OUT OF THE SOFTWARE CRISIS

SYSTEMS-THINKING FOR SOFTWARE PROJECTS

BALDUR BJARNASON



Out of the Software Crisis

Out of the Software Crisis

Systems-Thinking for Software Projects

Baldur Bjarnason

© 2022 Baldur Bjarnason
All rights reserved
Published in Hveragerði, Iceland
Typeset in the Literata typeface

Contents

INTRODUCTION 9

PART ONE: SOFTWARE DEVELOPMENT SYSTEMS 12

1. It was great until it wasn't 15
2. Try not to kill your software development system 19
3. How do you make good software? 31
4. Variability is poison to software development 37
5. Design practice requires experimentation 49
6. Intrinsic versus extrinsic innovation 57
7. Programming as product design 61
8. Research & taste 67
9. Programming as theory-building 71

PART TWO: HOW WE FAIL 78

10. The first Software Crisis 81
11. The agile reaction 85
12. The many kinds of software failures 91
13. Silver bullets 99

PART THREE: FRAMEWORKS 104

14. WTF do you mean when you say 'framework'? 107
15. You need to study exemplary software 113
16. Check your needs 117
17. Understand the paradigms 123
18. Maturity, Complexity, & Layers 133
19. In the end, what else is there but work? 139

PART FOUR: SYSTEM FAILURE (BONUS ESSAYS) 142

20. The Collapse of Complex Companies 145
21. Failure demand: when mistakes improve your metrics 155
22. Disruptive Innovations: Clayton Christensen's mistake 159

WHAT NEXT? 164
23. Further reading 167
24. References 171

Introduction

This book is born out of frustration.

It's common for men (especially men) my age to lament the decline of this or that. New inventions suck. Old inventions that were new when you were young are great. It's a complaint that dates back to Cicero's time. People idolise the culture of their youth. The hardships of the past fade from memory.

This book isn't that.

We did have, once upon a time, apps that were… okay. Possibly good, even. There was a period when many of the major tech companies did regular user testing to prevent regressions in usability, and when a meaningful proportion of the apps available on a few select Operating Systems seemed to have a decent design and user experience.

Software has always been bad, but many of the hardships of poor software have been lost in a haze of nostalgia. These apps may have been better designed, but they also didn't have to account for as much variety in screen sizes or inputs. Most of them didn't support proper multitasking, and *stability* was next to nonexistent. They crashed, ***a lot.*** Accessibility just wasn't a thing. Admiring retro software is like admiring the architecture of a classic building while forgetting that it's a condemned deathtrap and every other building like it burned down, killing every inhabitant.

Great apps are rare.

We remember the designs and how well they worked for those limited contexts but forget the crashes and the data loss. We remember the initial awe at seeing powerful collaborative tools at work, forgetting how hard it was to use in every other way and how limited it was for anything other than collaboration.

My frustration doesn't come from pining for a lost past that never existed. I'm not hoping for a restoration of a mythological golden age of software. We've always been shit. I'm glad for the exceptions, but that's what they are.

My frustration comes from a desire for us to be *better*. I've been making websites for over twenty-five years. I've seen the exceptions to the rubbish norm happen. I know we can do better; I know we can *be* better; I'm not expecting us to fix the unfixable or achieve the impossible. I want us to suck a little bit less at what we do.

I want designs that are just a little bit nicer to use.

I want web apps that are just a little bit more reliable.

I want user interfaces that are just a little bit less confusing.

I don't think that's an unreasonable goal. I believe that software quality is dictated by the system that made it. It isn't decided by the process, team, platform, or programming language, but by the system that integrates those elements into a functioning whole. That means we can improve most software simply by looking at how elements flow from one part of the system to another and how they interact. We can improve without throwing away what we already know, without adopting this week's new process or framework. We can make that enterprise app a tad more pleasant. We can make that productivity web app less flaky. We can make it so that we can find our way through even the most involved intranet.

I believe we can do this. We just have to want to.

Part One:
Software Development Systems

1

It was great until it wasn't

> In most organizations I have been exposed to around the world, bosses and supervisors still make all important decisions. The more important the decision, the more important the boss assigned to make the call. This is especially true of decisions that have financial implications. We still have bosses "above" the working people, filled with staff [...] and supervisors who, without consulting workers, make decisions that dramatically affect their lives. Many layers of bosses and assistant bosses control the behavior and performance of the people below them.
>
> *Dennis W. Bakke, 2005, Joy at Work p. 47*

I am in the habit of complaining about software. Not just the poor experience that individual apps offer but also the overall experience that 'software' gives you. Computers are frustrating, even to experienced users, and it's entirely the software's fault. Almost all of it is poorly designed, unforgiving to mistakes, and frustratingly opaque about what is happening. People blame themselves. I complain loudly about the software.

At one point, when I was full-steam ahead into one of my usual rants, a friend of mine just laughed.

—*That's how modern business works. You just notice it more with software. It's more susceptible to the effects of mismanagement. Makes it harder to hide.*

There is something to that. Companies habitually ship well-made physical products with poor software. Toyota, Apple, Boeing—it's a recurring pattern.

It could be that software is plain harder than 3D-printing a large passenger airliner out of carbon fibre or manufacturing precision-assembled high-performance personal computing devices. But I doubt it. I doubt that software is a greater challenge than mass-producing a modern car. Pretending so is a comforting thought to those of us who work in software. *This job is impossibly hard. We must be so smart*. It's unlikely to be true, but it is comforting.

What's more likely is that the process of making software is more sensitive to the usual consequences of mismanagement: churn, burnout, poor communication, and stress.

Churn is devastating for software quality as it destroys institutional memory and sabotages many of the fundamental mechanisms of programming, which require stability and consistency. Churn in manufacturing or physical product design isn't nearly as disruptive as in software.

Burnout is similarly serious because it has an outsized effect on the productivity of programmers and designers. With physical products, designer burnout will be limited to a single phase in the process. Solving it by rotating out employees doesn't have the same detrimental effect as churn has on software. In software, burnout will affect the entire project end-to-end. Design work isn't limited to one stage. Programmers are involved from start to finish. Burnout from mismanagement can stall everything. The mismanager will blame the victim, exacerbating the burnout, causing churn, devastating software quality, and inspiring burnout in the rest of the team.

Before you know it, everybody is stressed and burning out, year-on-year team churn reaches 100%, and the project is in its death throws.

Meanwhile, the manager will blame everybody except themselves. "*It's **their** fault for being flaky and miserable*."

The way most companies are run, a software team can't win.

They could stick to their best estimates about a project at the beginning when the information provided by management is sketchy and

poor. That would only force them to work themselves to the bone to keep up as they discover the actual needs of the project as the scope expands.

If they change their estimate as the facts on the ground change, they will get a reputation for being unreliable.

If they take the vacation days they are owed—that they have negotiated for in their contracts—they get a reputation for being layabouts.

If they only work overtime when there is a genuine crisis, managers call them lazy and start to constantly manufacture crises to keep them working.

If the team asks for more consistency in the project's requirements or goals, they get accused of being inflexible and not 'agile'. The managers will look for ways to use a process to 'keep the team on track.'

Burnout becomes endemic. Churn increases even more. Managers start to complain about software developers or designers in general. If the managers are older, it's "kids these days don't know how to work." If they're younger, they'll blame software developer culture.

They take steps to counter churn. Not constructive steps. They will be awful, counter-productive, and destructive steps. But they take them.

The simplest way for an authoritarian, command-and-control manager to reduce churn is to prevent people from leaving or selectively hire vulnerable people who can't leave. Immigrants are ideal. They are utterly dependent on their employer for their work visa, and the manager can feel good about themselves as they manipulate and coerce the employee into submission. "*I'm giving them a wonderful opportunity.*"

As the company grows—because you don't need good software to make money in software—recruitment becomes their number one concern. Manipulation and shenanigans of dubious legality only prevent churn so much.

You make up for it by hiring more and more people. Training becomes an issue, so you standardise on a limited set of frameworks and languages, even if they are inappropriate for most of your software projects. This framework-project mismatch increases stress and burnout, which leads to worse churn, so you up your hiring efforts.

Old employees return after recovering from burnout and a stint at a competitor, this time with a substantial pay bump, certainly higher pay than they'd have gotten if they stayed. They don't hang around for long, only a couple of years.

The cycle continues as long as there is money, of which there is a lot because software has *amazing* financial margins. Not everybody is lucky enough to stumble into a business where you've captured a corner of the market with barely functioning software. But that happens often enough for there to be a seemingly endless supply of optimistic Venture Capitalists willing to throw buckets of money on abusive CEOs who don't know what they're doing but talk a good talk.

Except... The money hose is going to stop. It might even be over as you read this. Boundless optimism can only withstand wars, supply concerns, inflation, and a general economic malaise for so long.

It's great while the money lasts. Even the burned-out employees who keep switching jobs get decent pay. Some even get stock options. Then the line stops going up, and it isn't great anymore.

Recruitment gets harder. Churn means people are leaving the industry entirely. Nobody is switching back after a couple of years away.

The software ship begins to sink. The only option you have is to finally try to do the job properly and stop sabotaging the software development process.

2

Try not to kill your software development system

> In spite of the fact that management is responsible for the system, or for the lack of the system, I find in my experience that few people in industry know what constitutes a system. Many people think of machinery and data processing when I mention system. Few of them know that recruitment, training, supervision, and aids to production workers are part of the system. Who else could be responsible for these activities?
>
> *W. Edwards Deming, Out of the Crisis, p. 312-313 (Deming, 2000)*

Most software companies are systems in a death spiral. Software development works for a while, then it goes wrong, and the system never recovers. Everything becomes harder and harder until the system is effectively thrown out and replaced. It happens in a number of different ways:

- The company is sold or acquired. A development system in its final throes gets to expire gracefully. The buyer integrates the product into their own development system.
- The product gets phased out. Sometimes dropped entirely, as even the brand has lost its value. Or, a second system is built from

scratch with a rewrite. A new product is made under the old name.

- The software coasts on like a zombie. Nobody knows how to fix it if it breaks. Everybody prays that it will keep going. Sometimes you keep building on top of it even though you don't know what makes the foundation tick, adding new and new code, like sedimentary layers around a collapsing core.

We know how to build software development processes, and we know how to give the design team its best chance at creating value.

What we are bad at, what lets the software industry down every time, is our inability to integrate all of these parts into a resilient system that lasts. Not because we don't know how to, but because we don't want to. We can't help it, but most managers in the software industry prefer to bolster their own feeling of being in control over a functioning software development system.

Our systems die because we keep killing them.

- We use processes to control employees to extract work from them. We should use processes as a feedback loop to help team members keep themselves on track.

