The Developer's <code>

50 LESSONS, OBSERVATIONS, + UNADULTERATED OPINIONS ON WEB DEVELOPER LIFE THROUGH THE EYES OF ONE OF THEM.

KA WAI CHEUNG

Table of Contents

Introduction

This year, I turn thirty-two. By year's end, I will have been involved in the web development industry for over 40% of my life. It's been an awfully fast 40%.

When I started college in 1997, campus-wide *high-speed Ethernet* had just been installed across the campus. You plugged a T1 cord into your rather large desktop to log on to the WWW. Cell phones were a fashion statement. "Text" wasn't a verb. We left phone messages on answering machines. We still wrote most things down by hand...with a pen...in a notebook. We brought our CDs with us from home. Social networking still meant going out to a frat party, striking out with a few girls, and passing out on the front lawn.

Today's twenty-five year old entered college with a cell phone that was used primarily for calling, and left with one that let her answer calls, play music, write emails, read articles, and, most importantly, make farts. There's an app for that.

And pity today's poor twenty year old. She is currently spending more time (I mean *way* more time) chatting with friends on her laptop, phone, or iPad than together, at some real, physical place in the world. I recently peeked at my younger cousin's Facebook news feed. It looked more like a New York Stock Exchange ticker.

Society and culture is changing because the web is changing. Browser-based, mobile-based, and tablet-based apps, built by people like us are rewriting our rules of engagement and our societal ethos on the order of months, not decades.

And if society is morphing this rapidly because of the things we are making, it certainly has changed our very own jobs as web professionals. But, when you're in the middle of experiencing it, as you and I are, right this very moment, it's hard to gauge how fast change is happening.

This book is that gauge

My name is Ka Wai Cheung. I'm a web programmer, designer, author, blogger, speaker, and founding partner at We Are Mammoth in Chicago. We build applications for some of the world's best known companies, and create some of our own web-based software. You'll hear a bit more about those later.

This book is my gauge — a collection of lessons, advice, and opinions I've developed over the past decade in web development — from dot-com startup intern, to corporate developer, to tech author, to my current post at We Are Mammoth. I hope you'll make it yours too.

When I first started designing web sites as a summer intern at a .com startup in 1999, I had no clue how to go about all the intricate aspects of daily life as a programmer/designer. I didn't really know what the questions were.

Over the next decade, I'd encounter a plethora of them. Here's a sample:

- Aren't we all getting rich by coming up with a half-baked idea, finding some VC funding, going public, and having an exit strategy?
- What technologies should I be using?
- What makes me more happy, a bigger paycheck or a better way to work?
- How long do we have to keep wireframing before we can start coding?
- When is coding a bad thing?
- How do I convince a client that they're wrong, and I'm right?

No one had these answers written in stone in 1999, or 2005, or even today. Just like HTML markup and programming frameworks have evolved on-the-fly over the last 15 years and are continuing to change with the times, so too are the rules we've made for thriving in the industry.

Why I wrote this book

And, that's why I wrote this book — to paint the real picture, both to ourselves and to everyone outside the world of web development, to what this job is really about.

This is a book for software developers of all kinds, tailored to those of us building web-based applications. But, it has little to do with code. It doesn't matter if you program in C# or Ruby or Python or PHP or Java or JavaScript or ActionScript. It doesn't matter whether you're working on databases, server-side code or scripting the interface. This book is about everything that surrounds the professional web developer. These are topics that aren't just bound to class definitions, interfaces, and objects.

At the same time, there will be *some* talk about code. But, when I do talk about code, I'll be approaching it in a less technical, and a far more holistic way. You won't find a laundry list of best practices or design patterns. There are plenty of books that do a great job of this, and I'll mention a few along the way.

This book is about what makes our jobs difficult yet rewarding, when things go wrong or right, and what makes us tick. It's about why our industry is in constant flux, how to maintain motivation, using code to make our lives easier, understanding why things become complex, and becoming a better teacher. This book is about how a web developer should and shouldn't approach the craft of building web applications.

This book is about *us*. I hope it makes you think about, question, and conquer the world of web development a little bit better.

Metaphor

A web programmer *programs*, and a web designer *designs*. This is what we do. But what does that really mean? There are no reality TV shows, movies, or public interest stories that showcase how we really work. So, when I'm asked what I do, I often resort to analogy. Our industry is chock full of them. It's the way we describe our work to everyone else.

A chef doesn't have to come up with metaphors for cooking. A musician doesn't have to describe songs in some roundabout way. People get it. They are self-describing forms of work. Pipefitters and bricklayers have the job description succinctly written right in their names.

Perhaps it's because programming is just not common knowledge. The average person can't see what makes code elegant or unorganized. A broth is too salty because you can taste it; a melody too cliché because you've heard the same rhythm before. These things are easier to comprehend than things like poor encapsulation, memory leaks, or scalability.

Our medium, even dating back to the first computers, is *very* new. Humans have been cooking, drawing, dancing, and building for thousands of years. Archaeologists have yet to discover those cave paintings of *Man At His Desk Typing*, just yet.

So, metaphor has to become our meta-language. Not only is it how we connect the uniqueness of programming to the general public, but it's often how we make decisions on how to approach software problems.

#1: Be wary of metaphors in web development

And this is where things get dangerous.

Metaphor can make us value things that aren't that important, and undervalue the things that are. Metaphors are like your in-laws: Learn from them, but be weary of them.

Sometimes, the line between analogy and reality blurs. When we take these comparisons too far, we don't make our best decisions. Metaphor has a devilish way of blinding us from the truth. Our decisions might make perfect sense in the context of the metaphor, but when you strip it down to just the business of building applications, you can easily be led astray. We sometimes rely on the perceived benefits of the metaphor too heavily rather than paying attention to the reality of the situation.

For instance, we often use traditional architecture as a metaphor for building software. It's the origin of most of our organizational titles. That's why we call ourselves *software architects*, *information architects*, *senior developers*, *junior developers*, and *project managers*. That's why many of us still swear by wireframes, specifications, workflow diagrams, Gantt charts, and waterfall development. We've built a large portion of the software development process by piggybacking off of another industry.

These concepts are essential in another medium and they have *some* merit in our world as well. But, they can just as easily imprison us. If you've ever stopped to think about why you approach a problem a certain way, you might be able to trace it back to following the metaphor of traditional architecture (or another metaphor) too closely.

So, how has metaphor hurt us? Let's look at a few examples where we've stretched a concept from real architecture into a less-than-ideal fit for software.

#2: Plan *enough,* then build

In traditional architecture, planning is essential. Certain things unequivocally have to happen before others. Studs go in before plumbing, plumbing before walls, walls before paint. `Undo`, `Cut`, or `Revert` aren't viable options when you build a library, baseball stadium, or a skyscraper.

The software `Undo` is `CTRL + Z`. The software `Cut` is `CTRL + X`. The software `Revert` is a code rollback in source control.

"There's no control-X. That's what your scissors are for."

That's why buildings require specs, blueprints, and CAD drawings. In a traditional architect's dream world, materials would be infinitely available, a life-size model of a building could be risen in a week, and a to-scale suspension bridge stress-tested in a few days. If architects could instantaneously replicate a building to ten new lots at once, I'd imagine they would consider letting go of all the formalities of documentation.

Of course, all this is mere fantasy. So, writing detailed specifications in excruciating detail makes the most sense when you've decided to build a skyscraper.

On the other hand, these *are* the luxuries of our industry. Software components don't need to wait on a shipment of letters and numbers from the factory. You type, compile, test, and repeat. We can write and test code on what will become the real product — not some model of the real product. We have the luxury of watching our suspension bridges break hundreds of times while in development, in all different places, under all different conditions without the worry of wasting materials or putting people's lives in jeopardy. Working this way is completely feasible.

When we finish software, the exact same application can be shipped, sent, deleted, copied, and re-copied a thousand times with negligible human effort. When the developers of the Wynn Hotel in Las Vegas built a virtually identical twin hotel called the Encore in 2008, they didn't have the luxury of copying and pasting the 2005 version over into the vacant lot. They had to start with specs and planning even to build a nearly identical structure over again.

Even when software meant shipping floppy disks and CDs, planning extensively still made a lot of sense. But, web-based software is a different game. Specs, wireframes, detailed information architecture, and complete planning prior to writing a line of code still has merit, but it doesn't fully take advantage of the medium. For instance, we can release new builds daily or hourly — whenever we want, with very low overhead, from the comfort of our cushy Aeron chairs.

The "plan, plan, plan" metaphor overvalues the time we spend trying to get everything perfect, and undervalues the time we could be spending writing code, building, and iterating as we go.

Only recently has iteration, starting with code, and working without strict functional specs become more widely accepted. Agile development is not revolutionary — it's just untying us from a metaphor that doesn't make as much sense today as it did in the past.

That's not to say that traditional waterfall development is obselete. It still has its merits on many software projects in large organizations. But following that metaphor completely blind might also blind us from an approach that better fits the medium we work in.

#3: Understand launch and Version 2.0

Traditionally, we've approached a launch date as a mission critical point in time when software must be *final*.

For buildings and structures, it's essential. At one time, the metaphor made sense in software too. When we shipped software on floppy disks and CDs, things had to be just right. There were huge cost and time implications for bugs. Projects were delayed for the sake of getting it perfect, or for the sake of shoving in a new feature. I'll talk about what that does for morale in the next chapter.

Today, web-based applications aren't launched — they're uploaded, released, and pushed. Software lives and matures over time.

And, once we've launched, iteration #2, #3, and #4 can come a few days or even hours later. Even the concept of formal version releases of software is antiquated. It made sense in the bygone days of shipping software on disk.

On the web, we continuously integrate and constantly iterate. Unlike the auto industry or food industry, there's no need for mass "recall." Today, a critical bug can be patched, tested, and deployed immediately. It's not version 2.0 anymore. It's version 2.0.12931. Or, it's simply *today's* version. Is anyone in the public eye really keeping track anymore?

Society is growing accustomed to iteration too. Did you see the new image gallery on Facebook? Did you see Google's new auto-suggest feature? Did you see Twitter's new layout? Nobody warned us with a month-long advertising campaign. New changes just *appear* now.

IMVU (http://www.imvu.com), a popular 3D-chat application boasts over 100 million registered users, and they ship releases 20 times *a day*.

In today's landscape, the initial launch shouldn't feel like the end-all-be-all like it once did, or still does in many other industries. It's just one of hundreds (if not thousands) of mini-launches that take place during the lifespan of software. Keeping that perspective can relieve the mental pressure of launching software.

This mentality can be easily abused. Don't use this change of construct as an excuse for being lazy or leaving loose ends untied. The launch of any web app should be very, very good before others have at it. The big things need to be right. Proper security needs to be in place. But, the small stuff, the ones that are OK to fix afterward, shouldn't keep you from releasing software. You'll be surprised how often things you thought were important when you launch suddenly aren't now that it's out there.

And, you still ought to celebrate when software launches. Grab a drink, make a toast, and take your team out to a fancy dinner. But, don't spend all your emotional currency on just the wedding. There's an entire relationship you'll have with software afterwards. There's time to make adjustments, add in a family of new features, and right wrongs.

Launch is just another pinpoint in software's life. Not the end-all-be-all.

#4: The "Ivory Tower" Architect is a myth

I've never liked the idea that technical architects should stop coding.

In physical architecture, architects perch in an ivory tower, living in a world of only planning. They don't nail, drill, paint, or solder. Requiring architects to do the physical work of drilling holes and laying concrete is simply impractical. Architecting and developing are two distinct career tracks.

In our industry, you work your way up to the role of a technical architect by actually developing — by doing the "physical" work of building applications. But, as you move up the software development ladder, you typically write less code. You immerse yourself more with planning than with discovering the problems on the front-line. You concern yourself more about an overall vision and less about the intimate details of code. As an industry, we've really latched on to the metaphor that architects should plan, and developers should develop.

This creates a false perception that once you've reached a certain level, programming is no longer where you're most valuable. Leave the dirty work for the junior developers. At the same time, it pushes lower-level programmers away from thinking about the overall goals and direction of the project. Just concentrate on implementation. The architect-developer model makes both parties less accountable for the application as a whole.

When you split up roles into the somewhat arbitrary hierarchy of those that think about the technical "big picture" and those that only think in `if` statements, `for` loops, and markup, you fracture disciplines that really belong together.

Pure technical architects can make a guess at the best architecture, the best design pattern or the best practice. But, it's only when you're knee-deep in code that you discover where the real challenges exist. At the same time, the developer who isn't given the reigns to think high-level also doesn't get the chance to voice a second opinion. Often, it's the guy who's doing the actual work that can see the bumps ahead.

We've taken the architect-developer analogy too far. The corporate ladder in the software industry needs a better analogy.

To build a building, architects architect and developers develop. Traditional architects know how to create elaborate plans and specs in fine detail. But, they don't build. It's simply not reasonable. The separation between those that think high-level and those that work in the trenches is largely for practical reasons.

In software, it doesn't have to be that way. Great developers can be both "in the trenches" and "high-level" at the same time. Compared to developers, an architect might spend the most time thinking high-level, but she ought to have a foot in development.

And another thing...

And when was it decided that programmers have to be social outcasts? Sometimes, a developer is best equipped to explain to a client why a feature won't help or would put the project over budget. Project managers aren't the only people that ought to communicate to stakeholders.

For software to really succeed, I'd rather have a group of developers that are capable of coming up with a plan, executing, and then communicating than a group of people that *only* plan, *only* execute, or *only* communicate.

#5: Specialization isn't mandatory

Be wary about creating hard lines across expertise within a development team. In traditional architecture, it's not practical for the electrician to also be the cement poorer, the brick layer, and the painter. They are specialties in and of themselves. They also occur in physically different places. You need to have a group of specialized doers honing each craft separately for intellectual and practical reasons.

In software, you can be a designer, programmer, and DBA. You can be well-versed in Oracle, SQLServer, PHP, Java, .NET, C++, Python, and SQL while knowing your way around HTML, CSS, JavaScript and Flash. Why is this not only possible, but practical?

First, all of these toolsets live on the screen right in front of you. If you're currently working in SQL, you don't have to go somewhere else to write HTML, or to cut an image in Photoshop. You switch programs on your computer. There is no physical barrier between any programming disciplines.

In addition, software concepts tend to transcend language or tier. MVC (model-view-controller) is an application architecture adopted in many UI platform applications, like Adobe Flex's Cairngorm platform, along with many server-side development frameworks, like .NET. Programming languages today have an extraordinary amount of overlap. Design patterns and refactoring are concepts that live everywhere in the programming landscape.

At We Are Mammoth, most our development team knows multiple programming languages and splits time between front-end, middleware, and database work. It helps us even out everyone's workload because we're all adept at working on all layers of an application.

A .NET developer, a strict-standards HTML whizkid, and a data modeling expert can all live within the same person. You may have an expertise and interest in one or the other, but there's no reason you can't be great at many disciplines.

"I used to be in advanced business platform solutions, now I just make stuff work."

Even further, why can't great programmers also be great user interface designers also? All too often I hear a programmer instantly denounce even the possibility that he could also be a great visual designer.

Conceptually, designing user interfaces is not that far off the map from designing a sound software architecture. Great functional UI is about clear affordances, organization, scalability, and intention. It has many of the same qualities we cherish in software design.

The reverse is true too. Too few talented user interface designers consider themselves capable of becoming great programmers. Perhaps programmers look at user interface design as making things "pretty" and designers look at programming as writing a lot of "technical stuff."

But, in the end, the goals of software design from both the interface level and the engine are the same. There is no reason why we can't be great at multiple disciplines.

#6: Metaphors can hurt both ways

You've seen why metaphors can hurt how we approach software. When we take them too far, we develop habits built around false pretenses. Not only do metaphors make us do things less efficiently, but they keep us from thinking about *better ways* of doing things.

Wireframes and detailed specifications take away time from building and reviewing the real thing. It doesn't take advantage of the opportunities to iterate and refactor. It makes us think through the entire process of writing code without actually having written any code yet.

The over-emphasis on launch hides the fact that software today can be modified and redistributed with relative ease. We don't "ship" software anymore. We upload and download it off the internet, or the software itself is entirely web-based. When launch dates get pushed back because features absolutely need to be crammed in, developer morale suffers. It's a complete buzzkill for a development team.

The traditional roles in software development, between architects, developers, and project managers, inhibit those that have talents in multiple areas. You can be a great visionary, a thoughtful programmer, and clear communicator at the same time. Following the metaphor too closely inhibits really talented people from all the opportunities this industry provides.

A search for new metaphors

Building software isn't like building a building, or sending a rocket into outer space. What we do is more like writing a novel or music. Consider how these kinds of professionals "plan" their work.

If you're an author, you might write a chapter outline to get a general guideline of what you want to write about. But after that, you start writing the real thing. Then you edit, and repeat. A word change here or an entire chapter removed there. Writing is a lot more like how we program.

A musician doesn't write out sheet music for months, and then hope the notes sound right. He plays and plays, and finds a riff or a hook that works. He might have a few lines of lyrics and then finds the right chords around it or vice-versa. He builds the song in pieces and tests it out in pieces.

In both cases, the cost of materials is cheap. Paper and pen are readily available. Guitars don't get paid by the note. Sound is cheap. Just the same, code is our own cheap material. Once you've got your development environment setup, there is no material cost of writing code.

No metaphor at all

Metaphors have a place in the beginning. They offer a stake in the ground when you're not sure how to approach a software problem or when there's not enough information to make sound decisions. But, once you've run with the metaphor for awhile, see where it's helping *and* where it's actually hurting you.

Identify the rules that persist solely to maintain the essence of the metaphor, and shed them once you've found a better way of doing your work. The best approach to building software might be not having a metaphor to lean on at all.

Motivation

Regardless of how skilled you are, if you're not motivated to write code, get out. Accountants might get through writing a spreadsheet just fine without motivation; a cashier can get by his day without passion. But, unmotivated developers kill a software project.

Motivation must be sustainable. It must be unearthed and cultivated continuously throughout development. What keeps you coding with passion at the beginning of a project might not be your source of inspiration at the finish line. Different things can get you going at different points in the process of building software.

Sustaining motivation isn't unique to software. You see it all the time in the media. The star athlete who signed the big multi-million dollar contract now doesn't give his full effort during a game. The band we grew up loving now starts "cashing it in" by churning out mediocre albums. Many athletes, musicians, and artists burn out even with the guarantee of wealth. It's proof that one thing alone isn't enough to *sustain* motivation.

You need different ways to keep passion running through your veins. A food critic's review keeps a chef on her toes. But so does a busy restaurant, a happy staff, a quality set of knives, and a steady supply of fresh ingredients. Motivating yourself to run a restaurant comes from all different kinds of sources. So does building software.

#7: Motivation starts with the opportunity to build something well

In this industry, long-lasting motivation lives not in a big salary, a bonus, a free lunch, or a ping-pong or pool table. Long-lasting motivation comes from the *work you do*. Every passionate programmer I've ever met is far more excited to tell me about an elegant solution to some technical problem they've spent hours agonizing over than that 10% raise they just received at their corporate coding gig.

That's why I'm baffled when, time after time, code-geeks, especially the young, carefree ones that only have rent to pay, settle for that new, yet completely uninspiring gig with the slightly larger salary and the promise of a bigger bonus.

"I dunno, lately I'm just not incentivized..."

If you fit that mold, and the difference between salary `X` and salary `X * 1.05` is really the difference between a few more wild nights out on the town a year, go for the gig with the more interesting problems, with the more impassioned and driven employees, with the better software and hardware. Most importantly, go where the things being built are at the center of everyone's concern. Go where you have a chance to *build something well*. It's an investment in passion, happiness, and fulfilliment.

