


Top 10 Career-Changing Programming Books

 rodhilton.com/2014/02/05/top-10-career-changing-programming-books

This post previously appeared on my blog, 'Absolutely No Machete Juggling'.

When I graduated with a Computer Science degree ten years ago, I was excited to dive into the world of professional programming. I had done well in school, and I thought I was completely ready to be employed doing my dream job: writing code. What I discovered in my very first interview, however, was that I was massively underprepared to be an actual professional programmer. I knew all about data structures and algorithms, but nothing about how actual professional, “enterprise” software was written. I was lucky to find a job at a place willing to take a chance on me, and proceeded to learn as much as I could as quickly as I could to make up for my deficiencies. This involved reading a LOT of books.

Here I reflect on my 10-year experience programming professionally and all of the books I've read in that time, and offer up the ten that had the most profound impact on my career. Note that these are not the “10 best” programming books. I do feel all of these books are very good, but that's not the only reason I'm selecting them here; I'm mentioning them because I felt that I was a profoundly different person after reading each than I was beforehand. Each of these books forced me to think differently about my profession, and I believe they helped mold me into the programmer I am today.

None of these books are language books. I may feel like learning to program in, say, Scala, had a profound impact on how I work professionally, but the enlightening thing was Scala itself, not the book I used to help me learn it. Similarly, I'd say that learning to use Git had a significant impact on how I view version control, but it was Git that had the impact on me, not the book that I used to teach myself the tool. The books on this list are about the the content they dumped into my brain, not just a particular technology they taught me, even if a technology had a profound impact on me.

So, without further ado...

Top “10”

The Pragmatic Programmer

I know, I know. Every list you've ever seen on the internet includes this book. I'm sorry, I wish I could be more original, but this book really is an eye-opener. *The Pragmatic Programmer* contains 46 tips for software professionals that are simply indispensable. As the name implies, the book avoids falling into any kind of religious wars with its tips, it's simply about pragmatism.

If you were to read only one book on this list, this is the one to read. It never goes terribly deep into anything, but it has a great breadth, covering the basics that will take a recent college-grad and transform him or her into someone employable, who can be a useful member of a team.

Many programmers got into the field because they liked hacking on code in their spare time, writing scripts to automate tasks or otherwise save time. There is a set of skills one develops just to sling code that makes a computer perform specific tasks, and that exact same skillset is needed by many, many employers. But there are many people who see programming professionally as simply an extension of their hobby, and do things the same way whether they are hacking at home or at work. *The Pragmatic Programmer* permanently altered how I view programming, it's not just extending my hobby of coding and getting people to pay me for it; there's a fundamental line between professional coding and hobbyist coding, and I am able to see that line and operate differently depending on what side of it I'm on thanks to *The Pragmatic Programmer*.

How groundbreaking is this book? Groundbreaking enough that it launched an entire publishing company. It's a big deal, if you've somehow managed not to read it yet, go do so.

What it changed: How I view “programming” as a job instead of a hobby I get paid for.

Continuous Delivery

Releasing software is one of the most stressful parts of the job. I can't tell you how many times in my career I've been part of a botched launch, or up until the wee hours of the morning on a conference call trying to get software into the hands of customers. When do we branch, what goes in what branch, how do we build the artifacts, what process do we walk through to get them where they need to go? It can be one of the most complex, error-prone, and difficult parts of professional programming.

Continuous Delivery means to do away with all of that difficulty. It describes a mindset, toolset, and methodology for completely turning releases on their head. Instead of doing them less frequently because they are difficult, do them more frequently so they're forced to be easier. In fact, don't just do them more frequently, do them **all the time**. *Continuous Delivery* describes, with real-world practical examples, how to version control all configuration, how to test integration points, how to handle branching and branch content, how to safely rollback, how to deploy with no downtime, how to do continuous testing, and how to automate everything from checkin to release.

In a lot of ways the book describes a pie-in-the-sky ideal. It's difficult to achieve truly continuous delivery, though GitHub, Flickr, and many other companies seem to have done so. But as the old adage goes, aim for the moon, even if you miss you'll end up among the stars. Wait, that adage is insane, stars are further away than the moon. Who came up with that phrase? Where was I? Oh right, even if you don't ever reach the true ideal, every step you made toward it makes deployments at your company that much better. I've worked in various environments where the principles of this book have been applied at different levels, and I can personally attest that there is a near-perfect linear relationship between how much you adhere to the advice in this book, and how smoothly releases go.