- We substantially underestimate the complexity of state management *in the long term*. We build software whose complexity is only manageable at the very beginning of the process and is guaranteed to go out of control.

- We always take on software projects that are a little bit bigger than we can handle.

- We underestimate the harm in letting bad ideas go into production software. We never fix it in a later version until it causes disaster.

- We let bugs pile up until they cascade into critical failures.

- We let actively dangerous and harmful designs out into the world because we know we won't be blamed personally.
- We drive the team so hard that we end up with code nobody can understand or fix.
- We let feature proliferation kill off otherwise viable software products.
- We blame the team for system failures.
- We escalate project complexity with demands of broad cross-platform support and make up complex needs that only exist because of organisational politics.
- We let the software's dependencies go out of control.
- We optimise all value out of the design process by either dropping all experimentation or demanding that most of the experiments succeed.
- We drive out all design innovation by over-emphasising technical novelty and pursuing unrealistic goals that don't match the organisation's capabilities.
- We don't do any user research, or any research at all, for that matter.
- We treat team members as easily replaceable commodities when they are integral to keeping the code alive.
- We let the programming pop culture run rampant and lose control of everything.
- We treat the value the user gets out of the software *after* purchase as an afterthought.
- Every sprint gets harder, requiring more and more team members and code until something gives and the existing development sys-

tem is thrown out. A new process is started, possibly with entirely new people, and the cycle continues.

Only a few of these are actual causes of software project failure. The rest are symptoms. Some of it is just that there isn't enough money in making good software. Crap software makes enough money or doesn't lose enough to derail the rest of the organisation.

But most of it is because managers think their job is to extract work from the team. They think variations in performance are because of variations in the work they can extract from each employee.

That is not the case. I'm fond of citing Deming's 95% rule, ("95% of performance is governed by the system"), and it turns out there is some evidence to back it.

The authors of the book *Peopleware* have done a few productivity surveys over the years. One of the patterns they found in these surveys was that experience and salary didn't have much of an effect on productivity, but the *organisation did*.

> The fact that they had nearly identical performances suggests that the wide spread of capabilities observed across the whole sample may not apply within the organization: Two people from the same organization tend to perform alike.
>
> *Peopleware, p. 46* (DeMarco & Lister, 2013)

The programmers in the best organisations were often *ten times* more productive than those in the worst while delivering code that passed all acceptance tests. 10x coders aren't a thing, but 10x systems are.

Part of the reason some organisations do better, at least if you believe people like Deming, is they treat the people integral to the system with respect. Make sure there isn't too much variation in workload. Keep the inputs to their work, requirements, feature requests, issue submissions, and the like to a consistent quality. Minimise output variation as well.

You don't accomplish this by tampering with the system or—even worse—doing it by hand. Don't step in and do quality control yourself. Don't rewrite and polish all of the tickets. Don't edit all of the commu-

nications from the team to the client. Adjust the system instead wherever you can. Trust the process to have a feedback loop that will help them self-regulate. If something keeps going wrong, that means there isn't a feedback loop telling the team that it's going wrong. Shouting at them doesn't count.

Even if you don't trust your team, even if they are genuinely incompetent to a person, *distrusting* a software development team is counterproductive. It will only make their performance worse. You need to trust the system, make sure the feedback loops are in place, let them adjust to the best of their ability, and make do with what you have. They may surprise you. Or, they may not. Either way, they'll do better than if you micromanage them.

Most importantly, don't expect the team to do more than it can. *Software is complicated.* If you've landed a big project and it turns out that it's well beyond the team's capabilities, forcing them to get it done will *kill the system*. It will trigger burnout and churn, which will cause a code die-off, design iterations will suffer, and experimentation will be non-existent. The software development system died.

Because you killed it.

Biting off more than you can chew happens all the time, but being in denial about it will choke you. Face the facts and figure another way out. Find some way to reduce the scope or split the project into smaller ones.

Don't kill your software development system.

This leads you to an obvious question:

WTF do I mean when I say 'system'?

The concept is that the world operates as multiple layers of interconnected systems and that the systems, not the elements of the system, are what defines overall behaviour and output.

A company's performance isn't down to the CEO or the employees but the system they operate in. A hospital's overall effectiveness at healing people isn't down to the skill of its doctors but how it's organised and structured.

Much of the field of systems-thinking is focused on theories on how you can change or modify systems—the levers you can pull to effect lasting change—but my focus is simple: let's try not to murder the system. We quite often *begin* with functioning development systems in software, and we don't seem to have problems getting started. But we do have a knack for just absolutely murdering the hell out of the system afterwards.

The best person to explain systems-thinking is Donella Meadows:

> A system isn't just any old collection of things. A system* is an interconnected set of elements that is coherently organized in a way that achieves something. If you look at that definition closely for a minute, you can see that a system must consist of three kinds of things: elements, interconnections, and a function or purpose.
>
> *Thinking in Systems: A Primer* (Meadows & Wright, 2008)

The most important elements in a system are stocks, which form its foundation. Stock is the substance of the system, the part that gives it meaning and a purpose, a store's inventory, and the trees in a forest. Flow is what changes the stock over time. Feedback loops are signals from stock to a flow and are the primary method of regulating a system (Meadows & Wright, 2008).

The elements of a software development system include:

- Working code that the team understands
- Validated designs
- The team
- The platforms underlying the code
- The platforms underlying the designs
- Your version control system
- Your issue tracking or ticketing system

The system is how everything works together. The process isn't the system but provides it with its feedback loops.

It doesn't matter what development process you're using—what flavour of agile or which take on the waterfall. The process can only provide feedback loops to the software development system. Unless you carefully integrate all the elements into a working system, all you get from the development process is randomness and noise—feedback loops triggering disorder and chaos. Since most software projects fail, we can take it as a given that most organisations do not have an integrated, working software development system.

How a system works

Thinking in systems gets tricky. We focus so on the individual elements or view the feedback loops *as* the system that we miss seeing the system itself.

Think of a warehouse full of various products. Deliveries increase stocks. Orders decrease it. Let's assume that it doesn't have any form of automation beyond a simple database that tracks inventory, which means it's up to the warehouse employees to manage the system.

The processes the employees and their managers put in place are *feedback loops*. They are mechanisms by which the state of the Stock triggers actions that change Flow. There are endless variations in the process, each with its strengths and strategies. Which you choose depends. The system is the Stock (goods ready to be shipped), warehouse, inventory management software, employees, and delivery team. It's also a subsystem in a larger distribution system. What makes the system is how these elements integrate.

(Actual warehouse management for retail or eCommerce distribution, for example, is way more involved than this. Apologies in advance to anybody who is actually familiar with the subject. The warehouse is a metaphor.)

Ad hoc management

—Managing the warehouse is up to the manager and their staff, right? That's their job.

You could have an ad hoc process where the warehouse is managed entirely by the staff and coordinated by a manager. Each warehouse employee schedules resupply deliveries of a product when Stock runs low. The manager sets guidelines for inventory levels but leaves the actual details to the employee. If the employee sees a sudden increase in orders for a product that seems to be escalating, they would be free to order a resupply *even though inventory levels are still above the guideline*.

There is a system. It's just unguided and evolves organically.

This leads to variability in resupply rates. Those variations will depend on who is on shift and on who is responsible for which product. This can work for a small warehouse where each employee learns how everything is *supposed* to work from a manageable group of peers. Experienced employees develop a sense of how product orders flow and become very adept at ordering resupplies at the right moment.

This process works, sometimes even very well. But it falls apart as soon as you experience employee churn. The greater the variety in products and product types, the more sensitive this feedback loop is to employee churn, and where the feedback loop goes, so goes the system. Once the warehouse reaches a certain size, the sheer variety in orders and products makes this process too complex to reliably manage. The variability in Stock resupplies increases, which makes it impossible for the manager to plan anything in advance.

Specialisation helps. Workers on the floor might run up resupply issues up the ladder to somebody in charge of inventory, but that still depends on their judgement. Otherwise, the Inventory Manager will quickly get overwhelmed. A workable system can evolve organically under these situations, with its own formal and informal routines and feedback loops, but these can be fragile. It's easy for a novice manager to come in and kill the system, sending the entire organisation into crisis.

Meetings, training, and documentation

—You could add meetings to coordinate the warehouse staff. And use documentation and training to get them all up to speed, maybe? That way churn won't matter as much. Documentation tells them what to do. The meetings tell them when. Right?

You could respond by adding documentation and training, possibly a Warehouse Employee Handbook. Maybe set up daily meetings where everybody reports on what happened the day before and what they're about to do so you can adjust the inventory management policies in response to changing circumstances.

You could call them 'scrums' like they do in rugby. A sports metaphor, right? Those always go down well.

Low inventory for a product is filed as a ticket and goes in a queue. Let's call it the Resupply Queue. Resupply staff works on the queue, resolves tickets when a resupply has been ordered and updates the database when the resupply has been delivered. Delivery staff fulfils orders and adds tickets to the Resupply queue. Orders keep coming in after a resupply has been ordered, so the Resupply team is constantly triaging duplicate tickets. You create another queue to manage pending resupplies and supplier follow-ups. Maybe call it a Resupply-in-Progress queue? RiP, for short. The delivery staff has no insight into the state of pending resupplies (they only check the resupply queue, not the RiP queue), so you add a daily report on pending resupplies to the scrum, making the meeting even longer. There's a meeting to update everybody on the state of the queues. Managing the balance between resupplies and deliveries becomes so complicated that you add another set of meetings to cover issues like how the queues work with each other and ticketing templates. Delivery needs to make sure their tickets have the details Resupply needs!

This has the same elements, with a couple of additions (the queues), as the ad hoc system, but is overall a completely different system.

It works. Kind of. Any friction, such as disagreement between teams, personality conflicts, or supplier issues and the warehouse will

become dysfunctional. Every time something goes wrong, the manager adds a meeting or a process until most of the work in the warehouse is time spent working the feedback loop. The manager sees productivity go down, blames the staff, and starts to 'manage' them: strict bathroom breaks, fewer sick days, more reports and meetings, performance reviews, and employee monitoring. Morale goes down. Churn increases. Productivity slides even further. The manager begins to wonder what it is with the current generation that makes them incapable of work. Maybe they blame unions or 'weakness' in today's youths. They blame everybody and anybody except themselves.

Automation

—*Well, this is something you solve with automation, obviously. Automate everything you can.*

You could buy warehouse management software that automatically orders resupplies when inventory reaches a certain level set by the manager. All the staff needs to do is correctly enter orders and resupplies into the database. The software takes care of the rest. You move as much of the system into software as you can.

Brilliant, huh?

