Software Crisis 2.0

Baldur Bjarnason

An essay on our industry’s core expertise: failed software projects.

The First Software Crisis

Back in the early days of computing, we were, as always, extremely bad at making software. Except it was fine as most computers were practically custom-made to their purpose and only had the power to be able to tackle relatively simple problems. Simple problems meant simple software. This didn't last long, Moore's law being a more concrete reality back then, computers steadily became more powerful which led them to being applied to more complex problems.

This did not go well.

By 1968, this was a full-blown crisis:

> There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well.
>
> David Gries and A.G. Fraser, from Software Engineering: Report of a conference sponsored by the NATO Science Committee

Once the field, in general, had accepted that there was indeed a crisis they could begin the work addressing it.

Which they did up to a point. Fred Brooks wrote The Mythical Man-Month in 1975. Edsger W. Dijkstra and others worked hard to improve the craft of programming. Douglas Engelbart pioneered UI/UX research at the Augmentation Research Center in the 60s. Alan Kay and others took over the UI research baton at Xerox Parc in the 70s. This led directly to the Mac, Hypercard, the mainstreaming of hypertext, NextSTEP and the web.

Software development had a clear, upward arc. You could argue that software in general still wasn't good enough. But it seemed to be a pragmatic inevitability that would at some point be just that: good enough. Even software that was objectively bad in multiple dimensions like Windows 95 or 98 was a clear improvement on what came before. To paraphrase William Gibson: good software was already here, it just wasn't evenly distributed *yet*.

Most software was still bad. Most UX designs were still horrible. Bug-free software still seemed to be an elusive, mythical beast. But you could look at the industry and think to yourself that this was going to get fixed. The industry was in the process of being 'modernised', turned into a proper field of practice.

Then, in 2001, 'Agile' stepped in with the Agile Manifesto. Finally, hopefully, at last, good software development was going to be popularised.

Are Things Improving?

Twenty years have passed and you can no longer plausibly claim that software is improving.

If there is one universal truth in software it's that there's a lot of bad software and failed software projects. If you've worked in the industry for more than just a few years, a minority of the software you've worked on will have been a success. If it wasn't buggy, it'll have been over budget. If it wasn't over budget, it'll have been late. If it wasn't late then it probably didn't fulfil all of the set requirements or features you had planned. If it fulfilled the requirements, you'll have found out that the requirements were utterly wrong.

Most of the software projects we've worked on are failures at some point. A few recover. Many don't. This is the norm.

As an industry, we're really bad at what we do and we all know it. Thus Weinberg's Law is still in effect: "*If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.*"

You often see people cite a statistic like "90% of software projects fail!" Or, "60% of all software projects fail!". It's useful when you want to dismiss some standard practice in the industry without making an actual argument about that practice. If somebody asks you to back that up with a reference, just handwave it away. *Everybody knows this*. Which is usually a very reliable indicator that the stat is a made-up urban legend.

Except, in this case, it isn't. Not only are both of the percentage stats backed by research, but they both also come from the *same* ongoing research.

The Chaos Report

The Standish Group has been writing reports on the prevalence of software project failures since the 90s. They call them Chaos Reports. They have a database of participating companies, with a decent range in size from small to huge. They survey the companies regularly and collect stats. It isn't perfect and it probably doesn't reflect the industry at large. But that's fine because the industry proper is almost certainly much worse. It has the benefit of being done by the same people using the same overall methodology over an extended period. It's also the only real long-term research we got on the subject.

The 90% stat originally comes from their 1995 report:

> Only 9% of their projects come in on-time and on-budget. And, even when these projects are completed, many are no more than a mere shadow of their original specification requirements. (Standish Group – Chaos Report)

Not only is the 90% stat marginally understating the scope of software failure, once you factor in requirement failures the rate of success drops even further. The industry knew what good software development looked like, but it wasn't the norm.

The 60% stat, however, also comes from the Standish Group, just a much later report.

	2011	**2012**	**2013**	**2014**	**2015**
SUCCESSFUL	39%	37%	41%	36%	36%
CHALLENGED	39%	46%	40%	47%	45%
FAILED	22%	17%	19%	17%	19%

The Chaos Report 2015 traditional resolution for all projects

Their 2015 report shows that the success rate for a software project is now in the 36-39% range, giving us a 61-64% unsuccessful rate which sounds much nicer than 90%. Also, the research is now more representative of the value delivered as it shows that around 40% of the projects delivered results in some way despite being unsuccessful. This statistic is in the same ballpark as that shown in other research studies, one of which in 2008 put the unsuccessful rate at 68%.

—*We've gotten better at software development, yay agile!*

Yes and no. These stats only measure whether the project is on time, on budget, and on target. This reflects our experience as developers. While a good portion of what we work on is unsuccessful by any reasonable measure (business failure, unusable, poor software quality, inaccessible, poor performance, negative return on investment, technical debt, septic code etc.) we do generally manage to get paid and deliver what was asked. For the most part. It isn't our fault that managers keep asking for dumb requirements they haven't backed with research. Software projects do tend to get delayed so a 40% success rate sounds accurate and you can imagine that most of the 'challenged' projects are just a bit late or just a bit over budget. It feels like it reflects the state of the industry.

If you're a software developer, that is. Sounds completely unreasonable to anybody who needs to *use* or rely on our crap.

This is why in 2015 the Standish Group changed their 'resolution definition'. Now, to qualify as successful, a project needed to be on time, on budget, *and have a satisfactory result*.

Uh, oh. Did a cold sweat just run down your back or was that just me?

—*Satisfactory result? What has that got to do with software? That is not something we do, dear sir!* (Flounces off.)