Don't stick around at the corporate gig you hate just because they're luring you with more cash. Leave that kind of job mentality to people who actually have horrible jobs — you know, accountants, lawyers, and members of the L.A. Clippers.

For me, it doesn't matter all-that-much what I'm building. It could be a one page website, a search engine, an online rolodex, an interactive map, or a game. It could be used by millions or thousands or eight people. It could be a 6-month build or a 2-hour exercise. I could be mostly writing markup, working on UI, writing server-side code, or building a database.

I'm gung-ho about my work when I know that I have a chance to *build something well*.

What gives me that chance? Projects that have distinct goals. Projects that have all the pieces either in place, or a plan to get them in place. Projects as ambitious as they are thought through. Projects that have a defined time to deliver. These kinds of projects give my work a purpose.

What projects make building something well harder to do? Projects that aren't well-defined. Projects whose conception is missing too many pieces. Projects mired with great ambition but devoid of forethought. Projects that have little, or even worse, a false sense of urgency.

I can look at the hundreds of programming projects I've worked on in the first dozen years of my programming career. All the great ones had those motivating qualities. All the poor ones didn't. Find the gigs that handle projects in a way that gives you the chance to succeed and feel great about your work. The residual perks will come a far second.

There's a great TED talk given by New York Times best-selling author, Dan Pink, on the surprising science of motivation* (You can watch it here). He argues that traditional motivational factors in business (the raise, the perk, or the bonus) do work – but only on trivial matters – the kind of tasks that are simple and directed.

In contrast, tasks that involved critical analysis and creative problem-solving, like the ones that we face every day, weren't aided by dangling a monetary carrot over someone's head. In fact, there was an inverse correlation between monetary incentive and performance — the greater the monetary reward, the *worse* they did.

When you're choosing that next gig, remember what really keeps you motivated for the long haul. It isn't the external rewards, it's the work itself.

**- (If you want the long version, read his book Drive: The Surprising Truth about what Motivates Us).*

#8: Begin where you love to begin

Sometimes the hardest place to find motivation is the very start. Thinking about code is easy. Code always compiles perfectly in your head. You don't obsess over the hundreds of minor obstacles you'll face along the way. But, once you commit to actually writing code, the game changes.

It's not too different from writing this book. I spent far more time thinking about what I wanted to write about than actually writing. Writing can sometimes be a soul-sucking game of uninspired lines, mental blocks, and distractive internet surfing.

Natalie Goldberg's Writing Down the Bones is an entire book on motivation for writers. She offers a simple tip on starting. Instead of focusing on the big opening, start writing somewhere in the middle of the story. Begin at the point you're attracted to the most. Don't try to write from the very beginning.

We spend so much time concerning ourselves with the big, attention-grabbing opener, when, in reality, it's a rather insignificant portion of the entire story. There's a considerable amount of work after the opening paragraph. That's the approach I took for this book. Nothing was written linearly. In the beginning, I focused on a specific topic when that topic inspired me.

You can apply the same concept to building software. You don't have to start with the homepage, or with the database before the business logic. Instead of starting software at the beginning, start at the place you're most comfortable or excited about. And, keep in mind we have the luxury that other builders don't. Unlike building houses, cars, or anything physical, we do not need to start anywhere specific. We can always *refactor* later on. You may take a few circuitous routes, but, if they are inspired rather than labored, you'll get more good work done faster.

So, if you have the freedom to be greedy about where to start writing software, be greedy. Pick a feature you find most interesting and work your way out from there.

This is especially helpful when you're about to embark on building a big piece of software. Rather than spend three days formulating a timeline and release schedule, commit those days working on the part of the application that most interests you. A week in, you'll know how much motivation you really have, and you'll have a far better idea of when the other parts can fall into place.

If you find yourself quickly losing steam, you can cut your losses then. But, more often than not, you'll find the daunting task of building software not so insurmountable. Three solid days (or a week) of building an application and you'll know a lot more about what you're building and how quickly the rest can get done. Putting together a realistic timeline is much easier after you've got a bit of work under your belt.

#9: Be imperfect

Every passionate programmer cares, first and foremost, about his code. It's our canvas, our playing field, and our stage. While no user will ever look at our code, the passionate programmer labors over every line. We care about how our code might perform under the most severe of conditions. We make attempts to reduce excess calls to the server, to the service, and to the database. Even when we know, full well, we're building a small app for a small audience, a lot of us still care our applications will perform under the biggest of stages.

And yet, to survive in this industry, you better not be a perfectionist. There is no such thing as a perfect piece of software – especially web software. Our products breathe and live through our users. They morph as our user base grows. Features beget new features. Bugs beget new bugs. Trying to be perfect can be exhausting.

The approach you took the first day you wrote the first line of code is likely completely different from the approach you're taking today. Software changes over time. You build, tweak, iterate, and occasionally have to rewrite. And, you better be OK with that.

A great software developer is obsessive-compulsive, yet accepts imperfection all the time. Trying to write "perfect code" is crippling. The quicker you can accept imperfection, the more motivated you'll be to keep moving forward with your work, and the more work you'll actually get done.

#10: Stop programming

You probably program too much.

Just when you've really gotten into your work, when your brain is entirely wrapped around your code, when your hands, eyes, and thoughts are working in harmony, stop. Look up. Think about when you're going to finish for the day. Look forward to shutting off your computer. Get outside a little.

Programming, for all its mental exercise, is a very comfortable physical activity. We usually program while sitting, and as the hours waste away, sitting lower, and more out-of-posture in our chairs. Some of us even eat and drink at our desks, while we code away. You can tell just by examining the keyboards — the somewhat slick ones with a pound of crumbs under the keys.

This comfort is dangerous. It means we can do this for hours and hours and hours without realizing we've exhausted our own resources. When you've hit that point where your code gets a bit sloppy, or better yet, just before it, stop. Great programming is about maximizing the amount of time you're working at your best, not the cumulative hours you spend in front of a screen.

Two hours of quality programming time is better than 8 hours of struggle.

You're far more susceptible to taking shortcuts or breaking standard conventions when you're coding tired. Those extra hours are spent creating bad code — code that you might regret the next day. So, cut your programming time down, get outside, and live a little.

#11: Test your work first thing in the morning

Test your software first thing in the morning. It's when you're most fresh and motivated to continue building something good.

During the day, we spend so much effort building software, that we lose steam testing each piece we write. It gets hard to see the big picture as the day wears on. By late afternoon, you're too close to the software. Your perception of what makes sense or feels right competes with fatigue, disinterest, and hunger.

Should this feature be here or there? Should we move this function to another screen? Will this make sense? Is this latest tweak really that important? At 5pm, it's hard to know what your software feels like because you've been in it for too long.

However, at 9am, fresh from a night's sleep, you can usually answer these questions better. Your mental cobwebs are gone. Before diving into the build, this is the best time to give your software the once-over.

In the morning, your software feels new again. You approach it like someone less fettered by what's behind the scenes. You can approach it from a less biased viewpoint because you've had time away.

The morning has a way of making you forget some of the copious details of code you may have obsessed about the night before. No longer pre-occupied with the slightly inelegant implementation that made something work, your mind is totally devoted to what you see in front of you rather than thinking about what's happening underneath.

When you test, start from the beginning. Don't dig into a particular section. Just experience it again. The night before, you may have been working on a piece of functionality that a real user may only use once or twice...or never. In the morning, focus on the things most people will use most of the time. It's a much better way to focus on the priorities of your software and focus on what needs fixing first.

Testing your software in the morning, before adding more code, is a great way to make sure you're still making good software. It's when you're most fresh and motivated.

#12: Don't work in your bedroom

If you have the "luxury" of working from home on a consistent basis, the one place you should never code is in your bedroom. Or your living room. Have a confined area to work, preferably a second room, that you can physically leave from after your work is over.

Why? Having that separation between work and home is important. As much as you can love coding, you need to define a physical boundary between when you are working and when you are off doing something else. Otherwise, your work and non-work life become physically (and psychologically) the same.

Separating the two will keep you fresh and motivated to sit down and get back to work again.

#13: Don't let bad first impressions of your work unravel you

As web apps go, how important are users' first impressions? I don't think they are very important.

No doubt, bad first impressions could be a sign that something really is wrong with your software. But, there are two things I've learned that account for many bad first impressions.

Sometimes bad first impressions really mean you simply haven't used the software before.

These are the ones we should really take with a grain of salt. For instance, the first time I used Gmail, I had to get used to it for a few days.

"These emails, they're like [sic] mini-forums. They're like threads of discussion...not email. Interesting. Do I like it? No. Yes. I don't know...maybe?"

Gmail email threads were novel, if not strange. I heard many people rave about it, but I heard many people rip it to shreds. A year later? I stopped hearing about it altogether. Here's a typical conversation you might have with someone that uses Gmail today:

> Male in red cap: "Hey dude, do you use Gmail?"
> Male in blue cap: "Yeah."
> Male in red cap: "What do you think?"
> Male in blue cap: "It's fine. Lately it's been slow. Hey let's go grab some beers."

It turns out that both systems work for me, and these two fine gentlemen above. We've stopped obsessing over it. In fact, today, I use both Outlook and Gmail. I also know lots of others who use both a non-Gmail email client and Gmail. When I use Outlook, I expect normal, old-school email. When I use

Gmail, I expect "special-forum-like-email-madness." In the end, I'm comfortable with both.

"I'm more of a late adopter."

I'll admit it. It takes some huevos to try and redefine paradigms as firmly implanted in society as email. In my line of work, fortunately, the stakes are a lot smaller. Radio buttons or drop down list? Search box on every page or just some pages? The likely answer? Yes. Yes. And Yes. And Yes. In the end, when you are accustomed to seeing the same software over and over, there's a good chance you'll get comfortable with whatever design decisions you first had a problem with.

I hear the naysayers knocking. Users conform to software? Are our reactions to software not important? Is this coming from the same guy that helped bring you Flash Application Design Solutions: The Flash *Usability* Handbook (purchase it at any of your favorite online bookstores!)? Yes it is!

I'm not saying that first impressions don't mean anything. But, those initial gut reactions to having seen something for the first time are often just that — gut reactions. As users, we're naturally daunted by something new. Yet, too often, as developers, we take those initial user reactions too seriously.

Here's the other problem:

Sometimes bad first impressions are not indicative of what's really important.

If Google just opened shop and I was a usability tester, here's what my first impressions might be:

- The "Google Search" button should flip with "I'm Feeling Lucky" because I'm used to clicking the right-most button when I submit information, and I'm usually going to search rather than "press my luck."
- I don't get what "I'm Feeling Lucky" is. That's confusing. There should be instruction there as to what might happen when I click it.
- You asked me to do an advanced search, and I had to look around a bit to figure out what you meant. Oh, and that advanced search page was hard to use.
- The navigation links at the bottom should be above the search bar, because that's where I'm used to seeing navigation.

Ask me now, and here's my rebuttal to my initial first impressions:

- I'm used to that now.
- I get it. I clicked it a few times and now I know it just takes the first search off the list and sends you right there.
- I don't ever do an advanced search. And that link is small so I'm not bothered by it.
- I never use those links, so I'm glad they're underneath the search box.

First impressions are often skewed because we don't really get how we'll ultimately conform to the software. Initially, something you think might be important (an advanced search option) ends up not being important at all. Something that's a little off from what you're accustomed to (those buttons should be reversed), you simply get acclimated to after a short period of time.

So, when you're confronted with negative feedback from your customers, clients, or co-workers, stick to your guns. Explain why you did the things you did. Ask them to let your work simmer for a few days or a week. If those problems still persist, then, perhaps there really is a flaw in your application.

But, you'll be surprised how often those initial, instinctual, negative impressions fade away.

#14: Never underestimate the emotional value of finishing

In the previous chapter, I talked about launch. In web development, launch is just one of the many events in the lifespan of software. It isn't the end of development, it's just another phase. We shouldn't postpone launch to perfect software. But, a launch date *still* keeps us motivated.

Why? Because a launch date carries *emotional* value. Knowing your software is no longer waiting on the sidelines, and is alive and ready, is a huge lift to the senses. You've *finished* something.

By finished, of course, I just mean software that's built, launched and usable, rather than software that's still in the throes of development. Too often, we underestimate the emotional value of work that is simply *finished*.

Launch is not an application's end-all final state. Perhaps there are even tectonic changes to be made in the future. But the mental uplift of knowing that you've finished is often underrated. Launch at the earliest moment you can, when your software is good enough to go so you can feel the mental boost.

The good feeling of *finishing* has a huge implication on how well and how efficiently you *continue* to do your work. Contrast that with the unmotivated feeling of the endless tweaking of software still not ready for prime-time. That has an equally negative impact.

While we underestimate the emotional uplift of launch, we overestimate the perceived headaches of trying to tweak an application after the fact — even if they are deep changes to functionality.

Very few add-ons, removals, or logic shifts are undoable. Good programmers prepare themselves for this all the time. The basis of design patterns, methods on refactoring, and best practices is largely to accommodate for the very real fact that what your code is doing today will likely get tweaked (endlessly) tomorrow.

So, get your software to launch as soon as you reasonably can. It's a natural high.

Productivity

Motivation may be what we need to get started. But, productivity is the tangible measure of success. It's an everyday, consistent level of impassioned work.

The corporate world, as it does with just about any concept, equates productivity to some calculable metric. In this case, a metric like *utilization* or *throughput*. Productivity is often equated to how *much* work you do or how *many* things you do at once, not the quality of your work.

Multi-tasking is the quintessential act for pretending like you're being productive, but rarely a great way to actually do good work. "Working" through lunch is one of these acts. How much quality code is really being written typing with one pointer finger while the other hand grips that footlong sandwich?

And, plus, that's not enjoyable. Leave your desk. Eat your lunch in peace, and get some fresh air. The code will be there when you come back. And, before you get to that, read my thoughts on really being productive.

#15: Just say "no" to the pet project

Every one of us has a pet project archive. Software you started and saw part of the way through, but never quite finished. Code that began strong but came to a screeching halt because more pressing issues came up. Other work simply got in the way, or, you just lost interest.

Pet projects fail when there are no time constraints and nothing's on the line. When your launch date is "one of these days" you likely won't be finishing it anytime soon.

That's why *time is the most important parameter in maintaining your passion for writing software*. With a pet project, it's fine to just start writing code for your own enjoyment and learning. But, when you're ready to turn it into something real, define your time boundaries. Answer the following. It'll turn your project into something real.

- How much time will you spend working on your pet project each day and each week?
- What's the first day you can show it to someone else?
- What day do you launch to the public?
- What day will you release your first major iteration?

The first question sets your everyday expectations for yourself. And, be reasonable. Maybe its two hours a day or three days a week. Make it achievable, but, most importantly, make it consistent.

The second question sets your deadline for almost-ready software. It's a stake in the ground a short time from now where your co-worker, friend, or spouse gets a crack at what you've done. It helps you work backwards to figure out how much you need to do between now and then. Couple this with how much time you spend per day and week, and you'll know how much needs to get done every session.

The third question gets you ready for "good enough" software. Software that's got all the big things right and is good enough for the general public. It's launch-ready. Don't set this date too far from the date you show your friends. And, it's web-based software. It doesn't have to be perfect.

That's where the fourth question comes in. It sets you up for everything thereafter. You've launched and now you've set up a time to push new releases. Perhaps, it's a week after launch. Because we're talking about web-based software, it can be just a few days (or even hours) after launch. Once you've actually gotten to this stage, you're off and running.

These time constraints create the walls you need to fill in with work. It helps you define your most important features. It gives purpose to each moment you put into your software. Without rigid time constraints, you can go on forever wondering if you've put enough time into something or make excuses that other things take priority.

Instead of getting ready to deliver something, you'll be feature creeping, re-evaluating, and procrastinating every step for as long as you please. Productivity dries up when you don't have that sense of urgency. Time constraints keep you progressing.

But, what if you truly have a pet project? What if it's just something you're doing for fun on the side and learning new skills along the way, and nothing more? Set the time parameters anyways. Even if they are arbitrary stakes in the ground, **set them**. If you care about staying productive, turn your pet projects into real projects, with a real deadline. Force yourself to learn what you need to learn in that given amount of time.

#16: Set a deadline, even if it's arbitrary

I started building my company's first product, DoneDone, in October, 2008. DoneDone is a simple, web-based bug tracking tool that focuses on clarity and workflow over features. It started because we didn't like the other bug tracker we were currently paying $120/month for. I could do something better.

When client work sputtered for a few weeks, I began coding. If we were willing to pay for our current bug tracker, others would certainly pay for something better. And so, motivation started with those two thoughts:

- Make something better.
- Make some money.

For the first few weeks, I worked with only ideas. No wireframes, specifications, or even Photoshop files. I just wrote code, built UI, tested, refined, and wrote some more. I was still in the honeymoon period of development. Though I was directionless, the thought of making money off a product was enough motivation to start.

Fast-forward a few months later. It was November, 2008. Client work trickled in, and naturally, my pet project shuffled to the back of the priority line. Every few days, I could carve out a couple of hours to build DoneDone, but the hours were sluggish and unproductive. It was hard to decide what to do with the infrequent buckets of time I could carve out for myself.

Instead, we needed a new approach. We had to treat DoneDone with the urgency of a client project. What was the difference between this project and projects we'd done for other clients? We were our *own* client. And just like a client project, we needed a set of dates. A date to release DoneDone internally. A date to launch the product to the public. And dates to release iterations thereafter.

In the end, we came up with the completely arbitrary date of April 15, 2009, roughly 6 months from the start. A pet project suddenly became a real project.

From that stake in the ground, we could now work backward. We needed to add in the payment gateways, figure out our cost structure, build a marketing site, and clean up our feature set. With a deadline set, we could now fill that time in with a requisite amount of work. The sense of urgency came back. I could be productive again.

In early April, I set up a DoneDone week on our company blog. Each day focused on a different aspect of the application. I chose 5 things to focus on, one for each day of the business week. Even the marketing write-ups were based on an arbitrary set of time.

We launched on April 15th, simply and solely because that's the date we set. There were certainly other features to add. Looking back on it, it's hard to imagine DoneDone ever not having them. We did not have an email-to-ticket or tagging system for issues, both core pieces of the product today. But, they simply did not make it to launch. We focused on the most important features a bug tracking tool of 6 months requires. On April 15th, the project was launched to the public, productivity still running high.

Have I really written nearly a whole chapter's worth about setting deadlines? Yes. In all it's un-revolutionari-ness, it's amazing how many people still simply do not set deadlines for their own work.

It's not just about setting an end date. Deadlines create a sense of urgency that gets you to the finish line. It's making you your own best boss, especially when you've got no one else breathing down your neck.

Finally, deadlines keep your work relevant. When you let your project bleed from months into, even, years, your product might not have the intended worth it had when you began.

Jack Dorsey spent less than three months from initial concept to launching the first version of a, then, little-known SMS messaging service that later would be called Twitter. Imagine if, instead, he had spent years and years on development instead of the short burst from start to launch. Things might have turned out differently.

#17: Constrain your parameters

It's not just about time. If your software costs money to build (hosting, hired help, coffee), put a cap on it. What can you do with a few hundred bucks a month? Capping cost makes you creative. It means you might have to figure out a more efficient way of using your resources. It forces you to think smart. And smart thinking keeps you productive.

