



Keep it beautiful

Everything here is my opinion. I do not speak for your employer.

[An epic treatise on scheduling, bug tracking, and triage](#)

[A little bump in the wire that makes your Internet faster](#)

2018-07-24 »

Books that explain (parts of) how the world really works

I spend a lot of time answering various people's tech-business questions. Occasionally, I will say something that sounds brilliant, and an unlucky listener will exclaim, "Avery, how do you know so much stuff?"

Answer: I read it in a book. Now, I don't actually read very many books. But I've been lucky enough to receive some great recommendations. Here are mine. Now you, too, can sound like a genius.

To make it more interesting, I'll start with the questions, then tell you the book that answers them.

Q: Why is your multi-year epic project still not profitable?

Because you don't understand how the "technology adoption life cycle" (the one with visionaries, early adopters, mainstream, laggards, etc) really works. The ultimate book on this topic is [Crossing the Chasm](#), by Geoffrey Moore in 1991.

(You're going to notice a pattern to my book suggestions: they're old. It turns out really good advice stays relevant.)

Crossing the Chasm is very dear to my heart: it was recommended to me by one of the VCs that invested in my first startup - unfortunately a bit late. Chapter by chapter, it was like

reading the post-mortem for my startup, except it was all written before we ever started.

Don't do this, because that will happen. Oh, crap, that's what we did, and that's what happened. *Whatever you do, avoid this common mistake, because it always results in xyz.* Yup, xyz everywhere. And so on. When we finally turned things around, I gave all the credit directly to this book. It is that good.

My favourite part is their definition of a "market segment." Most people don't know what they're talking about when they talk about a market segment. You think you do, but you don't. I thought I did, but I didn't. You can't run a startup without understanding what a market segment is. You will fail. Read the book instead.

One of their bits of advice: don't try to capture 10% of a big market. Capture 100% of a small market. Anyone who says "if we can just get 1% of this 10 billion dollar market..." is admitting defeat. Nobody will buy your product if it resolves only 90% of their obstacles, and for every market segment, it's a different 90%. You need to find a market segment, even a small one, for which you can solve 100%.

Q: Why are all these Cloud service providers losing money and offering terrible products?

Because the "Cloud" market has already Crossed the Chasm and has moved into the mainstream ("hypergrowth") phase, where the rules are nothing like what you're used to.

After the chasm has been crossed and you've made a product that serves several adjacent market niches, it gets easier to win even more market segments, partly because your company is actually making money and growing. Each new segment makes you more profitable, so you grow more and expand more, in a positive feedback loop.

The only problem is that, once you start taking off, you attract competition. Crossing the Chasm is largely about staying orthogonal to your competitors, but as you enter the mainstream, that stops working.

That's Cloud services right now. And ride sharing.

[Inside the Tornado](#), also by Geoffrey Moore, a few years later in 1995, talks about this situation and how to deal with it.

To be honest, I read this book because I liked *Crossing the Chasm* so much, but I recommend it a lot less often. Almost nobody is lucky enough to get into the middle of a hypergrowth market, so the book is mostly irrelevant to almost everyone. (One exception: by reading the book and learning what a hypergrowth market is really like, you might realize you aren't in one after all, and save yourself a lot of mistakes. Or you might realize you don't have the capital needed to compete, and bail out.)

An interesting observation in the book is that hypergrowth phases are temporary (eventually the entire mainstream is saturated), and the market share of various competitors, after hypergrowth ends, stays mostly fixed. Customers sign up with a particular product, and are reluctant to change. That's why, say, desktop PCs have run about the same set of OSes for a long time, the same word processors and spreadsheets have been popular in the same proportion for a long time, and the relative market shares between Coke and Pepsi or Colgate and Crest rarely change.

(After saturation, customers still switch from one supplier to another, but there's a dynamic equilibrium: everybody keeps spending money on sales and features, but none of them can do it any better than the others, so you might lose one customer and gain another.)

So, to oversimplify, the book's recommended strategy during hypergrowth is to invest like crazy to acquire market share, because the big customers you acquire now might stick around for decades. It's worth losing money in the short term. You have to move impossibly fast and land those customers at any cost, sacrificing short-term profitability, product elegance, and so on.

The tornado is ugly. It results in ugly products and weird business models (at least temporarily). There'll be time to fix it all later, when the market is saturated.

Q: Won't Big Company X just clone your product and steal all your customers?

Maybe! But maybe not. It depends on some surprising factors.