It can be. Provided the software is flawless, has no major issues, and is guaranteed flawless before you roll it out. That happens, right? It isn't that 90% of the software projects this size and of this complexity fail or anything, right?

Right?

Software automation has two important flaws:

1. It encodes a specific strategy into code. No matter how configurable the software is, it will still enforce a specific strategy. The exact one it encodes is often implicit and undocumented, only to be discovered through use. If that strategy isn't what you need for your warehouse, if your research was mistaken about your organisation's needs, then you are sunk. Completely sunk.

2. Any flaw in the software becomes a disaster to your entire operation. These don't even have to be outright bugs. A general UX flaw or mistaken assumption in its design can crater your productivity. These are the kinds of issues that lead that can bankrupt a company.

All that, and you need to pay for the goddamn garbage heap of a software as well. It can work, but it's always a huge gamble. What's your appetite for risk? How much can you invest?

If the company that owns the warehouse has *deep* pockets, a need for massive scale, and the patience to get it right before they switch, no matter how long it takes, then, yes, this could make sense. But even in that case, it would be better if the software automation grew *with* the company as it grew. That way, the software system and the warehouse system could start simple and only evolve into complexity over time.

Checklists

—What then? Training and meetings don't work. Disciplining the layabouts doesn't work. Automation is risky. Are you telling me that it's all impossible? Might as well give up?

You could always use a checklist? It may sound ridiculous. But checklists (and similar abstractions) work. They are one of the most basic forms of system feedback available, but they absolutely do work. They save people's lives before and after surgery. They prevent planes from crashing, and factories from exploding. They are good stuff.

The simplest version would be a literal checklist. Printed, photocopied, and distributed to the team. Whenever a team member handles an order, they go through the checklist, mark off items, and the result tells them whether they should flag the product for resupply or not. When in doubt they go and ask somebody more experienced. If they don't have an answer, the question goes up the management ladder. Checklists add structured feedback loops to an otherwise ad hoc

system. They give the team information they need to adjust their work to maintain the system's desired Flow-Stock equilibrium.

A slightly more advanced version would be a software-mediated checkpoint. You see this all the time with deliveries. The driver scans the package, and if the package matches what comes up on the screen, they and you sign for it. It's a checklist. The great thing about software checklists is their relative simplicity. As long as you avoid turning them into complex automation hell beasts that trigger random primary and secondary systems with every action, you can make a software checklist with a fair amount of reliability.

This strategy for managing systems can become remarkably sophisticated. It relies on humans to press the actual buttons that make the orders or mark a product as sold, but that's the part that can make the process remarkably robust. If the checklist, package signing, or whatever fails, the driver can still make the delivery, and the inventory manager can still pick up the phone to the supplier. The process can be robust to software failure because it doesn't rely on it for interconnectedness.

Processes aren't about *managing* the team or the project. They are *feedback loops* that provide elements of the system—in software that would be the team—with the information they need to adjust Flows and thereby affect Stock. The process's effectiveness depends on the circumstance and how it connects to the rest of the system. There is no such thing as a prefabricated process. You always need to implement and adjust it to your needs. In software development, all processes work up to a point and in specific contexts. When in doubt, start simple and only add complexity if you absolutely have to. You may be told that you need to implement scrum or SAFe, but *how* you do it and hook it up to the rest of the system matters almost as much as the process itself. Details matter.

Some details matter more than others.

3

How do you make good software?

Good software isn't a precondition for success in the software industry. Otherwise, we'd have more of it. Good software can be made well, every release successful, and still fail because the market size isn't big enough. Good software can be driven out of existence by a cluttered VC-funded freemium alternative or a half-baked shoddy Open-Source-Software 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 software is rare because quality is relative to the needs of its intended audience. Hundreds of books have been written on the subject. Many great; many contradictory. Best practice in one context becomes a bad practice in another. Software development is large, and there is no one true answer to how to do it well. One size doesn't fit all. No one solution will solve even a single problem in every circumstance.

Making software that's *great* is difficult and intimidating. When we feel stuck in a dysfunctional organisation and haven't been given enough resources to do everything the right way, the thought of being able to make great software seems impossible.

But almost all of us are in the position to consistently make software that's *not-bad*. We can avoid the big mistakes that lead to a proliferation of bugs and bitrot. We can adopt modern management techniques to improve teamwork and outcomes. We can stop undermining the design process.

We can stop sabotaging ourselves and consistently make not-bad software.

If we make not-bad the baseline, good will come with practice, even when the circumstances aren't optimal, and great will come with luck.

There are many actions we can take and practices we can adopt to get there. The process is a part of it, but we already have plenty of processes that work. We need to get out of their way.

Where to start? What is the first step we can take to reliably make software that inspires the end-user to say *huh*?. *Interesting*? Not *bad*.

The first step is simple.

The *step* isn't simple. It's genuinely quite difficult. It's the *software* that has to be simple or at least start that way.

Simple software tends to work. We know how to make simple software that works.

Crucially, it needs 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.
>
> **Galls Law**, *from The Systems Bible (John Gall, 2002)*

An app needs to begin as simple as it can be while still working. If you start with an already complex idea, you will fail miserably.

Then the struggle is to *keep* the software simple.

This isn't just a question of less code making it easier to find all the bugs. Smaller codebases provide fewer chances for interacting bugs to cascade into a major flaw. They are easier to understand and reason about, meaning that a new developer is more likely to build an accurate mental model for how it works, which means that more of your team is able to work with it and fix its flaws. They tend to come with

simpler user interface requirements. This makes for simpler designs, which are easier for the designer to do well. They are easier to test in terms of automated testing of the code and in terms of user testing.

Or, if you're looking at it from the other end: the effort it takes to deliver correctly functioning software increases nonlinearly with its complexity. You don't know in advance just how nonlinearly. You might think a feature would only lead to a marginal increase in effort. And, if you're lucky, you might be right. Or, it might require a complete re-architecture of the entire app for it to work. Usually, a developer can tell in advance which it is, but not always. The difference between simple and even just moderately complex software can be dramatic.

This might lead you to think we should design a software development methodology or system that emphasises simplicity and simple software. If simplicity is so important, then that system would end up taking over the world!

You'd be right.

The industry did try that. It did take over the world. They called it *Unix*.

The history of the Unix operating system is quite long. It began as an in-house operating system at AT&T and ended up getting licensed broadly and widely copied over the decades. A big reason behind its success is what has been described as The Unix Philosophy, and the summary that most use is this one by Douglas McIlroy:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.
>
> *Basics of the Unix Philosophy, The Art of Unix Programming* (Eric S. Raymond, 2003)

The original Unix systems followed this principle religiously. It served its descendants well and has resulted in enviable reliability and consistency. It and its descendants have a near-global dominance of the software scene.

Most servers run Linux, which is a Unix-inspired system. Android phones run Linux as well. The operating systems of both Macs and iPhones run on a Unix descendant called Darwin. Your TV probably runs on Linux. Windows has a Linux-compatibility subsystem because it's become the primary platform for many developers.

You'd think that once people saw how well the Unix philosophy worked—with programs that did one thing—they'd try to apply that philosophy everywhere. Life isn't that simple. Even beyond the incompatibility that the Unix philosophy has with the irrational economics of software (feature-proliferation is antimatter to Unix's matter), you quickly run into a host of problems.

The biggest one is that graphical user interfaces, whether in a native or web app, do not lend themselves to the Unix Philosophy. A key aspect of the text-based apps that the Unix philosophy focuses on is that the operating system makes it easy to shuttle (or 'pipe' in the jargon) text from one program to another. Everything is a file. Everything is text (even when it isn't).

Graphical systems don't have that elegant interoperability. Imagine trying to build a Photoshop replacement out of dozens of individual apps? Even if we assume that we can get them all to work with the same image format, modern operating systems just aren't designed to fluidly move files and data from app to app in a way that would make this usable. Some hoped that browsers would be able to break that trend, but that doesn't look likely.

To accomplish consistent simplicity in program design in a graphical operating system (which is all of them today) would require redesigning the operating system from the ground up to support it.

This is a huge task—a major software project—and those are extremely risky and likely to fail. It would be the opposite of the kind of software it's supposed to support.

We can make our software as simple as we can. Make our applications work with standard file formats when we can. Help the user switch more easily between relatively simple but specialised apps to get their work done. Support all of the inputs and outputs that come with the operating system—copy and paste, default apps for file types,

URL scheme handlers—to make that switching easier. Interoperability should let our apps do less, making them more reliable and less likely to fail.

Just this principle should increase our chances of making reliable software substantially. But for that to happen, we need to work in organisations that enable it, under management that genuinely understands and cares about software quality.

And because software quality is a people problem, it needs to genuinely understand and care about *people*.

A tall order, I know, but you can begin in your corner of the world.

4

Variability is poison to software development

> The key idea is the elimination of the virus of variability. The key lies in the improvement of process capability. This idea was taught widely during World War II and was important to the successful development of America's mighty war machine.
>
> *The Germ theory of Management* (*Tribus*, 1992)

The germ theory of management

There have been many theories that claim to explain everything about management. *Learn this one magic rule to become a great manager!* They all claim to be revolutionary, solve everything, be used by the greats in the industry, and be completely unknown secrets you can access for the low price of a hardcover book in an airport bookstore.

No theory explains everything, but I have found that one particular approach has helped me think much more clearly about the problem of management and strategy. It can be found in the writings of Deming, Juran, and Sarasohn. Myron Tribus has a solid go at defining it in the paper I quote above.

It is a particular application of systems-thinking to management: the idea that the manager's job is to manage *the system*, not the employees. The idea is that a manager's role is to reduce the variability in the system so that its performance becomes predictable. A predictable system with consistent capabilities makes for better plans. Consistent processes and output become known quantities you can reason about, and being able to reason about something is key to forming successful *strategies*.

There is more to management than that, but *reducing variability* is a strong foundation. This is especially important in software because software is *extremely* variable and has failure rates far greater than most other forms of economic activity we have in modern society.

Even restaurants and book publishing are far more predictable than software development. The economic risks of those industries aren't down to variability in production—almost every restaurant, no matter how shitty it is, consistently delivers its menu with little variation. They are horrible businesses to be in, but the risks are in the reception. Books don't fail because the typesetting process failed halfway through, turning the last few chapters into unreadable inkblots (though accidents happen). Books fail because, after their production process has been completed in an almost disappointingly predictable manner, nobody cared enough to buy a copy except the author's family, who then all lie about having read it.

Only a fraction of the software industry has to deal with the reception risk that publishers face. Most software projects are internal projects intended to solve specific problems for the parent organisation. Most of the rest are business-to-business services that solve problems for *other* organisations. Consumer software, like games, is especially risky because they combine the reception risk of book publishing and other media with the production risk of software.