I worked in an environment operating at about a 70%-level of adherence to the philosophy outlined in this book, and it was heaven. When I left that job, my new employer was at approximately 0%, and it was complete misery. I set about implementing the ideas of the book and even a 10%-level of adherence was like a fifty-ton boulder being removed from my back. It worked so well that it was like a blinding light of epiphany for co-workers, and we wound up hiring someone whose sole job it was to help get us further along. Today we're at about 50%, and it's easily five times better than it was at 10%, and infinitely better than at 0%. Still hoping to get to 100%, obviously, but there's no doubt that every aspect of the book makes releases smoother and less stressful. I simply don't think I could ever work any other way ever again, it's like finding out you've been coding with a blindfold on for years.

What it changed: How I release software and bake releasability into my code.

Clean Code / The Clean Coder

Look at this, only a few items into my list and I've already cheated by including two books as a single entry. Yes, *Clean Code* and *The Clean Coder* are two separate books, but honestly they're both very short, and very similar. Both books are about how a programmer should conduct him or herself professionally, they simply cover different aspects. Professional software developers communicate with their coworkers in two ways: through code and through everything else. *Clean Code* is about how you communicate with your co-workers (fellow programmers) through code itself, and *The Clean Coder* is about how you communicate verbally, or through e-mail.

Both of these books, written by "Uncle" Bob Martin, really could easily be a single book with two large sections. Bob's philosophy toward professional software development is honest and direct, some would even say blunt. He makes no bones about it: fail to communicate in the way he describes, and you're bordering on professional negligence. It seems harsh but, frankly, I'm convinced. Call me a believer.

I definitely treat my code differently in light of his suggestions from *Clean Code*. It may seem strange that I categorize *Clean Code* as a book about communication, given that it's all about how to write code. But in the words of Abelson and Sussman, "Programs should be written for people to read, and only incidentally for machines to execute." Machines will run code whether it's "clean" or not, but your coworkers will only be able to understand and work with your code if it's clean. *Clean Code* is about how to structure your code for others to read, or even for the future version of yourself to read.

Even more than *Clean Code*, *The Clean Coder* had a profound impact on me. It drastically altered how I talk to bosses, product owners, project managers, marketers, salespeople, and other non-programmers. It advocates taking ownership of your screwups, being honest about abilities and deadlines, and up-front about costs. Not every co-worker you encounter will appreciate the approach outlined in *The Clean Coder*, but ultimately your customers will, because your products will be better for it.

What they changed: How I conduct myself professionally.

Release It!

A product's life doesn't begin when you first create the source code repository, or write the first line of code, or even finish the first story. It begins as soon as it's launched into production, into the hands of real users. Everything before that is just bits, just plain text files on disks. So in a lot of ways, it's astonishing how much thought is put into the code for the period of time before it's really born.

Release It! places the stress on the real life of a program. It's all about monitoring, health checking, logging, and ensuring that applications remain operational. It's about baking in concern for capacity and stability from the start, and what needs to be done to keep a program operating even when there are outages, or broken integrations, or massive spikes in load. Most of all, it's about **assuming** that code will fail, backend servers will die, databases will timeout, and everything your software depends on will eventually go to hell. It's a completely different approach to software development, and it's completely eye-opening.

Not to be too pejorative, but if you do enterprise application development, you probably shouldn't write another line of code before you read this book. I consider pretty much everything I've written before it to be inadequate for real production use, even all the stuff currently in production. It covers patterns and anti-patterns to support (or subvert) stability as well as capacity, and the section of the book covering these topics is simply excellent. But then it goes beyond that to also discuss Operational enablement. Even if you're not into DevOps, and don't want to really be involved in DevOps work, this book gives you the tools and tips to do what aspect of DevOps is the purview of pure developers.

Release It!'s tactics will make you your operations team's favorite person, and greatly help cover you and your team's ass in the eventual case of catastrophic failure somewhere. The patterns sections alone are worth the price of admission here, and the fact that the book is chock full of even more useful content beyond them is kind of stunning.