	2011	**2012**	**2013**	**2014**	**2015**
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Chaos Report 2015 modern resolution for all projects

All the stats drop. Not by as much as you'd think. Which is good? We still have about a 30% success rate.

That isn't the full picture. The horror story reveals itself when you break it down by project size.

	SUCCESSFUL	**CHALLENGED**	**FAILED**	**TOTAL**
Grand	6%	51%	43%	100%
Large	11%	59%	30%	100%
Medium	12%	62%	26%	100%
Moderate	24%	64%	12%	100%
Small	61%	32%	7%	100%

The Chaos Report 2015 resolution by project size

For medium, large, and grand projects, the success rate is only in the 6-12% range. For most project sizes that we are likely to deal with in the industry, we're back in "90% of all software projects fail" territory. Even moderate projects (too big to be small, too small to be medium) only succeed 24% of the time. The only reason the final number hovers around 30% is that it turns out that we are damn good at making small software projects: 61% are successful.

To refer to one of my favourite books on systems, The Systems Bible by John Gall: "The larger the system, the greater the probability of unexpected failure." It should be surprising that the odds of failure increase fast with software project size.

This would be fine, just make all the projects small, except we all know that that's often not possible. We are, generally, really *really* bad at reducing scope. Everything escalates in software development. We know no moderation and in node.js development, habitually add thousands of dependencies right at the start.

—We're gonna need this. We're gonna need that. And that. And that as well. We're gonna need all of this at some point. Might as well add it now.

We know it's wrong. We have kind of figured out how to do this job. We just lack the discipline to follow through. It's also just a small part of the picture. Some of what we have seen as progress in software development is more akin to stat manipulation. The state of a software product, for example, isn't reset at the end of every release. Over time, your small projects start to feel less and less small.

I don't know about you but this feels depressingly familiar. Since the 90s we've learned how to deliver what people ask for, but we remain depressingly incapable of acting on that knowledge.

—Surely that's somebody else's problem, right? We deliver what we're asked, in the way management wants it done, and we're going to get paid whether it delivers any value to the organisation or not.

> Vronsky, against his own will or wishes, followed her lead, hoped too that something, apart from his own action, would be sure to solve all difficulties.

Leo Tolstoy, *Anna Karenina*

Something is sure to solve all difficulties

The true reason for the state of software and even knowledge work in general: we, as a whole, don't know the answers to even the most basic questions:

- What is good management?
- How do we prevent bad software?
- How do we prevent *harmful* software? Software that is harmful to society, culture, the workforce, or the environment?
- *What is good software*?
- How do we recognise good software?
- How do we develop a culture for talking about, analysing, and enjoying well-made software?
- Are we in a position where we can even tell whether our software is good or not?
- Do we have any capacity for acknowledging that the quality of our software is declining?
- Or, not increasing when it should?
- Are we capable of recognising when the process is serving the product's end-user?
- Or, when it's not?

These seem basic but that's where we're at as an industry: we do not have answers to the most foundational questions of our work. Agile hasn't answered any of them, even though its various flavours have tried. Like Vronsky in Anna Karenina, we're waiting for somebody else to solve all our difficulties.

Most people involved in software development don't know what truly good software looks like because they've only ever used mediocre or awful software.

Most of those who make software don't know what a sensible software product development process looks like because they've never seen it happen.

Most of our managers know less about management theory, organisational psychology, or even basic principles of collaboration than an abusive high school gym teacher.

At least the abusive bastard *wants* to hurt us. The manager does it unintentionally through ignorance or inexperience.

> It always happened with Levin that when his first shots were a failure he got hot and out of temper, and shot badly the whole day. So it was that day. The snipe showed themselves in numbers. They kept flying up from just under the dogs, from under the sportsmen's legs, and Levin might have retrieved his ill luck. But the more he shot, the more he felt disgraced in the eyes of Veslovsky, who kept popping away merrily and indiscriminately, killing nothing, and not in the slightest abashed by his ill success. Levin, in feverish haste, could not restrain himself, got more and more out of temper, and ended by shooting almost without a hope of hitting.

Leo Tolstoy, *Anna Karenina*

Shooting without a hope of hitting

It's a miracle we ship anything at all, between the overwork, poor communication, misguided research, and the rampant fad-seeking. *Software is born doomed and it's a constant struggle to keep it alive.*

All of our errors, mistakes, and misdirected obstinancy are all winding paths to the same destination: we keep building software that people don't want to use. ***That's our original sin.*** Sometimes they just don't need it. Sometimes it just doesn't address a problem anybody has. Sometimes it's done in such a way that nobody *wants* to use it, even those who do need it.

The end is the same.

It's tempting to look at the Chaos Report and conclude, as most of the agile cosplay industry has done, that the key is to slice everything up into smaller slices. Behold! The two-week sprint, a salve for our wounds, saviour of all our projects, a boon for all managers! A *silver bullet*, if you will. Because there is such a thing.

Except, no. It only works to a degree. The short sprint model makes scope creep harder but doesn't prevent it and it doesn't prevent cross-sprint interdependencies. A lot of the time the sprint's conclusion is just punctuation, a little bit of breathing space between sentences, and doesn't work as a chapter break that concludes an idea while setting you up for the next one. Sprints can often serve to exacerbate death marches by making a grand project's feature production relentless and never-ending.

Which, doesn't work that well. Clearly. (*Gestures wildly towards the Chaos Report tables.*)

Even done well, sprints don't fix the problem as software is *ongoing*. Your project state isn't reset at the start of each sprint so you rarely get the scope benefits of a small project. There will come a time that a sprint will fail, badly. That's just a fact of life. Sometimes things go wrong. Instead of taking time to recover, fix what's broken, figure out what went wrong, most project leaders in software development will do the absolute worst thing you can do under the circumstances: push everybody a bit harder.

