Working with Coders

A Guide to Software Development for the Perplexed Non-Techie

Patrick Gleeson

Apress®

WORKING WITH CODERS

A GUIDE TO SOFTWARE DEVELOPMENT FOR THE PERPLEXED NON-TECHIE

Patrick Gleeson

Apress®

Working with Coders: A Guide to Software Development for the Perplexed Non-Techie

Patrick Gleeson
London, United Kingdom

IISBN-13 (pbk): 978-1-4842-2700-8 ISBN-13 (electronic): 978-1-4842-2701-5
DOI 10.1007/978-1-4842-2701-5

Library of Congress Control Number: 2017946183

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Coordinating Editor: Rita Fernando
Copy Editor: Larissa Shmailo
Cover: eStudio Calamar

For Sylvia, next to whose crib much of this was written.

Contents

About the Author

Patrick Gleeson has been a coder and a manager of coders for the past ten years. He has worked in a variety of organizations, from bespoke software consultancies to multinational corporations to tiny start-ups, and is currently CTO of Think Smart, a company that provides tools to help young people make better career choices. He holds a degree from the University of Cambridge in philosophy and classics, and another one from the London Academy of Music and Dramatic Art in technical theater. He also sidelines as a composer for film and theater, and once spent a year building animatronic puppets as part of a robot circus, including a mechanical octopus that played the xylophone.

Acknowledgments

Many thanks to the fantastic team I've worked with at Apress: Rita Fernando, Laura Berendson, Susan McDermott, and Larissa Shmailo. I am also particularly indebted to Robert Hutchinson for seeing a kernel of potential in my initial pitch, and championing the project at its earliest and most vulnerable stages.

Finally, endless gratitude to my wife Ellie, who not only tolerated but even encouraged my work on this book during one of the busiest and most hectic times in our lives so far.

Introduction

A couple of years ago I went for an interview with a start-up based in a busy shared working space in East London. The founder, Ali, was a brash ex-City trader type, who'd carved out a career for himself working first in credit derivatives and then real estate. He'd had an idea about a way to disrupt the property rental industry, and had pulled together enough funding to hire a small team of coders to build him a prototype app. After giving me the standard grilling about my past experience, and apparently satisfied by my responses, he gave me a chance to ask some questions of my own. I inquired as to the current state of his prototype, and he assured me that great progress was being made.

"The way I see it—right?—building software is just like building a house," he told me, nodding in agreement with himself as he spoke. "First you design it, then you plan the build. At that point you know how long it's going to take, and from there you just get on and build it. Simple, yeah? Now, we've got the design and we've got a project plan, and our beta is set to launch in June. At the moment I'll admit we're a little bit behind schedule, but I'm hiring in another dev—that could be you—so we'll be back on track in no time."

He sat back, satisfied with himself, and in that moment I realized that something peculiar had just happened. Even though all I had to go on was what he himself had told me, I had just come to know something about Ali's company that Ali himself did not know, namely that there was absolutely no chance *whatsoever* of the beta launching in June. His explanation of the current state of play had thrown up so many red flags I felt like I was at a Maoist rally. But before I could respond, Ali had turned the conversation on to his exit strategy, and what that meant for the options package he could offer to any new hires, and the opportunity to talk about timelines didn't come up again.

I didn't accept the job that I was eventually offered (discreet inquiries among the existing team confirmed my suspicions that Ali was not the most easy-to-get-along-with employer), but I kept tabs on the company nonetheless. Sure enough, June came and went without a launch. So did July, and then August. I think they eventually got something out the door in October.

I found myself thinking more and more about *why* I had been so confident that they would miss their deadline. I realized that there were some hard truths about software that I had learned through bitter experience, to which Ali wasn't privy. How could he be, never having worked with coders before in his life? For example, I knew that software development is nothing like building a house. That trying to finalize a design for a consumer-facing app before any prototyping has been done almost *guarantees* that you'll have to redesign at some point before you can launch. That if you're behind schedule on a software project the very *last* thing you should do is add more developers to your team. But he, understandably, didn't have a clue about any of this.