What it changed: What I consider to be “production-ready”, and how I view Operations.

Head First Design Patterns / Patterns of Enterprise Application Architecture

No list like this would be complete without a book about design patterns. But where’s the famous “Gang of Four” book, you ask? Not on this list, that’s where. Honestly, GoF was a pretty groundbreaking book at the time, but I personally think the presentation of the information it contains is awful. I believe everything presented in GoF is presented better in *Head First Design Patterns*. I know that not everyone is crazy about the Head First series, and even I find the structure and layout of the book grating at times, but I think the diagrams and visuals are light years better than those of GoF.

I also think Head First does a better job of providing *contextual* examples. While GoF provides sample code implementing the pattern, I feel that *Head First Design Patterns* provides a more valuable context for its examples, with more explanation about what the code is doing and what it’s for. This helps readers understand *when* to use specific patterns, which I feel is the most important thing to learn when learning patterns. Too often, people read their first design patterns book and immediately decide to implement as many as they can. This is the wrong approach to take with patterns, and I think Head First’s contextualization and strong visuals make it easier for readers to avoid this mistake. Jeff Atwood disagrees and I can see his point, but I think overall this book is better in this regard than the classic GoF.

Patterns of Enterprise Application Architecture is the GoF book, but at the level of architecture rather than code. Like GoF, it is extremely dry, and somewhat difficult to get through cover to cover, working better as a reference book than a reading book. It does a very good job, however, of managing to still provide ample context, describing when you’d want to use (or avoid) a particular pattern. I can’t tell you how many times I’ve referenced this book.

Patterns provide great “templates” to use when solving common problems. They need to be reached for with great care to avoid overuse, but when utilized appropriately can give developers a great deal of confidence in the time-tested designs they outline. Additionally, they provide a shared vocabulary among developers that greatly aids communication about complex topics. Describing the exact kind of hamburger you want to a Burger King employee is difficult when you have to describe every single element of the meal, but it’s much easier when you can simply say “number 5” and you both know exactly what is being ordered.

What they changed: How I design and discuss my software, both at the code and architecture level.

Working Effectively with Legacy Code

My first job out of college was replacing a developer who had left the company, as the sole responsible engineer on a massive and extremely complex codebase. Working in this codebase was terrifying, any change I made had the potential to break almost anything, and there was no way to test any changes without pushing a jar to the production system and watching it go. I checked over every change I made about a thousand times, and hand-constructed little `public static void main` classes just to instantiate classes and invoke methods, and then hand-check results. I had never heard of unit tests at this point (evidently, neither had my predecessor), so everything was done with kid gloves.

It wasn’t until 2 jobs later that I actually read *Working Effectively with Legacy Code*, which describes exactly how to deal with systems like these. The book explains how to take yourself from having no confidence in the codebase or your changes, to having complete confidence in them. It’s not simply about how to effectively manage yourself in the hole you’ve found yourself in, but exactly the tactics you can use to dig yourself out of the hole. It’s organized extremely well, indexed largely by actual complaints you might have about an inherited codebase. If I’d read this book earlier, my first job experience would have been much less stressful, and much more rewarding.

One important thing to realize is that “Legacy Code” doesn’t refer exclusively to million-line Cobol codebases. As soon as code is written and deployed somewhere, it’s legacy code from that point forward. Every codebase you’ve worked on that you didn’t write yourself as a greenfield project is a legacy codebase, and the methodology of the book will help. Once upon a time in my career, inheriting another developer’s codebase was frightening for me, and I’d often react (as so many developers do) by immediately wanting to do a full-scale rewrite of any codebase that’s too complex for me to manage. Thanks to this book, I have no problem inheriting code written by others, even if they’re no longer around.

Moving to a new job is less intimidating to me now, and I often spend the first few months of my time somewhere new simply getting the scaffolding in place to make changes confidently later on, increasing unit test coverage and breaking code into smaller and more isolated chunks. The full-scale rewrite is no longer the first tool I reach for in my toolbelt, it’s the last one, and I feel confident that I can refactor nearly any codebase into something I’m comfortable working on.