It's rare for a trade publisher not to be able to publish a book that they want to and have paid for. It has happened, usually for legal reasons, but it's rare. Whereas in software, we routinely run into organisations that *want* to deliver a piece of software, have the *funding* to build it, the *staff* needed to work on it, and still *fail* utterly at completing the

project in a way that delivers working software. The risks involved in the software industry are on a different level.

It's the management's job to try and reduce those risks. Otherwise, the variability inherent in software production will overwhelm any attempt to form a coherent business strategy.

How exactly you can do that will vary from field to field, project to project. A process that works well to manage the risks in developing a new service at *Amazon Web Services* might increase the risks of developing an intranet site for a mid-sized company. One size definitely does not fit all. Are you trying to build utility-level software services, or are you trying to build a single-OS productivity app?

From a systems-thinking point of view, the goal of the development process is to make sure that the stock and flow of the system are consistent and predictable. That lets you plan and form strategies.

In software development, an important *Stock* is working code. Not just any working code. It has to be *working* code that *serves the need it was designed to and is understood by the team members well enough to modify, adjust, and fix as needed.*

Whenever the team writes more code that works, fulfils the needs, and is understood, stocks increases.

If the team doesn't understand how the code works— doesn't have a theory of how it functions—then that code will need to be rewritten. Maybe not today, maybe not tomorrow, but someday. It's zombie code —bitrot. You just don't know it yet.

That means that stocks flow out when:

- Needs change, which causes requirements to change, and means the code is now useless.

- The environment changes, so the code no longer functions or is no longer reliable.

- The last team member around when the code was written leaves. It doesn't matter how much training their replacement gets, they're never going to fully understand the code, and any time they fix or extend the code they will have to replace bigger chunks

of it than the predecessor would have, taking more time and effort. The code lost value.

- The team implements a feature, but they discover later that it was put together and released in haste. Nobody has an actual working theory of the code in question. It seems to work in production, but now everybody is afraid of touching it. The code is disposable with no future value and not actual stock.

Code is prone to losing value for almost any reason, but the outflows for most software projects are predictable, which means you can manage them.

Software platform

One source of variability in software development comes from the platforms your projects will run on. You have to deal with variations in the system's environments and variations between system versions. Operating systems tend to run on many different devices; each with its own behaviour. Both language environments and systems are developed over time and come in many different versions and releases.

An immature language environment that changes in every release is a substantial source of uncertainty, risk, and variability. Every change devalues your code—stocks flow out—because you need to adjust it to match.

When you support many such environments, such as multiple operating systems or different languages, those risks are *compounded*. Supporting three operating systems means your code gets devalued through platform changes three times as often and if the implementations share any code, those changes will interact, causing more errors and bugs. The changes *don't share a direction*. A feature might get simpler on one and more complicated on another. Platforms both diverge and converge, at the same time, in different areas.

Developing one app that works on two operating systems is often *more* than twice as hard. You can mitigate the risk by targeting a single cross-platform environment that abstracts away the differences

between the platforms. That makes it easier to deliver cross-platform software but comes with risks because it adds an entire additional layer between your code and the base platform. All dependencies in software are risks. The cross-platform framework is itself a source of variability. Another is that abstracting the underlying platform away means your app won't integrate as well with that platform as it otherwise would have. The OSes have different defaults, different affordances, and different integrations, so making an app that's equally easy to use on both is a tough problem to crack. No matter how you look at it, going cross-platform is always going to be more complicated than not. It often turns out to be an extremely costly strategy for software companies.

This is a big problem for web apps. Even when delivered as native apps through technology like *Electron*, they fail to integrate fully. Their menubar is often either obviously not native or bare-bones. Their context menus are more parodies of the native context menu than actual replacements. They might be a single platform in terms of code and APIs, but they are a user-experience hinterland for platform integration. Making a web app *design* that works equally well, *as an app*, on Windows, macOS, iPadOS, iOS, Android, and desktop Linux is a problem that requires imagination and originality. Sometimes it's best to go the other way and lean away from platform norms and into web norms. Focus on creating a great user experience *within the browser context*. Take advantage of the fact that the web has its own norms and conventions, and stop pretending to be a desktop app. It can work with the right design. A web app that works well in a browser tab and *then* gets installed as a *Progressive Web App* on the desktop is framed as a well-made web app that now has better OS integration. A web app that's packaged to look like a native app from the start will be understood as a *poorly native app*. Framing matters a lot in all design, and software design is no exception.

More platforms mean more complexity. More complexity makes project failure ***always*** more likely.

This is pretty easy to see in practice. Servers tend to be some of the most reliable pieces of software we have. Not just because that's their

selling point (reliability would be a selling point for *a lot* of software), but because it's much easier to control all the various sources of variability in a server-side project. You can usually pick a single version of a single server platform with long-term support and a single version of a programming language runtime. The only source of variability is you—your own processes and requirements.

Server-rendered web services deserve a special mention here. Web apps where all of the business logic runs on the server and the user interface is mostly server-rendered HTML with a bit of JavaScript are some of the most predictable work environments available for delivering software services with graphical interfaces. They bypass most of the issues with cross-platform development by running all of the value-generating code in controlled environments. There is a good reason why Ruby on Rails and Python's Django are popular choices for delivering intranets and mid-sized web services. They might not offer the same power as a native app, but they tend to be the graphical software projects with the least amount of variability. Low variability opens up its own possibilities.

Even in more native apps, small teams deliver reliable quality single-OS applications. Award-winning apps such as Acorn for macOS or Overcast for iOS are each developed by a *single developer*. The costs involved in making cross-platform apps with similar functionality, even using a framework, would likely be *five to six times higher*. I doubt you could deliver a version of Acorn for both macOS and Windows with anything less than a five-person team, maybe even a ten-person team. You would have to make numerous usability sacrifices to accommodate the various design and user interface differences between the platforms.

The biggest win you can get, in terms of complexity and cost, is to minimise platform variability. If you can get away with targeting just one platform and ignoring the rest, do so. Conversely, a key to massively reducing software production risk for your project is figuring out an overall product strategy that works with only single platform support. Bear in mind that the web, Electron apps, Web apps,

Windows, macOS, iOS, and Android are all separate platforms whose APIs and interface norms differ.

This is a consistent challenge for web apps. Progressively enhancing from a simple but functional baseline tends to help. Choosing only to support so-called *evergreen* browsers also helps. But the downside of the web's massive reach is a requirement for broad platform support and a variety of user experience contexts. Web app user experience design is a challenge that shouldn't be underestimated.

iOS and Android have disadvantages of their own: they are controlled platforms. Every app you develop for them runs the risk of being blocked from distribution for spurious or opaque reasons. You cannot rely on consistent release and distribution processes when targeting these platforms, bringing considerable *business* variability and risk into your project that's entirely out of your control.

There is no easy answer to platform choice.

Dependencies

Dependencies—third-party software modules imported into your project—present similar risks. Any time they change, your own code loses value. Stock flows out.

It's the Achilles' heel of most web apps. Unlike the platforms you build on, your dependencies can grow unbounded. Importing a bundle of a hundred or so packages can help you reduce the time it takes to deliver the initial version of your project from weeks to days, if not hours. But with that shorter project startup time comes a downside: it magnifies the variability your project has to deal with almost exponentially—each of your dependencies has dependencies of its own. With every additional one, the overall project's risk increases.

Projects with a large number of dependencies feel great, to begin with. The dependencies make it easy for you to create a lot of code that works, does the job, and you understand. Your prototype is up and running in record time. The project's initial success means more feature requests and more iteration. Then one of the dependencies gets updated, and you face a choice.

You can ignore the update—after all, it isn't security related—and carry on. Your code has been devalued, but the overall Stock level in the project is fine. You understand the code you have, and it still works, so you put off the update. You start to fall behind on all your dependencies, but that's fine. You don't need any of the new features. The value of your code is dropping but not zero, which is the value it would have if you had to rewrite it. Then a dependency of a dependency gets a security update. You are faced with the prospect of upgrading a hundred dependencies, which have all changed their APIs in the meantime. Some of those changes are mutually incompatible, and none have backported the security fix to earlier versions. Development of the project stalls. Managing your dependencies begins to suck up all of your development time. You have no sense of how long anything will take because you have no idea what will happen with the dependencies. Your code no longer has value as stock in the software development system. It has become a l*iability*.

You could also religiously update your code to use the latest version of each dependency, turning dependencies into a constant *overhead* throughout the project's lifetime. The value of the code you make is slightly devalued, but by what's hopefully a manageable margin. Your job becomes that of the dependency wrangler. Third-party code is developed at its own pace with its own priorities, and it's unlikely that it will match yours. Some dependencies will inject security flaws into your code. Others will represent constantly moving targets that require constant changes in your code. Some of the dependency updates will, again, be mutually incompatible. Two packages might get updated, but one depends on an earlier version of another. Your job won't be to write code but to wrangle a mess of misbehaving third-party code.

Sometimes the value of a software project is short-term, and the sharp devaluing that comes from many dependencies isn't important to the company's finances. Sometimes you are working on a software project you hope will run for *years*, and the compound devaluation that comes from all those dependencies would threaten the company's viability. Most projects are somewhere in between. Remember that

each dependency is like a loan. It gives your software development system Stock right at the beginning when you need it, but the repayments can destroy you.

The economics of tests and types

Computer science is, understandably, preoccupied with *correctness*. How to write code that you can be reasonably confident is correct and free of defects.

It's important work.

Two approaches that come up as solutions in this space are unit tests and various kinds of typed programming languages.

Both of these approaches are useful beyond just ensuring correctness. They improve the in-flows of stock in a software development system and reduce outflows.

Unit testing and types increase the value of your code and protect it from losing value.

They do so in different ways, which is why, ideally, you want both. Their actual correctness doesn't matter as much as you'd think.

Unit tests work like per-module checklists and provide current and future coders with a tight runtime feedback loop exclusive to that one module. It won't guarantee the software's correctness as a whole, but it gives you an early warning of failure. If the unit test fails, the software would have failed, but conversely, a unit test succeeding doesn't guarantee correctness.

Because they are checklists, unit tests also provide documentation for how the code in question is supposed to behave *in situ*—in the dynamic context of running software. As a checklist, you can also use it to guide development. Write the checklist before you start, and you'll know when you're done and have working code—all bundled with an automated checklist that will protect it from being devalued by

changes, such as those that come from shifting platforms or dependency updates.

All of which is to say that test-driven development is as much of a modern management tool derived from systems-thinking as it is a Computer Science tool for proving correctness.