The more I thought about it, the more it seemed to me tremendously unfair that the Alis of this world, whose professional success depends on successful software projects, and who are responsible for taking the decisions that will either sink or save such projects, don't know about all those quirks and idiosyncrasies of software development that cause the right decision to be so often the most counterintuitive one. Someone should tell them, I thought. About all of it.

Then it dawned on me that that someone might as well be me, so I wrote this book.

CHAPTER

1

Introductions

You, Me, and This

This is a book about how software is created, and the people who do the creating. In particular, it's about how weird and idiosyncratic the process of creating software is, how fickle, and how disaster-prone. And believe me, it *is* disaster-prone. A study of IT projects at large organizations showed that the average budget overrun of projects that included software creation was 50% higher than of those that didn't, while the average schedule overrun was *ten times higher*.[1] It's not just that software creation takes longer than anyone predicts and costs more; it's that the extent to which it defies prediction and consistently disappoints is staggering.

The premise of the chapters ahead is that things don't have to be this way. Software development disasters normally occur because the weirdnesses and idiosyncrasies of the process are misunderstood and ignored, and there is a tendency to treat building a piece of software like building a house. But, and this is a point I will return to again and again:

Building software is nothing like building a house.

[1]Michael Bloch, Sven Blumberg, and Jürgen Laartz, "Delivering large-scale IT projects on time, on budget, and on value," McKinsey & Company, October 2012, http://www.mckinsey.com/business-functions/business-technology/our-insights/delivering-large-scale-it-projects-on-time-on-budget-and-on-value

P. Gleeson, *Working with Coders*, DOI 10.1007/978-1-4842-2701-5_1

This book is a guided tour of the process of software development, complete with a look at the anatomy and psychology of software developers, to help avoid the misconceptions and misunderstandings that blight software projects. I wrote it for you, for a given value of "you," so let's address that before anything else.

Who you are

You are someone who needs some software to be written, and you're not going to write it yourself. We can go a little bit deeper. I'm going to draw three thumbnail sketches of you, and if any are a likeness, you can be assured that this book is for you.

The Project Manager

You've got your Prince2 accreditation under your belt, and the first couple of projects you worked on at your company gave you plenty of confidence putting the theory into practice.[2] You make a Gantt chart like other people make coffee, and you could risk assess a banana sundae as easily as eating it. Your success with the roll-out of the updated accounting software got you name-checked by the COO in the monthly all-hands meeting, and your line manager is telling you there are big things in store if you keep it up.

But then came the big internal systems revamp. Unlike your previous projects, this one is about building and launching a new piece of software. You've been allocated some developers from the Basement, and they seem pretty friendly, but the senior Java engineer has been slightly less than helpful when you've tried to get her help in the project planning stages. She keeps saying that the right thing to do isn't to try to "reinvent the wheel" with a whole new system but rather to take the time to properly rework the existing system. Which isn't very helpful, because the *whole point* of the project is that people are sick of trying to work with the existing system and you've finally been allocated the budget to make something better. You won't get the stakeholder buy-in if you just give people more of the same, but she doesn't seem to see the importance of that. And she keeps insisting she can't possibly tell you how long it'll take to build the new system until you rewrite your carefully written specification documents as a set of "user stories", which seem to be exactly the same thing except that every sentence has to start with "As a user." And she keeps complaining that you're under-resourced, and that you either need to offshore the work or bring in some contractors, even while maintaining that she can't quantify how much work is actually involved that needs resourcing. Meanwhile the project kick-off is getting closer and closer…

[2]If you're not familiar with it, Prince2 is a project management methodology used extensively in organizations like the United Nations.

The CEO

Well, you did it. You took a chance and made a change, and now you're a founder in the exciting world of start-ups. You've got a great idea for a web app, a kick-ass investor deck, enough seed funding for 9 months of runway, and you're raring to get started putting everything you read in Eric Ries's book into practice.[3] You've even made your first hire, and have your CTO in place. He's a little bit young, but he's got some amazing previous experience and really impressed you at interview with how knowledgeable he sounded.