What it changed: How I feel about inherited codebases, and how I manage my confidence working with them.

Refactoring / xUnit Test Patterns

I think most recent college graduates, myself included at the time, are “cowboy” coders. I used to have all the changes in my head, and just tried to get them fed from my brain into the compiler as quickly as I could, before I forgot all the stuff I wanted to do. Today, I cringe when I think about how many characters of code I’d type between actually running or testing my software; “waiting for the

compiler is just going to slow me down, let me get all the code written first and then I'll debug it!"

Learning the technique of refactoring, in which you change the structure of code without changing the behavior, forces a mental split. You realize that "coding" is really two jobs, and that structure and behavior should be altered and tested independently, never at the same time. Martin Fowler's *Refactoring* is a collection of structure-but-not-behavior changes that really provides the toolset for a lot of other books on this list. *Refactoring* is so important that, depending on what language you work with, you may not even think you have to actually read it: your IDE probably supports many of the operations it describes out of the box. Nonetheless, it is a critical read, as it puts the reader in the mindset to understand the two hats they must wear as a coder, and how to intentionally change from "coding" to "refactoring".

Of course, refactoring goes hand-in-hand with unit testing. There are hundreds of books covering unit tests and test-driven development, but none of them that I've seen break things down as well as *xUnit Test Patterns*. The book covers everything a programmer needs to become a unit testing badass, how to work with mocks and stubs, how to recognize problem smells in tests, how to refactor tests, and tons more. It's not about a specific technology or tool, it's about unit testing best practices in general, and my attitude toward testing and the kinds of tests I write are much improved because of it.

Refactoring and testing are essential tools in the programmer's toolchest, and these two books cover all of the mechanics and tools one needs to master those tools. *Refactoring* focuses on improving the structure of your code, *xUnit Test Patterns* focuses on improving the structure of your tests, and your code and tests form a symbiotic bond of code quality. These two books are, in a lot of ways, two sides of a very important coin.

What they changed: How I approach altering existing code, and how I ensure I've done so correctly.

The Passionate Programmer / Land the Tech Job You Love

Okay, I get it, I'm terrible at making these lists, and clearly should have just done a "Top 15" or something. In any case, landing that first job out of college is tough, but eventually the day comes when it's time to move on. *The Passionate Programmer* is largely about how to find the right kind of job for you, what to look for in tech companies, and how to manage the direction of your career. It's pretty high level, but full of extraordinarily important advice to ensure you find yourself at companies that fit you and that you fit into well. *Land the Tech Job You Love* is more about the mechanics of this process, how to write a resume, how to interview, how to negotiate a salary, and the like. This is another situation where really two books are so closely related that they'd be better as a single larger book.

These books helped give me confidence to understand the process of hunting for and getting a job as a programmer. It completely shifted my mentality, from being the unqualified person begging a company to give me a job, to being a competent and capable engineer simply searching for a mutually beneficial fit. It changed how I view the job hunt, and how I conduct myself in interviews. After reading these books, I completely scrapped my entire resume and created a new one from scratch.

In a lot of ways, these books inspired me to create this very blog, or at least adjust what I used it for. I view my various online profiles as part of my "brand" and I think my viewpoint shift in this regard informs a great deal of what I post here, on twitter, and elsewhere. Yes, even all the inappropriate swearing (companies should probably know what they're getting into with me).

I have a lot of confidence about my career now, and I don't live in fear of losing my job or being unable to find a new one. I think about my career differently, as a very planned and deliberate thing, not just a series of jobs. It makes me excited about my future as a programmer, rather than concerned and fearful, which is a liberating sensation.

What they changed: How I view and manage my career.

Apprenticeship Patterns

Apprenticeship Patterns isn't really a patterns book as the name implies, but it's content has been kind of shoehorned into the format, I assume to increase sales. Ignoring that flaw, *Apprenticeship Patterns* is the best book on Software Craftsmanship I've read, and I've read quite a few. I actually recommend it above Pete McBreen's *Software Craftsmanship*, because it covers pretty much everything useful from that book, but excises some of the more unrealistic or naive bits, as well as the extremely long and pointless section about salary. *Apprenticeship Patterns* is a bugfix release for *Software Craftsmanship*.