Consider all the sad stories of lottery winners. People working 9-to-5 jobs just to make ends meet and struck it rich. They buy cars, jewelry, yachts, and homes. They give away thousands to re-emerging long-lost relatives. Then they go into debt and are worse off than not having won at all. They didn't realize that, even 50 million dollars is a finite amount of money. The walls, while spread further apart, were still there. They just didn't think to find them.

When you don't have walls around you, whether they are walls around time, cost, or a feature set, you lose sight of reality. You make questionable decisions because nothing is forcing you to make them firm. If you want to develop great software, set up and obey the walls around you. Then build.

#18: Cut the detail out of the timeline

In twelve years of web development, I've never seen a project go exactly as planned.

Functionality changes. Unanticipated obstacles arise. And, sometimes things that you thought might take longer, simply don't. And yet, all too often, we put too much *detail* into a project timeline. Putting a delivery time toward every single page, view, function, UI component, or interaction means you've become a slave to your timeline. You've decided how long every single step will take without having taken any of the steps yet. If you're going to plan, you might as well plan with less detail.

Build timeline deliverables in sizeable chunks, not in small breadcrumbs. If you're estimating an eight week project, give yourself eight weekly deliverables rather than 40 daily ones. Instead of defining when each individual interaction of your application can be delivered, decide when complete section can be delivered. In the end, it's the same amount of time, but with less checkpoints in between.

When your timeline is too detailed and your delivery dates to frequent, there's no wiggle room to experiment or reconsider the details of an application as you go. You're forced to stick to a rigid schedule of guessed tasks. And, when one or two of those small tasks goes wrong, suddenly, the entire timeline feels like it's crumbling right in front of you. That's not motivating, nor is it how good software gets built.

Giving yourself a reasonable amount of time between deliverables enables you to play. It breaks down a large project into bite-size mini-projects where you *still* get the chance to begin where you love to begin. It gives you the chance to iterate a few times before your next deadline. A week (or two) before your next deliverable gives you the opportunity to make a few mistakes and still recover.

#19: Make your product better in two ways everyday

Development can get dull. At times, I'd like to do something else — paint a picture, wash my car, perform open-heart surgery on an endangered animal — *anything* but this.

In the proverbial dog days of development, sustainable productivity has to come from small victories. Money, fame, delusions of grandeur, and the simple satisfaction of coding won't keep you productive all the time. There has to be an everyday nugget of inspiration — a baseline motivator that exists even when the big motivators grow stale.

I've spent a few years building a web-based data modeling application called X2O. X2O creates the software framework we use for every web project at We Are Mammoth. It generates an interface to build an application's data model, and generates the database, data access layer, and web services, so you can quickly build customized database-driven apps. You'll learn about it in a lot more detail in the next chapter on automation.

X2O is an application with lofty goals. It is wildly complex in places, with a whole lot going on underneath the hood. When broken down, it's a synthesis of a several dozen applications that help generate different parts of your custom app. Maintaining passion to keep building it is hard because each of these apps is a big application in itself. Building big stuff like this requires small victories.

While I was knee-deep in development, I put a new rule into my day-to-day efforts on X2O: Make two things better about it each day.

I have an end-goal. One day, I'd like to wrap this entire framework up into a bow tie, figure out a way to distribute it to the outside world, travel the world and market it. But, spending too much time daydreaming about the end game doesn't help X2O get better. Real work has to be done.

When productivity is running low, I keep to a simple rule: Make two improvements to your product every day. They don't have to be big improvements, like wrapping a security layer around the application. They can also be little ones, like creating more elegant, friendly error messages on the UI. Some days I'll have the energy to tackle a big task and a little one. Other days, maybe two minor ones. In the end, everyday I know that X2O *today* is quantifiably better than X2O *yesterday*.

There's significance in the number 2 as well. One improvement can sometimes be daunting. Which one do you choose? One is also too close to zero. One makes it easy to convince yourself that you could skip today and make up that extra one tomorrow. On the other hand, three seems like too much to sustain every day. Two is a magical number. I can find time, even on a busy day, to squeeze out two new little nuggets of goodness into my product. It's hard work, but sustainable.

And remember, it's two ways to make your product better, not necessarily two new features. "Better" can also come from getting rid of stale code, refactoring, commenting, and re-organization.

Deciding to make your product better in two ways everyday is a good mental exercise to keep those large projects moving forward. In a working week, you'll have 10 better things to say about your product than you do now. In a working month, you'll have 40 better things to say about your product than you do now. That's real, everyday progress.

#20: Invest in good hardware

Distractions are the enemy of productivity. The more little things that get in your way each time you try to get work done, the less productive you'll be. It's especially true for us programmers, where any little distraction can disrupt a sometimes complex train of thought.

So, productivity starts with every *little* thing that surrounds your work environment. There are so many things that can distract you when you should be working at your best. Your work environment should do everything to minimize that distraction.

For starters, invest in the right hardware. The financial costs you put in up-front will invariably pay off every day, in *the currency of productivity*.

At We Are Mammoth, we recently made the upgrade from using fiver year old Dell towers running Windows XP to a MacBook Pro running Windows 7 on Parallels. The investment was monetarily substantial, but, for that singular, one-time down payment, we reap the benefits every single minute we're working.

Why I love my MacBook Pro

Full disclosure: And now, I will preface the remainder of this chapter by saying that I am a converted Macophile. You may skip to the next chapter if you'd like.

Running Windows on the Mac lets me take advantages of both platforms. I can run Photoshop on the Mac, in a pinch, while still working on .NET apps with Visual Studio on the Windows side. Plus, I can test IE, Firefox and Safari on both PC and Mac without having to setup a cast-off computer (the ones companies typically denote, the "Browser Test Machine").

With our new laptops, our developers are more mobile now. Our applications take fractions of the time to load on newer, faster processors.

I've even fallen in love with the keyboard. The keys are flat and thin. They have just the right responsiveness to the touch. They let me type more fluidly and seamlessly with the code I see in my head. I make less mistakes. Compare that to the bulkier, stickier nature of traditional keyboards. Even if every keystroke is just 10% more efficient, and I can reduce my errors by a few less a day, that multiples over time.

"Still nothing. You sure this is the best computer we have?"

And when you buy a better machine, make sure you get a video card that can support multiple monitors. Along with our glistening new laptops, we supply every developer with a secondary monitor. The more screen real estate you have, the more open applications you can have in front of you without needing to flip between states. If you have the luxury of a two-monitor setup, keep your main monitor (or laptop) directly in front of you, off to either the left or right of you, or splurge on a laptop stand.

In a dual monitor setup, keep your programming environment in the full-screen directly in front of you. In the other screen, keep up your test browser and have any other programs (like email or chat clients) accessible from there as well. This way, you can stay focused on your development and testing on one screen at all times.

If you have a triple monitor setup, keep your programming environment directly in front of you, your test browser up on one screen and all the other programs (email, chat) on a 3rd.

Scan a typical web development office and you can quickly tell whether management's in-touch with their development team. Count the number of monitors in the room and divide by the number of employees. This number is your *In-touch* quotient.

***In-Touch* Quotient**	**Diagnosis**
1	Not very in-touch
1 - 2	Somewhat in-touch
2.1 - 3	Maximally in-touch
> 3	You're actually in a day-trader's office. Leave **immediately**.

The machine you've been running on for a few years may seem fine to you, and you might be accustomed to its temperaments. But spending those extra few thousands of dollars is an *investment in productivity*, not just an expenditure.

#21: Establish "off-time" and let your co-workers do so too

Productivity can only happen when there's time allowed to be productive. It's less obvious than it sounds. Basic events in everyday work — meetings, phone calls, shoulder-tapping — are so commonplace now that we forget these are all *distractions*.

How have some companies solved the distraction issue? 37signals preaches staying away from your employees. Interruption not only stops you from working on the thing you're working on, but breaks you out of your natural zone — that state of being completely concentrated on your work. Google is well-known for their "20% rule" (allowing engineers a day a week to work on their own projects).

Companies that want their employees to thrive, not just survive, need ways of allowing that to *actually* happen.

In our business, we are far more productive when we take the occasional day to work from home. Extracting yourself from the nagging feeling that there's a client call looming, or someone needing your help on something *right now* is a huge burden off.

But, we also serve clients that sometimes need lots of attention. And, sometimes *we* need lots of attention from each other as well. If you work for clients, you may not have the luxury of full days of pure programming. There will always be fires to fight, clients to call, and emails to answer.

Welcome, OFF-time

A few years ago, I instated OFF-time at We Are Mammoth. It's a way of mimicking the "get-out-of-my-face-and-just-let-me-work" rules that other companies implement, without much compromising to our client's needs (a.k.a. calls, emails, and general attentiveness). Here's how it works:

We have 2 hour shifts of OFF-time for each developer, every day. Here are the 5 commandments of OFF-time:

- No emails need to be answered.
- No meetings. You are unavailable during this time.
- No phone calls.
- No co-worker IMs you.
- No co-worker talks to you.

During OFF-time, you place a white flag on your desk. If you're at home, then you just put your away status on AIM. After 2 hour's off, you check your email, your phone calls and generally come back to the world. Then, proceed as normal. The golden rule is simple: Just don't bug the next person on OFF time.

It would be great to have company-wide off time, but it's unreasonable for most companies. It means a company essentially shuts down from any client or internal communication for two hours. It's a hard feat given we are still primarily a client-oriented business. Instead, we set up three staggered shifts. They look something like this:

- Shift 1: 10am-noon daily – Ka Wai, Michael
- Shift 2: 2pm-4pm daily – Anthony, Mustafa
- Shift 3: 4pm-6pm daily – Tom, Craig

Because shifts are staggered, only one or two people are off at any time. So, the company doesn't just shut down. We are, for the most part, available.

Someone else can help you

Another key is pairing up different roles for each OFF-time shift. I tend to work closest with Mustafa, Tom with Mike, and Anthony with Craig. So, we don't share the same OFF-time.

That means, if there's an urgent issue, a natural counterpart is available to discuss it. If Craig has a question for me at 11:30am, he goes to Mustafa. If I have one for Tom at 4:30pm, I can ask Mike. Find a counterpart that can handle all the questions that you could handle, and differentiate your off-time hours.

Interruption as the last resort

Since OFF-time cuts you off from the person on it, it also makes you think harder about your own problems. Is this something you can solve with a little research? Interruption is becoming the last resort.

OFF-time gives each of us 10 hours of interruption-free time a week, with almost little disruption to our availability to clients and ourselves. It's a great way to sustain productivity.

#22: Keep a personal to-do list

It's amazing how something as simple as a checklist of items can alter your productivity. Enter the personal to-do list.

A personal to-do list is **not** a project timeline or a Gantt chart. Those documents serve a group. They are too sweeping for a single person. They make projections about the broad scope of a project rather than lay the path for the next couple of paces ahead of you. While they are useful "big picture" documents, they don't help you organize.

A personal to-do list is **not** an overflowing email inbox. Using your emails to remind you of things you have to do is futile. Email is a smorgasbord of fragmented conversation and questions interspersed with un-prioritized tasks — not a clearly defined list of things to do right now. Email isn't made for quick scanning.

A personal to-do list is **just a checklist, alright????** Nothing more, nothing less. It's quick and simple, but deceptively powerful.

When you receive a task in your inbox, write the task down in your personal to-do list. When you've adjusted your timeline to accommodate a must-have feature, breakdown that feature into small to-dos on your personal to-do list.

At first glance, the personal to-do list seems like just another thing to manage — all the items in it likely originated from some other document. That smells like bad practice because it violates the "don't repeat yourself" mantra most of us follow in our code. But, a personal to-do list is a rare occasion where duplication is *essential*. That's because it doesn't serve the same purpose as other, more rigid, documents.

Unlike other documents, the kind of to-do list I'm advocating is made for constant daily adjustment. It is never set-in-stone. To-dos are added, checked off, pushed back, pushed up, and thrown out daily. Unlike project timelines and Gantt charts, a personal to-do list doesn't care about the past. Its starting point is always right now. It also doesn't project anything. It's full of organized, real tasks that have to get finished in the very near term.

A good personal to-do list has the following qualities:

- It is one, and only one, list.
- It has four divided areas: *Today*, *Tomorrow*, *Two days from now*, and the *Future*.
- It is one-level deep, not a multi-level tree. This makes it easy to scan.
- It's easily modifiable. You can move items up and down the list easily.
- Each to-do in any of the first three buckets should take an hour or less. Items in the *Future* bucket can be broader.
- It's online. You have access to it wherever you're working.

I use Tadalist from 37signals because it's simple and free. To-do lists shouldn't be robust software. Tadalist lets you do only a few things: Create lists, add items, edit items, delete items, and re-order items. Here's how you set up a personal to-do list using Tadalist:

- **Step 1:** Create a list called "Personal To-Do List"
- **Step 2:** Add 4 divider items (In Tadalist just make them 4 to-do items that you'll never actually check off):

```
--- TODAY ---
--- TOMORROW ---
--- TWO DAYS FROM NOW ---
--- FUTURE ---
```

As you start adding to-dos, put them under their appropriate divider. If it's something you need to do today, drag it under `TODAY`. If it's something that needs to be done tomorrow, drag it under `TOMORROW`. If it's just further out, put it under `TWO DAYS FROM NOW`. If you're not sure when, or need to keep a broader task in mind, drag it under the `FUTURE`.

Remember, nothing about a to-do list is final. If you're unsure whether to put something under `TOMORROW` vs. `TWO DAYS FROM NOW`, lean toward the closer date. If you get to tomorrow and it still hasn't become top priority, you can leave it for tomorrow again.

Breaking down features into to-dos

Any to-dos you bring under `TODAY`, `TOMORROW`, or `TWO DAYS FROM NOW`, should be small tasks (no more than a couple of hours). For instance, "`Build registration and login`" is a bad to-do. Too much time goes by before you're able to see progress on your to-do list. Instead, "`Build registration and login`" might be added as a series of these bite-size tasks:

```
--- TODAY ---
Registration: Build HTML form
Registration: Implement JS validation
Registration: Verify email uniqueness
Registration: Send email confirmation
--- TOMORROW ---
Login: Build login form
Login: Implement JS validation
--- TWO DAYS FROM NOW ---
Login: Build "Forgot password" form
Login: Send "Forgot password" email
Login: Build "Reset password" form
Login: Build "Reset password" confirmation
--- FUTURE ---
Something down the road...
```

We've broken down building a registration component into a 3-day, 10 to-do task. It's organized without seeming overly complex.

Each to-do is a small chunk of work. Once you finish one, you can check it off. It gives you instant gratification each time you *finish* something. Instead of waiting until you complete an entire component, you see progress frequently as you go.

How tomorrow becomes today

To-do lists start with today. So, what happens when you wake up tomorrow? Spend 5 seconds to get your to-do list updated again. In Tadalist, it's two moves with your mouse:

- Drag the `TOMORROW` divider just above the `TWO DAYS FROM NOW` divider. Everything that was set for tomorrow now falls under `TODAY`. Anything you didn't get to yesterday remains in `TODAY`.
- Drag `TWO DAYS FROM NOW` just above the `FUTURE` divider. Everything that was set for 2 days out now falls under `TOMORROW`.

```
--- TODAY ---
Login: Build login form
Login: Implement JS validation
--- TOMORROW ---
Login: Build "Forgot password" form
Login: Send "Forgot password" email
Login: Build "Reset password" form
Login: Build "Reset password" confirmation
--- TWO DAYS FROM NOW ---
--- FUTURE ---
Something down the road...
```

Back to the future

Each day, glance at your growing list of FUTURE items. If you'll need to finish one of those items in the next 2 days, break that task down into bite-size tasks, and move them appropriately. In a few seconds, you have a quick gauge of what your workload looks like.

Re-evaluating what matters each day

A personal to-do list is perfectly made for adjustments to priority. It rolls with the everyday uncertainty of software development. Perhaps an item that's destined to be done today doesn't seem as important anymore. Just drag it into the TOMORROW bucket or even further down. Similarly, an item set for TWO DAYS FROM NOW may be something you have the energy for today. Move it, finish it, and check it off.

There will be many days when you won't get to all of your TODAY items. You may have an item stuck on TODAY for day's on-end because other priorities got in the way.

But after awhile, you'll notice something extraordinary. Certain to-dos always seem to linger around your TODAY bucket or routinely get pushed back to TOMORROW. These "bad egg" to-dos might not be as important as you thought when you first added it. *Get rid of to-dos that grow stale on your list*. Avoiding unimportant work is just as productive as completing important work.

"Well no wonder! I had it on my *not* to do list!"

A personal to-do list is not black magic – it will not do your work for you. But, it helps you organize and adjust simultaneously. It leads you to productivity.

#23: Write code as a last resort

I love the story of the unhappy tenants in a New York city office building. When residents complained about the increasingly poor service of the elevators, a consulting firm concluded that long wait times were the issue. Solving the tenant's complaints meant potentially adding new elevators and implementing new computer controls to improve elevator efficiency – all very costly adjustments.

Enter the young psychologist hired in the building's personnel department. He recommended, instead, placing mirrors in the elevator lobby. The problem of waiting times, in his mind, was really a problem of boredom. His suggestion worked. People stopped complaining about waiting for the elevator when they had something to do — observe themselves. Same problem solved with a very different solution.

The most passionate of us are the ones who spend most of our work time thinking critically and creatively, often to find simpler, "lazier" solutions. The answer isn't always to plow ahead with the obvious, brute-force solution. In our world, the brute force solutions would be simply writing more code.

Sometimes, the best answers are found somewhere else. Think to yourself, the next time you're confronted with a New York office building elevator problem...

- Has someone already done this task before? Can I use something off-the-shelf to take care of the dirty work for me?
- Is this piece of functionality really important to the goals of the application? Is the task already there, but just through a different user experience?
- Is this an automatable task? Can I write software to write this algorithm for me, so I don't have to repeat this work again?
- Is there a simpler way to code what I'm coding right now that might be worth the tradeoff, even if it doesn't solve the problem

entirely?

As a last resort, the passionate coder actually...writes code. In today's landscape, there are so many opportunities to not write it.

When you go straight into "writing code" mode at every task, you lose the opportunity to think about why you're really writing it. When you, instead, think "does this need to be implemented in *this* way", you focus your coding efforts on the elements that, truly, are important for solving your problems.

Automation

Cooks, cleaners, baristas, accountants, cashiers, construction workers, bankers, mortgage lenders, and doctors. What do these jobs have in common? They require you to repeat the same skill set each day. Sure, the most talented chefs are inventing new dishes. The best doctors are researching innovative new ways to make people better. But, most find a specialty, perfect it, and repeat the craft over and over for years and years. Their work becomes by-and-large, algorithmic.

Software developers do this too. You can be great at building shopping carts, content management systems, web services, or login components. You can spend years working on the same types of things over again and again. Granted, it might be harder to find that kind of work over time. Technology shifts at a blazing rate. The languages, platforms, and hardware you're using now will likely become a liability a few years from now. But, do realize, programmers are still forced to use and program against IE6 these days. And, even Microsoft doesn't want you programming for IE6 anymore.

There will always be work for the dinosaurs.

By contrast, look at sculptors, musicians, architects, actors, writers, and directors. They find inspiration in expanding and forging new ways of doing their work. Many actors choose roles specifically to avoid being typecasted. It's every actor's biggest fear — the realization that they are *always* the villain, the victim, the comic, or the outcast every time. Think Bill Murray's break from comedy in *Lost In Translation*.