But there are a few things that are worrying you. Your CTO has told you that the prototype you got built isn't fit for purpose because the original contractor used something called PHP, but apparently that's not a "proper" language, so he's going to have to re-make it from scratch using something else called Node that sounds like basically the same thing. Plus he wants you to spend an awful lot of money on something called Assembla but hasn't really explained what it's for. And he keeps insisting he can't possibly give an estimate for how long it'll take to build any new features once the rebuild is complete, unless you rewrite your user stories as a "functional spec," even though as far as you can tell that's exactly the same thing except it uses the word "shall" a lot. Obviously you trust him implicitly on technical issues, but somehow everything seems to have gotten just a little a bit harder since he came on board, and you have a nagging sense after every meeting that you're not really understanding each other…

The Client

Business has been good recently. After a few years of lean times the company has built up a little bit of a reputation locally, and a combination of word of mouth and some positive reviews online has brought in a fair amount of business. There's even been a nomination for some industry awards—although to be fair, a lot of people get nominated, and they charge quite a lot to attend the awards ceremony, so your business savvy tells you it's probably not worth it. But nevertheless, it's good to be noticed.

If there was one thing you could wish was going slightly better, it would be the website. You found a friend of a friend to build it originally, back when all you needed was basically a home screen with a phone number. Over the years you've added more and more to it, from some "contact us" pages to a blog, and most recently an online reservation system. Eventually the friend of the friend left town, and recommended a local digital agency to take the work over. For a while you've been noticing how slow the website seems to have gotten recently. You keep mentioning the slowness to the agency, and they say

[3] More on Ries's book in Chapter 3.

they'll take a look, but nothing ever seems to get done about it. Then there's that ongoing issue where pages look all broken sometimes, but whenever you try to get them to fix that they say things like "we couldn't repro it," or "it's probably a caching thing, so it won't affect new visitors." You're not sure you trust that. Now you've set aside some budget to expand the website to include selling some products online directly, but when you talked to the agency about it they got a bit cagey and started talking about something called "technical debt." Plus when you tried to give them a thorough description of the new pages you'd envisaged they shot you down and said they prefer to work in an "agile" way, which as far as you can tell means making it up as they go along rather than planning it out in advance. You like them, and they say you're one of their favorite clients, but sometimes you feel like they're taking you for a ride…

Sound familiar?

If you can even remotely identify with any of the above, then this book is for you. But even if you can't, you might get something from this book. If you're not a software developer and you have software developers working alongside you, under you, or (this is rare) above you, and you'd like a better understanding of how they work, read this book. If you have to make decisions that affect or are affected by the work of software developers, read this book. If you *are* a software developer, and you want some insight into how your work relates to the work of your non-technical colleagues, read this book. If so far I haven't said anything that even remotely resonates, then… well, you've made it this far. Only a couple of hundred pages to go. What have you got to lose?

From now on I'm going to be making some assumptions about what you know. I'm going to assume that you don't know the first thing about computer code. I'm going to assume, for example, that you'd struggle to explain the difference between HTML and HTTP. I'm also going to assume that you don't particularly *care* about that difference, except insofar as the knowledge can be used to get the job done faster and better, whatever "the job" happens to be.

That's you covered, then. Let's talk about me.

Who I am

I was like you once. That is, I used neither to know nor to care about the difference between HTML and HTTP, or any such technical guff. It was a simpler time. A happier time? Possibly.

Then I got a job as a software developer. Helpfully, my first employer didn't require any prior knowledge or experience, which meant that my BA in philosophy and classics wasn't held against me. Over the ensuing decade I worked in a variety of roles: as a software developer, a manager of software developers, a project manager, a product owner, and a carpenter.[4] I've been fortunate enough to work in a variety of organizations, from start-ups to large companies to software consultancies to freelance gigs.

