your source under control, your bugs in check, and your process repeatable with these three concise, readable books from The Pragmatic Bookshelf.

Pragmatic Version Control



- Keep your project assets safe—never lose a great idea
- Know how to UNDO bad decisions—no matter when they were made
- Learn how to share code safely, and work in parallel
- See how to avoid costly code freezes
- Manage 3rd party code
- Understand how to go back in time, and work on previous versions.

Pragmatic Version Control using CVS

Dave Thomas and Andy Hunt

(176 pages) ISBN: 0-9745140-0-4. \$29.95

Pragmatic Version Control using Subversion

Mike Mason

(224 pages) ISBN: 0-9745140-6-3. \$29.95

Pragmatic Unit Testing

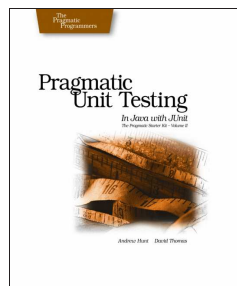
- Write better code, faster
- Discover the hiding places where bugs breed
- Learn how to think of all the things that could go wrong
- Test pieces of code without using the whole project
- Use JUnit to simplify your test code
- Test effectively with the whole team.

Pragmatic Unit Testing

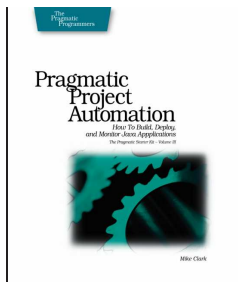
Andy Hunt and Dave Thomas

(176 pages) ISBN: 0-9745140-1-2. \$29.95

(Also available for C#, ISBN: 0-9745140-2-0)



Pragmatic Project Automation



- Common, freely available tools which automate build, test, and release procedures
- Effective ways to keep on top of problems
- Automate to create better code, and save time and money
- Create and deploy releases easily and automatically
- Have programs to monitor themselves and report problems.

Pragmatic Project Automation

Mike Clark

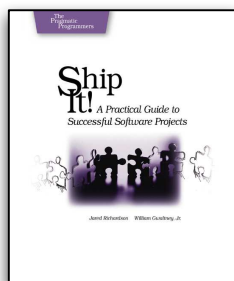
(176 pages) ISBN: 0-9745140-3-9. \$29.95

Visit our secure online store: <http://pragmaticprogrammer.com/catalog>

Help for Programmers

A large part of the message in this book is that “just a programmer” doesn’t cut it anymore. Developers increasingly have to branch out into project and management areas if they are to stay competitive (and employable). Here are some books that will help.

Ship It!



This book shows you how to run a project and *Ship It!*, on time and on budget, without excuses. You’ll learn the common technical infrastructure that every project needs along with well-accepted, easy-to-adopt, best-of-breed practices that really work, as well as common problems and how to solve them.

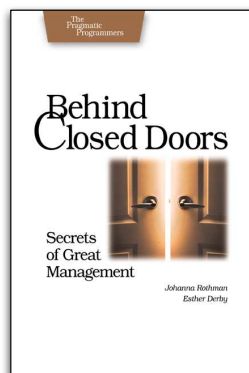
Ship It!: A Practical Guide to Successful Software Projects
Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. \$29.95

Behind Closed Doors

If you’ve been lucky, you may have seen the results of great management, but it’s not always easy to see how managers achieve those results. Great management happens in one-on-one meetings with team members and in meetings with other managers fall in private. It’s hard to learn management by example when you can’t see it. This book opens the doors wide so you can see exactly how it’s done:

- Delegate effectively
- Use feedback and goal-setting
- Develop influence
- Handle one-on-one meetings
- Coach and mentor
- Decide what work to do and what not to do
- ...and more.

Behind Closed Doors
Johanna Rothman and Esther Derby
(200 pages) ISBN: 0-9766940-2-6. \$24.95
(Available Fall 2005)



Visit our secure online store: <http://pragmaticprogrammer.com/catalog>

The Pragmatic Bookshelf

The Pragmatic Starter Kit series: Three great titles, one objective. To get you up to speed with the essentials for successful project development. Keep your source under control, your bugs in check, and your process repeatable with these three concise, readable books.

Facets of Ruby series: Learn all about developing applications using the Ruby programming language, from the famous Pickaxe book to the latest books featuring Ruby On Rails.

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

My Job Went to India

pragmaticprogrammer.com/titles/mjwti

This book's home page, including errata and other resources.

Register for Updates

pragmaticprogrammer.com/updates

Be notified when updates and new books become available.

Join the Community

pragmaticprogrammer.com/community

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

pragmaticprogrammer.com/news

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/mjwti.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com