Nowadays in the tech industry we use the words "disruption" and "innovation" interchangeably. We like to talk about "disruptive innovation," but by now we've forgotten why we insert the adjective instead of just saying "innovation." Is it just because disruption just sounds cooler?

We can trace the term "disruptive innovation" back to the book that invented it in the first place: [The Innovator's Dilemma](#), by Clayton M. Christensen, in 1997. That book is about the difference between *two kinds* of innovation: sustaining innovation, and disruptive innovation.

As techies, what we've forgotten is that sustaining innovation is the common one. It's so common we forget to name it. When Intel makes a newer, faster chipset (sadly less often lately) or there's a new model of laptop, that's sustaining innovation. It happens all the time. And notably, big companies are really great at it. If you're a big company and you make laptops, then if you make an even better laptop, you'll probably make customers happier, sell more of them, and make more money. Everybody wins. Companies pour money into that.

If your startup is making a product that's the same thing only a bit better, then yeah, Big Company X is going to clone it and eat your lunch. You're doing sustaining innovation, and all that takes is money. They have more money than you. The end.

Very different is disruptive innovation. A disruptive innovation generally has some fatal flaw that makes it laughable to incumbents. A smart phone... without a keyboard? Come on. A taxi service... where the drivers are untrained, unprofessional, and unlicensed? An ad algorithm that tries to show you *fewer* ads? A tiny little 1" hard disk that's way more expensive per gigabyte? Who wants that?

The Innovator's Dilemma explores what actually makes innovation "disruptive," across several different product areas,

from digging machines, to printers, to hard disks, and shows how the same trend repeats over and over. As the incumbent, you keep making your products better (sustaining innovation). Customer requirements increase, but not as fast as your products improve. Meanwhile, some competitors are working on some obviously inferior technology selling to your lowest-value cheapskate customers you're happy to have off your books anyway, while you sell to ever-higher-end customers and make ever-larger profits.

The competing technology is far from meeting your customers' demands, and its trajectory clearly shows it will *never* catch up to your product line. Then, one day, the inferior technology - still much worse than yours - finally gets *good enough* to meet your biggest customers' needs. Boom. You're dead. It's still not as good as your product, but that doesn't matter. It's *good enough*.

What makes this a "dilemma" is that, right up until that crossover point, your company would lose money by investing in the new thing. The old thing is far more profitable, and your customers don't even want the new thing; the new thing can't yet do what they want! Any profitable company has been highly optimized to deliver what customers want, and reject what customers don't want, so of course you won't invest in the inferior product. And then, one day, all your customers change their minds all at once, and you're too late.

You can plot the whole thing on a graph. You can even plot it before it happens, knowing it will happen, and still not be able to stop it. Math is beautiful. (The book suggests a few tricks to try to save yourself, but the tricks are very non-obvious.)

If you want to know why IBM missed the PC revolution, and why Microsoft was caught off guard by the Internet, eventually exited the smartphone market, still runs apps with tiny mouse-optimized toolbars on touchscreen tablets, and still can't run your Access databases on the web, this book is the one.

It'll also help you answer that very important question: what's really stopping Big Company X from cloning my product?

Q: I have lots of money. How do I clone a competitor's product, only better?

Don't do that.

Perhaps it's too obvious when I phrase it that way, but let's be honest, people try this all the time. Imagine, say, I don't know, instant messaging apps. There's an existing product getting popular, it looks pretty trivial, I could clone that in a week! Then I'll just spend more on marketing, or bundle some product integrations, or my better brand name, and rake in the customers.

Maybe. I mean, you're not automatically doomed if you try this approach. It's worked before. Like the U.S. invasion of Afghanistan... oh wait. Well, okay, the U.S. didn't *lose*, right? They did spend like 10,000x as much money as their opponent though. Perhaps there was a better way.

This is where I recommend [The Art of War](#), by the famous Sun Tzu, in the 5th century BC.

I'll admit it, the book is a bit of a cliché at this point. The advice all sounds obvious, in its obtusely-phrased way, perhaps because people have been studying it for millenia and then quoting it in Hollywood movies. Problem is, most people still aren't getting it.

The book starts with "The supreme art of war is to subdue the enemy without fighting" and moves on from there. Why are you fighting? Why are you fighting on the opponent's turf? Why is your strategy so transparent? Do you even have a strategy?