Types perform a similar role. They generally don't operate as runtime checklists but let you perform automated checks that tell you whether the software is cohesive and that the modules fit together as designed.

It should be obvious why this *complements* unit testing from a systems-thinking perspective.

Types, depending on the language in question, can help you automatically check whether the platform or the project's dependencies have broken their interface contract in a way that matters to you. It doesn't solve the problem posed by dependencies changing but will aid efforts to update your code. Types, therefore, mitigate the devaluation that comes from shifting dependencies.

Typed programming languages help prevent dependency changes from devaluing your code.

Together these two approaches can go a long way towards minimising the value deterioration that can come from the environment shifting around your software project.

Instability comes in many forms. Some of it comes from the outside. But, sometimes, it comes from the inside: from your team or even yourself.

Sometimes variability is baked-in, and you have to figure out how to make that work in your favour.

No time for tests

Skipping tests to save time is akin to setting a good portion of your budget on fire. Unit tests and types have a *business purpose*.

From a programming point of view, the purpose of unit tests and types is to attest that the code is free of certain kinds of defects. Programmers will debate endlessly which is superior for this purpose, unit tests or types, and whether you truly need unit tests if types prevent the same kinds of defects.

While defects are very important to any software project, the *business* benefit of unit tests is to preserve the future economic value of the code. This isn't quite the same thing. Unit tests make the code resistant to being devalued by change.

In software, the biggest threat to the economic value of code is that everything is always in flux. The business purpose of unit tests and types isn't to prevent defects but to prevent *variations* in quality. Unexpected *improvements* in quality are just as undesirable as discovering a bug. Other modules may well be depending on the previous unimproved state. Unit tests serve as an early warning check that tells you something changed when it shouldn't have.

Code that doesn't have unit tests does not have a runtime checklist of functionality or the interface contract each module of code offers to the rest.

Code that doesn't have types cannot provide assurance across the entire codebase that those interface contracts are being honoured. If you're working in a programming language that doesn't implement type checking, you will need to figure out another way to check that the interface contracts are being held up across the codebase.

You need both for the code to have economic value and not be a disposable waste of money.

Unit tests and types are first and foremost business tools. Dropping them threatens the economic value of the project. If the programmers on the team prefer other methods of proving that the code is correct, they can implement those *in addition to unit tests and types*, provided time and budget allow.

5

Design practice requires experimentation

> Thus, the *only* measure of the value of design is its economic value. Our designs are not important because they are new, or because they are beautiful, or because they are innovative; they are only important if they make money. Our designs will only make money if we create recipes to turn material, labor, and overhead into valuable functionality better than our competitors do.
>
> *Managing the Design Factory p. 11 (Donald G. Reinertsen, 1997)*

Economically useful information

The various definitions of design, especially at the management level, alternate between the facile ("make it pretty") and the reductive ("design is how it works"). The long-standing debate around what, exactly, design is in the context of software—while essential for the field of "whatever it is designers do for software" to resolve its decades-long identity crisis—isn't helpful when it comes to figuring out how to make our software projects less likely to fail. That's why I prefer Reinertsen's definition from the book I quoted: design is *economically useful information*.

In systems-thinking terms, design is a stock that either increases or prevents decreases in the value of other stock.

Reinertsen's books focus on the development of physical products. Applying their ideas to software takes some work. The books are also pragmatic to the point of being ruthlessly mercenary. Hence the no-nonsense definition that emphasises economic value.

The example he uses to explain his definition is that of the Sony Walkman. The design of the Walkman is itself economically useful information. That design created a new product category by repurposing existing technologies and capabilities at Sony. The Walkman let its buyers listen to music in a new way. This made Sony a lot of money.

The example I prefer is the disposable paper coffee filter. Even before the creation of filter coffee makers, just the paper filter with a simple funnel unlocked considerable economic value. Filter coffee was more convenient, faster, easier to use, and *more consistent* than earlier methods for making coffee. Not only did the paper coffee filter become a ubiquitous product, but its invention enabled the global dominance of coffee as a beverage, which means that its design had substantial positive economic externalities. One design led to increased coffee sales and the invention of coffee makers as a popular household device category. The concept alone of the paper coffee filter was information of enormous economic use. It created a simple but consistent system for making coffee entirely through design, with feedback loops such as the amount of coffee grinds in the filter, the water level as you pour over, and the heat of the water.

Design can also aid management by reducing variability *elsewhere* in the process. If we keep to the coffee example, one of the consequences of filter coffee and filter coffee makers was to create a system that reduced the variability of the coffee-making process. You didn't need any in-house expertise to deliver tolerable coffee at scale. This lets any business offer coffee at a level of quality that was entirely a consequence of materials and tolerances, not training or expertise. If you wanted to offer cheap coffee you bought cheap machines and ingredients. If you wanted to consistently deliver good (though maybe not great) coffee, you could invest in coffee makers with tighter con-

trol over water temperature and flow and better ingredients. The design of the coffee filter resulted in one of the most manageable processes in the hospitality industry. All from just information: a pattern and instructions for how to make something with it.

The information generated by design processes can generate economic value in many ways. It can reduce variability, open up product categories, and invent innovative uses for existing technologies. Decoration isn't high on that list and isn't that big a part of the design of software. It's important, sure. But it's the least important thing that design does for software. If you prioritise design work in order of economic usefulness, aesthetics sink down to the bottom. It's part of what design does. It isn't what you hire designers to do.

What do you hire designers to do?

A designer's job is to create information that's economically useful to the organisation employing the designer, given the resources and capabilities it can access. That involves solving a problem, with a product or a service, for a group of potential customers, external or internal.

How exactly the design process interacts with software development is going to vary. Sometimes the design work is all upfront. Sometimes it's tightly integrated with the overall development process. It depends on the organisation, product category, platform, and company strategy.

Common to all these processes is *experimentation*. You will need to do experiments and tests as a part of the design process. This is separate from your testing of the code itself because experimentation for design *is* design. The point of the design process is to generate useful information. Without validation through experimentation, what you have isn't information but guesswork.

You are going to need experiments. Some will be user tests, some A/B tests, some are tech tests to see if a design can be implemented, and some will be prototypes for the team or an in-house audience to assess. These are necessary because without experimentation you will never see any *genuinely* new designs.

User tests are necessary because they are the only way you can be sure that the software's design is genuinely doing its job. But they are also designed to generate insights and observations on the user and their needs.

No matter what your overall process or product is, you're going to need design and tech prototypes, and you're going to need user testing. The rest is an optional extra.

Design experimentation (or the 50% rule)

The purpose of design experimentation is to generate more information. To maximise the productivity and effectiveness of your design team, you need to maximise the information generated by each test. Reinertsen, thankfully, has already figured that part out for us.

> This suggests that *we cannot maximize information generation by maximizing the success rate of a test*. We maximize information content of the test by approaching a 50 per cent failure rate. *Our testing processes need to have an adequate failure rate to generate sufficient information*.
>
> *Managing the Design Factory p. 71 (Donald G. Reinertsen, 1997)*

Your prototypes, experiments, or user tests will not generate any *economically useful information* to speak of if:

- You are confident they will work.
- Or, you are certain they will fail.

In practical terms, if you aren't prototyping, aren't testing, and your team's interfaces go from wireframes to mockups, to implementation, to shipping with no errors, no failures, and no issues, then you aren't generating any economically useful information. You aren't designing; *you're copying other people's designs*.

This is strategically and tactically useless and has no business value whatsoever. You might as well outsource your design process to a photocopier.

You need experiments.

Some of them need to fail. Some need to succeed.

Many managers from a financial background are tempted to try to maximise their Return-On-Investment in design by minimising the number of failed tests and prototypes. Paradoxically, this attempt to reduce wasted work instead *increases costs and waste*. Without the failures, your tests aren't generating valuable information, and your design processes aren't delivering any value. You're overspending on design and getting nothing out of it. You are spending money on design, but that money isn't resulting in any inflows of Stock into your software development system. It's wasted money.

Similarly, many managers from a technical background tend to view the primary purpose of design as enablers of technical innovation. Instead of seeing design as delivering value, they view it as a necessary cost centre whose only role is to make a pretty widget that triggers the underlying technology that does the actual work.

Treating design as a cost centre becomes a self-fulfilling prophecy. It won't generate value if you don't expect it to.

The opposite end is also dangerous. Focusing too much on innovation, and trying unfamiliar and unusual ideas, can lead to a series of design failures that have taught you nothing and generated no value. When you let designers do their job, when they are allowed to make experiments that fail, and when they get to apply their skills, their work will be what makes your project successful and profitable.

Stick to the *50% rule*. If roughly half the design tests fail and half succeed, you're on track to get as much value out of the design process as possible.

Reducing variability: design and the tight feedback loop

Design is inherently a series of one-off processes. Without some risk and failure, you aren't generating any valuable information. This presents a management problem because the job is to reduce variability. There are few things more chaotic and unpredictable than one-off processes.

What most managers in software do is practically sit on top of the designers, have opinions on everything they do, provide designs of their own, and micromanage the hell out of everything they do. Even to the point of drawing up the user interface sketches themselves and telling the designers "make this design work". Anybody who manages a design team like this would get just as much economic value for their company as an eight-year-old schoolchild with crayons.

The systems-thinking way is to turn the design process into a system. For that, you need feedback loops that let the designers manage the flow of work themselves. The good news is that *this is what designers do*. Their training and experience are exactly this. The manager's best contribution is to let designers direct the design work. That doesn't mean that the designers are left entirely to their own devices. The environment and context of the design work have a huge impact. As does the pacing and structure of the overall project.

You can improve the feedback loops in the design process in a variety of ways. Some of the more common include:

- Short batch sizes. Keep each design-and-experiment cycle short. This tightens the feedback loop and helps you discover useful information earlier in the process, which helps control variability.

- Fixed batch sizes. Give each cycle a fixed duration. This can help focus the tests and experiments. The exact duration of each cycle depends on the project, and this tactic isn't universally beneficial.

- Standardise as much of the process as you can, how each prototype is put together and assessed, and the tools used to make it.

> Formalise your user testing process. Each cycle may well be a one-off in terms of what it's trying to accomplish, but that doesn't prevent you from standardising *everything else*. One-off doesn't have to mean ad-hoc.

There are more, but short batch sizes and iteration are the mainstays. They work remarkably well, and design process variability can become quite reasonable.

Getting the design process under control and making it effective at creating value is one thing. Fostering and creating innovation is another.

Nobody is consistently as good at product design innovation as Nintendo.

6

Intrinsic versus extrinsic innovation

We've just gone over the role design has in product development: economically useful information generated through prototyping and testing.