Musicians fear this same predictability as well. Some of the most well-known musicians reinvented themselves in their prime. Just listen to the Beatles' *Revolver*, Bowie's *The Man Who Sold the World*, or U2's *Achtung Baby*. They found hunger for something more than just repeating themselves, departing from a style they had already mastered.

The beauty in software development is that you can choose to fall under *both* buckets of work. You can keep working on concepts you've already mastered, that have become algorithmic, or push for something new. You can solve more complex, more interesting, and more rewarding problems.

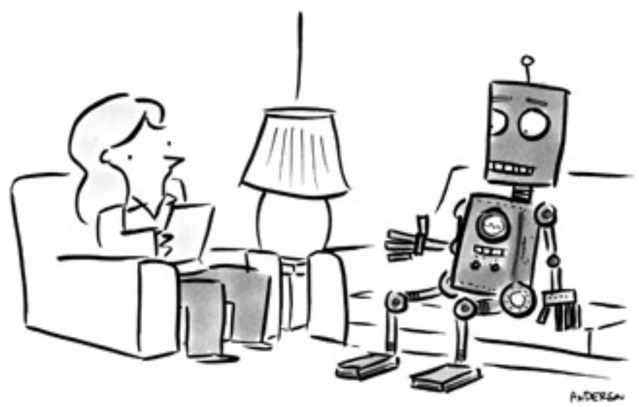
I'd like to think the latter is a more interesting place to spend your day. And, fortunately, you *can* spend it there.

#24: Separate human work from robot work

There is simply no reason a software developer today should resort to doing the same mechanistic things over and over again. We have the unique luxury of working in a medium that can help do *the work for us*. A musician can't write music that writes music and a director can't direct a movie that directs other movies — but a software developer *can* write programs that write other programs. It's our own dirty little secret.

Yet, as an industry whole, we aren't taking advantage of this as much as we ought to. We've all had moments of déjà-vu programming. You've pasted code from an old project and tweaked it slightly to fit the new one, or you've just spent hours writing functionality you know you've written many times before. This has got to stop. There are better, faster, and more rewarding ways to go about your work.

If you're finding yourself doing this often, be more cognizant of it. Separate the code you find yourself repeating from the code that's unique. Identify and extract work that feels like a repeat *process* (not just copy-and-paste code) and don't repeat them. Let software *write* the tedious and repeatable work for you. This is code generation.



"Lately it seems like nothing but zeroes."

You may have heard about code generation before, but if you're not doing at least a small amount of it, you may not have fully considered all of its incredible benefits.

The story of little Carl Friedrich Gauss

The famous mathematician, Carl Friedrich Gauss grew up in Germany in the late 18th century. As the story goes, his grade school teacher was notoriously lazy and wanted to keep his pupils occupied for long periods of time. In hopes of keeping his mind busy for a long while, the teacher had Gauss sum all the numbers up from 1 to 100. After only a few minutes, much to the teacher's chagrin, the young Gauss came back to his teacher with the answer: `5050`.

How did he come up with the answer so quickly? If Gauss had the tools to write code back then, he might have written some code like this to get to the answer:

```
public int sum_range_of_positive_integers_to_100()
{
     int sum;
     for (int i = 1; i <= 100; i++)
     {
          sum += i;
     }
     return sum;
}
```

After some quick thinking, he might have re-written his program to something a little more generic, in case the teacher was to challenge him again with a different range of numbers.

```
public int sum_range_of_positive_integers(int first, int last)
{
     int sum;
     for (int i = first; i <= last; i++)
     {
          sum += i;
     }
```

```
    return sum;
}
```

Of course, young Gauss didn't have such a luxury at the time. So how did he get to the answer so quickly?

Rather than adding up the numbers one-at-a-time, he approached the problem in a far more clever way. Instead of brute-force addition, he summed the numbers up pairwise, starting with `1` and `100`, followed by `2` and `99`, `3` and `98`, and so forth.

```
(1 + 100) + (2 + 99) + (3 + 98) + ... + (49 + 52) + (50 +
51).
```

```
101 + 101 + 101 + ... + 101 + 101 + 101
```

From here, a simple pattern emerges. It turns out there are 50 pairs of numbers that each sum up to the magic number, `101`. The problem of summing up all 100 numbers together was reduced to a simple multiplication equation:

```
50 X 101 = 5050.
```

Gauss's approach was a uniquely *human* one. He took a problem that appeared tedious at the surface and came up with an elegant way to solve it. He found a better heuristic instead of resorting to the über-tedious brute-force approach. As it turns out, he fell upon a nifty little formula.

```
Sum of all numbers from 1 to n = n(n+1) / 2
```

This formula is far more palatable for a human to calculate than brute-force addition would be. It's busy work *fit* for a human to do.

How would our code approach this problem? The code written earlier in this chapter would have produced the same answer, only doing it the "brute-force" way — the exact way we told it to. And, given how fast processors work these days, it would have arrived at the answer much faster than even Gauss did, despite the inefficiencies in its approach.

But, at a certain point, Gauss's approach would win out. If Gauss were asked to sum all the numbers from `1` to `4,273,558`, `sum_range_of_positive_integers()` would take a lot longer to compute.

This means that, at some number, Gauss would likely have been able to *beat the code* to the answer. Because, while Gauss cunningly knew that the answer lied in evaluating one simple equation, the code would get to the answer in the incredibly inefficient way it was told to do so:

```
1 + 2 + 3 + 4 + 5 + 6 + .... + 4,273,556 + 4,273,557 +
4,273,558.
```

Let's rewrite the function using Gauss's bit of ingenuity:

```
public int sum_range_of_positive_integers(int last)
{
    // A solution Gauss would be proud of...
    return (last * (last + 1)) / 2;
}
```

Now, we've turned the function into a highly efficient one. No matter what the input, it will perform at just about the same rate of efficiency. Summing `1` to `4,273,558` won't take much longer than summing `1` to `100`. Our program won't be slow and it will never be wrong. Gauss, unfortunately, for all his cleverness, can't compete with the machine anymore.

Let code thrive on the tedious stuff while you, the human being, concentrate on solving those same problems in more clever ways.

#25: Take advantage of a machine's strengths

Gauss's tale provides some interesting insight that you might have taken for granted in today's world.

Had we kept our initial function intact, a program would have willingly summed all the numbers from `1` to `4,273,558`. It might have taken a long time, but it wouldn't stop unless the machine ran completely out of resources.

On the other hand, a human being (like Gauss) approached the problem differently. Instead of just going at it, Gauss realized there was a better way. With a little bit of creative thinking, he found a way to simplify the problem — a uniquely human quality.

People have qualities not found in code — like the ability to reason and analyze outside the boundaries of perceived rules. On the other hand, code, unlike people, will follow directions (even unreasonable ones) and perform them precisely as directed, to the death.

Though these are obvious truths, they are often forgotten and under-utilized in our everyday work as programmers. Getting the *right* systems to work on the right *problems* is the key to real productivity.

Here are some other qualities about code you might not have thought about recently. Compare each of these to your own work habits.

Code doesn't get lazy

It will never decide to just take a shortcut or find an easier way of doing things. The code snippet at the beginning of this chapter would sum up all 4-million-plus integers without deciding to just skip over one. Code executes with impeccable precision.

Code doesn't get bored

Here are some nonsense instructions:

```
while (x != x + 1)
{
    // do nothing
}
```

As irrational as this task sounds, code will continue to crank at this nothingness forever, saved only by a run-time engine that would force an abort sensing this loop would be going nowhere fast. Code doesn't analyze the importance of a task. It has no interest in its own well-being. It simply executes.

Code doesn't forget

At the beginning of this chapter, I told the engine precisely this:

"Whenever I tell you to `sum_range_of_positive_integers(1, 100)`, create an integer called `sum`. Starting with the number `1`, add the value to `sum`, and then increment the value by `1`. Keep doing this until you've hit `100`. Then, give me back the value of `sum`."

Years later, I can go back to my software, call the method again, and expect the same result. Code doesn't forget what it's asked to do. Software systems, built upon thousands and thousands of lines of code, are equally adept. Code, no matter its volume, remembers what to do days, weeks, and years later. Could you?

Code is cheap

You would understand if the co-worker who would willingly sum the numbers from 1 to 4-million-plus asked for a raise in return. You could work out some commission per summation agreement or agree to an hourly rate. Summing 1 to 10 would be a lot cheaper than 1 to 1 billion.

We've, fortunately, never taught code about money, market economies, or vacation homes. It performs without monetary needs. Code asks for nothing in return. Once you've taught code something, you can take full advantage of it. There are no code-labor laws, at least as of this writing, to get in your way.

Code is fast

It executes at a rates incomparable to the way we can finish tasks. It's metered by the limitations of hardware, which will become less and less limiting over time. Humans are absolutely no match for speed.

What does this all mean? Imagine a Craigslist ad that went like this:

> **Diligent Software Programmer Looking for Work!!! (Earth)**
>
> Teach me anything. I'll learn quick and let you know if I need more information instantly. I will work whenever you want, however much you want. I will never forget anything you say and will never complain that I'm not being challenged. I am particularly good at tedious work.
>
> - Location: Anywhere
> - It's NOT ok to contact this poster with services or other commercial interest
> - Compensation: $0

Code is inarguably the greatest junior developer that ever lived. It is uniquely adept at *tedious* yet, *definable* tasks. It never complains — unless your instructions don't make sense. It's cheap, fast, diligent, consistent, and unemotional. Don't you wish that on programmers sometimes? The power of code is extraordinary.

Programs work amazingly at the tedious but automatable. People are terrible at it. When you've figured out what *is* the tediously automatable stuff, you can pull apart the work that you should be doing from the work that you should be building programs to do.

#26: Think automation

When my company, We Are Mammoth, first went into business, we built Flash-based web sites for a variety of clients. Yes, Flash sites. Forgive me, but, this was pre-HTML5, pre-Scriptaculous/jQuery effects, pre CSS3, pre Steve Jobs' declaration of war against Adobe.

Moving on.

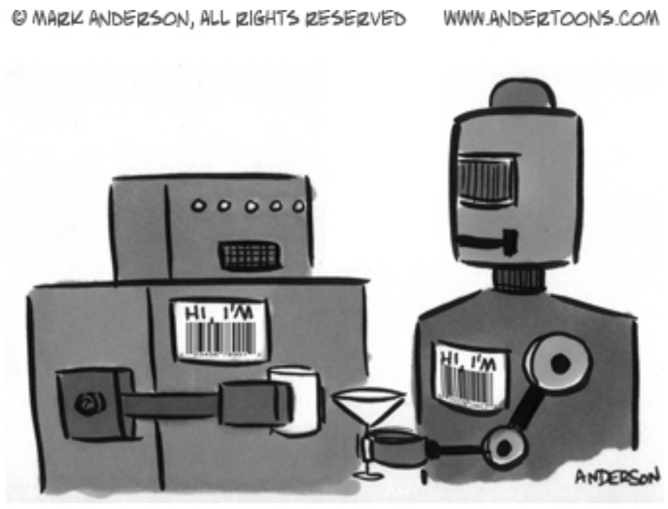
A few months into our business, I began smelling repeatable work – work that we were doing in the same mechanistic way each time. Having seen the process for a few iterations, I could separate the common tedious, yet automatable elements from the custom work that applied to each project we built. They bubbled up like oil in water.

Every application we built followed a common set of themes. We used the same sets of languages (C# and AS3) in roughly the same way each time. We used SQL Server to store and retrieve data. Each application needed a data access layer, a series of common web-service commands, and a way to parse and fill data from those web services into AS3 value objects on the Flash end. It turned out that variability only lived in two places in our repeatable architecture.

First, we couldn't trivialize the data model. We worked on applications for car companies, bed manufacturers, brokerage firms, software distributors, fast-food chains, and airlines. Their databases were custom-tailored toward solving their own unique business problems.

Second, we couldn't trivialize the front-end. Design, functionality, and the user experience are what make each application different.

What we could trivialize was a lot of the work that sat in between the database and user interface. A data access layer (DAL), common web services (transferred via XML), and AS3 value objects (VOs) could all be deduced from the data model. In other words, once we defined what the database model and user experience looked like, much of the stuff in the middle was automatic. It was obvious what had to be built. It was really tedious, banal stuff, fit for robots.



A rough architecture of a typical application looked like this, with the potentially automatable parts in bold:

Model → **DB** → **DAL** → **XML** → **VOs** → UI

Let's see how the automatable parts of an application could be turned into instructions for a program to do.

Building a blog from scratch

Imagine building something as simple as a blog, soup to nuts, by hand. Start with the database. The data model might contain three tables that look like this:

- `Posts (ID, Title, CreateDate, Body, AuthorID)`
- `Authors (ID, FirstName, LastName)`
- `Comments (ID, Comment, EmailAddress, CreateDate, PostID)`

If you're familiar with database schema modeling, this model is a straightforward one. `Posts` have a title, body, creation date, and a relationship to one `Author`, via the foreign key `AuthorID`. The `AuthorID` keys into the table `Author` by matching on its `ID` column. `Comments` have an email address, creation date, and a related originating `Post`, via the foreign key `PostID`. The `PostID` keys into the table `Post` by matching on its `ID` column.

Next, let's write some SQL procedures to get the data we need out of the database. For every table, I can write four procedures for creating, reading, updating, and deleting a record from a particular database table. They are widely known as `CRUD` methods. Here's how they might look for `Posts`:

***C**reate post:*

```
CREATE PROCEDURE CreatePost (
    @Title NVARCHAR(255),
    @CreateDate DATETIME,
    @Body NTEXT,
    @AuthorID INT)
AS
INSERT INTO Post VALUES (
    @Title,
    @CreateDate,
    @Body,
    @AuthorID)
```

***R**ead post:*

```
CREATE PROCEDURE LoadPost (@ID INT)
AS
SELECT * FROM Post WHERE ID = @ID
```

***U**pdate post:*

```
CREATE PROCEDURE UpdatePost (@ID INT,
    @Title NVARCHAR(255),
    @CreateDate DATETIME,
    @Body NTEXT,
    @AuthorID INT)
AS
UPDATE Post
SET
    Title = @Title,
    CreateDate = @CreateDate,
    Body = @Body,
    AuthorID = @AuthorID
WHERE
    ID = @ID
```

***D**elete post:*

```
CREATE PROCEDURE DeletePost (@ID INT)
AS
DELETE FROM Post WHERE ID = @ID
```

I'd then repeat the same process for `Authors` and `Comments`.

I then can extrapolate on the relationships between these tables. A post has an author. So, I could write a stored procedure to get all the blog posts by a specific author's ID. Let's call it `GetAllPostsByAuthorID` In fact, there's a simple formula to the madness. For any foreign key `[Y]` in a table `[X]`, I could write a stored procedure of the form: `GetAll[X]By[Y]ID`:

```
CREATE PROCEDURE GetAllPostsByAuthorID(@ID INT)
AS
SELECT * FROM [Posts] WHERE AuthorID = @ID

CREATE PROCEDURE GetAllCommentsByPostID(@ID INT)
AS
SELECT * FROM [Comments] WHERE PostID = @ID
```

I might also want to load records by filtering on a specific field. For instance, I'll need to get posts for a given day (`GetAllPostsWhereCreateDateEquals(CreateDateParam)`, or authors by their last name (`GetAllAuthorsWhereLastNameEquals(LastNameParam)`). Another formula emerges.

For any filterable field `[Z]` in a table `[X]`, given a parameter `[P]` I could write a stored procedure of the form `GetAll[X]Where[Z]Equals([P])`. For fields that are of type `DATETIME`, let's add a stored procedure that filters posts by a start and end date. Now, we have an additional set of *deducible* stored procedures, by simply introspecting the database schema:

```
CREATE PROCEDURE GetAllPostsWhereCreateDateEquals
(@CreateDate DATETIME)
AS
SELECT * FROM [Posts] WHERE CreateDate = @CreateDate

CREATE PROCEDURE GetAllPostsWhereCreateDateInRange
(@StartDate DATETIME, @EndDate DATETIME)
AS
SELECT * FROM [Posts] WHERE CreateDate >= @StartDate AND
    CreateDate <= @EndDate

CREATE PROCEDURE GetAllAuthorsWhereFirstNameLike
(@FirstName NVARCHAR(255))
AS
SELECT * FROM [Authors] WHERE FirstName LIKE @FirstName

CREATE PROCEDURE GetAllAuthorsWhereLastNameLike
(@LastName NVARCHAR(255))
AS
SELECT * FROM [Authors] WHERE LastName LIKE @LastName

// ...and so forth.
```

Stored procedures finished. I can now set my sights on the data access layer and middle-layer objects. Again, a pattern emerges. The code that grabs data from a database, and the objects that store the data, can be deduced from the structure of the database schema.

Here's a shortened version of what the classes for `Posts` might look like in C#:

```
public class Post
{
    private int _id;
    private string _title;
    private DateTime _create_date;
```

```
        private string _body;
        private int _authorID;
        private Author _relatedAuthor;
        //...
    }

    public class Posts : Collection
    {
        private List<Post> _posts;
        //...
    }
```

By examining the Posts table in the database, I can then create parallel methods in C# to access the stored procedures I wrote earlier.

```
public void InsertPost(Post post) {}
public Post LoadPost (int ID) {}
public Post UpdatePost (Post post) {}
public void DeletePost(int ID) {}
public Posts GetAllPostsWhereCreateDateEquals(DateTime date) {}
public Posts GetAllPostsWhereCreateDateInRange
(DateTime startdate, DateTime enddate) {}
public Posts GetAllPostsWhereCreateDateInRange
(DateTime startdate, DateTime enddate) {}
//... and so forth.
```

Once these data access layer objects are written, I'd move on to create web services, write code that would consume and parse these services in AS3, and write corresponding value objects in AS3. Again, this code can again be largely figured out from examining the structure of the database. For the purposes of this book, I'll spare you the details. But, only when all that work is done can I finally pull data from the database into the UI.

Defining the recipe for building an app

In pseudocode, the recipe for building an application from the bottom-up looks something like this:

- **Step 1:** Define a database schema
- **Step 2:** Build SQL stored procedures

- **Step 3:** Build C# data-access layer
- **Step 4:** Build web services
- **Step 5:** Build objects to consume web services
- **Step 6:** Build AS3 value objects
- **Step 7:** Hook up data to the user interface
- **Step 8:** Design the user interface

After building the database schema, steps 2-7 were surprisingly straightforward for much of the work we did. Much of it is deduced by just examining the database. Sure there are custom stored procedures and a fair amount of business logic to write that are unique to a blog, but a significant amount of the work is automatable.

That automatable work is fit work for the fictitious Craigslist job seeker. Or, more realistically, I can *write smart programs to do the job for me*.

In this blog exercise, we've extracted out the common processes of building every application from the processes specific to each application. In other words, it would be hard to teach a program to "create a blog" but you could tell it how to "create some useful stored procedures given a database schema."

Every tiny little repeatable process that I can teach a human being to do, I can write a piece of code to do a million times better. Once I've taught the generator how to properly write something given a set of inputs, I could always expect the same fast, predictable results over and over again. It now frees the human from doing repeatable work.

It was with this very recipe in mind that I began writing a code generation framework called X2O. X2O provides an interface to build a data model, and then generates all the underlying tiers of an application based off the data model.

After a few months, we saw other opportunities to expand X2O. We could generate developer documentation, JavaScript classes, and even fairly high-level CMS tools based just off an application's database schema.

#27: Know the ingredients of a good code generator

So, how do you actually write a generator? If you want a truly in-depth source on code generation, I recommend Jack Herrington's *Code Generation in Action*. It covers detailed techniques and high-level patterns for generating code of all kinds. But, you don't need that level of detail to get started. Here's what you need to know.