—*Let's catch up.*

A sprint failed so you work just a little bit harder for a sprint or two, just to catch up. Except everything just gets harder and the failures start to cascade. Before you know it, the project is spiralling and you're on your way on a death march towards failure.

The manager will blame the staff, of course. *They weren't as good as we thought. They made mistakes. They aren't willing to put in the effort. They gave us the wrong estimates. They keep adding new bugs and not fixing old ones. They keep wasting precious time on procedural faff like types, unit testing, and automation.* The manager who doesn't blame the staff, who remembers Deming's 95% rule ("95% of performance is governed by the system"), wouldn't have made the mistake in the first place, would have understood that a naive sprint format doesn't have the retrospective, analytical, and recovery features that the software development process truly needs. So, you either need to bake those into the process around the sprint, maybe add those as dedicated sprints, or switch to another process entirely.

Instead, we march and, like Tolstoy's author stand-in Levin, we resign ourselves to shooting without a hope of hitting. Software development takes more time than most managers or even developers appreciate. *Time to be introspective. Time to be analytical. Time to recover. Time to be unfocused. Time, that cannot be made up by adding more staff.* As Frederick Brooks observed in 1975 in the Mythical Man-Month, you can't partition a twelve-month, single-developer software project into neat units to be done by twelve developers in one month.

It isn't enough to minimise the odds of failure for individual projects. Software builds up to become complex systems, so you need to have sensible ways to deal with failure modes. John Gall, The Systems Bible: "*The Fundamental Failure-Mode Theorem* (F.F.T.)*: complex systems usually operate in a failure mode.*"

—*That means that if we give ourselves more time, and dedicate specific space towards recovery, analysis, and correction, we're going to be fine? The key is to do agile correctly, right?*

—*Right?*

> Vronsky, meanwhile, in spite of the complete realization of what he had so long desired, was not perfectly happy. He soon felt that the realization of his desires gave him no more than a grain of sand out of the mountain of happiness he had expected. It showed him the mistake men make in picturing to themselves happiness as the realization of their desires.

Leo Tolstoy, *Anna Karenina*

The realization of desires

In established companies, the way you survive and advance your career is to serve the organisation, not the product or the end-user. The work that gets done is in service to the organisation itself. Sometimes that results in a benefit to the end-user. Often it doesn't because, again in Deming's words: "nobody gives a hoot about profit". Or, if you prefer Russel L. Ackoff's words: "*the principal function of those executives is to provide themselves with the quality of work life that they like and profit is simply a means*."

Everybody, including management, is in it for their career and both software failure and financial losses have surprisingly little effect on most people's prospects. Especially in the software industry. What harms your career is going against the organisation, challenging the consensus, and not being a team player. That shit gets around and gives you a bad reputation because that's what the organisation cares about.

Getting people to do the right thing, at the right time, in the right order, is surprisingly difficult, even when you know all of the basic principles.

The standard way to counteract this is through a process or a system. You make sure that research and end-user benefit is ensured, somehow, as a part of the process. You don't just do stuff that the organisation thinks it needs. You find out what it truly needs.

The way we usually do this, which rarely works, is to do a bunch of 'research,' read a lot that we personally find interesting, interview a few potential stakeholders, and then set out to make what we think is the right solution.

This is the famous waterfall process in spirit, if not in fact. It doesn't matter if you're using all the agile processes, or if you're scrumming up the place left, right and centre. If you've decided exactly what you're going to do in advance, before any of your work encounters a real end-user, you are following the spirit of the waterfall process.

1. You decide what to do based on a few interviews and surveys.
2. You make some headway towards implementing it and then decide to test it or do more research because that smells like agile, design thinking, or something.
3. The results are a bit all over the place but do tell you that something isn't quite working.
4. So you adjust what you had made.
5. Test again.
6. And the loop continues until you run out of money.

The problem here is that you've jumped headlong into what Deming called "tampering", which is "taking action based on the belief that a common cause is a special cause."

You think that the mistake is an error in the latest thing you did when it was down to a faulty assumption underlying *everything* you're doing—the entire project is misguided.

A/B testing mitigates this but doesn't prevent it as it does nothing to fix faulty assumptions. A/B testing doesn't help you build a model or map out what it is that the end-user *needs*. It can help you find a local optimum for a variety of issues but can't help you find an overall direction. Because the problem is that the initial idea is always wrong in some substantive way. Software development that obsessively focuses on unattainable goals will always be doomed.

If you build software this way, start with an unwavering idea, alternate between testing it and A/B testing it, and 'fix' issues that crop up in the last test, you only end up tampering with the initial unworkable idea. All you accomplish is to *magnify* the project's error rate until it fails. *This* process will never help you discover actual needs.

At best this results in featuritis.

> She flew over the ditch as though not noticing it. She flew over it like a bird; but at the same instant Vronsky, to his horror, felt that he had failed to keep up with the mare's pace, that he had, he did not know how, made a fearful, unpardonable mistake, in recovering his seat in the saddle. All at once his position had shifted and he knew that something awful had happened.

Leo Tolstoy, *Anna Karenina*

Something awful happens

We love features.

For external-facing software, like a service or a product, you tend to get tacked-on features for marketing to hype up. These rarely make sense and leads to ongoing UI clutter until, like Microsoft with their ribbon toolbar, you're forced to try to invent completely new kinds of widgets to make the mess even halfway usable. At least in their case, the initial product was a sound one and remained excellent for many years. (RIP Word 5.1a. I used that app for years.)