Over the course of my career to date I've made a tremendous number of horrible mistakes. Real stinkers. Many of these have been purely technical and, being mercifully irrelevant to this book, can be set to one side and ignored. But many, particularly those that involve my decisions when managing software developers and software projects, have taught me, painfully, a tremendous amount about what works and what doesn't work when it comes to getting software done. I have also had the opportunity to watch (and be secretly comforted by) the mistakes made by my colleagues and superiors that nearly rivaled my own. And in reflecting, in the long, dark 3am crises of the soul that all managers go through, on all the mistakes I have made and witnessed, I have observed a few commonalities. It has occurred to me that there are some pieces of knowledge that, were they known by me and my colleagues beforehand, might have helped avert some of the many, many mistakes. Hence this book.

What this book is

This book is designed to tell non-technical people a bit about how software development works so that said non-technical people can make better decisions. It's a fairly unsurprising fact in business that since software development is generally a supporting function in an organization, software developers tend to work for non-developers, rather than vice versa. This means that the CTO answers to the CEO, the Senior Engineer reports to the Project Manager, and the digital agency does what the client pays them to do. In each case, therefore, it's the non-technical person who takes the decisions and gives the orders. Now, obviously on purely technical matters the non-technical person isn't qualified to take decisions, so for those, decision-making power is delegated to the technical people. It's all the non-technical stuff—the commercial, logistical, aesthetic stuff, etc. —that the non-technical person is in charge of.

But herein lies the rub: all that non-technical stuff affects and is affected by the technical stuff. And so to make the right decision, the non-technical person, not knowing much about the technical stuff, has to make sensible assumptions about what those effects will be, based on logic, intuition, and analogies with

[4]Long story.

better-understood domains. Which would be fine, except that it turns out that the technical stuff is illogical, counter-intuitive, and doesn't compare at all easily to anything else. I will repeat that, because it's a central theme to this book:

Software development is illogical, counter-intuitive, and doesn't compare at all easily to anything else.

That being the case, it's unsurprising that it's monumentally difficult to bring a project that involves software development to completion on time and within budget. It takes an understanding of the weirdnesses and mysteries of the process that few people who haven't actually spent years writing code as part of structured projects really have.

This isn't to say that if we put coders in charge all software projects would go swimmingly. Far from it: as with any project, the best people to have in charge are the people who have specific experience, skill, and training in the fine art of *being in charge*. But the point that I will repeatedly try to prove in this book is that when it comes to delivering software, the people in charge aren't normally equipped with the information they need to take the best decisions, because they've seldom had the luxury of many years in which to study the curious beast that is software development and learn its mysterious ways.

This book is an attempt to provide a shortcut to some of that information, to ensure that a good leader of software projects doesn't *have* to have had hands-on experience as a software developer. It's a primer in the arcane and obscure world of the coder to help you, the non-coder, make the right decisions, and not make the sorts of mistakes that I and my colleagues have spent our careers learning too late were mistakes.

To a large extent it's not, therefore, an attempt to say anything massively profound, innovative, or unusual. Rather it's an attempt to produce a useful digest of a lot of things that are, generally speaking, known, but that are typically not known by some of the people who would benefit most from that knowledge, i.e., you.

In Chapters 2 and 3 we will cover some of the biggest conflicts between traditional, intuitive ways of planning and running projects and the software development process, and evaluate some ways of avoiding these conflicts. We'll cover the ways in which software management has evolved, with a particular focus on Agile development. If you don't know what that is, don't worry, we'll go through the fundamentals. We'll evaluate its strengths and also its weaknesses.[5]

[5]If you know any Agile devotees you may want to not let them see you reading the bit about weaknesses—it's a movement that inspires a certain fanaticism that sometimes has very little tolerance for criticism. If you are an Agile devotee of a fanatical bent then please at least read the bit about weaknesses before sending me your hate mail!

Chapters 4 through 6 provide an introduction to what software developers do and how they do it. We'll cover the process of software development, the terminology involved, and everything software developers do that isn't actually writing lines of code.

Chapters 7 through 9 turn the focus to managing software developers, as distinct from managing software projects. This includes some advice on some very specific things, such as how to go about hiring a software developer, as well as a more general exploration of the psychology of the coder, looking at the pressures and priorities that occupy developers' minds.