Use a programming language with I/O capabilities

First, you **must** program in a language that can interact with files on your machine. Your language of choice must support reading, manipulating, writing, and deleting files to disk.

Fortunately, pretty much any of today's popular programming languages (C, C++, C#, VB, Java, PHP, Python, Ruby, Perl) support this. If you've never read or written files using your programming language, spend an hour researching it. Know how to read, manipulate, write, save, and delete files. Your code generator will be doing a lot of this.

Define your input source

Second, you need an input source. It's the place that houses all the parameters your code generator needs to...generate code. In X2O's case, the input source began with an XML file. The XML file defined the data model – it contained nodes for tables, fields (and their data types), and foreign keys. It gives the generator all the information it needs to create the database, SQL scripts, data access layer, and so forth. Here's a example of converting the blog data model into a simple, XML input source:

```
<input_source>
    <table name="Posts">
```

```
        <field name="ID" type="int" identity="true" />
        <field name="Title" type="NVarChar" length="255"/>
        <field name="CreateDate" type="DateTime" />
        <field name="Body" type="NText" />
        <foreignkey name="AuthorID" to_table="Authors" />
    </table>
    <table name="Authors">
        <field name="ID" type="int" identity="true" />
        <field name="FirstName" type="NVarChar" length="255"/>
        <field name="LastName" type="NVarChar" length="255" />
    </table>
    <table name="Comments">
        <field name="ID" type="int" identity="true" />
        <field name="Comment" type="NText" />
        <field name="Email" type="NVarChar" length="50" />
        <field name="CreateDate" type="NText" />
        <foreignkey name="PostID" to_table="Posts" />
    </table>
</input_source>
```

Over time, your input source will grow. As you find more things to generate, you'll likely need more kinds of inputs. For example, when I wanted to generate documentation, I added friendly descriptions as attributes to each table and field node. My documentation generator could then reference that data to spit out more customized documentation. You may want to define the order of each attribute as they appear in a form to generate a rudimentary CMS. You can add that attribute to each field node in your input source as well.

Of course, your input source doesn't have to be an XML file. It can be a text file, a database, an RSS feed – it doesn't matter. A year after we began writing X2O, we replaced the XML file input with a database. That allowed us to stop manipulating XML files and work with an elegant, web-based client instead. Today, our input source is just the database that lives underneath the X2O web client.

Extract your input source into something usable

With input source in-hand, write a program to extract its contents into something usable. In my case, we mapped the contents of the XML file into its own object in C#. This lets you have both a system that's easy to work with when constructing the input source (XML) and a system that's easy to work with when your generating code against the input source (like an object in C#).

In today's landscape, languages like E4X(ECMAScript for XML) makes converting an input source into a programmatic object pretty seamless. Whatever method you use, it's critical to have an easy way to loop through and introspect your input source. You'll see why in the next step.

Combine your input source provider with templates

With a usable programming environment and input source defined, the next step is to write templates. In our blog example, each tedious part of the development process had a formula. For example, to generate all CRUD statements, we do nothing more than loop through every table in our data model, and apply the same statements for each.

Take the CREATE statements. We can take the following bit of real SQL code...

```
CREATE PROCEDURE CreatePost (
    @Title NVARCHAR(255),
    @CreateDate DATETIME,
    @Body NTEXT,
    @AuthorID INT)
AS
INSERT INTO Post VALUES (
    @Title,
    @CreateDate,
    @Body,
    @AuthorID)
```

...and replace the custom portions with replacable variables...

```
CREATE PROCEDURE Create[cur_table] (
[List_of_attributes_as_input_params]
)
AS
INSERT INTO [cur_table] VALUES (
[List_of_attributes_as_sql_insert_params]
)
```

...to create a template for generating CREATE statements.

With this template, we can loop through each table node in our input source provider and fill in the template with the appropriate values. In this case, `cur_table` is just the name of each table, while `List_of_attributes_as_input_params` and `List_of_attributes_as_sql_insert_params` are found by inspecting the field nodes of the input source provider. We can create a `CREATE` stored procedure for each table in our data model as follows:

```
// This holds our template in memory
string template = file.read("SQLCreate.txt");

// This will hold the final generated output
string output;

foreach (table current_table in input_source)
{
    string cur_table = current_table.Name;
    string p1;
    string p2;

    /* Loop thru each field in the table to create
    List_of_attributes_as_input_params and
    List_of_attributes_as_sql_insert_params */

    foreach (field in current_table)
    {
        p1 += "@" + field.name + " " + field.type.ToUpper()
           + ",";
        p2 += "@" + field.name + ",";
    }

    // Get rid of the last comma...

    p1 = p1.substring(0,p1.Length - 1);
    p2 = p2.substring(0,p2.Length - 1);

    // Write the CREATE statement for current table
    output += template.replace("[cur_table]",cur_table]
    .replace("[List_of_attributes_as_input_params]",p1)
    .replace("[List_of_attributes_as_sql_insert_params]",p2);
}

// Write the final output to a file
file.write("my_create_statements.txt", output);

// Assuming the database was already generated,
// run the SQL file against the database.
```

In pseudocode, the creation of generated code looks like this:

1. Build an example file for the code you wish to generate.
2. Create a template by extracting out the custom parts and replacing them with variables.
3. Write code to read in the template file, loop through the input source, and replace the variables from the template file as necessary.
4. Write the newly created file to disk.
5. Do something with the files at the end (run them, compile them, etc.)

Component-driven design

A good rule of thumb: Keep all your generators as separate class files or libraries (depending on your programming environment). Early on, X2O was a mass of code in one large file. The code that generated the database, SQL scripts, data access layer, web services, Flash objects, and CMS files all lived in the same library. While it worked, it grew unmanageable. It was harder to maintain because any minor change to the generator meant recompiling tens of thousands of lines of code.

Once we pulled each part out into about three dozen separate libraries, it was a lot easier to maintain. You can then chain all the generators together by referencing them in one all-encompassing master generator library.

Encapsulating and componentizing is a good programmer habit anyways. But, it's especially important when you're building dozens of little generators.

With these five simple tips in mind, and plus the aid of a book like Herrington's *Code Generation in Action*, you can get out of the starting gates.

The very first time

The first time you get your code generator to generate something, I mean *anything*, is thrilling. It feels like your cheating. You've essentially hired a machine that will abide by your every word. So, get to that thrill as fast as you can.

As I mentioned in *Motivation*, get your code generator to "launch" as early as is achievable. Chew off something *small* to generate and generate it. It might just be generating getter/setter methods for a set of properties in your input source. Or, a program that writes data from your input source into an HTML table. Find something simple and useful to start – something that will get you excited when you realize you never have to write this little bit of code by-hand anymore.

When you get that first mini-generator started, motivation will stream through your veins.

#28: Avoid touching generated code with bare hands

You can write generators that produce code you intend to modify or you can write generators that produce code you'll never actually touch.

With X2O, we've made a strict rule that any generated code is not to be modified after the fact. Generated code is like fine China...

You break it, you pay for it!

We don't hit the "Generate" button and then noodle around with the freshly generated stuff. Why? Suppose we add a new field to our database and want to re-generate our new code against an updated data model. Each time we did that, we'd have to remind ourselves what we hand-modified and ensure the code is modified again.

Code generators that are "one-time only" rarely withstand the test of time. Inevitably, something will change in your process or requirements that will force changes to what was once generated.

If you find yourself needing to noodle around your generated code, there are elegant ways around the problem. In C#, you can mark a class as *partial*. This lets you define a class in multiple source files. In X2O, every generated C# class is partial, so that, if we ever needed to, we can add any additional methods or properties in a separate file marked with the same partial class.

If you don't have the luxury of partial classes in your language of choice, there are other elegant approaches. You can extend classes, implement interfaces, or write custom helper classes. There are all sorts of approaches to take if you find reason to modify your generated code.

#29: Writing a code generator makes you a better programmer

Writing your own code generator also makes you think critically about your development process. It requires you to figure out what parts of your development process are actually repeatable enough to be good candidates for code generation. It transitions you from *assembly-line developer* into *creative thinker*.

That's a healthy practice for programmers.

For example, X2O generates a content management system by introspecting the database. There's a one-to-one correspondence between add/edit/listing pages and tables. There's a one-to-one correspondence between database field types and input forms. There's a one-to-one correspondence between required foreign key constraints and required dropdown boxes in an "add" or "edit" screen.

But, there's a balance between how much code you *should* be generating and how much code *should* be left for custom craftsmanship. After the first few sweet tastes of successful code generation, you might feel that air of invincibility and start trying to wrap everything into a code generator – even the things that really aren't automatable (but certainly tedious). It's easy to try to cram too much automation into things that are still too custom.

This is where you really have to consider the benefits of code generation. If you're output code requires too many custom inputs to generate or requires too many hacks to use, you probably shouldn't be generating that bit in the first place. Just like bad code smells, there are also bad code generation smells.

Writing code generators gets you thinking about what is truly *automatable* and tedious vs. what is just tedious.

#30: Speak, think, and write code in a ubiquitous language

One incredibly valuable byproduct of code generation is the creation of a ubiquitous language. People working on all different areas of the application can speak on the same terms. It especially works when you have one centralized model (i.e. a database) that spawns off many different generated products. When scripts, code, front-end layers and documentation all spawn from the same central source, naming conventions become inherently consistent.

This means people using any part of the code generator can all speak the same meta language. Had people written this by hand, we might see small disparities in language convention. A .NET developer might create a class called `Person` that might be pulling data from a database table called users, customers, or employees, even when they might correspond exactly one-to-one.

Granted, the domain that a middleware developer works in doesn't necessarily map one-to-one with the domain that a database designer lives in. But, most web apps today do. The adherence to similar naming conventions across multiple tiers is the philosophical strategy behind web frameworks like Ruby on Rails or Django.

And even if that's not the case, starting off with the same conventions in nomenclature from the get-go (and then breaking convention as it makes sense), sidesteps some of the common inefficiencies in team development. Where models can translate from one tier to another, you ought to stick to the same naming convention. And, a code generator can enforce that.

A ubiquitous language means that all stakeholders talk about the same application in the same vernacular, instead of letting technological disconnects stand in the way.

#31: Generated code makes errors obvious and small modifications powerful

Writing your own code generator might seem like a lot more extra work at first. You're always one step removed from the actual production code. There's something slightly less tactile about the process – it feels like writing code with latex gloves on. But, once you get over that initial awkwardness, you'll find that writing code that writes code is actually easier at times.

Debugging is a perfect example. If something in your generated code isn't working right, the errors will become obvious. The same error will repeat itself *over and over again*, in the *exact same* fashion. Remember, it's your code that's writing the real production code, not a fallible human being. It's much easier to spot what's wrong when things are wrong in the exact same way, each time.

Compare this to a developer that makes the occasional sloppy, lazy coding mistake. It happens to the most diligent of us. A code generator makes debugging easier because there will be a consistency to the errors being made. And, once you've found the source of the problem, you make a tweak, re-run your generator, and replace the problem throughout your outputted code, all at once.

In code generation, errors are obvious and small tweaks are powerful.

#32: Address the naysayers

There are many out there that scoff at code generation. They say it creates bad habits and bad architecture. While this can be true, it's a human problem. You're in complete control to avoid these problems.

Some argue that code generation is a trade-off between rapid output and custom-fit code — it creates a lot of excess that rarely gets used by the end application. Because it's so easy for programs to write code, you may not care as much to have it generate concise, optimized code. But, this is something easily resolved.

Perhaps your next project doesn't need a certain set of data access methods. Use your input source to define some optional parameters so that you're not spitting out sheets of excess code for a project that doesn't need it. As your generator matures, you might want to enable and disable certain code from generating. This is where component-driven design really helps.

Some argue that code generators produce inelegant code. Blasphemy! This has nothing to do with code generators and everything to do with how you prescribe what your code generator should produce.

If your generated classes have duplicate functions or common methods, then refactor the templates that make up your code generator. Write the duplicate functions into a stand-alone class that live outside the generator. Write an interface and let each generated class implement it. You can still apply the same programming-by-hand techniques to your generated code.

When we code by hand, we usually follow this heuristic:

1. Make the code compile
2. Make the code do what we claim it should do
3. A long while later, refactor it so it's easier to maintain.

What's the number one reason we refactor code we write by hand? It makes the code easier to maintain in the future. That's why it's hard to motivate yourself to refactor generated code – you never need to actually *maintain* the code you generate. You only maintain the generator.

But, there is nothing that says your code generator can't follow good principles. If your generated code repeats the same methods 30 times in 30 different classes then, modify your generator so those methods live in a class that can be inherited by the other 30 generated classes. Apply the standards you adhere to when you write code manually into the code you generate. Resist the temptation to take shortcuts in your outputted code.

In addition, some argue that generated code makes your applications inflexible. They have to conform to whatever framework the generated code has imposed on you. Here, the onus is on you. If your generated code forces limitations on your end-applications, you may need to reconsider whether you're generating the *right* things.

It's tempting to push more of your work off to building a code generator, even at the expense of the end-application. Know where to draw the line between work that's automatable from work that you really should be doing yourself each time.

#33: Automating software is not like fast-food

The best pizzerias in the world, the ones you hear your friends and self-proclaimed TV food aficionados rave about, make their pizzas by hand, in-house. But, suppose one day your favorite pizza place fired the chefs and replaced them with an assembly line of machines. Each machine is in charge of building different parts of the pie – the dough pounder, the sauce layer, the cheese topper, the mushroom dispenser, and the perfectly timed oven. You could build machines to automate the construction of a pizza, and it would be perfect each time.

If measured by precision, machines ought to do a far better job than human pizza makers. They'd surely be more consistent and faster. Although consistency and expediency are huge components to a restaurant's success, the restaurant still might lose business.

While consistent quality is a key component to a good pizza, there's value in the handcrafted nature of pizza. A handmade pizza is shaped slightly differently each time. Certain ingredients might be fresher this week than last. There are desirable burnt edges here or there. The imprecise cutting of the pie usually creates a few guilt-free mini slices ("Oh, I'll just have this little one here"). The organic nature of handmade pizza has value to the consumer.

Whether it's our curse or our good fortune, building the *algorithmic* parts of software by hand adds no intrinsic value to the product. It's simply not a selling point. As much as you enjoy the building of software, consumers don't care. The public doesn't crave inconsistent bits of handmade software — they want the very opposite. Automating that-which-can-be-automated saves us time and energy to focus on those parts which aren't automatable and require more delicate thought without the drawbacks you might find in industries like the food industry.

Back in the 1950s and 1960s, fast food was hip. It was the future of food —

that which showed the might of American ingenuity. It fit the lifestyle of a nation who predominantly travelled by automobile and wanted a quick, yet still satisfying way to enjoy food "on-the-go."

Yet, a few decades later, that mystical magic food began to backfire. The quasi-automated nature of making such food led to major problems in our society. Fast-food culture has produced a major obesity epidemic due to the non-natural content of the food, health-care scares due to the things we feed our animals and the pesticides we spray to mass produce crops. And besides all of the social, environmental, and moral benefits, a culture simply fell back in love with *real hand-crafted food*.

Fortunately, we don't have that problem with "fast code."

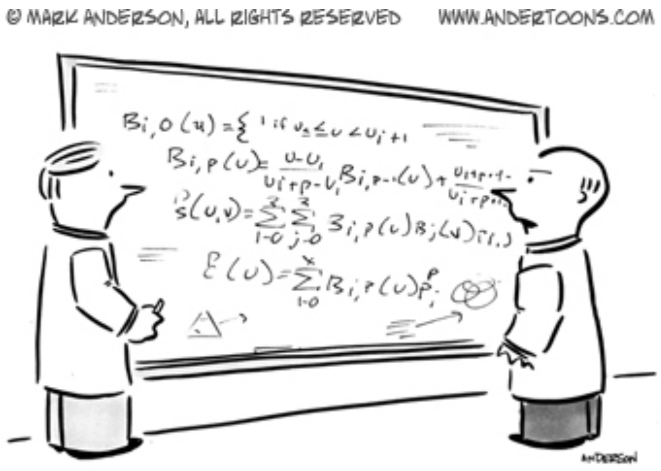
While we love thoughtful software, we can still mass produce the algorithmic bits with the same quality of code as if we crafted it manually. There's no gain in typing code by-hand that can more quickly and more accurately be generated.

That's why we *ought* to build our automated dough pounders, sauce layers, and cheese toppers in our software when and where we can. It gives us more time to craft the custom parts of our software — the parts that require more human thought and craftsmanship. Automating, where applicable, makes our software better, at no loss to the consumer.

Complexity

Aside from death and taxes, complexity may be the only other sure bet in life. Complexity always grows over time. In our industry, it's the unavoidable consequence of maturing software. Unless you're willing to remove features, there is simply no way to sidestep it.

If you can't get rid of complexity, your next job is to stifle its growth. Recognize when complexity *isn't* necessary. Develop a finely tuned nose for it. When you know what it smells like, you can pick up the all-to-familiar stench from every nook and cranny of your software. You'll be better off in the end.



"What the hell is that supposed to mean?!"

#34: Snuffing out bad complexity

Bad complexity is complexity that just *doesn't have to be there*. And it's not always easy to spot. Sometimes, even the pieces we think have to be there really do not.

In 1776, Thomas Jefferson sat on the committee that drafted the Declaration of Independence. The committee gave Jefferson the privilege of writing the first draft. When the draft was completed, he sent it to his friend, Benjamin Franklin for review. His draft was returned with much of Jefferson's handiwork erased or severely reduced.

Jefferson was none to happy, but Franklin tried to convince his friend why this was for the better. As Congress analyzed the draft, Jefferson recounted the story that Franklin had told him:

> "When I was a journeyman printer, one of my companions, an apprentice hatter, having served out his time, was about to open shop for himself.
>
> His first concern was to have a handsome signboard, with a proper inscription. He composed it in these words:
>
> John Thompson,
> Hatter,
> makes and sells hats
> for ready money
>
> with a figure of a hat subjoined; but he thought he would submit it to his friends for their amendments.
>
> The first he showed it to thought the word 'Hatter' tautologous, because followed by the words 'makes hats,' which showed he was a hatter. It was struck out.
>
> The next observed that the word 'makes' might as well be omitted, because his customers would not care who made the hats. If good and to their mind, they would buy them, by whomsoever made. He struck it out.

> A third said he thought the words 'for ready money' were useless, as it was not the custom of the place to sell on credit. Every one who purchased expected to pay. They were parted with, and the inscription now stood, 'John Thompson sells hats.'
>
> 'Sells hats!' says the next friend. 'Why, nobody will expect you to give them away. What then is the use of that word?' It was stricken out, and 'hats' followed it, as there was one painted on the board.
>
> So the inscription was reduced ultimately to 'John Thompson,' with the figure of a hat subjoined."
>
> *Courtesy: PBS.org*

Plenty of software could take a cue from the hatter's sign. Does this button need to be there? Is this line of copy adding anything of value, or is it merely repeating something else that's already there? Does this new feature actually help make the task easier?

#35: The simplicity paradox

Here's the maddening thing: *Everyone. Loves. Simple.* That's why people say "I just want things to be simple." Who says "I just want things to be complicated"...*ever*?

I decided to find out.

As of this writing, if you Google the phrase "I want things to be simple," you'll get approximately 954,000 matching results. There is **one** unique matching results for the phrase "I want things to be complicated."

One. The only unique matching result? My company blog post that I wrote about this very subject in October, 2009. Extract myself out of the annals of recorded human civilization, and, apparently, no one has ever wanted, or even mused the idea of voluntary complication.