Features tend to result in momentary boosts to sales and attention. They lend themselves to being measured, both by management and by reviewers. The negative effect of clutter and complexity is generally indirect and can't be traced to an individual feature and so avoids individual blame. Adding a feature to an existing design is almost always good for your career and rarely a benefit for the end-user.

Most software is layers of cruft, wrapped around strata of mediocre designs and good intentions, sometimes with a sound idea at its centre.

Not everything has to be genuinely excellent. You don't have to be good to succeed. Bad software can succeed and good software can fail. If the need is great and a bad app addresses even a part of it, that can turn into an amazing blockbuster success. This leaves you open to a future competitor whose app isn't bad but that's a problem for then, not now. In the meantime, you can bank your bonuses, pile on the feature cruft, and leverage your luck into a higher paying position elsewhere. And if you make a fantastic, well-designed, reliable app that does not solve a single real problem for anybody anywhere it's going to fail. Miserably.

The key here is a true need. Or, if you want to be a little bit fanciful about it, *how does it make the user feel*?

There happened to him at that instant what does happen to people when they are unexpectedly caught in something very disgraceful. He did not succeed in adapting his face to the position in which he was placed towards his wife by the discovery of his fault. Instead of being hurt, denying, defending himself, begging forgiveness, instead of remaining indifferent even—anything would have been better than what he did do—his face utterly involuntarily (reflex spinal action, reflected Stepan Arkadyevitch, who was fond of physiology)—utterly involuntarily assumed its habitual, good-humored, and therefore idiotic smile.

Leo Tolstoy, *Anna Karenina*

Good-humoured idiotic smiles

Every application is judged by the user in terms of how “awesome” it makes *them*.

(If that sounds like a very American way of putting it, that’s because it is. But Kathy Sierra, in her book *Badass: Making Users Awesome* and her blog Creating Passionate Users, does a good job of backing that North American style of grating hyperbole with some of the best design advice you can find.)

Making it easy for users to solve the problem at hand is an effective way of inspiring that feeling. If you are spending hours and effort on a feature and you don’t know whether the user wants it and it has no obvious bearing on the awesome factor, you are probably doing something wrong.

This is why one of the many many things you need to do to prevent disaster is research and exposure. You need to do research that’s based on what people do, not just what they say they do. You need exposure hours for most of your team members. You need to make sure that your team is healthy. You need to let people do what they know best – let the specialists be the specialists and not just implementation monkeys for the mediocre designs and ideas from management. *Autonomy*. That sort of thing. You need to do all of these things and so many more and you still might fail because you also need a tremendous amount of luck.

Which is another reason why software projects fail: *simple bad luck*.

But, if you're addressing a genuine need, for a sizable group of people, whose needs you know either through research or experience, and you figure out a way to make them feel awesome while addressing that need…

… then your software product *might* not be doomed.

I say "might" because, as I noted above, almost everything is wrong with how most software projects are managed. As the Chaos Reports demonstrate, most projects are likely to be unsuccessful.

She took no interest in the people she knew, feeling that nothing fresh would come of them. Her chief mental interest in the watering-place consisted in watching and making theories about the people she did not know. It was characteristic of Kitty that she always imagined everything in people in the most favorable light possible, especially so in those she did not know. And now as she made surmises as to who people were, what were their relations to one another, and what they were like, Kitty endowed them with the most marvelous and noble characters, and found confirmation of her idea in her observations.

Leo Tolstoy, *Anna Karenina*

Finding confirmation of our ideas in our observations

It usually isn’t that hard to find out what people *do*, how they solve the tasks at hand. It is a bit tricky.

Don’t ask them directly what they tend to do whenever they tackle a specific task. That way you’re going to get detailed descriptions of how you’re supposed to use those same common apps. Most of us have little awareness of how we use these tools in practice. When asked, we often end up just reciting the user manual.

We rarely remember the workarounds we use to make it work for our particular problems. We rarely linger on the persistent issues that we keep tackling and have been tackling for so long that the gestures have become muscle memory. We’re remarkably good at training our muscle memory to solve problems we encounter regularly. The thing about muscle memory is that we don't have to think about it that much. You aren't going to discover any of it in an interview.

Most importantly, we often lie, consciously or unconsciously, about whether we'd pay for any given piece of software.

This is why field visits and exposure hours are so important. (Can be virtual, the world is full of online spaces.) Without the understanding of the end-user that comes from observation, your software project is very unlikely to succeed.

But it could survive for a long, long time because our industry economics are broken.

So broken.

The mare had broken her back, and it was decided to shoot her. Vronsky could not answer questions, could not speak to anyone. He turned, and without picking up his cap that had fallen off, walked away from the race course, not knowing where he was going. He felt utterly wretched. For the first time in his life he knew the bitterest sort of misfortune, misfortune beyond remedy, and caused by his own fault.

Leo Tolstoy, *Anna Karenina*

A bitter misfortune by our own fault

—Finally, we talk about why money and profit aren't enough to ensure software quality.

The economics of tech and software are fundamentally irrational.

A common issue for VC-funded startups and grant-funded not-for-profits is that they keep funding software that has no business being in business. If it does eventually become sustainable, let alone profitable, it's usually due to a monopoly or oligopoly position, attained by driving out all of the paid competitors with a rubbish free offering, or through sheer dumb luck. Millions of dollars will buy you many rolls of the dice.

The story of the past two decades in software is the story of solid apps being replaced with shitty VC-funded freemium SaaS offerings. Those that don't get replaced lose their market to shitty OSS projects that are mostly funded, directly or indirectly, by the same people.