The final chapter might seem to be a rather dispiriting affair, focusing primarily on how to manage failure. However, what I hope to show is that, this being an imperfect world, things never go entirely according to plan, and the mark of a great leader is the ability to triumph in adverse circumstances. Chapter 10 offers some advice on how to move forwards when things go wrong, and hopes to end on the positive message that no disaster, no matter how great, is ever as bad as it seems.[6]

What this book is not

This is not a book about young white male nerds

Let's talk about stereotypes. I'm sure I don't need to tell you that in western society there are persistent, often somewhat derogatory stereotypes associated with software developers. It is easy to assume that developers will be geeky and/or nerdy,[7] and for all that "geek culture" has done something to rehabilitate geeks in general, the prevailing sentiment in business contexts, in my experience, is that geekiness entails a wealth of undesirable personality traits. Furthermore, certain demographic assumptions tend to be attached to the stereotype of a software developer. The assumption is that, when you are talking about a developer, you are talking about someone male, white, and probably under 35.

And let's be clear: based on the makeup of the coder population of Europe and North America today, these assumptions and stereotypes have some foundation. The majority of software developers here and now *are* male, they *are* white, and they *are* under-35. There are studies and statistics that demonstrate this.[8] It's harder to find statistics that quantify geekiness, but

[6]Unless your disaster happens to involve the UK's National Health Service and an IT project worth billions of dollars. In that case it's every bit as bad as it seems, and you should be ashamed of yourself for what you have done to the reputation of IT, the NHS, and the UK as a whole. More on this in the next chapter.
[7]There are many rival schools of thought concerning the distinction between the two terms. I won't risk partisan outrage by expressing my own opinions on the matter.
[8]`http://www.bls.gov/cps/cpsaat11.htm` and `http://stackoverflow.com/research/developer-survey-2016` offer particularly clear insights.

speaking as a proud geek who has worked around a fair spread of software developers, I would be prepared to posit that the average developer is significantly geekier than the average person.

The question then arises: this being a book on how to work with software developers, and there being a trend in software developers towards being young, white, male, and geeky, should this book deal with how to work with young, white, male, young geeks? My answer to that is emphatically no, for three very serious reasons.

The first is that only catering to the majority is always an unreliable practice. The majority of diners at a restaurant don't have lethal peanut allergies but that doesn't mean it makes sense to sprinkle nuts aplenty over every dish with merry abandon. The majority of students in a classroom will be in the bottom two thirds of the class academically, but it would be a foolish teacher who made no effort to engage or challenge the top thirty-three percent. Likewise, even if it is the case that majority of software developers belong to a specific demographic, and it is possible to devise ways of working that are particularly suited to that demographic at the expense of others, those ways of working will still probably be deeply unsuitable when applied to a team that doesn't exclusively match the demographic. It would be irresponsible and pointless to try to identify and recommend them.

The second reason is that nothing dates a text like outdated assumptions. While it may be the case that at the moment there's a tendency for programmers to be white, male, young, and nerdy, there's no good reason to suppose that this trend will continue forever. In fact, there's good reason to suppose it will change. There's nothing fundamental about software that means only young white male nerds are suited to creating it. First of all, let's not forget that the author of the first computer program ever to be published was a woman—the brilliant and possibly unhinged Ada Lovelace, daughter of Lord Byron, worked with Charles Babbage to write software for an entirely theoretical piece of hardware of his devising, and credit for the entire concept of software is due in large part to her. Furthermore, the gender skew is not a global phenomenon—in his fascinating book, *Geek Sublime*, Vikram Chandra reports that in 2003 in India, 55% of Bachelor of Science degrees in computer science were awarded to women. Furthermore, the change is coming to America: in an attempt to gain more students, universities and colleges are working to attract more women, and in some cases it's working spectacularly well. Thanks to a concerted effort to address gender imbalance, in 2016 more than half of computer science majors at Harvey Mudd College, California, were women. Likewise racial diversity among software developers is increasing, albeit painfully slowly. And the thing about all those bright-eyed twenty-somethings who were lured into coding by the modern mythology of the tech billionaire? They're getting older every year, balancing out the age skew. It also goes without saying that an increase in demographic diversity will reduce the applicability of simple personality stereotypes as well.