This book was the one that made really see the value in the Software Craftsmanship movement, and truly embrace it. I've written elsewhere about why I like the Software Craftsman title, but this was the book that convinced me to consider myself part of that crowd. Software Craftsmanship isn't just about what customers can expect from you, it's about what your fellow developers can expect from you, and what you should expect from yourself. It's not just about writing clean code, it's about having a clean career, if that makes any sense.

I now put a much greater stress on my fellow engineers than I used to, and I care more about the team as a whole. In a lot of ways, this book takes the practices and techniques of many other books on this list and codifies them into an over-arching set of guiding principles. Software Craftsmanship as a movement can get a little culty at times, but I generally consider myself part of that cult, and I largely have this book to blame. The night time is the right time.

What's especially great is this book is it's been licensed under Creative Commons, and is now completely free on the web! Cool!

What it changed: How I view my responsibilities as a professional, and what I consider my true title.

The Art of Agile Development

The first job I had out of college was pure chaos. No process, no estimation, no planning, nothing. Generally someone from marketing would stop by a programmer's cubicle and inform them that they just sold a few thousand dollars worth of seats based on a feature that didn't exist yet, so how long would it take to implement it? Being my first post-college job, I was in "sponge mode," so I simply thought this was how it worked in the real world. It wasn't until my next job that I was introduced to Agile Development methodologies by way of Scrum, which was like mana from heaven. I was hooked.

My job after that was at a company that wasn't just into Agile as a methodology, their core business was actually developing agile tools for other software shops to use. The entire company lived and breathed agile, so knowing agile was the same as understanding the core company domain. It would have been impossible to do my job without understanding agile, in more ways than one. So when I first took the job, I decided I needed to read an Agile book to make sure I knew my stuff. Based on the title, I picked up *The Art of Agile Development*. What I didn't realize at the time was that there were a lot of different agile methodologies, and in fact this book wasn't about Scrum, it was about XP.

I became a die-hard XP programmer without even realizing it. My first exposure to "XP Programming" was a failed experiment in college that ruined it for me, I never would have knowingly bought a book on XP. But *The Art of Agile Development* changed how I do my job, it changed the processes I like to use when working with managers and other developers, and the practices I like to adhere to myself, such as Test-Driven Development, Spiking, Evolutionary Design, and the like. What's ironic is that I read this book to work at an agile company, only to find most of them disliked XP, and considered themselves Scrum only.

What's nice about XP is that it's pretty individualistic. You can employ XP principles as a developer while working within Scrum, Kanban, Crystal, Lean, or whatever else. In fact, that's exactly what wound up happening: a small contingent of developers at this company including myself began working in a more XP-style within the confines of the company's Scrum processes, and our successes wound up infecting larger and larger groups of people until pretty much the entire engineering team was working similarly. When the company switched from Scrum to Kanban, it had little effect on how we worked.

Today, my preferred way of working is with XP-style practices within a Kanban-style process, and an enormous part of that is because of this book. I wish I had a Kanban book to recommend as well to round this part of my list out, but 100% of my Kanban experience was gained on the job, with no books of any kind. What's more, having worked for three years at a company where agile was something bordering on a religion, I'm pretty burned out on the topic in general, so other process-centric books on my "to-read" list have found themselves migrated towards the bottom. Nonetheless, in all of my reading, *The Art of Agile Development* was easily the most influential book on how I like to work. This one is pretty subjective, as I'm pretty sure **any** good XP book would have had the same effect, but this was the one that did it for me, so I had to include it here.

What it changed: How I like to work in terms of processes and practices.

Update: Domain-Driven Design Distilled

When I first posted this list, I gave an honorable mention to Domain-Driven Design (DDD) even though I had never read it. My rationale was that, I had a hunch that once I finally read a book about Domain-Driven Design, it would have a career changing impact on me, but unfortunately I didn't like the style (or size) of the book itself, so I gave the honorable mention. Basically I wanted a book that gave me an overview of Domain-Driven Design without being a 560-page reference book. I wanted the "Head First Design Patterns" to the original Domain-Driven Design's "GoF". Something to make the material easier to digest more quickly.