These companies don't have to worry about income, at least not at first. They indulge their engineer's worst instincts for fancy technological innovations. They jockey around dealing with office politics and organisational power plays. They pile on features in the name of 'product-market fit'. They interview users in random and haphazard ways that can never give you solid ideas about their needs. They drive their team hard in the name of growth and disregard *their* needs, which leads to burnout and churn, which in turn leads to shortened institutional memory and an inability for the organisation to retain any actual lesson learned. They juke the stats to make their product look like they're growing meaningfully when what they're dealing with is escalating failure demand because the UX is going downhill, fast.

They rarely examine whether their core idea, the fundamental proposition they are building on, is viable, even once they start running out of money. Pivot, as much of an industry buzzword as it is, pretty much exclusively happens at Medium, which despite regular disruptive business model rotations manages to always ***look the same, work the same, and feel the same***. They keep making small changes that ruin their partners and disrupt the work of their end-users but nothing really changes. Software industry pivots are meaningless because nobody has the courage for actual change.

Addressing an actual need – one that supports an actual business model – makes up for a lot of disorganisation and inexperience. Provided you manage to avoid getting hit by the scorched earth tactics of a VC-funded competitor. I've been involved in horribly mismanaged projects that had stumbled upon a sound product with a solid business model. I've also been involved in projects that were staffed with talented, smart people who were great at their jobs, individually, but who were collectively hung up on an unworkable idea that nobody was going to pay for.

Most of the former projects are doing okay. A decent business model gives you room to grow and time to learn. Those of the latter who are still around won't be for that much longer.

Neither case has much incentive to make good software. What makes the former competitive is almost always pricing and the commodification of their competitors. Nothing will save a company with no viable market.

> Levin, looking at the tiny, pitiful creature, made strenuous efforts to discover in his heart some traces of fatherly feeling for it. He felt nothing towards it but disgust. But when it was undressed and he caught a glimpse of wee, wee, little hands, little feet, saffron-colored, with little toes, too, and positively with a little big toe different from the rest, and when he saw Lizaveta Petrovna closing the wide-open little hands, as though they were soft springs, and putting them into linen garments, such pity for the little creature came upon him, and such terror that she would hurt it, that he held her hand back.

Leo Tolstoy, *Anna Karenina*

Disgust and pity

This is why 90% of all software projects fail: even if you build your product through small projects, one of them will fail, and odds are you don't have the processes in place to recover from that failure and learn from it. A series of small projects effectively has the failure statistics of a large project. Our industry economics and dynamics offer little incentive to recover from these failures.

90% is almost certainly still the number and the popularity of short sprints is little more than an industry-wide tendency to juke the stats. Everything is successful until you run out of money and turn off the lights.

—*Everything was great right up until it wasn't*

There are dozens of different kinds of processes and methods that work when it comes to software development. Several of those work when you're making a product. Several more work when you're making an internal service. Quite a few work when you're making an external service.

There is no one true way, no silver bullet, except agreement *and* disagreement. Those who need to work together need to be on the same page. Those who don't need to be autonomous. You need to figure out ways to coordinate the two, discover differences, and settle them. Everybody needs to exist in both states simultaneously depending on context and relationship. *Autonomy* means that team members need to be *empowered* and it's the most important thing in modern software development.

Tech industry management culture is allergic to any form of autonomy. It doesn't matter how effective it is, how well it's proven to work in research, how many successful products get their start with autonomous development teams, autonomy is a fundamental threat to authoritarian culture.

Authoritarian doesn't mean that everybody who has ever worked in management is a pseudo-fascist or prone to violence. Nor does it mean that they are all xenophobic or racist. (Though some are, to be fair.)

That our management culture is authoritarian means that it's built on the belief that authority is inherently right and that authority *makes* it right. That the hierarchy is more important than facts—decisions are correct because they were made by somebody in authority. Making authority look weak or foolish is a bigger crime than *being* a weak or foolish manager.

I've never encountered an English-language organisation in my life that wasn't fundamentally authoritarian. I've seen a few Nordic organisations that I'd call egalitarian (which would be the antonym to authoritarian in this context). Never an English-language one.

I've heard of egalitarian English-language companies but have never witnessed one or spoken directly to a person working for one. I'm sure they exist, but many of the companies who have presented themselves as egalitarian in the past turned out to be not so. *There is cause to be sceptical.*

Here's a test.

- A manager has slaved over a new plan for the project.
- They meet with their team and tell them what the new plan for success is going forward, describe in detail the idea that is going to direct their next few months.
- One of the employees responds in that meeting by saying that it doesn't sound like a good idea, based on what they know of the problems ahead, and, as described, seems likely to fail.

Should the manager:

1. ***Reprimand that employee*** (in private, to 'preserve morale'). Whatever concerns they have about the plan, no matter how valid, should have been raised through proper channels. Otherwise, you're disrupting the team and the process and that's worse than letting a single bad plan go through.
2. ***Apologise to the team***. The manager made a mistake in deciding on a plan without discussing it with the team first and getting a rough consensus in advance on what was the best course of action.

If your first language is English, odds are you chose option 1 even though it dramatically increases the odds of failure. A solid proportion of English-speaking managers would still choose option 1 even knowing it made the plan's failure a near-certainty. Their goal isn't to successfully execute the plan but to control because control is success for a manager in these organisations.

If you've read any management theory then authoritarian—command and control—management is almost always labelled as obsolete and outdated. Especially in product development where team autonomy is widely considered to be key to a successful product.