So while it may well be the case that certain assumptions about who developers are will be likely to be broadly accurate, statistically speaking, right here and right now, relying on those assumptions in this ever-changing world in which we're living would be short-sighted.

My final reason for avoiding stereotypes is that the stereotypes about what sorts of people become software developers have a tendency to be self-reinforcing. Believing in them and assuming them can lead to setting up an environment in which it's harder for people who don't conform to them to make their way in the software world. I don't think it's controversial to state that diversity and inclusivity are valuable things both in their own right and for the benefits that they precipitate. Therefore, it seems to me that the best thing I can do to promote those two goals is to avoid regurgitating and reinforcing the narrow stereotypes that can stand in their way.

This book will not, therefore, indulge in caricaturing software developers. If you were hoping for a manual on how to work with nerds you will be disappointed. We will be looking at the psychology of the coder, and making actionable generalizations about how coders are likely to think and act, but those generalizations will be based on assertions about what it is to develop software for a living, not what it is to conform to a stereotype.

This is not a book about how to code

I should also be explicit in stating that this book will not try to teach you how to write or even read code, and nor will it attempt to convince you that you should learn these things for yourself. I am not an evangelist for the profession[9]. Everything we will cover will be focused on helping you work *with* software developers, not *as* a software developer, and where we do look in depth at a technical topic it will normally be to make it easier for you to have productive conversations about that topic with the experts you work with, rather than become an expert in your own right.

On that note, it's worth being clear that this book is going to simplify a *lot* of things. I am going to give a lot of definitions of technical terms and explanations of processes, and they will not be entirely complete and entirely accurate. When it comes to tech and process, every definition has caveats and exceptions, and I will not be jumping down every rabbit hole. There are two reasons for this. The first is that I could easily fill up a book debating, for example, the finer points of what constitutes a database in the post-SQL

[9]Or rather, in general I am, because it's an intellectually stimulating type of work with an engaged and engaging international community around it, normally offering very good pay, decent hours, a low barrier to entry and good career security. But outside of this footnote I won't try to push this belief on you anywhere else in this book.

age, and whether BDD has to involve writing unit tests that simply exercise the same code pathways that the higher-level functional and integration tests already cover. But if I did I'd never get around to writing the parts of this book that I actually want to communicate. The second, more important, reason is that you really shouldn't need to care about the fine detail. What you need is working definitions that help you get the job done. Therefore please note that when I say, "A database is…" or "BDD involves…" you should be aware that in my head I am adding to the end of each sentence: "(with some caveats)."

This is not an attack on non-technical people

Lastly, this book is not an attempt to absolve software developers from their share of the blame for a terrible track record of software projects in the last fifty years. Yes, it is a premise of the book that we can make the process of building software much less painful by changing the behavior of the non-technical people involved, but that doesn't imply that the non-technical people are the sole ones at fault. Rather, I believe that the non-technical people (i.e., you) are the ones who can do the most to address the problems with software development, regardless of whose actions or attitudes might be the original cause of those problems.

CHAPTER

2

Why Writing Software Is Nothing Like Building a House

Three Big Problems in Software Projects

You might think it would be a truth universally acknowledged that in order to get something done, first one should work out what to do, and then one should do it. In this chapter we will demonstrate that this maxim, self-evident though it may be, does not apply at all well to software development.

P. Gleeson, *Working with Coders*, DOI 10.1007/978-1-4842-2701-5_2

The sad truth about software projects

Let's start with an example.

My first introduction to the wonderful world of software came in the form of a job as a junior code-monkey[1] at a software agency. One of our regular clients was an insurance firm, for whom we built systems that allowed their call-center teams to provide renewal quotes, and other similarly thrilling projects. The firm's latest flagship initiative was a partnership with a chain of high-end auto dealers where, at the point of purchase of a new car, the dealers would try to sell customers various forms of "premium" insurance that were entirely unrelated to car ownership. The customers would end up with some insurance they hadn't even known they needed, the insurance firm would get some more business, the dealers would take a cut—everybody would win.