Then, why do we run into this Jeffersonian problem of complexity when we're building our own stuff, so often? Why do the things we produce often wind up festered in complication? How do so many well-intentioned pieces of software matriculate from simple idea to functional nightmare?

Sometimes it's because simple products are, quite simply, hard to build

Most ideas, simple at the surface, are viciously complicated when you get into the details. Because ideas, at a high-level, are always simple. Every business idea must be accompanied by the elevator pitch — 60 seconds to get the message across from beginning to end. You can't pack complexity into a 60-second description.

When ideas start feeling complex, you've left the comforts of Idealand and actually started thinking about the implementation. Once you dig into the details, you discover new states, edge cases, and question marks. It's the nature of *detail*. An idea that hasn't been thought through completely (read: most of them) has little chance of surviving Complexityville at this point. Rather than rethinking the idea altogether, it's sometimes easier to plow through the problems with head down and blinders up. Half-baked decisions are made; features are added all for the sake of sparing the sanctity of the "Big Idea." Then, complexity festers.

But, not all simple things are hard to build.

If everyone likes simple, and not all simple things are simple to build, you'd think that most software should be both simple-to-use and simple-to-build. It's a win-win for both user and developer. But, that kind of software rarely exists in our world. There has to be something more to this mystery.

More often than not, simplicity is lost, because, when you build something simple, it just doesn't feel like it's *enough*. You mislead yourself into believing your customer isn't getting his money's worth. A simple thing that's *also* simple to build feels value-less. An idea that's easily implemented is rarely considered a "big idea" at all. Truly simple ideas seem inferior; if it's simple to build, it must not be a good idea.

Businessmen, venture capitalists, and angel investors don't throw millions of dollars at simple ideas. They throw money at the Donald Trump superlatives — innovative, cool, cutting-edge, best-in-class. Often times, these are just other ways of saying an idea is *complex* enough to be worth its weight.

Herein lies the paradox. From a builders point of view, we often equate the worth of the software we build to its complexity. More complexity equals more value. Even when we've all agreed to build something simple, inevitably, our fear of inadequacy leads us to something more complex.

The view from the other side of the mirror is a bit different. The reality is, 90% of the world uses only 10% of the features built in the average enterprise-level software. When users can't find the few functions they need because they're buried amongst the many features they don't need, they either take it out on our own perceived shortcomings or blame it on the software itself.

Users see complexity as the shortcoming; builders, marketers, and stakeholders see simplicity as the shortcoming.

Standing behind something simple feels too easy and too cheap. But, we forget that the real end-user often has different parameters for success.

At We Are Mammoth, the natural urge to complicate is something we resist constantly in our business. We have to re-sell and re-pitch simple to ourselves each time.

Countless internal arguments about features in our own software end up with incredibly simple solutions — changes in text, moving a link somewhere else on the page, removing something, or deciding the new feature just wasn't worth it to begin with. Yet, these simple conclusions came out of hours and days of discussion, thinking, and re-thinking.

It's natural to feel that the amount of time you spend on something should parallel the amount of measurable output you put into the product, regardless of the benefit of that new feature. But, free yourself from that debilitating thought. Once you've let go of the vulnerable feeling that simplicity cheapens your worth, you can finally get on with building good software.

Simplicity empowers. It is one reason why experts keep doing the things they excel at – because those things come more naturally (read: simply) to them than to the masses. We crave things that make sense, have rules, boundaries, and reasonable expectations. It lets us wrap our head around a problem completely. Simplicity is not boring – it's comforting. Simplicity is timeless.

A simple solution shouldn't be thought of as "not enough" of anything. Sometimes it is exactly enough of everything.

#36: Complexity as a game of pick-up sticks

Software sometimes feels a lot like *Pick-up sticks*: the children's game where you try to remove individual plastic sticks from a pile without disturbing the others. You start the game by holding a bundle in your hand and letting them go. Most of the sticks fall in a pile in the middle, while a select few roll their way away from the center.

Each stick represents a feature. Sometimes, a feature just lives outside the whole mess of others. Sometimes, it touches a few components. Sometimes it's completely entwined around everything else.

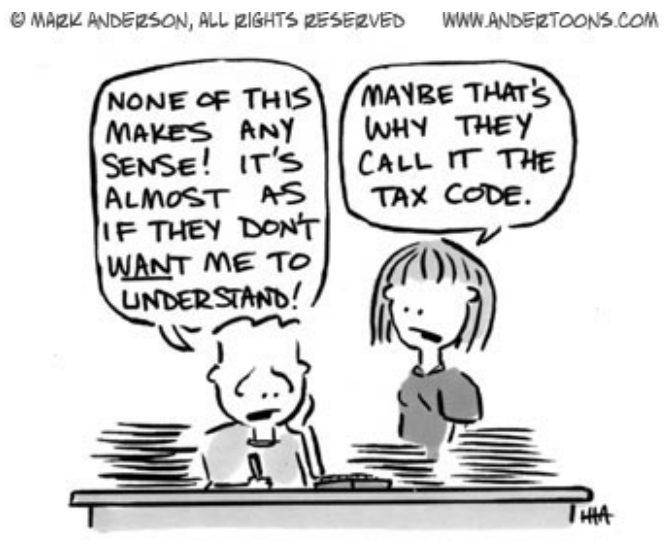
Implementing a new feature is like adding a few new sticks into the mix. At a certain point, trying to remove any stick without disturbing the rest is nearly impossible. Complexity adds up fast. We try our best to circumvent this by following good habits – encapsulating, scoping variables at the right level, breaking apart larger pieces of logic into bite-size chunks, introducing patterns. It's our way of aligning the sticks in parallel, side-by-side, so they don't touch each other. But, consistently refactoring code into the "right" places while continuing to add more sticks into the mix can get hairy. It's easy to let your guard down.

Every time you add a new feature, you stand the chance of disrupting a host of other features that might not at first seem directly connected. As you add more features, those disruption points grow pretty rapidly.

#37: Complexity under the surface != Complexity at the surface

"An extraordinarily complex mess."

That's what Nina Olson, the national Taxpayer Advocate for the U.S. Internal Revenue Service calls the American tax system. The official book weighs in at about 6,500 pages. Can you imagine? If you think I'm lying, you're right. It's actually about **65,000** pages long.


© MARK ANDERSON, ALL RIGHTS RESERVED WWW.ANDERTOONS.COM
NONE OF THIS MAKES ANY SENSE! IT'S ALMOST AS IF THEY DON'T WANT ME TO UNDERSTAND!
MAYBE THAT'S WHY THEY CALL IT THE TAX CODE.

This is why I love TurboTax®. It's taken these 65,000 pages of documentation and miraculously curated some sort of usable software out of it, for commoners like me. TurboTax® could so easily have just reproduced the Federal and State 1040 Forms digitally, turning each line item into a text field, calculating a few fields here and there, *gratis*, submitted the form for you via the Information Superhighway and be done with it.

It probably still would have a lot of fans.

Instead, TurboTax® is like a personal tax wizard who understands that no one really wants to hang out with it. "Get your W-2 out and copy the numbers into me." "Do you own a farm? No?! Then let's not ever mention it again." It even knows when it's asking me esoteric questions, by letting me know that something is very unlikely or uncommon. "This probably doesn't apply to you."

TurboTax® has done a heroic job of making tax filing, at least, palatable. I can only imagine what a stinky mess the underlying code must be. Logic not just for federal law, but for each of the fifty states, and for each of the thousands of local counties and beyond. And within those parameters, logic for single and married people, small-business owners, investors, students, philanthropists, first-time home buyers, the unemployed, the just-retired, the forgetful, the poor, the rich, the richer, and — yes, farmers.

Couple that with the annual changes to tax laws — every little rule taken out or put in to account for some earmark in government spending that make up those 65,000 pages of dead trees. If you think you're angry about a couple extra hundred dollars you owe the government, imagine being a TurboTax® developer writing yet another weird bit of conditional logic for someone who's just bought an environmentally-friendly motorboat in Mississippi within the last six months.

Is any of that code even worth refactoring? What would you refactor, when the very code you're writing today might be obselete after the next Congressional recess, or might depend on a totally different set of parameters down the road?

TurboTax® is proof that, even when the task at hand is an *extraordinarily complex mess*, the software itself doesn't have to be. You can squeeeze all that complexity underneath the surface, interpret that mess into some reasonably digestable set of usable functions, and create some really helpful, far more simplistic, software.

#38: Hard to code might mean hard to use

TurboTax® takes something nearly impossible to comprehend, and makes it approachable to the masses. Complexity is shifted from the user to the code.

However, not always is this shift a zero-sum game. Sometimes, overly complex logic is just a sign that the function of the application is confusing. Complex code supporting a complex interface? How about we don't.

Confusion in the elevator

Back in Lesson #23 ("Write Code as a Last Resort"), I mentioned one of my favorite anecdotes — the story of the poorly-performing New York City elevators. Let's propose a similar problem. Instead of the tenants complaining about waiting at the elevator lobby, suppose they're now bickering about how long it takes to let each person off the elevator.

"Listen to the young psychologist! Put mirrors *in* the elevator!" you cry.

Yes, that might do the trick. But, that approach seems too simple. Instead, let's try to change the way the elevator lets people off the elevator. An elevator typically goes in one direction all the way until there aren't anymore requested floors in that direction. Is there a better way than the common algorithm?

"No, I must have missed that."

Imagine you're part of a team of engineers attempting to build better software to control an elevator for a 50-story high-rise. The elevator can monitor what floor people enter and which floor button they press. Your manager (The Idea Guy) walks in with one, simple commandment:

> "Build the elevator so that people collectively spend the least amount of time on it!" — The Idea Guy.

A seemingly simple goal, your team starts running through the scenarios. Suppose John steps in from the ground floor and pushes the button to go to his penthouse on floor 50. The elevator goes up but stops at Floor 8. In comes Steve, the UPS delivery guy, with a large brown box. He has to deliver the package to Floor 5, so, he quickly presses "5".

What should the elevator do? Should the elevator stop at 5 first, because it's closer, or stop at 5 second because it's already heading north? A typical elevator keeps going up.

But, this isn't a typical elevator. This new state-of-the-art elevator will go in whatever direction gets everyone off the fastest.

Using this metric as our end goal, the engineers agree that the elevator should come back down to floor 5 first. If the elevator heads back down to 5, John and Steve will pass a total of 59 floors:

John: 8 up + 3 down + 45 up = 56 floors passed.

Steve: 3 down = 3 floors passed.

Total floors spent by John and Steve: **59**.

Compare this to the alternative. If the elevator, instead, keeps going up, before coming back down to floor 5, they'd collectively pass more than twice that many:

John: 50 up = 50 floors passed.

Steve: 42 up + 45 down = 87 floors passed.

Total floors spent by John and Steve: **137**.

Now, let's tweak the scenario a bit. Suppose Steve, the UPS delivery guy, had entered the elevator at floor 30, instead of floor 8. Here's what happens if the elevator comes back down first:

John: 30 up + 25 down + 45 up = 100 floors passed.

Steve: 25 down = 25 floors passed.

Total floors spent by John and Steve: **125**.

And if, instead, the elevator kept going up to John's penthouse before coming back down:

John: 50 up = 50 floors passed.

Steve: 20 up + 45 down = 65 floors passed.

Total floors spent by John and Steve: **115**.

In this scenario, we save 10 "man" floors if the elevator goes up! So, depending on when Steve gets on, the elevator may decide to continue its ascent or descend first before re-ascending.

Programming for the optimal elevator ride for two people is pretty benign.

But, suppose a third person, Samantha, enters the elevator. Now, we need to calculate six scenarios. Think of the problem more simply as, what order should the elevator deliver the three passengers?

- Case 1: John, Steve, Samantha
- Case 2: John, Samantha, Steve
- Case 3: Steve, John, Samantha
- Case 4: Steve, Samantha, John
- Case 5: Samantha, Steve, John
- Case 6: Samantha, John, Steve

In fact, the number of scenarios that need to be tested is just the factorial of the number of people on the elevator at a given time:

```
2 people = 2! = 2 comparisons
3 people = 3! = 6 comparisons
4 people = 4! = 24 comparisons
8 people = 8! = 40,320 comparisons
```

Once we get past just a few people, the number of cases to test becomes impractical.

But, that's only *one* aspect of the complexity problem. People are getting on and off the elevator at different times. Each time a new person enters the elevator, we would need to keep track of how many floors the existing passengers have already passed before attempting the new set of calculations.

In other words, we couldn't sufficiently deduce the path the elevator has already taken just by looking at who's currently on-board. If Mike enters on Floor 25, then Sanjay enters at Floor 35, did the elevator travel 10 floors between Mike's entrance and Sanjay's, or did it go back to Floor 21 first to drop off Samantha?

At more than two elevator-riders, we'd *also* need to track when people leave the elevator so we can omit them from future calculations.

And, what happens when someone forgets to hit their floor number, or hits the wrong floor number and then presses a button during mid-ascent? Does our software recalculate and potentially shift gears in mid-flight?

If you are new to programming, this is, sadly, not an exaggerated example of how complex a seemingly simple goal (like getting people off an elevator as quickly as possible) can be.

Complexity with little pay-off

Suppose, somehow, you've built the perfect system. Your team has managed to write code in such a way that everyone collectively leaves the elevator in the shortest amount of time — calculating thousands of scenarios in a split-second, taking into account everyone's already-taken-path. It's a true feat of technology! But, how are John, Steve, and the rest of the gang faring?

Code is great at the tediously automatable. And, this is certainly a case of the extraordinarily tedious. But, humans aren't good at it. John and Steve are at the complete mercy of the elevator, not knowing which general direction they'll be going when the next person gets on, nor why.

Here we have a case where *hard to code means hard to use*. Complexity, in this case, hurts *both* ways. By the time even a fourth person gets on the elevator, there are just too many scenarios for a human to know what route the elevator wants to take. Even when the elevator is accomplishing the goal of "get everyone off in the fewest amount of collectively traveled floors," the people inside are left wondering when it's their turn.

They might well prefer a couple of simpler solutions like:

- The elevator's path goes in the order each floor was selected
- The elevator goes all the way up first before coming all the way down

Sure, we might not reach the optimal path, but, by favoring a simpler solution, the people in the elevator have better knowledge of what's going on from the outside. And, sometimes, that's more important than anything else.

When details become egregiously hard to code, it *may* be a smell that the actual function of the system is difficult to understand. And, while you may pat yourself on the back that you've programmed something inherently complex, others are punching you *in* the back.

#39: Design patterns and the danger of anticipation

Another smell of complexity arises when we think too far ahead in our code. There's a price to pay for being too cute or too cerebral about the actual thing you're trying to build. A classic case — implementing a design pattern too early.

Don't get me wrong. Design patterns are wonderful things. When a common programming approach happens over and over again, we get excited. We've all experienced it – that sense your code could be doing something *greater* than just solving the concrete task-at-hand. That feeling you've solved a problem you didn't even realize you were working on.

When you've had this feeling a few times, successfully architecting your code to more abstract patterns, it's easy to feel invincible. You work like a crime dog, sniffing out any small sign or clue, any hint, even false ones, that another abstraction lives above your straightforward piece of code.

But, very quickly, your sixth sense can come back to bite you where it really hurts. Most of us have heard, or experienced, the horror stories of "architecting yourself into a corner." It's where you've taken an approach to solve a problem in a more abstract way.

For example, you're working on an intranet for Burgeoning Web Company. They're small – they only have two departments – IT and sales. They want the ability to calculate anticipated bonuses for their employees based on a bunch of employee parameters. But, each department wants to emphasize different things.

The IT department only wants to give bonuses to employees that have stuck around for 5 years, and as a percentage of their current salary. The sales department wants to give everyone a base bonus, plus a standard incremental bonus for each year they've worked at Burgeoning Web Company. The execs are generous folk.

You start coding. You've built out an `Employee` class that will contain all the information you need to calculate an employee's bonus. You then write a simple function, which, for now, contains one simple conditional statement to return a given employee's anticipated bonus.

```
public decimal GetBonusForEmployee(Employee employee)
{
    if (employee.department == Departments.IT)
    {
        // Calculate bonus the "IT" way
        if (employee.Years >= 5)
        {
            return .1 * employee.Salary;
        }

        return 0;
    }
    else
    {
        // Calculate bonus the "Sales" way
        return 1000 + 500 * employee.Years;
    }
}
```

You write a little tool to load all employees into a collection of `Employee` objects, and then apply the method above to each. Your work is done. Beer time.

But, your mind begins to think ahead to the other possibilities that lie ahead when Burgeoning Web Company really begins to...burgeon. What do I do when the third or fourth department comes along? Swap the conditional for a switch statement! But, why wait now. Let's anticipate it now so we're ready for the future:

```
public decimal getBonusForEmployee(Employee employee)
{
    switch(employee.department)
    {
        case Departments.IT:
            // Calculate bonus the "IT" way
            if (employee.Years >= 5)
            {
                return .1 * employee.Salary;
            }

            return 0;

        case Departments.SALES:

            //Calculate bonus the "Sales" way
            return 1000 + 500 * employee.Years;
    }
}
```

Beautifully done! The switch statement is a safe, elegant, anticipatory move. It explicitly identifies the Sales department instead of relegating it to the `else` statement. When Marketing comes along, you know just where it fits. The change from conditional to switch statement is a good one.

This small refactoring makes sense. Our code is now more explicit. It's easier to scan. Another developer could come in and pick it up right away.

Is it good enough for now?

Sensing your higher calling, you decide to do more. What happens when two departments becomes....10? There's HR, Legal, Production, Accounting, the Janitorial staff. *It's gonna happen*. The switch statement is going to eventually get unwieldy. It will be tainted with complex calculations that just have no business lying there, exposed so nakedly at the surface of the bonus calculation method.

You rifle through your Patterns book (I do recommend this one) and, voila — the strategy pattern! Move all those one-off bonus calculations into individual strategy classes (e.g. `ITBonusCalculationStrategy` and `SalesBonusCalculationStrategy`) that each implement a Bonus Calculation Strategy interface (`IBonusCalculationStrategy`). This interface will require each implementing class to define a `CalculateBonus()` method.

Once that's done, you modify the `Employee` class to contain a concrete strategy instance, and create one new public method that will return the employee's bonus.

With the strategy pattern, you can now remove the `getBonusForEmployee()` method altogether. The calculation of an employee's bonus can now live in the class itself. Now, all those nasty algorithms lie elegantly in the soft cushiony pillows of individual implementations of the `IBonusCalculationStrategy` interface.

And, since you've gone this far, you decide to embellish your code further. You abstract the creation of employees into a factory pattern. This way, you can create department-specific employee creator classes that can assign an employee's corresponding bonus strategy.

You've completely removed the conditional switch on departments (it's taken care of in the employee creator classes), and the nasty calculation logic (it's buried in department-specific strategy classes). Wonderful!

Once department #15 comes along, this architecture will be a site to see.

Weeks and months go by. The winter hits, and times are tough for Burgeoning Company. No new departments. Meanwhile, the bonus logic has changed. You go back in, stepping through code, wondering, where the #$&*#&$ is my simple logic? Ah yes. It's been strategized and factory-ized.

Another couple months go by. Burgeoning Company calls and says, they've fired the sales team, and it'll just be the CEO working the phone with his team of developers. Wanting to keep developer morale, the CEO still wants to offer bonuses, but, now based solely on seniority.