If you've worked in a software company you also know that command and control is the status quo, even when they've wrapped it in an 'agile' flag. They've just gotten better at masking it with brainstorming meetings, discussion formats, Slack emoticons, performative 'wokeness' (that never leads to *actual* 'woke' policies) and chirpy HR lackeys. The mask is just a mask.

When it comes down to it, you're still getting death march development implementing top-down dictated features and designs, ultimately based on managerial whim, justified after the fact with cherry-picked data. Dissenters are still punished.

Software companies coast on money. Either through excessive funding, where VCs and other funds throw good money after bad or through high margins after an initial lucky break. Which is great for investors and executives but makes for a miserable industry to work in.

You'd think that a company that's profitable despite having crap software would attract a ton of successful competitors. Except, as I've been trying to tell you, making good software is *hard* even when you're doing everything right. Almost nobody in the industry is interested in doing *anything* right.

(It's amazing that in all of the papers economists write trying to figure out the Solow Paradox they rarely seem to consider 'maybe it's just that all software is bad, we're bad at making software, and we should feel bad,' as a possible explanation.)

Pick a tech startup founder at random. Odds are that they're in it for the feeling of power, not to make good software, a successful business, or even to make a profit. The worst want to be idolised and don't care how much of other people's money they waste to that end.

The software industry just *burns* money. Most of the startups never turn a profit. Many of the big companies (*cough*Uber*cough*) never turn a profit. Even a few of the biggest companies on the planet manage not to turn a profit except on rare occasions (*cough*Amazon*cough*). Those that do mint money, like Apple, Facebook or Google, are profiting through hardware or advertising while making atrocious software. Even when there is profit, that's usually because the monopoly or oligopoly profits are just so goddamn huge that there is no way to spend it all, now matter how badly you mismanage everything.

Startups are the *absolute worst* at this.

People don’t get into the startup racket for constructive reasons. Half of them resemble cults more than they do companies. They certainly aren’t trying to make good software.

Shouldn’t that be the goal? If we’re supposed to make software that’s *valueable*, not just on-time, on-budget, and to requirements, then doesn’t that mean that we need to aim higher? Much higher?

So how do you make good software?

Good software isn't a precondition for success. Otherwise, we'd have more of it. Good software can be made well, every release successful, and still fail because the market size simply isn't big enough. Good software can be driven out of existence by a cluttered VC-funded freemium alternative or a half-baked shoddy OSS version. *Free beats 'good' every time*. Good software can fail if it's marketed incorrectly, priced too high, or doesn't run on an appropriate platform.

Good, successful software is rare because it's *hard*. What's more, I don't truly know how to make genuinely good software. I have ideas—theories on how it's done based on research, study, and observation—but I've never had a chance to put them all into practice. I've, so far, only worked on other people's software projects and other people's ideas, using other people's processes.

The following are just guesses. Theories that I hope to try out at some point. Remember, I'm talking to myself here as much as I'm talking to you and I guarantee you that I'm wrong about a lot of this.

Taste

First, I think you need to develop a taste *for* good software. You need to behave like a painter, writer, or filmmaker and study your field. I'm biased here as I didn't study computer science back in the day but interactive media at an art, media, and design faculty. I was raised in an arty media household. Had one foot in TV and radio when I was a teen. My holiday family gatherings have multiple journalists, authors, and artists attending.

What I'm trying to say is that, because of my background, when it comes to creative work I'm a firm believer in the traditional kind of discourse and structure of film, tv, radio, and publishing.

If your job is to make something, you study the field, its history and current practice. *That's what you do in publishing*. *That's what you do in radio*. *That's what you do in* TV.

That's what I think you should be doing in software. That's what I tell myself as I obsess over old apps, collect new ones, and pore over screenshots and screencasts.

I don't care if you're a backend developer, front end developer, product manager, designer, or Swift/Java engineer, you need to collect good software, try it, experience it, get to know what well-made feels like. *You think a boom operator on a film set isn't a film enthusiast?* Half of them, easy, can recite the best lines from Coppola's *The Conversaton* from memory.

You need to develop a curiosity for classic software, patterns, tropes, and conventions that used to be the norm but have fallen out of vogue. Like a filmmaker studying Eisenstein or a novelist studying Tolstoy, you need to become familiar with the great works of your field: great Mac Classic app, early iOS apps, the peaks of indie Mac OS X, and the die-hards who survived the death of OpenStep. Then get back to the bad software that you're forced to use to figure out which bits of it are praiseworthy. Imagine how you'd redesign a piece of steaming garbage like Google Docs into something pleasant that married its top-notch engineering with considered UX design. Argue with others about how to fix it. Debate.

Software development is a creative field. It has more similarities with filmmaking than it does with bridge-building. Develop a taste for interesting software and then talk to others about them. Discuss, disagree, find common ground and start again.

Find or build small teams

Once you become more familiar with software history you'll notice it. It's quite obvious after a bit of study, even though it's counter-intuitive to those who are a product of the big tech/startup scene. It can take a moment for it to click.

Most of the truly great software in computing history was created by small teams. Many of them were even made by individuals or duos.

Some of those teams grew later on, but even those are a solid minority. Most of the time, if a classic app or service was any good, it was, at least initially, made, developed and maintained by a Small-to-Medium-sized-Business.

It's a mistake to adopt the startup mentality of believing that you have to go from a good idea to a multinational corporation in the space of a few years. That's a risky way to make money if you can pull it off, but even when successful it has led the software to suffer.

Do you truly believe that Dropbox is a better experience now than it was in 2009?