Of course, this was contingent on the dealers being able to tell the customers what insurance they were eligible for, and how much it would cost, and for that they would need some software. The insurance firm had an in-house development team, but their time was booked up overhauling their internal systems, so my consultancy was brought in. We were given a set of requirements, for which we produced a bid, and once it was signed off a team of four of us got to work on building the quote generation website, codenamed Project Upsell.

That was when the problems began.

First of all, because our software was going to interface with the insurance firm's in-house pricing software, we needed to work closely with the insurance firm's in-house development team, who knew how it worked. However *they* resented our presence because the higher-ups had a tendency to refer to us as the "crack troops" who had been "parachuted in" to "rescue" their "overwhelmed" team, and they tended to make such references *in front of the in-house developers*.

The next problem came when we were trying to set up a database[2] of quotes so that we had a record of what quotes we had generated for which customers. It turned out we only needed a very simple database, and we had quoted the cost of the project on the premise that building it would be easy. However, two weeks into the development process, one of the in-house developers peeked at the source code we were writing and raised an immediate concern to his superior: we weren't using "EntityCapture"! This was deemed so serious that crisis teleconferences were arranged.

[1]In this book we will cover many technical terms that can be used to facilitate conversation with technical colleagues. This is not one of them.

[2]We'll talk a little bit more about databases in later chapters; but for now, if you don't know what they are, think of them like spreadsheets—big grids of information where each row is a record and each column houses a particular type of data about a record.

EntityCapture, it transpired, was a piece of enterprise[3] software for which the insurance firm had bought a very expensive license a few years previously, for use in their in-house systems. It was designed to handle storing a certain sort of insurance-related customer information in a database, but with a bit of pushing and pulling you could just about use it to store other things. The downside was that it was phenomenally complicated to put in place. We, having never heard of it, didn't think to try to use it, and when we did hear about it we decided we *definitely* didn't want to use it—it was sort of the equivalent of hooking up a tap in your bathroom to the hot water supply via the hydraulic system that powers construction site diggers: technically possible, but it'll make your plumbing objectively worse, no matter how good the hydraulic system is at being a hydraulic system.

It turned out that the business analysts at the insurance firm felt differently. An awful lot had been spent on EntityCapture, and there was some pressure from the higher-ups to demonstrate that it offered value for money. Therefore politically speaking it would be very convenient if it could be shown to be a key part of Project Upsell. And besides, pointed out the business analysts, the very compelling salesperson who had sold them EntityCapture had assured them that for this sort of project EntityCapture was essential, and would speed up development and improve "synergy" with other in-house systems. My manager protested that synergy wasn't a real thing. He pointed out that integrating EntityCapture would in reality put the project back by weeks, and warned that we would have to charge for additional time spent; the response came back that we should have anticipated this requirement in our initial quote, and so on. In the end we agreed to use EntityCapture and they agreed to pay us a little more, and everyone came away feeling slightly put upon.

There were further squabbles over short-notice changes to the requirements (or "clarifications" as the BAs so charmingly called them), delays as we waited for logos and branding details that never appeared, and, to be fair, a few really terrible decisions on the part of me and my team that contributed to the delays and disputes. But after much to-ing and fro-ing we had a 90% complete, demo-able version, and a meeting was called with some representatives of the car dealerships to start training them in how to use Project Upsell and iron out any last details.

And that was when the wheels really fell off.

It turned out that while the car dealers had enthusiastically agreed to the project in principle, the insurance firm hadn't consulted them when it came to drawing up the requirements. It was only in the meeting at the end of

[3]"Enterprise" is a term we will definitely come back to, but in the meantime, the best summary I can give is *Remy Porter's Law of Enterprise Software*, to wit: "If a piece of software is described in any way, shape, or form with the word 'enterprise' it's a piece of garbage."