It's time to shed a tear. You've placed all your bets on department-specific bonus rules. It was, by all accounts, a safe one a year ago. You built the walls, ladders, and slides to account for every possible department bonus for the next hundred years. And here, the entire refactoring has gone to waste. It's not just over-architected clutter, its spoiled clutter. You unearth your strategy classes, remove the factory methods, and submissively decide that, a potentially-nasty-but-currently-quite-alright conditional statement will do just fine for now.

Patterns are wonderful concepts. But, they should be implmented with the utmost caution. Anticipating logic in the future, more often than not, leads to unintended complexity.

If there is a golden rule, it's that an application shouldn't be forced into a well-documented design pattern or set of patterns. Rather, a design pattern (or set of them) should be implemented as fully as needed to fit the desired tasks of the application, and the most likely scenarios for the near future.

When you study a design pattern, read it as a general approach to solving a particular problem, not as a strict, rigid solution to a problem. Patterns all have pros and cons. While patterns make some tasks more elegant to perform, you always lose something else. And, since most of today's web applications are constantly changing based on new customer or client requirements, finding the "perfect" set of patterns from the get-go is more dream than reality.

Does this mean you shouldn't anticipate at all? No. The same problems manifest when you don't pay any attention to where your architecture is headed.

Examine any code-base whose authors decided not to make simple refactorings. The problems reveal themselves quickly. Variables are scoped at the wrong level, or even worse, accessible globally with some bizarre naming convention to ensure they'll always be unique. Conditional logic reads more like the terms on a Terms of Use page – a bunch of unrelated truths stitched together with *ands* and *ors* that have collectively lost all meaning to the next unlucky soul that has to modify it.

You'll find this often in legacy code in larger companies – code that's been handed down from generation-to-generation of developers that came into it without much passion and came out of it with even less. Method signatures become unruly, and method calls look like the code itself isn't sure what you're doing:

```
calculateBonusesForTeam(.02, 155000, null, 0, 0, null, new
Employee(), null, null, false, true);
```

Over time, unconsidered refactorings get expensive. Maintenance becomes asymptotically slower. Forget big changes — even small changes, the ones you always take for granted, might collapse a code base that's long since forgotten any basic set of good habits. This kind of code gels into a sticky molasses.

Back to the Burgeoning Company story, at some point, refactoring the bonus calculations into a strategy pattern might have made sense. Moving employee creation methods into a factory class might have been useful. *It just wasn't at the time.*

Over-architect too early in the development life-cycle, and you're left with a hole waiting to be filled. Under-architect and you're left without any option or motivation to evolve your software any further.

> "The hole and the patch should be commensurate." *-Thomas Jefferson to James Madison*

Perhaps, Jefferson was channeling his conversation with Benjamin Franklin a few years earlier.

Refactoring to patterns is incredibly powerful. But, it really depends on when. I highly recommend Joshua Kerievsky's book, aptly named "Refactoring to Patterns." It discusses when, why, and how to refactor code into (and also, out of) common software patterns far better than I could ever do.

Anticipate, but anticipate cautiously. Whether it's just a small change or a large pattern shift, know what your gaining and losing each time you decide to refactor.

#40: Programming Cadence

So, how do you manage anticipating too early to change vs. reacting too late to change?

Consider software development like you're driving a stick. As you start, you're on first gear – coding away at a steady pace. The more you code, the less efficient you become. At some point, you have to shift up a gear.

Shifting up a gear, in programming terms, is cleaning up your code — taking a step back to refactor, abstract, or implement a pattern. It means taking time to consider how to change your habits at a particular point in the development process. Doing this does not mean you've made a mistake (as some might argue). It's natural and necessary.

You have to shift in code just as you have to when you're driving. But, if you do it too soon, you'll spend a lot of time trying to regain your speed. Do it too late (or not at all), and you're code will burn out. Knowing when to shift is essential. It keeps the development process running as efficiently as possible. You don't shift just to shift... you have to do it when it's right. You have to find your programming cadence.

"I was going so fast I figured it was better to keep my eyes on the road instead of the speedometer.

There is no set amount of gears in software development. We can choose to have just 5 or 500 gears in our programming cadence. This depends on the complexity and scale of the project as well as your own willingness to shift gears. For more complex projects, allowing yourself more gears means we can shift a lot more often. It means that if we shift a little too early, it won't take us too long to get back to speed. A little too late, and we haven't experienced too much burnout. For smaller ones, just a few gear shifts will do.

Develop your complexity nose

In the end, software complexity is necessary. It's the debt paid for more functionality. But, know when complexity feels right and when complexity feels wrong. Listen to your sixth sense when it tells you that, this time, complexity makes things worse on everyone. It's why what we do is much more art than it is science.

Teaching

If there's anything I want you to take away from this book, it is to *learn* how to *teach*. Programmers who can teach well will become better programmers. That's why this is the most important chapter in the book.

There are plenty of experts in the world, but far fewer great teachers. You can be a Rock Star programmer, but if you can't teach, you're suffocating your value in this business. Teaching isn't just *regurgitating* what you know. It's an art that requires you to step outside of your own knowledge and into the mind of someone who's learning something new. Simply knowing something is just one ingredient in the recipe for a successful teacher. Too often, we mistake someone with knowledge for someone who can teach.

Isiah Thomas was a hall-of-fame point guard in the NBA. He was a 12-time NBA all-star and led the Detroit Pistons to two NBA championships as a player. As a head coach, he had a pedestrian record of 187 wins and 223 losses. Sometimes people with great knowledge and skill can't transfer those same traits over to the people they're trying to teach.

Teaching programming concepts to a relative newbie is even that much harder. Learning to code just isn't as transparent as learning to play basketball, cook, write, or sing. It's overly cerebral. There are no obvious indicators whether someone is thinking about a software problem right or not.

But every great programmer should work at becoming an equally astute teacher. It will help you work with passionate people and cultivate other future great programmers.

Teaching also teaches you. The better you can explain to someone what reflection is, how and why you'd use an MVC framework in your web app, the differences between an abstract class and an interface, or when you should use a specific kind of SQL JOIN, the better you'll understand those concepts as well. Teaching makes you better at what you do.

Let's first air the dirty laundry. Every one of us can be a better teacher than we are right now. We need to first recognize when we're not teaching at our best, and then figure out how to go about teaching better.

#41: An expert coder doth not an expert teacher make

At first, it might seem like a great programmer should have the skills to be a great teacher.

After all, coding has many of the same traits as teaching. Underneath the fancy patterns and elegant frameworks, code is just a set of concrete instructions to do something. Even if one, seemingly obvious, detail is left out, you'll know soon enough. Code has an order too. You can't implement a concept before it's been defined yet, just like you couldn't teach someone how to multiply before they understand how to add.

On the other hand, the act of coding isn't like teaching at all. In fact, it promotes habits that are entirely counterproductive to the art of teaching. A really good programmer just might make for an awful teacher.

First, rarely do coders code linearly. You don't start from the top of the page and just work your way down to the end. Along the way, a clear concept in your head turns into a half-truth. Part way through writing a method, you might decide you need to track things in an array. After a few minutes, you'll decide a hashtable works better. If coding against a platform is at all like talking to a student, you'd sound rather unsure of yourself.

Second, coding lets you cheat on the details. We compile our code not because we think we're done, but because we want to find out what we may have missed. You can usually bucket most compiler errors in the "I was just being lazy" category. A missed instantiation here, a data-type mismatch or non-returned value there. I'm a habitual compiler. A compiler is a lazy programmer's best friend. Ditto for unit tests, code-hinting and auto-completion.

All these niceties are great for programming. They give us softly padded walls to bounce our code off of. They let us focus on the hard stuff first and not worry too much about perfection in our code-speak. A good programming platform is simultaneously wiping our chin, correcting our grammar, and telling us what we really mean while we spew out semi-coherent lines of instruction. And, the faster and more efficient we are at coding, the more we rely on the platform to steer us in the right direction.

Teaching a newbie is entirely different. Every missed detail is a lost detail. You can't start your sentences expecting your student to finish them — at least not early on. And unlike a compiler, who invariably will forget your missteps once you correct them, people don't have as much luck separating the wrong details from the right. You may compile your code a dozen times before you finally get it right, but imagine correcting yourself twelve times before your teaching lesson finally makes sense.

An expert programmer does not make an expert teacher.

How do you teach people well? It starts by knowing that what may make you a great programmer will not make you a great teacher.

#42: Prevent the "Curse of Knowledge" from creeping into your teaching

In the popular Chip and Dan Heath book *Made to Stick*, the brothers argue that, once you've become an expert in a particular domain, it is nearly impossible to understand what it feels like to not understand that domain. Think of how you might explain color to a person born without sight or sound to a person born without hearing. They call this the *Curse of Knowledge*.

In a less extreme example, think of a lawyer who can't give you a clear answer to a legal question without all sorts of abstractions and qualifications.

© MARK ANDERSON, ALL RIGHTS RESERVED   WWW.ANDERTOONS.COM

"I think I speak for all of us when I say what in God's name are you talking about?"

Undoubtedly, one of the biggest abusers of the *Curse of Knowledge* is *us*. A great coder would have a hard time trying to imagine not knowing how to code.

That's also why teaching someone else your code after it's been through dozens of iterations is often difficult. It's a LIFO problem. Your knowledge of your code works from today backwards. You remember the last tweak and maybe even the one before that. But you've already forgotten the major overhaul that took place last month. And, you don't even remember what it looked like when you first threw it together. To you, the story of your code is like looking in the rear view mirror.

And, that's what makes retelling the story of your code difficult. The person your teaching would best learn what you've written by experiencing it the way you did – from the start, going forward, up through this present moment.

So, when you're teaching a newbie a seemingly basic concept, or taking your colleague through your own code, understand the *Curse of Knowledge* is in full effect. Teach twice as slow as you think, explain the "obvious" details until it's crystal clear to your pupil.

#43: Teach with obvious examples

Good examples are devoid of abstractions. They are concrete and clearly — almost too obviously — convey the intentions of what you're teaching. They provide good *context*.

For instance, imagine you're teaching a beginning programmer the basics of object-oriented programming. You might start, naturally, with object instantiation. You might decide this is a good example line of code to start the discussion:

```
Object myObject = new Object();
```

For the already-acclimated, this is a ho-hum line of example code. It says, create an instance of an object of type `Object`, called `myObject`. It's known to us that the name `myObject` is just any-old-name we decide to give this newly birthed instance. On the contrary, the constructor `Object()` isn't just named in any old-fashioned way. It's the constructor — it has to be named exactly the same as the name of the class.

But, to someone learning object instantiation for the first time, the line of code looks like this:

```
Object myObject = new Object();
```

First impressions? **Object** is clearly significant. It shows up *three* times in one line of code! It's the name of the class, and, by rule, the name of the constructor. But, the embedded word `Object` in `myObject` has absolutely no significance. We just happen to call the instance `myObject` for lack of a better name. But, for the newbie, its insignificance is *not* obvious.

So, here's a better rewrite:

```
Object mary = new Object();
```

Now, we're getting somewhere. `Object` is still important, but it's clear that it has no implications on the name of the instance of the object.

But, still, for a first-time object-oriented programmer, this is still a little too in-the-weeds. What kind of object is `Object`? Back to the drawing board.

```
Human mary = new Human();
```

Ah yes. Now, `Human` is significant. The relationship between `Human` and `mary` is intuitive, even for someone who doesn't write code for a living. It's a more palatable example of a class you could write. The newbie now envisions everything a human named Mary could do.

But, there's still something hard to understand. If we already say "Human Mary," isn't it obvious that she's a new `Human`? What, exactly, is the point of a constructor?

Here, a conundrum exists. Constructors that don't accept any parameters is fairly common. For the seasoned developer, we're accustomed to working with classes that don't accept any parameters when instantiated. And so, the conventional `new Object()` (or `new Human()`, or `new List<DateTime>()`) feels intuitive.

To the newbie, it seems mindless. Constructors that don't accept parameters, and, even worse, don't do anything in its definition (aside from instantiating the object), baffle the OOP neophyte. So, if you want to ingratiate the masses that are looking up to you for help, make the example constructor accept a parameter (or two).

```
Human mary = new Human("female", 45);
```

Ask the newbie if he can create another human named Doug, who just got his driver's license, and there's a good chance they can figure it out.

The moral of the story? When showcasing examples, be *explicit*. Be more explicit than you think you have to be – we went through four iterations to get to a better example for something as basic as object instantiation. Cut out the generalities, generic names, and theory for something tangible and obvious.

#44: In the beginning, go for blanket statements over the truth

When you teach something new, never start with the notion that everything you're going to say from here on out is 100% correct. Teaching a concept perfectly from the get-go is neither practical nor efficient. Any advanced concept is inherently difficult to understand. That's what makes it advanced. It's full of nuances, exceptions, and special cases that don't always fit into a nicely-wrapped unified theory.

But, when you learn something new, that nicely-wrapped set of facts is exactly what you want. We crave the hard and fast truths, whether they really exist or not, because they are the scaffolds that help you build your knowledge of any subject.

So, when you're the expert, let go of the intricate details of your domain at first. Let go of the "except when" and "but not if" cases – they just *aren't that important right now*. In the beginning, reveal the handful of rules that will get you most of the way to understanding a concept well.

When you pare down a complex topic into a less-than-perfect set of rules, it gives someone new a chance to build a solid foundation of understanding in their mind. When you teach subtleties and exceptions too early, before people have had a chance to soak in the general concepts, their learning becomes fragmented. Piecing together the whole story at once becomes difficult. It's hard to digest both rules and exceptions to those rules at the same time.

And so what if the understanding isn't 100% correct immediately? A solid foundation of understanding is motivating. And motivation will get them to an advanced level *faster*.

#45: Encourage autonomous thought

Teach the "rules" in the beginning as if they were set by law: Death by a thousand cuts if broken. It provides a structured starting point for a novice. And, it's probably not a bad thing to instill a little fear...

This is also what the *Dreyfus model of skill acquisiton* preaches for novices as well. The rules part of it, not the punishment part. The Dreyfus model is, put simply, a model of how students learn. It was proposed in 1980 by a couple of PhD brothers (Stuart and Hubert Dreyfus) in their research at UC Berkeley.

According to the Dreyfuses, at the novice level, students follow rules without context. There's no questioning. Why do you want to normalize a relational database? Why do you want complete separation of structure and presentation in your HTML markup? Why do you want to adhere to the DRY principle in software development? Because that's what you've been taught, son.

But, when they start heading toward the expert level, those stone-etched laws are meant to be broken. A proficient coder treats those rules as guidelines — the orange plastic cones on the highway. For instance, there actually *is* a time and place to denormalize a database (e.g. when the database is a "read mostly" one, like an OLAP cube). But, a novice shouldn't know about the benefits of denormalization in the beginning until she's fully grasped the benefits of normalization. In order to understand the cracks in the foundation, you need to intimately understand the foundation itself.

When you begin to master a subject like programming, you stop using rules to guide your work. That thing that just comes naturally, takes over. There is no more recipe, there's just intuition.

And, that is what your student needs to know. One day, on the road to becoming as good as *you*, your student has to cross that intersection where he questions the rules that you taught him in the first place. One day, some of those beginner rules won't fit with, what your student thinks is a better approach to the problem. *Encourage that type of autonomous thought*.

In Arthur J Riel's *Object-Oriented Design Heuristics*, a book of metrics for good object-oriented design, he states:

> I refer to these 60 guidelines as "heuristics," or rules of thumb. They are not hard and fast rules that must be followed under penalty of heresy. Instead, they should be thought of as a series of warning bells that will ring when violated. The warning should be examined, and if warranted, a change should be enacted to remove the violation of the heuristic. It is perfectly valid to datte that the heuristic does not apply in a given example for one reason or another. In fact, in many cases, two heruistics will be at odds with one another in a particular area of an object-oriented design. The developer is required to decide which heuristic plays the more important role.

In other words, break the rules when it makes sense to. That's the big leap from novice to expert.

Clients

In this business, clients, stakeholders, and customers are our lifeline. Without them, what we do amounts to nothing more than a hobby.

But, quite often, the working relationship between us and them feels more like Ali v. Frazier than Penn and Teller. In an ideal world, our client is giving us backrubs and feeding us scoops of vanilla ice cream all while dabbing the corners of our mouths as we labor over their application. In an ideal world, the client knows the agony we sometimes go through to fit nascent ideas into real code.

The harsh truth — clients rarely see what pains we go through to bend to changing requirements. Customers only think about that one new feature they want – the one that, in their eyes, involves "just changing this one little thing" when there's so much more too it. Stakeholders only care about the bottom line.

#46: Difficult clients are not uniquely our problem

But, this problem is not uniquely ours. In fact, we have it easy compared to some others.

When an architect designs a house, the homeowner sees only what's easily visible. He sees the obvious qualities of the home – the granite countertops, hardwood floors, and crown molding – not the subtle nature of a floor plan that the architect may have anguished over for months.

When a chef cooks a meal that's off by a salt grain, a picky critic delights in sending it back. The chef's work, completely nullified, is tossed away. I once saw a rather pretentious family's entire set of orders sent back to the kitchen because their rather spoiled teenage son lost his appetite over a hairy bug in his meal. Gone was the work of an entire staff of laboring cooks because an ignorant client mistook a fibrous piece of ginger for a cockroach.

We've all been the client at some point. Clients rarely appreciate the delicate, intricate, advanced thought that goes into the products they consume. And, perhaps that is the cruel irony of it all. When I hire a plumber to fix the low water pressure in my shower, I simply want it fixed. I don't care if it's due to the main line, the flow constrictor or a clogged shower head. A cheap bill and some skin numbing showers will do just fine, thank you.

In just the same way, when we build software for the consumer or client, the people we work with likely can't tell that you programmed the application with such elegance and cunningness.

What does this mean? No one you work for *cares* about your code – at least not immediately. It also means that, when they want to change your software, they haven't the slightest clue whether that change in code is easy, hard, impossible, or annoying. They don't know if their one "would-be-nice-to-have" request is really a "would-be-awful-to-build" one as well.

It's frustrating, yes. But, don't pity yourself. You are not alone.

#47: Teach clients what programming actually is

So how can we get clients to appreciate our labors more?

Sometimes, it starts with teaching clients how we do what we do. This is especially true when you're working with someone that's never worked on their own web application before. Even the most completely obvious things to us are not common knowledge to everyone else. I've learned this lesson many times in my career.

Years ago, I took on a freelance gig with a startup client who wanted to build a recommender system. It would offer the cheapest prices on bulk liquor purchases for bars and restaurants. Sounded easy enough. I was 22, a relative programming newcomer, and naïve about how quickly complexity happens.

In my head, I envisioned I'd get a bunch of data — a list of alcohol brands, distributors, addresses, and costs. The user would request something, and I'd write some code that would go find the cheapest price in the system that fit the search parameters. My client and I met over coffee, and went our merry ways. He'd consolidate the data and I'd start building this rather *Simple, Elegant Example of Exquisitely Crafted Software*.

A few days later, we met for lunch over an Excel spreadsheet. I opened it up, and it looked a bit hairy. It wasn't the simple 5 column table I was expecting. "Well, the prices change based on how much quantity of the product you buy," my client said. There were two other columns — a maximum and minimum quantity of alcohol that had to be purchased to obtain a specific price.

Fair enough. After all, I thought, that is the whole point of bulk purchasing. Back to the drawing board for a few slight tweaks, and off we go.

The following day, I had the solution. I added a couple of additional quantity range fields (`BeginRange` and `EndRange`) in my database. The application would accept a quantity value and I could write a simple SQL query on the table where `@quantity >= BeginRange` and `@quantity <= EndRange`. It would then search the database based on these additional parameters as well. Newton, you've done it again! The system worked brilliantly.