There are exceptions, of course, but you have a better chance of finding and joining (or even founding) the next Panic or Omni Group than the next Apple. There might not even *be* a new Apple. Even Apple isn't Apple anymore as their software efforts as a whole have been declining steadily for the past decade. Now in the era of specialised Software-as-a-Service, there are many more Small-to-Medium-sized software companies around than you'd think.

Start simple

A small team generally also means that there is an upper limit to how complex you can make the app. But you also *always* need to **start simple**:

> A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

John Gall, *The Systems Bible*

The starting point of the app needs to be something extremely simple that also happens to work. I've lost count of how many projects I've been involved with have made this mistake. If you start with a complex idea, odds are that you are going to fail miserably.

This presents a problem if your goal is to 'transform the world' or become famous, infamous, or noted in some way. As we established above, that sort of motivation rarely leads to anything but awful software, let alone anything good.

You can change small bits of the world. Like that bit over there. Make that corner nice.

What seems to work for many is to build an app that helps people out with one thing, preferably one work-related thing because we all like to get paid.

Be the right kind of ambitious

If you have to start simple and stay small (to medium) that means you can't be ambitious in the traditional technology-will-take-over-the-world sense. What you can do is to be ambitious in how you combine the elements you have.

Nintendo is one company that does this regularly. They build innovative consoles and games, mostly using well-established technologies. Gunpei Yokoi, a game designer who worked for Nintendo for a very long time, called it "*lateral thinking with seasoned technology*".

(It's variously translated as seasoned or withered. I prefer seasoned.)

Use technology that's well tested, robust, and has known characteristics and then combine them in interesting designs.

Another way to describe it (and probably the more common, less poetic North American way) is that every project has an *innovation budget*: the amount of 'new' you can safely handle with your resources without increasing the risk of project failure. Most software companies blow this budget out of the water and then keep on piling until it's a miracle they even manage to get started in the first place.

The managers that like to think of themselves as sensible try to pick and choose technologies—balance the budget.

The Nintendo way would be to spend the entire innovation budget on *design*. Use only old, unexciting technology. Make the designs interesting.

'Innovation' here is relative. If your team has been using AWS Lambdas and Step functions for a few years, using Postgres would qualify as an 'innovation'. If all you know is Postgres, then even a little bit of DynamoDB takes a chunk out of your innovation budget.

Use familiar technology whose shape and characteristics you know. Be inventive in your designs. *Be interesting*.

Know your customers

It helps if you have something in common with them. What I wrote above about exposure hours applies here. As Jared Spool says:

> The solution? Exposure hours. The number of hours each team member is exposed directly to real users interacting with the team's designs or the team's competitor's designs. There is a direct correlation between this exposure and the improvements we see in the designs that team produces.

If this sounds like hard work, impossible to accomplish during a pandemic, do remember that the observation can be virtual by scouring forums and the like (a technique pioneered by Amy Hoy) or it can simply be of your own environment.

This is why a lot of the software that's *pretty-good-actually* are development tools. Building software for yourself is if done thoughtfully, building on countless 'exposure hours'. You still need to do research but it's a big head-start.

Effectual reasoning

If you look through the ‘failure’ section of this essay, one of the biggest causes of software project failure is top-down management. Deciding in advance what to do, even before you do the research. Deciding the plan in advance, before you get a proper sense of the work. Deciding in advance what all of your team members should do, before you even discuss it with them, or before any of you has a real understanding of the situation or the tasks ahead.

Saras D. Sarasvathy, who has made a point of studying entrepreneurship, calls this predictive or causal reasoning in her paper “What makes entrepreneurs entrepreneurial?”

> Causal rationality begins with a pre-determined goal and a given set of means, and seeks to identify the optimal – fastest, cheapest, most efficient, etc. – alternative to achieve the given goal. The make-vs.-buy decision in production, or choosing the target market with the highest potential return in marketing, or picking a portfolio with the lowest risk in finance, or even hiring the best person for the job in human resources management, are all examples of problems of causal reasoning. A more interesting variation of causal reasoning involves the creation of additional alternatives to achieve the given goal. This form of creative causal reasoning is often used in strategic thinking.

This type of thinking begins with a specific goal and then you proceed stepwise towards that goal, no matter how impractical or undesirable that goal is later revealed to be.

Entrepreneurial thinking or "effectual reasoning" as Dr Sarasvathy calls it:

> Effectual reasoning, however, does not begin with a specific goal. Instead, it begins with a given set of means and allows goals to emerge contingently over time from the varied imagination and diverse aspirations of the founders and the people they interact with. While causal thinkers are like great generals seeking to conquer fertile lands (Genghis Khan conquering two thirds of the known world), effectual thinkers are like explorers setting out on voyages into uncharted waters (Columbus discovering the new world).

Causal: how can we implement this specific solution to this specific problem?

Effectual: what problems can we solve with the skills and resources we have at hand?

It's about finding something you're likely to accomplish with your current resources. As opposed to attempting to do something and then finding out whether you can accomplish it or not.

This may well mean that the right thing to do is something other than software: books, courses, services, talks, or even hybrid semi-automated services.

Don't be afraid to not make software

At least to begin with.

Dr Sarasvathy outlines a few other principles that she's found to be common among entrepreneurs:

1. *Affordable loss*. The risk calculation is inverted. Instead of trying to calculate the odds of success, you calculate the impact of total failure. "If we try this and fail, how much will we lose? Can we afford that? What's the potential upside?". A lot of the time, decisions that are seen as risky in normal organisations are not risky from an effectuation perspective: odds might be unknowable or extremely low, but the cost is also low and upside high.
2. *Strategic partnerships*. That is, you tend to partner without doing extensive competitive analysis. This is the reason why a lot of entrepreneurs tend to go heavy on open source and open source collaboration.
3. *Leverage contingencies*. In most organisations, anything that's surprising is a threat. In entrepreneurial thinking, anything that's surprising is an opportunity. If your user base turns out to be very different from what you thought, lean into it. If one aspect of your app works wonderfully while the rest is ignored, turn that into the product. In Dr Sarasvathy's words "*surprises, whether good or bad, can be used as inputs into the new venture creation process*."