The technology used in the design process and the eventual product doesn't have to be *new*, which is an insight that is arguably the only reason Nintendo still exists as a company. It's one of the few companies that do this habitually. They build innovative consoles and games using well-established technologies. Gunpei Yokoi, a game designer at Nintendo, called it *"lateral thinking with seasoned technology"*. It's been described as an approach that fosters creativity with mature tools (Scott Warren and Greg Jones, 2014)

(The phrase is variously translated as seasoned or withered. I prefer seasoned.)

If you have to start simple and stay small—smaller than the rest, at least—that means you can't be ambitious in the traditional *technology-will-take-over-the-world* sense. Be ambitious in how you combine the elements and resources you have. Instead of investing in technological wizardry with an army of programmers and engineers working on accomplishing the impossible, you let your designers drive the process. This can be cost-effective as design innovation isn't generally a ques-

tion of team size but more about having the resources and *freedom* to experiment towards a flexible goal. More seasoned technologies tend to be reliable and consistent, which helps with costs, design (when the tech is a known factor, prototypes can focus on design), and strategy.

One part of the process that Nintendo uses is that, unlike most companies who make software, they don't decide on their exact plan before they start. They have a broad direction—a very general concept, maybe a specific property like Mario and a general desire to make something enjoyable—but they don't have a game in mind from the outset. Instead, the first step is to generate ideas. They assemble several small teams of people who pitch ideas and test them out. They have a process that helps them converge the ideas of those teams into a single product through feedback loops. They work with their existing software, design, and hardware development systems to create a genuinely new experience (Stuart and MacDonald, 2018).

Most other companies look at their goals first, decide what outcomes they want (or need), and then work out ways to change their systems to accomplish those outcomes. They decide exactly what product the team will build right at the beginning. This is the norm. It also rarely works. Most software projects fail, and this is one reason. *They keep tampering with their software development systems*. The idea is, if it works, they will have a genuinely new and innovative capability that is intrinsic to their system and therefore highly repeatable. Success means big profits. But failure is the norm.

This is intrinsic innovation

Intrinsic innovation is when your path to innovation is by *altering an existing system or building a complex one from scratch*.

The problem with deciding on strategically desirable outcomes *at the beginning* and then changing the system to achieve those outcomes, is that tampering with systems, especially software development systems, has unpredictable results. It is rare for any organisation to have warranted confidence in its own capacity for intrinsic innovation. Companies like that are rare and revolutionary. You almost cer-

tainly aren't working for one. Most successful companies have stumbled onto a functioning software development system and are in a constant internal battle to preserve it. Attempts at intrinsic innovation have sunk entire companies. It's a good way to turn your software into a mess.

The Nintendo alternative is to first look at what kinds of outcomes your preexisting systems are capable of with, at most, some incremental changes. This is why they prefer to use seasoned technology. This is why they begin by generating ideas. This is why they converge on designs and projects that they already know are within the capabilities of their existing development systems.

This is extrinsic innovation

Extrinsic innovation is when your path to innovation is by *exploring what new* ***outcomes*** *a preexisting system is capable of.*

Extrinsic is inherently less risky than intrinsic but is just as capable of novel software. It turns innovation into a design process: idea, design, test, repeat. The problem for most organisations is that extrinsic innovation is a *creative* process. You need space for ideation and experimentation. Management also needs to live with not being able to set specific strategic goals. Instead of managing the organisation like a ship, steering it in one direction or another, they need to concentrate on managing its capabilities, nurture the systems it has, and plan around the *kinds* of outcomes the company is capable of instead of specific ones. Let the organisation go where it needs. By building on the existing system, extrinsic innovation gradually expands the system, both piecemeal and in layers. Going with extrinsic innovation doesn't prevent your system from growing, but it does tend to make that growth slower and more sustainable.

Counter-intuitively, entrepreneurs generally favour extrinsic innovation. A core aspect of most entrepreneurial thinking is to work with the resources available and build on them (Sarasvathy, 2006). This is one reason why software startups pivot. They build their organisation—their *system*—around specific capabilities and approaches.

What matters is to find an outcome that the system can generate reliably and has a viable market. They call this *product-market fit*. (Some things in tech are buzzwords, marketing phrases devoid of true meaning. Some of them are jargon—specialised phrases that look like buzzwords to outsiders. This is jargon.)

Startups fail most of the time. Startups that begin with a specific outcome in mind and then build towards that fail more than the rest. Every new organisation begins with specific capabilities and emerging specialisations based on the founders, first staff, local community, economy, environment, and initial resources. To have even a hope of success, they need to build on that proto-system, look for opportunities it can be applied to, and feed it into existence. You need to be consistent.

The worst thing you can do is follow every new trend and fashion in software development as you try to build the system. Which is hard to avoid. Most give into it a little because software development is a pop culture.

That culture is one of the biggest risks a software project faces.

7

Programming as product design

> So I think what happened is computing has turned into pop culture and the universities are not helping in general, at least not in the US.
>
> So, Cicero—anybody know a good Cicero quote having to do with the present and past? Let's check your classical education here. So, you know who Cicero was. He was one of those old Roman guys.
>
> So, Cicero once wrote: 'He who knows only his own generation remains forever a child.'
>
> *Programming and Scaling* (Alan Kay, 2011)

Software development is an immature branch of design that likes to dress up as engineering.

Or, more correctly, programming is a branch of the same root discipline that design is a branch of. Design and programming are related species—programming is to design what a penguin is to an ostrich. Design and programming are both structured knowledge work whose purpose is to generate economically useful information—it's usually instructions of some sort. The value of that information is the Stock in a software development system.

In programming, those instructions are executable, which makes the field unique.

From a systems-thinking perspective, programming is not in any way, shape, or form an engineering discipline. It doesn't work like engineering; it doesn't manage like engineering; it does not have the consistency or reliability of engineering; it does not have outcomes that resemble those of engineering.

From a systems-thinking perspective, programming behaves like *design*. You have specialists who are given a problem they need to solve. They do not know how to solve that problem at the start of the process. Through trial and error, they build both the instructions that address the problem and a mental model for how those instructions can be changed and updated. They work with a system that has tight, iterative feedback loops to build up information-based stocks.

The discipline of programming results in economically useful information and the knowledge required to ensure that information's continuing usefulness in the future. Just like design.

The trial and error part is important. Recall that the information created by the design process only has economic value if there has been a test or experiment of some kind to see if it does something of value? The value of both a design and code is not in what it does but in the knowledge that it does something useful. Value comes from knowledge. It does not come from the design or from the code alone.

Software development is managed as a design discipline, not an engineering discipline. This is *not* as controversial a take as it sounds. First, design can be plenty rigorous when it needs to be. Second, when people talk about managing software development as a product development process driven by autonomous teams, this is what they mean. This is *already* what all modern software development processes do. They focus on establishing feedback loops. Then they build value through iterative work that responds to those feedback loops. The details of how these processes work vary enormously from industry to industry, market to market, and organisation to organisation. But many of the principles are the same.

A downside to programming's nature as a design industry is that it has many of the flaws and idiosyncrasies you find in design, fashion,

and other pop cultures. This is what Alan Kay was complaining about in the talk I quoted at the start of this chapter.

Programming is a pop culture industry. It is insistent on chasing novelty, trends, and arbitrary technological innovation to the point of self-sabotage. The notion that the safest, most valuable kind of innovation could be innovation that's extrinsic to the software development system with no technological novelty is incompatible with the norms and established values of the software programming culture. This is why Nintendo are so frequently presented as rebels or iconoclasts.

Programming as a pop culture

The programming pop culture defines change—any change—as progress. Most developers, myself included, have a fascination with novelty. If it's new, then it must be an improvement. You even hear this stated outright as an argument by developers: it's newer and therefore better. Trends in software development are rarely based on objective observation or sensible practice. This endless chasing of trends leads to projects being needlessly rewritten, code being abandoned, and new projects being started when fixing the bugs in an old project would have done the same. The stocks of the software development system are flushed out at a moment's notice simply because the developers found something shinier.

Experienced developers are aware of this tendency in themselves and work to mitigate it, but younger developers are often under the mistaken impression that *this is how software development works*. Unless they can pare back this tendency or are matched with teammates who hold them back, this tendency can lead to immense destruction of value for an organisation.

Pop cultures favour the visual aesthetic of the day. We all know what sort of aesthetic designers commonly favour. Small, low-contrast text, lots of whitespace, no pure whites or pure blacks (just greys). The details vary with fashion, but each generation of designers has a preferred visual aesthetic. That aesthetic tests poorly; the text is illegible;

the layout doesn't have enough contrast. The information density is so sparse it's effectively non-existent.

Looks pretty, though.

Coders have a similar tendency, their preferred aesthetic is just a bit different, but as with designers, it tests horribly when put in front of genuine users. The exact details of the preferred aesthetic tend to vary from generation to generation. One group prefers light-on-dark text (despite not suffering from conditions that benefit from dark mode) and unusable hyper-complex layouts where everything is configurable. Another group goes for ultra-minimalism where nothing is shown by default. You constantly scrub around and hunt for a button, a widget—anything that even vaguely resembles an affordance. This is usually not an issue if you have designers on the team. If you're letting the programmers design the user interface or are a programmer designing a user interface, you need to be aware of it.

But this adherence to a specific aesthetic isn't limited to designs. It's also an issue when it comes to the code itself.

The programming pop culture favours specific ***code*** *aesthetics based on the trends of the day*. I'm not talking about code style or formatting. The code in a project should adhere to a single style, simple as that. The issue is that the programming pop culture demands that code exhibit the latest popular *aesthetics* of rigour, formality, and cleverness. Whether the code actually is rigorous, formal, or clever matters less. A few years ago, as the popularity of the Ruby programming language peaked, a certain dynamism and trickery were *en vogue*. It didn't matter if you were writing in Ruby, JavaScript, or Objective-C. Your code had to have a level of "magic" to it. Metaprogramming, syntax-hacking languages to create ad hoc Domain-Specific Languages, tricks with extreme late binding, and more were frequent topics on developer weblogs and forums. Even a phrase like "objective-c runtime metaprogramming" will date you to a specific generation of native app developers almost down to a single year.

As with all of these pop culture trends in programming, this led to unreadable code that was impossible to work with or fix as soon as it faded from popular consciousness.

The current trend is towards the aesthetics of *correctness*. Everything has to look like it has strong or static typing. It doesn't have to really *have* static typing. That can all be made up after the fact in a declaration file. It merely needs to have the aesthetics of types. Type annotations everywhere, implementing logic through type system trickery, and forcing any and all dynamism out of the system in the name of *correctness* is the name of the game.