Strategy is a thing that can be learned. You won't learn it all from reading this book, which is pretty short and a bit outdated (although it does have some parts about supply chain management). But it's a start.

If you want to know how Microsoft really *did* get "a computer on every desk and in every home, running Microsoft software," in a world where they had an inferior product and lots of competitors, this is a good place to start.

Q: I have the Best Architecture and the Fanciest Developers. Why do people hate my product?

Lots of reasons, of course! But an interesting reason is that designers and architects have a tendency to look at their products from a bird's eye, high level, super abstract view. They want the abstraction to look beautiful, and they spend not enough time on the ground, in the dirt, fine tuning things to make them work right.

It's virtually impossible to explain this, in terms of programming, to the programmers who are doing it. And we do it, all of us. But sometimes we can learn from analogy. That's one reason I really liked [The Death and Life of Great American Cities](#), by Jane Jacobs, in 1961. (It was recommended to me long ago by a one-time roommate of mine, and it really changed how I think about cities and about design.)

It has absolutely nothing to do with programing, but like The Innovator's Dilemma talking about bankrupt digging machine companies, we can learn from Jane Jacobs's excellent rant about terrible-brilliant urban planners, with huge budgets, following Best Practices and optimizing literal aerial views, making most of the cities of North America into awful messes, and also severely screwing up large parts of even the cities that are better.

Like the people who said Dependency Injection was a bad idea, Jacobs was ridiculed for a long time, but history has proven her right.

It's fascinating to read why flattening neighbourhoods and rebuilding them from scratch doesn't work; results are vastly worse than simply refactoring (although the term "refactoring" hadn't been invented in 1961). It turns out that a lot of subtle stuff was built up in that neighbourhood over time, and when you start from scratch, you lose it all. You might have [heard this somewhere before](#).

I love this book because not only can you learn how Car Culture happened (while assuming people were doing their best, rather than resorting to a conspiracy theory about car manufacturers and public transit), but also what causes crime, what motivates

people to care (or not care) for their neighbourhood buildings, and the mental traps that confound anyone in search of elegance.

Q: Why is software development so unpredictable?

Actually it isn't, you're just predicting it wrong. Sorry. I [already wrote way too extensively about that](#) so I'll resist going over it all again.

Out-of-control software development processes turn out to be a special case of general out-of-control processes, and who's the expert on process control? W. Edwards Deming, of course, in his several books, including [The New Economics](#) from the year 2000. I [wrote about some of his work](#) about a year and a half ago.

Since then, I've had more time to contemplate what I read, and I'm now pretty sure I know the very most important thing I learned: the difference between random variation and outliers.

When we're modeling any kind of statistics, we have a tendency to assume our data fits some kind of standard statistical distribution - for example, Gaussian.

What Deming was teaching was that real-life human processes often don't cleanly fit a continuous model. The reality, in manufacturing and in software, is only *mostly* Gaussian. Every now and then something wacky will happen that is not just part of the distribution. Imagine the difference between drilling a hole slightly off center, vs breaking the drill bit. How many standard deviations out is a broken drill bit? Wrong question.

Imagine doing a least-square-error curve fit while including those outliers. Because errors are squared (hence the name), a few big errors will move the distribution as much as a large number of small errors. But the outliers are by definition not predictable or reproducible, so including them in the curve fit only stops you from predicting the remaining predictable parts.

Deming's greatest lesson, in my opinion, is that you have to treat outliers and common errors completely differently. When you overdrive a software development process, then things don't just stretch, they break. Teammates getting tired and needing

recovery (worse standard deviation) is one thing; teammates quitting (outlier) is another thing entirely. You want to reduce standard deviation, but you usually do that by adjusting a continuous variable (eg. to make recovery time more predictable, work people less hard). To prevent outliers, you need a more discrete change (eg. to keep people from quitting, figure out what is making them quit, and get rid of it).

Deming also provides actual techniques - albeit very handwavy techniques that seem to make "real" statisticians cringe - for detecting outliers vs continuous variables. Something about the standard deviation and the interquartile range. This margin is too small, etc.

...

Well, that got long. Maybe I should have just listed the book titles.

Related

[Systems design explains the world: volume 1](#) (2020)

[On the causes of poverty](#) (2008)

Unrelated

[How to design a replacement for C++](#) (2010)

I'm CEO at [**Tailscale**](#), where AI runs on us, not the other way around.

You can [**find me on Bluesky**](#). Or subscribe with [**RSS**](#).

apenwarr on gmail.com