A five-year plan is never going to work for new software. You can't apply top-down, causal reasoning to the creation of new software. This is also why command-and-control management styles are especially bad under these circumstances, no matter how much of a norm that is in startups or enterprises today.

> "This new feeling has not changed me, has not made me happy and enlightened all of a sudden, as I had dreamed, just like the feeling for my child. There was no surprise in this either. Faith—or not faith—I don't know what it is—but this feeling has come just as imperceptibly through suffering, and has taken firm root in my soul.
>
> "I shall go on in the same way, losing my temper with Ivan the coachman, falling into angry discussions, expressing my opinions tactlessly; there will be still the same wall between the holy of holies of my soul and other people, even my wife; I shall still go on scolding her for my own terror, and being remorseful for it; I shall still be as unable to understand with my reason why I pray, and I shall still go on praying; but my life now, my whole life apart from anything that can happen to me, every minute of it is no more meaningless, as it was before, but it has the positive meaning of goodness, which I have the power to put into it."

Leo Tolstoy, *Anna Karenina*

Going on in the same way

Towards the beginning of this essay I wrote that we don't know the answers to our fields most basic questions. We don't know the answers *as a field* but we generally do know as individuals. Deming's ideas and the germ theory of management have been broadly uncontroversial for thirty years. When Marty Cagan argues for autonomous and empowered teams he is basic that on what many already accept as best practices. Dr Sarasvathy's *effectual reasoning* feels like common sense to most entrepreneurs. We know that software reliability is best served by a judicious mixture of unit and integrated tests. Types help under some circumstances. We know that user-oriented designs, based on research not surveys, is the best way to create software that delivers value. We know that software is iterative and ongoing and best developed in as small bites as is possible. We know that software reliability and usability declines *fast* as it grows in size and scope. Increased very quickly ceases to result in a positive return on investment. We know that many of the problems we're facing have been solved before through designs that we can study and learn from.

We know all of these things but we don't apply them as a field.

I'm tired of this treadmill. I'm not just tired of my part in this charade, where half the projects I work on just don't pan out in any meaningful way, but I'm also tired of *your part in it*. I'm tired of bad software. I'm tired of having to fight my way through app or OS updates to recover destroyed productivity. I'm tired of plodding through awful forms and awful websites. I'm tired of the sheer instability and bugginess of *everything*. I have a high end workstation and most of the apps are harder to use *and feel slower* than the apps I was using fifteen years ago on a budget Mac Mini G4. For example, I haven't found a browser-OS combination yet where it takes less than ten seconds to save a group of tabs as bookmarks on a powerful machine and I've tried them all.

I'm tired of our industry's constant failure to cover the basics.

I'm also tired of our lack of art, of our faddishness, uniform aesthetics, and complete lack of polish. I would like us all to do better. I'm not expecting us, or me, to start making amazing software after decades of mediocrity. That's unreasonable. And you can't get rid of bad software entirely. Making something that's bad is a part of the learning process. We both need to make it and need to discuss it when it happens if we are to be able to improve ourselves. Sometimes things just don't work out.

If you ask somebody who has a decade of relevant production experience to make a two-hour radio documentary on a subject, if you give them the resources that they're used to, their odds of success are not 10%, 20%, 30%, or even 40%. Success isn't guaranteed but you can generally count on a decent listen delivered on time and on budget. *That's what a decade of experience is supposed to deliver.*

Bridges don't collapse while being built 40% or even 10% of the time.

Ask a similarly experienced software developer to make a project that's similar in scope and budget and you're solidly in the grand/large project danger zone in the Chaos Report tables. A 30% success rate would be amazing. If it's on time or on budget, it won't also match the requirements. If it matches requirements then those requirements are probably wrong and it doesn't deliver value. If it delivers value, it probably went over budget or over time, harming its return on investment.

Software development manages somehow to be less reliable than *both* the creative industries *and* engineering while preserving many of the liabilities of both.

We have to do better. We need to have aspirations of *quality*, not disruption or world domination.

I have to do better.

In Anna Karenina we follow the trials of a number of characters. Vronsky, the narcissistic user who ruins lives. Anna Karenina who destroys herself through her naivity and immaturity. Alexei Alexandrovich Karenin, her husband, who supports the status quo, the system as it is, even as it harms him and who is as much to blame for what happens. Kitty Alexandrovna Shcherbatskaya who naively romanticises the horrifying aristocratic society of Tsarist Russia.

The final words of Anna Karenina are Levin's, the character who most closely matches Tolstoy himself in position and outlook. After battling depression and doubt throughout the story, attempting to improve the world around him through both grand gestures and hard work, he finally realises that certainty is impossible—he might not even be able to change the way he feels, works, or behaves, let alone the world around him. What he can change is his intent, his will towards good, and the meaning he puts into the tasks of his life.

Given how rare good software is in the world, the ability to deliver it is probably down to luck.

Probably. Not *certainly*.

I have to hope that the substance of success—the ability to create great software—comes from understanding and effort, not from the vagaries of fate. A lot of it is likely to require an attitude change or change in reasoning. Some of it is down to trial and error. We can put in the effort.

We still might fail, but we can still put the 'positive meaning of goodness' into our work.