A part of this trend is the unpopularity of the approaches and languages that are seen as *less rigorous*. CSS is dropped in favour of statically typed *CSS-in-JS* approaches. HTML is dropped in favour of a strict inline XML-like markup format called *JSX*. Just a few years ago, everybody in web development hated and dropped XML and XHTML specifically *because* it was too strict and felt less dynamic and flexible than HTML. At some point, pop culture will bore of this and swing its attention back the other way.

It's a fashion industry. Trends come; trends go. The lack of historical awareness is considered by most to be a feature.

This rigour is useful in moderation. Static typing does prevent bugs. Usually, they are the same sort of bugs unit testing prevents. Both have immense value as tools to manage your software development as described earlier. Currently, the fashion is to favour static typing over unit testing for establishing a certain base level of correctness in your code. At some point, they are likely to switch again. They have a couple of times in the past. That you could use both at the same time and get the benefits of both doesn't enter the discourse. Static typing with compile-time correctness checks has its uses. So do dynamism, extreme late binding, and metaprogramming. Most of these approaches can be used together, but that isn't how pop culture works. Pop culture demands there be only one *winner* at a time. Choose one, not whichever works the best at each time.

Product development can't indulge in being pop culture. Be wary of these popularity contests.

These issues with programming culture aren't new.

Jamie Zawinski calls it the "Cascade of Attention-Deficit Teenagers" model. We can't do much to change the nature of the field by

now—Alan Kay has certainly tried—but we can mitigate the harm done by the trend-seeking. We can work to ensure that everybody on the team, programmers and designers, is aligned, and have the same understanding of what matters and how to accomplish it.

For that, you need everybody to understand the context they are working in—the works of their field and how they are received. You need to develop taste and understanding.

You need *research*.

8

Research & taste

> 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.
>
> Fast Path to a Great UX – Increased Exposure Hours (Jared Spool, 2011)

Exposure hours

The most effective way to counteract the pop culture tendencies of both design and programming is research. This is a big reason why the programmers on a team should participate in the research process. It gives them a context for how the software should be used that balances out the constant drill of *newer is better* from the programming community itself.

User research is huge. It varies. It's a subject entirely unto itself. But one approach is especially effective at bringing pop culture tendencies under control, which is useful even without the research value. It's what Jared Spool calls *exposure hours*. The concept is almost worryingly simple: *the more your team members observe real users doing the sort of work your project aims to support, the better the team's designs*

tend to be. This specific term might be his, but the observation isn't unique to him.

Not a rule or a law, but it is a workable rule of thumb.

Exposure hours that are shared across the team have a few benefits:

1. The designs tend to be built with a more holistic model of how the work is done. This improves overall flow.

2. The design begins to address various issues the team witnessed that would be hard to quantify in a report.

3. It *improves collaboration in the team*.

As Spool observed, when all of the team, design and non-design members were exposed to user interactions, their collaborations improved. They argued less; they had fewer communication issues; it helped align the entire team.

Developer-oriented software

This helps explain why—even though it's usually a recipe for disaster—you'll occasionally encounter startups and teams that launch extremely successful software projects whose User Experience is lauded, despite having done little to no user research.

Sometimes these success stories are just flukes, but some are instances where the team was designing and making software for itself. Developers writing software for other developers have many of the same advantages you see with exposure hours. Because you tend to have a decent understanding of the various issues and problems that come with your job.

Building software for yourself is, if done thoughtfully, building on countless 'exposure hours' that come from doing the work yourself and witnessing your coworkers do the work.

If you don't have the budget to do a lot of user research, the familiarity that comes from making software with yourself and your teammates in mind can be invaluable.

Taste

One of the unstated advantages of user research and exposure hours is that it helps the team develop *taste* for what sort of software works.

Taste is one of the biggest advantages both a programmer and a designer can have.

It's popular today to believe that 'good' in books, film, music, or design is entirely subjective. It isn't. Quality is *relative*, but it isn't subjective. At least not from the perspective of those *making* or *creating*.

In the broader picture, whether a book is well written isn't something you can assess objectively. But once you narrow it down to a context, your criteria for quality become more concrete. The more specific the context, the more concrete the criteria.

Is it a good romance novel? is a much easier question to answer. *Is it a good faith-based romance novel*? is even easier. *Is it a good* M/M *urban fantasy werewolf erotic romance novel*? is about as concrete and objective a context as can be for this sort of question.

Quality isn't a question of whether you or anybody else enjoyed it. Enjoyment is subjective and an entirely different topic. The question is how well it plays with the rules of the form without *breaking* the form. *How well does it operate within its specific context*?

That's relative but not subjective. That's why writers don't just study writing. They study *genres*. Even literary fiction is a genre with its own conventions. To understand what makes a book good, you need to understand its context.

The same principle applies to software designs, and *user research provides the context*.

You need to behave like a painter, writer, or filmmaker and study your field, study the work, and understand the contexts of that work.

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

That's what you should be doing in software.

I don't care if you're a backend developer, front-end developer, product manager, project manager, designer, or Swift/Java engineer, you need to collect good software, try it, experience it, and 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 apps, 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 mediocre software you're forced to use and figure out which bits are praiseworthy. Imagine how you'd redesign a piece of steaming garbage 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 in common with filmmaking than it does with bridge-building. Develop a taste for software and then talk to others about them. Discuss, disagree, find common ground and start again.

An awareness of history directly counteracts the ill effects of over-excited pop culture. It is an advantage at every stage of your work: to build software you understand well enough to change.

9

Programming as theory-building

> The building of the program is the same as the building of the theory of it by and in the team of programmers. During the program life a programmer team possessing its theory remains in active control of the program, and in particular retains control over all modifications. The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. The actual state of death becomes visible when demands for modifications of the program cannot be intelligently answered. Revival of a program is the rebuilding of its theory by a new programmer team.
>
> *Programming as Theory Building* (Peter Naur, 1985)

Software is a temporary garden whose fate is inextricably intertwined with its gardeners. Beyond that, software is a theory. It's a theory about a particular solution to a problem. Like the proverbial garden, it is composed of a microscopic ecosystem of artefacts, each of whom has to be treated like a living thing. The gardener develops a sense of how the parts connect and affect each other, what makes them thrive, what kills them off, and how you prompt them to grow. The software project and its programmers are an indivisible and organic entity that our industry treats like a toy model made of easily replaceable lego

blocks. They believe a software project and its developers can be broken apart and reassembled without dying.

What keeps the software alive are the programmers who have an accurate mental model (*theory*) of how it is built and works. That mental model can only be learned by having worked on the project while it grew or by working alongside somebody who did, who can help you absorb the theory. Replace enough of the programmers, and their mental models become disconnected from the reality of the code, and the code dies. That dead code can only be replaced by new code that has been 'grown' by the current programmers.

A successful software project is grown from a small living thing to a larger living thing. Building the project large from the start and will never come to life. Replacing the gardeners that brought it to life will lead it to whither.

Most of our errors with software development are category errors. We treat these software projects as one kind of animal when they are a different species entirely. They are grown thought-stuff. We're treating them like lego blocks.

This is why the Stock that code provides to a software development system isn't the code itself but the value the code represents. That value depends on how well it is understood by the team that needs to fix, improve, and modify it. It depends on the stability of the underlying language and platform. It depends on how often it needs to be rewritten or modified to continue to function normally as its dependencies change. Code isn't valuable in and of itself. Code can be a liability. What you want is code that has value and whose value depreciates slowly.

Misunderstanding the nature of software development will increase project failures. Going against the grain of software development makes catastrophe more likely, much like how the wrong type of wood, steel, or concrete will lead a bridge to collapse.

Software is the insights of the development team made manifest. Software has no life on its own but exists as a kind of cyborg simultaneously in the programmers and the code. To reuse Donna Haraway's words, software is *simultaneously fiercely material and irreducibly ima-*

ginary (p. xii, Gray 1995). Software can be real and unreal at the same time. The written code is how software interacts with end users and other software systems. The insights and knowledge that exist in the minds of the developers are how software lives, changes, and grows. Without the code, it has no way to interact with the real world. Without the knowledge in the minds of the developers, it has no way to adapt and survive.

Documentation only works up to a point because it can both get out of sync with what the code is doing and because documenting the internals of a complex piece of software is a rare skill. A skill that most developers don't possess. Most internal documentation only begins to make sense to a developer *after* they've developed an internal mental model of how it all hangs together. Most code documentation becomes useful *after* you have built the theory in your mind, not before. It operates as a mnemonic for what you already know, not as a tool for learning.

This has practical implications for software projects.

The first is that it's a partial explanation for *bitrot*, the phenomenon where working software seems to deteriorate even when the code is untouched. Sometimes bitrot is straightforward: the platform or dependencies has been updated. The code "rots" because its context has changed. But sometimes bitrot happens because the *programmers* have changed. Memory is fallible. The mental model that a programmer has integrated about a particularly convoluted piece of code can fade away. The coder returns to the code after a period, and it no longer makes sense. The programmer no longer has a cohesive, integrated theory about that part of the software which means it has to be replaced.

This can happen in the other direction as well. The programmer revisits a module after working on other parts of the software. That work has left them with a sharper, more accurate theory of the software. Now they realise the original module didn't fit in with the rest and, with their improved insight, see that if left untouched the old module would become a source of friction, bugs, and defects. The code didn't change, but the programmer did. *The code rotted because the programmer changed.*

Team structure and churn

Another consequence of the theory-building model has to do with team structure and churn. The most reliable method a programmer has for building an accurate 'theory' of a piece of software is to have been there when it was first written. That programmer will have the best understanding of how the various components interact, how best to change any given part, how to minimise bitrot, and which pieces of code are actually unused. This is the *first generation* programmer.

The *second-generation* programmer—the second best method for building an accurate theory of the code—is somebody who worked on the code with somebody who was there when it was first written. Whenever you encounter unfamiliar code or modules that don't make sense, you have a first-generation developer on hand to help you understand—help you develop a mental model of how it all fits together. Over time, the proportion of the code written during your tenure will grow until you have effectively become a *first generation* programmer.

Team stability is *vital* for software development. Each team needs to be composed of a majority of first-generation developers. Too many second-generation developers and the first generation gets overwhelmed, and work stalls. The work slows down either because too many core tasks are waiting on the first generation or because the second generation keeps having to rewrite components from scratch because they didn't understand the theory behind the original component. Too few second-generation developers and there is no renewal—each developer that leaves the team is a potential catastrophe.

Many teams in the industry constantly rewrite parts of their code. Not because they keep figuring out better ways of approaching the problem but because nobody on the team has an accurate mental model for how it works and how the code fits in with the rest. If you can't hold onto the original team, if you grow the team too quickly, you end up running to stay in place. Code keeps getting written without any improvements of substance to the software itself.