But, when we met again, my client looked puzzled. My code was beautiful, but there was something missing in the *behavior* of the system. As he played with the software, he noticed, there were a few more levers missing.

As I would soon find out, in real life, you get discounts for coupling similar products together. The concept of bulk didn't just live per product, but buying `X` amount of whiskey afforded you a discount on `Y` amount of vermouth (Manhattan anyone?). And, the discounts differed based on how much whiskey you purchased. And, perhaps we ought to throw in a few free jugs of Maraschino cherries as well.

From his end, the behavior of the system seemed off. In his experience, these deals, found by calling real human distributors directly, were commonplace. From my end, I didn't have the data to *deduce* any of this. And, even if we could get all the data I needed, I'd need a lot more time to figure out exactly how to organize it. Did I need to build some separate table of dependencies to handle discounted products based on the purchase of another product? Did I need to build a "common mixed drink" concept so the app could intuit what drinks you could make out of your purchases in order to offer other discounts? Was there more madness I would find out after this?

It dawned on me one day that I was not, in fact, building a concise, well-defined system to mine spreadsheets of data to harvest the singular, right answer. Instead, I was building Larry^{TM}, the alcohol distribution manager. I was trying to account for decisions that were not easily deducible. Larry^{TM} gives offers based on his relationship to his customer, his own 40 years of experience, and a general hunch or two. He knows what will drive customers back to his company as opposed to the other hundred distributors he competes with.

Why wasn't I told all of this new information in the beginning? Was he just hiding it from my prying ears or did he just not figure it was important at the time?

And here, in full daylight, arose the fundamental misconception that lots of non-techies have of software, and the web, and ones and zeroes. As web developers, we are primarily organizers of logic and information. Our jobs are mainly about pushing, pulling, manipulating and displaying data. Most of us aren't in the business of artificial intelligence. We can't easily write programs that recommend or guess. Even "recommending" or "guessing" is a product of some set of defined, describable logic. Yet, sometimes, that's exactly how the outside world perceives this work — some kind of black magic.

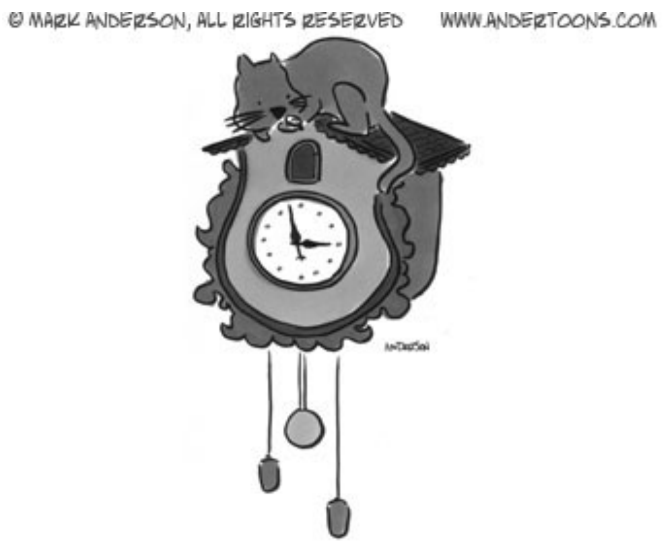
When I started to explain this difference to my client, he said he'd get back to me. Years later, my code is still sitting on my Dell Inspiron 2600 laptop — affectionately known as my *Dust Collector / Paper Weight*.

Looking back, the client-developer relationship became clearer to me. From my client's point of view, the web, software, databases – all these "technical" things were a foggy, mysterious haze of magic. There was some part of him that believed that code could magically take care of a few loose ends in logic, even if these small loose ends were the things that made this type of application really complex to build.

In any project, there will be a certain amount of unknown. This is the nature of our work. Rarely is an idea completely flushed out before we transition into development. And, even when everyone thinks it is, it really isn't. Weirdness has a way of dodging our minds when we're still talking about software; it tends to surreptitiously unveil itself only when we start building.

Maybe that's why we build a love/hate relationship with our clients and customers. What is seemingly so obvious to those of us toiling "in the box" is sometimes lost to those on the outside. Because, it's in the box where the idea must finally be realized. It's in the box where you know, full well, whether you have something concrete or you don't. It's in the box where the real struggle occurs.

That's why there's frustration from the outside as well. From their vantage point, they've thrown you lots of information and requirements and detail. Certainly, enough to get you started. They've let you know what they want, and they are waiting...



And, when, days or weeks or months later, they see something that isn't quite what they hoped for, they too are somewhat deflated. The box isn't as magical as they had thought.

So, when these times arise, take your customer/client/stakeholder/user/dreamer into the box. Take them there early if need be. Show them what you've been working on. Step them through the actual code if you must. Get them involved in thinking through the questions that naturally come up to you when you're in the midst of programming.

Show them your kitchen

The box is our kitchen. When the customer is yelling at you, bring them in. Show them the line of piled up orders, the furiously busy staff, and the half-seared steak that fell on the floor. Explain to them why customizing their order isn't simple — that, even though we have ice cream and tomato basil soup, ice cream floating on top of hot soup simply doesn't work.

You'd be amazed at the respect and admiration you'll gain from the people you're working for, when you let them peek at what's going on under the hood. A simple "idea," when manifested into an actual "thing," is full of little details that just never came up before the code arrived. The very details that you've now taken for granted, weeks into development, are still new to those around you.

Get them in early so you can demystify what we do — it is not soothsaying, prophecy, or black magic. It is simply, programming.

#48: Define the goals of your application, seriously.

Working with clients isn't easy. The disconnect between you — programmer/designer/messiah — and them — unreasonable dictator-at-times, will always be there.

But, not all clients are difficult. The better ones, the ones you want to keep for the entirety of your consulting career, the ones you hope will always have new projects and new ideas ready-to-serve you at a moment's notice, all seem to have one thing in common.

Great clients put the *application* above themselves. When the application is the **most important part of the project**, everything falls into place. Each feature decision can be scrutinized by simply asking the question, "Does this make the application better?" Feelings, both your's and the client's, aren't what dictate the outcome. When the product is not put at the forefront, clients will justify a feature request by other measures.

- "...it's something cool I saw on another site."
- "...because it's 1996, and everyone's using `<blink>` and a counter!"
- "...because it's 2005, and everyone's using RSS."
- "...because it's 2009, and everyone's got a Facebook and Twitter badge."
- "...because my usability book told me that content below the fold never gets read!"
- "...because our CEO loves pink!"

At the beginning of the client relationship, establish the goals of the application. Decide, together, what the end-product is hoping to achieve — and write it down. Doing this turns ammo like "cool" and "cutting-edge" into blank bullets. Establishing the goals lets you say "no" with more conviction.

#49: Make your work interesting

One of the biggest misconceptions non-programmers have with programming is that it's mechanistic, algorithmic, downright robotic. But we know it's not. Programming is as much art as it is science. We are passionate about our work in much the same way as artists, musicians, and chefs are passionate about theirs.

It's our job to make our work *interesting* — to promote web development as a craft, not as mere wage work. By doing so, you can change your relationship with your client. It stops being just service work. Engaging your client with work that's interesting will make them appreciate and respect your work more. When your client respects you, they listen to you. They place their trust in you. They're more likely to give you the benefit of the doubt when your opinion differs from their's.

How do people in other industries make their work interesting to the consumer? It's easy to talk about Jamie Oliver, the "Naked Chef" who has popularized British cuisine, healthy eating, and generally mashing everything together with your hands. It's easy to talk about a musician like Jack White who, in Davis Guggenheim's rock guitar documentary, "It Might Get Loud," talks of his blood-stained guitars. It's interesting because cooking and music tickle the senses, and people will listen to famous people talk about their craft.

But, let me vouch for some lesser known names.

Lou Manfredini is "Mr. Fix-It." He's an exuberant handyman who knows *everything* about fixing up a home. On his weekly Chicago radio show, he helps homeowners fix every type of problem. Whether it's installing a new HVAC unit, combatting a leaky roof, or sealing a deck, Manfredini has a recommendation and opinion on everything. He's equally passionate about the type of paint you should use in your kid's room as he is about getting vermin out of your basement.

Jeffrey Ruhalter is a 4th generation Master Butcher in Manhattan. Watch him butcher a pig or trim a piece of dry-aged steak. His eccentric style of communication oozes passion. You can't help but find his work *interesting* — unless porterhouse ain't your thing.

Manfredini and Ruhalter prove that you don't have to be in Hollywood to make your work interesting. They are heroes in otherwise unsexy vocations. We can do the same. Plus, I'd like to think programming merits more interest than leaky-sink fixing.

#50: Be forgiving and personable

Passionate programmers have a penchant for being irritable. Providing the absolute last line of defense between nascent idea and functional reality can be frustrating. The entire thought-chain prior to the code we write often lives in diagrams, functional specs, wireframes and the brains of those who claim to be the "idea guys." Yet, we're the ones faced with the daunting, often under-appreciated task of getting the damn thing to actually work. Bugs only live in code, never in napkin drawings.

In years past, software was this unapproachable, often command-line driven thing that only geeks, dorks, and nerds used. It was built by even geekier geeks, dorkier dorks, and nerdier nerds. Back then, it may have been OK to play the stereotypical role of the anti-social, generally off-putting curmudgeon.

But today, software is *everywhere* and it's for *everybody*. Our clients are everyday people, not just other software guys. They use the products of our labor like they use furniture. It's just there. The line between when someone's using software and when someone isn't is quite blurry. Our work is mainstream these days. It means we need to button-up how we work with the people we're building software for.

And so, when one of our clients, one that isn't tuned into how we work, asks if they can just add another feature here that, in reality, breaks an already-agreed-upon assumption and undermines the entire architecture of your application, it's all too easy to quickly retaliate, feud, yell, or threaten.

Instead, we need to be forgiving. Understand the view from above the hood while we're so entrenched in working under it. If a client's request isn't practical, explain to them why. Give them an example scenario that opens up that "whole new bag of worms." Offer an alternative solution to solve the problem they're having.

And, make it a habit to talk to them in person. Hear their real voice and let them hear yours. Pick up the phone and call them instead of just emailing. You'll be surprised at how far a compassionate-sounding voice can go in getting things set your way.

Pride

The other day, I read an op-ed piece in the New York Times called "The Healing Power of Construction Work." In it, a carpenter from Middle America talks about how an unusual number of his hired construction workers were also in trouble with the law. Some of his best crafstmen were drug addicts, alcoholics, felons, and even a paroled murderer in the mix.

He wasn't suggesting that construction work attracted violent people, but instead, it provided some healing escape from their otherwise troubled lives.

There is a calm when you work with your hands, and a cerebral quality about using raw materials to build something. His hired hands didn't treat construction work as merely a job — it was an escape from reality and a chance to do something really *well*.

Though, by all accounts, construction work is a blue-collar job, there is a primal reward to it — the satisfaction of creating something that didn't exist before. Construction is something that anyone physically able can do, if they learn, work hard, and care about the product — even for those who otherwise have not found success in other areas of their lives.

As I read the piece, it struck me that I approach programming in the same way. I am not a convicted felon, nor, do I personally know any fellow programmers who happen to be running from the law. But, I do know many that, whether they'd like to admit it or not, coding can be a soothing escape from reality. Programming gives you that same joy of building something out of nothing.

Most programmers I know don't even care that much about *what* it is they build or who they're building it for. So long as they are solving an interesting problem and so long as there is an opportunity to build something elegantly. The mental exercise of dissecting a problem and solving it masterfully is the mental drug that keeps programmers addicted.

We build and design software because, whether found near the surface, or buried deep into our souls, we actually *love* doing it. The best programmers I know toil over every small, sometimes insignificant, development decision. Because, just like those construction workers, it isn't just about writing code, it's about writing code *well*.

Those who love this job aren't in it just for the money. There are easier ways to make money. This vocation was completely of our own choosing.

We have a marketing problem

The problem? Few others outside our relatively small tribe realize how rewarding software development can be. Even many among us don't fully realize it. That's why I cringe when admitting I am a web-software-application-developer-guy. It all reeks of someone sufficiently intelligent, just settling for something. Perhaps, it has a lot to do with the nature of our work.

At our worst, we are disgruntled and unhappy, hopping from one job to the next. It's no different than any other industry. But, the stigma comes because of how we exist when we're most passionate. As I mentioned, most of us don't care about what we're building, so long as we're *solving a challenging problem elegantly*.

And to do just that, we live inside our heads a bit more than most. We stare at screens in a trance – typing, deleting, typing, and pondering in one seamless, delicate dance. We look out the window, seemingly longingly, when in reality, we aren't seeing anything but pseudo-code running through our heads. We are not smiling, talking or seeking any sort of reciprocation. We simply want to be left alone, to our own thoughts, as the world does whatever it does around us. This is the content, happy, passionate developer who's completely escaped the world around her.

When we are least excited, we are also quiet. We aren't smiling, and we don't want to be talked to. The only difference is, we type with less vigor and look out the window *noticing* the world around us and wanting to get out. When encouraged, we will, in fact, sigh, bang a fist, and mutter how much we disagree with the work we are doing. The disgruntled software developer looks just like the most fulfilled one, just with a noticeable sigh.

And so, we have a *marketing problem*. The rest of the world sees programmers as a breed of recluse, anti-social, unhappy headphone geeks rather than, what we really are – passionate craftsmen, thinkers, and game changers. Why is this?

Lessons from the cooking industry

Take the cooking industry. Emeril Lagasse, Bobby Flay, Mario Batali, and Gordon Ramsay are exuberant (sometimes annoyingly so) chefs whose passion oozes from their pores. Their passion reaches, not just other chefs, but the *masses*. Our (less famous) contemporaries don't have that same kind of global appeal. There are no programmer celebrities whose reach stretches beyond the engaged eyes of other programmers.

At first, you may think it's because people generally want to cook more than they'd want to program. But, I can assure you that, while I routinely salivate when a chef prepares a *horseradish-crusted salmon with braised greens and smashed new potatoes*, I will not be making one for myself anytime soon. The cooking industry has found a way to sell its craft to everyone – even if many of us will never deglaze a pan in our lives.

Maybe it's because we just like to eat. Food is visually stimulating. Watching someone prepare a meal stirs our most primal emotions. Some call it *food porn*. But, the modern-day buzz about food hasn't always been like this. After all, cooking shows have been around for decades. Most of us have heard of Julia Child, but ever hear of Justin Wilson, Jeff Smith, or Graham Kerr? They had their own cooking shows for years, but lived in a far less cooking-crazed society. They never gained the omnipresent appeal their modern-day contemporaries have today. What shifted?

In the olden days, cooking shows felt like being in your grandmother's kitchen. A couple of cameras, some pedantic talk about a quarter cup of this and a teaspoon of that. Cooking shows were made for people that wanted to…*cook*. They never stretched farther beyond their audience. Cooking was just about…. cooking. Today's shows today slice from a completely different angle.

First, they've emphasized the detail. There's the close-up shot, then the closer-up – the one where you can see the marbling of a thick slab of tenderloin while double-checking that the chef's fingernails are clean. HD television has helped the food industry, as much as any other industry, sell its stuff. In bygone days, a steak was just a steak. Now, it's about the intricate marbling, flowing juices, and grill marks. The *detail* is where the appeal lives.

Second, today's shows make cooking approachable to everyone. Long-gone are the days where TV cooking was just about following a recipe. Today's shows emphasize simplicity. Everyone can do it. 30-minute meals, $5 dishes, and a good time with friends. Cooking is feasible and entertaining.

Chefs play up their food like royalty. Passion lives in their description of ingredients and flavors, even if only using non-descript adjectives like "fresh," "flavorful," or "zesty." Nowadays, chefs always taste their own food (usually at the climactic end of the show), exalting what magic it's doing to their tastebuds in sensationalized "mmms."

Even further, *the bad stuff sells*. Go to a real restaurant kitchen on a Friday night and see the real story. Screaming, sweat, dropped food, and a general disaster-waiting-to-happen. *System D* in Anthony Bourdain's *The Nasty Bits* tells a far different, if even slightly embellished, story from the pristine world of cooking that's sometimes portrayed on television. And, it's a New York Times Bestseller.

On TV, Gordon Ramsay has made infamous the state of affairs at many restaurants on their deathbeds. *Kitchen Nightmares* is the raw truth about how poorly a restaurant can operate, and still run. Watching a restaurant pull itself out of near-certain catastrophe is, apparently, *entertaining*. The cooking industry has learned how to sell their commodity to the masses.

Other industry leaders have found the magic elixir as well. They present their craft in a way that tickles our senses enough to make someone who has no real interest in their craft, care.

Don't agree? Just flip through your television on a weekday night. In the United States and Europe, there are wildly popular shows on crab catching ("Deadliest Catch"), dog training ("The Dog Whisperer"), children's choirs ("The Choir"), dieting ("The Biggest Loser"), babysitting ("Supernanny"), blue-collar dirty jobs ("Dirty Jobs"), and raising octuplets ("Jon and Kate Plus 8") . These aren't exactly glamour industries.

"A girl, a hunky bachelor and seven dwarves?! You *sure* this isn't a reality show?!"

Then, why not software programming? Why can't we be among those that have figured out what makes their line of business marketable? Code lets us play games, make friends, converse with them from anywhere on this planet, find love, buy anything, monitor the sick, organize our lives, and everything in between. These are the kinds of thing *our people* create everyday. We have a captivating story to tell. Just ask the guy over there who's not saying a thing.

I'm not suggesting *Top Programmer* or *Coding Nightmares* are in pilot production anytime soon. But, we ought to put ourselves out there with the masses. We build the tools that run today's society, and everyday, there are those among us figuring out how to build them faster, cheaper, and more beautifully than ever before. Programming is a fascinating job. It's up to us to show the rest of the world.

The software world is, at its best, a beautifully run kitchen. At its worst, it's a complete organizational nightmare. *Legacy* usually means something grand and eternal in every other context besides what it means in software. We also live in a constantly changing medium. What we're using today will seem archaic five years from now. These are all viable topics for the masses.

We need to do it in small steps. It starts with the way each of us treats our work. At restaurants, good waiters take pride in presenting a dish. It's not just the chef's dish; it's his or her dish too. The craft, care, and art of it distinguish mere sustenance from exquisite cuisine. In the same sense, we ought to take pride in our work. Take off the headphones and talk to your neighbor, then talk to your wife, brother, sister, neighbor, and pet dog. Software development is full of craft, care and art.

The process of building software can be interesting and entertaining. What we do *is* a marketable business. It's up to us to make it more than just about code, just as the cooking world has made their work more than just about ingredients. Absorb yourself in it. Then, write, speak, and teach it to others with passion.

It's a struggle I have every day. Whenever I'm asked what I do for a living, I shrug.

I want to say that I'm a web developer and designer — a modern day web programmer, if you will. But, "programmer" just doesn't have the ring I'm looking for. It lacks the chutzpah of doctor, architect, or President of the United States. "Doctor" means miracle worker, "architect" alludes to the dreamer and master builder, and I hear being President has a few perks as well.

To the layperson, programmer equates to "working with computers," which carries about as much validity as equating a surgeon to "working with sharp things." The next time someone asks me what I do for a living, I'll tell them I'm a country music star. It's just easier.

We are, in fact, sometimes doctors, architects, and rulers all at once. We work miracles with our code, dream, build, and lay down the law. This is the book I will give them when they ask me what I do.