I also said I had hoped *Implementing Domain-Driven Design* (IDDD) would be that book (so much so that I pre-ordered it), but it seemed to assume the reader has already the original DDD book, and it was even longer than the original DDD book, clocking in at 656 pages. That's over 1,200 pages about domain modeling, it just seemed like massive overkill.

Finally, *Domain-Driven Design Distilled*, by the same author as IDDD, was released. At a mere 147 pages, the stated goal of DDD Distilled was to get the information into the brains of as many people as possible without overwhelming them, give the basics and an overview of DDD in an approachable manner, and then allow readers to go deeper with DDD or IDDD later on. This was precisely what I was looking for and the book absolutely delivered - I really loved it.

I kind of always had this hunch that Domain-Driven Design was something of a buzzword fad, that it likely described something I was already doing regularly and that the book and the approach likely just lent formality and terminology to common sense activities. After all, the biggest thing I see referenced seems to be this Ubiquitous Language stuff, which I think just means using the same nouns for stuff as the domain experts, which I try to do anyway so I'm sure I'm already doing everything in the book, right? Nope. I was flat wrong, which is why I consider this book a must-read for engineers who do a lot of greenfield work, domain modeling, and architecture.

Early on, the author provides a sort of toy example that will stay with us for the duration of the book, designing the domain for a Scrum management product. I've actually worked a job where I did this very thing, so this resonated fairly strongly. The book suggests that, if engineers are left to their own devices, they'll design around code generality to reduce duplication, so there might be like a `ScrumElement` that could be a `Product` or a `BacklogItem`, and there's like a generic `ScrumElementContainer` which could be either a `Sprint` or `Release`. I'm just reading this section like, yeah, that's exactly what I would do... in fact I did that. Is that bad? But the rest of the book explains exactly why that's bad, and exactly how to do it better. Chapter after chapter, the book showed me the ways in which my approach to domain modeling was disastrously bad and how much better it could be. It also explained how, with this alternative approach, my domain would lend itself more easily to modular system design along service-oriented boundaries.

In short, this book is excellent and completely changed how I think about and model domain objects at work. The book can sometimes be light on detail, I often found myself wanting more information, or stronger examples of exactly how something should work, but at the end of the day that's the purpose of this book - a short introduction that encourages the reader to dive in deeper with Domain Driven Design or Implementing Domain Driven Design. As such, I can't really complain about the general lightness of this book, as it's the primary reason it was such an easily-digestible 147 pages.

Overall, this book is a must-read, I wish it existed years ago. I think back to all the times that a group of coworkers and I would gather in front of a whiteboard and model domain objects together without a single domain expert in the room. It makes me slap my head at how idiotic my approach has been for over a decade, the ways that I let database and technical concerns dictate the design of domain objects rather than the business's needs. I can never look at this regularly-performed process the same way, which is why it joins my list of Career-Changing Books.

What it changed: How I approach domain modeling; how I design service and module separations

Honorable Mentions

There are a number of books that I didn't include in the above list, but that nonetheless had a large impact on my career. This, of course, despite the fact that I completely cheated in my Top 10 and included more than ten books.

- **Presentation Patterns** - Only an honorable mention because it's not *really* about software development per se, but it's totally altered how I do presentations.
- **Pragmatic Thinking and Learning** - Learn more about your brain than you ever realized you needed to know. Though not specifically about programming, it's a very programmer-centric view of the mind, and how one can best work with your own mind and improve your ability to think and learn.
- **Effective Java** - I said I wasn't going to include any technology-specific books, but I can't help but mention *Effective Java* somewhere. I was programming in Java for years before reading this book, but afterwards I felt like a Java master. I almost never work with pure Java anymore, instead largely using other JVM-compatible languages, but the Java I wrote before reading *Effective Java* looks very different than the Java I wrote afterwards, and I definitely prefer the latter.

So that's my complete list. I obviously have many, many more books to read, and I look forward to writing another list like this one in the future after being profoundly changed for the better some more.

Have some books you want to add? Feel like telling me one of my favorite books is inferior to one of yours? Want to yell at me for not including *The Art of Computer Programming* (come on, you never read that shit and you know it)? Leave a comment!