The various different software processes will do little to counter these issues on their own. This is a team management issue that can only be addressed by a manager who understands the fundamental

nature of software development. There is no magic development process that will prevent these problems. It takes skill.

Churn is destructive

Constant churn in a software development team, both among the programmers and designers, is ***absolutely devastating***. It is the death knell for a software project. Makes deadlines meaningless. It turns software into a disposable, single-use product like a paper towel. *Anything* that increases team member churn threatens the very viability of the project and the software it's creating.

Rest and assimilation

Even though processes vary and can never be the entire solution on their own, there is one procedural aspect that is universal in software, if you want your software project to survive in the long term. No matter how you structure the work, divide the cycles, or how long each is, every software project needs to provide space for self-directed work.

The book *Shape Up* calls this the cool-down.

> During cool-down, programmers and designers on project teams are free to work on whatever they want. After working hard to ship their six-week projects, they enjoy having time that's under their control. They use it to fix bugs, explore new ideas, or try out new technical possibilities.
>
> *Shape Up*, (*Singer*, 2019)

John Cutler points out that if you don't provide a cool-down period, circumstances will force one on you:

> This happens with individuals when they dig too deep for too long. They hit the wall and get sick. They burn out. The slowdown may be acute. Or it may be chronic—a very long, prolonged period of numbness.
>
> It happens with companies. Quarterly goals increase. The company scales. Complexity increases. The pressure increases and cracks begin to form. As with individuals, sometimes there is a sudden collapse.
>
> Slow Down on Your Own Terms, (*Cutler, 2021*)

Cool-down periods are important for all kinds of work. For software development, cool-downs take on an additional purpose: they represent periods where the developers can explore the software and help build their mental models. They allow second-generation developers to understand the code base better. They let us clean up codes that need cleaning, fix structural issues that don't fit anywhere else and do some house cleaning. In the process of doing these chores, our mental models of how the software works—our theory of it—progressively become more accurate and more representative of the actual state of the project.

Without cool-down periods it becomes much harder for new developers to become familiar with the project. Slow down, and it becomes easier to grow the team and help existing team members work more effectively. That speeds the work up. In Cutler's words, you slow down to speed up. And you really do speed up. Having your house in order, so to speak, not only benefits team performance but also results in more *consistent* performance and makes planning, strategy, and execution much easier.

Theory-building and software

Understanding the mental processes that underly programming—what ultimately makes software work—won't turn you into a software developer overnight. You don't have to understand any of it to *be* a programmer. Most programmers are about as aware of their own the-

ory-building as the proverbial fish is of water. It's like breathing, *they don't have to be aware*. It's just how it works.

But it *is* essential to understand if you're collaborating with or managing programmers. It helps you understand some of the idiosyncracies that are specific to programming. More importantly, it can help you avoid many mistakes that frequently doom software projects.

As its benefit is in *avoiding* disaster, it obviously won't guarantee success, but what is success other than the consistent avoidance of failure?

Part Two:
How we fail

10

The first Software Crisis

> In Europe alone there are about 10,000 installed computers — this number is increasing at a rate of anywhere from 25 per cent to 50 per cent per year. The quality of software provided for these computers will soon affect more than a quarter of a million analysts and programmers.
>
> *David Gries and A.G. Fraser, 1968, Software Engineering: Report of a conference sponsored by the NATO Science Committee*

We haven't just suddenly, as a society, become bad at making software. *We've always been bad at software*. We've always been bad at completing and shipping our software and computing projects.

It's fitting that the first software project in history, Ada Lovelace's and Charles Babbage's Difference Engine was designed in the 19th century but wasn't finished until 1991. Too expensive and over a century too late. The software industry began as it intended to carry on.

The early days of modern computing (as opposed to 19th-century computing) were almost as hectic. They improved delivery times over the Difference Engine as projects were completed in years instead of decades. But, as always, we were bad at making software.

One mitigating factor was that the early, post-war 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. As we know, simple code is the only code we can make reliably.

This didn't last long.

Moore's law was a more concrete reality back then. Computers steadily became more powerful, which led them to be applied to more complex problems.

This did not go well.

By 1968, this was a full-blown crisis:

> 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, Software Engineering: Report of a conference sponsored by the NATO Science Committee*

We've always known that computers had great *potential*. But we've always had problems trying to figure out how to *unlock* that potential with the software we know how to make. Even back in the sixties, it was clear that computers would be integrated into our society at multiple levels; they'd interact with hardware, manufacturing, transport, energy, and infrastructure. They may not have known just how pervasive computing and software were going to become, but those working in the industry knew the direction things were heading.

Their concern wasn't with what that revolution meant once it happened. You can tell from the reports and various conference proceedings that they were more worried that our inability to make software reliably and on time would stop the revolution in its tracks. The future would end before it got started. It didn't seem to cross their minds that the computer revolution wouldn't be stopped by flaws, and instead, we'd get software bugs everywhere, in *everything*.

Instead, the consensus in software engineering was that we needed to get better at this.

Which, you know, is fair.

Once the field, in general, had accepted that there was indeed a software crisis they began the work address to address 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 their craft. 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. The 90s made memory garbage collection mainstream. This made a whole host of memory-related bugs impossible for entire classes of software. The 90s also brought with them the popularisation of various lightweight software development processes that emphasised *smaller* and *simpler* in every step and later coagulated into the amorphous blob that is "agile software development" in the early 2000s.

The heroic effort that went into fixing the year 2000 problem, again in the 90s, instilled in an entire generation of programmers the value of not cutting corners in software development.

We'd brought ourselves to the brink of disaster, but now we knew better. We had clear ideas of how software could be made reliably and concisely.

Things looked good.

Software development had a clear, upward arc. You could argue that our software still wasn't good enough. Java represented progress over its predecessors, but it was still *java*. The web was the software development equivalent of three kids stacked in a trenchcoat trying to pass as an adult. But it had the virtue of not being able to do much, which meant that most of what we did with it was simple.

And, again, we know how to make simple software work.

It seemed to be a pragmatic inevitability that general software quality would reach that long distant goal of being *good enough*. Even software that was objectively bad in multiple dimensions like Windows 95 was a clear improvement from 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 being 'modernised' into a field of practice.

Then, in 2001 after a decade of percolating through the industry as a variety of randomly named lightweight development methods, 'Agile' stepped in with the Agile Manifesto. Finally, hopefully, at last, good software development was going to be the *norm*. The future was about to happen.

Except it didn't quite work out that way.

11

The agile reaction

> The Agile Manifesto is an immune response on the part of programmers to bad management. The document is an expression of trauma, and its intellectual descendants continue to carry this baggage. While the Agile era has brought about remarkable advancements in project management techniques and development tools, it remains a tactical, technical, and ultimately reactionary movement.
>
> *Dorian Taylor, Agile as Trauma*

As Dorian Taylor writes above, Agile is a reaction to the trauma of bad management. It isn't the first reaction and won't be the last because the problem isn't the process. It never was. Each was a reaction to pervasive poor management in the industry. When that reaction didn't magically fix mismanagement, the development culture immune response triggered again and spawned another movement.

It carries on forever because there's nothing wrong with the process, any of them. All of the Agile processes work, even the ones that have fallen out of favour. Even the much-maligned Waterfall process works, at least as originally practised. It might not be appropriate for all software products, but it's iterative, favours prototypes that it called 'pilot models' (Royce, 1987), and is clearly a reaction to poor management disinterested in investing in software development:

> The virtue of all of this is that as the design proceeds the change process is scoped down to manageable limits. At any point in the design process after the requirements analysis is completed there exists a firm and close-up, moving baseline to which to return in the event of unforeseen design difficulties. What we have is an effective fallback position that tends to maximize the extent of early work that is salvageable and preserved.
>
> *Managing the development of large software systems: concepts and techniques* (Royce, 1987)

It isn't perfect. It's missing a few lessons that Agile learned from the management theory revolution in the seventies. That's when systems-thinking as outlined by people like W.E. Demings all but replaced command-and-control as accepted best practice in organisational psychology and management theory. Updating the process as described using systems-thinking wouldn't even be that hard. You'd manage process variability by reducing batch sizes and introducing Work-In-Progress queue management. You'd replace manual testing with automated testing and input quality control—making sure that the software's dependencies have unit testing or type coverage and that the services you use had adequate reliability guarantees.

If you update Royce's process to use what we've learned in management theory over the decades, you'd end up with a process that would fit right in with the Agile process hall of fame. You could even make the argument that it's *more* agile than many others because of Royce's emphasis on iteration and pilot models.

The processes aren't broken. They never were. But, because management continues to be dysfunctional, we keep getting new ones. Waterfall had its iterations for a few years. Then Agile took over and had its iterations over a few years. Now we have developers reacting to the Agile reaction and calling it a hoax, which will probably spawn another series of processes that all work.

We keep trying to fix software development by tinkering with the small corner where developers have some influence. Every year a movement or new process claims to fix everything and make software work. Then it doesn't work, and its failure spawns a new movement in

reaction. You try a new thing. It works for a while. Then it doesn't; because what worked was *the change* which forced you to pay attention to what you were doing. Once enough people have been disappointed, a new movement spawns as a reaction.

—*This time it'll be different!*

We try again as we hear claims that this version of agile is proper agile.

Every version of 'agile' is promoted by people who genuinely believe they have solved the problem of software quality. They are all wrong because the problem isn't with the process.

Agile, in particular, has become associated with micromanagement, process fanaticism, and addiction to unexplained 'magic' rituals of dubious value. This is unfair because the problems are with bad management. Replace Agile with something else, and those managers will give the new process an equally bad name.

Agile wasn't progress. At its heart, it was a coping mechanism whose acceptance gave us a temporary respite from the still pervasive mismanagement of software until, inevitably, the machinery of agile was coopted and became integral parts of the broken system we were trying to fix.

We've made a lot of headway into tackling specific types of bugs. Memory management bugs are rarer than they were two decades ago, thanks to the various ways modern programming languages automatically manage memory use. They haven't been eliminated entirely, as they should have been, but they are rarer. This means that applications don't outright crash as much as they used to. Automated tests such as unit tests and typed programming languages help, at least in some circumstances. Functional programming can also work well.

These are all positive developments.

But now that our devices and our applications can run for weeks on end without crashing, we've discovered that we, as an industry, are particularly bad at three specific aspects of software development:

- State management. Any app that has to manage a consistent state over time and over multiple interactions is an app that has bugs. It is 100% certain that this app will end up, at some point, in an in-

consistent state where it either stops functioning or, worse, continues to just in an incorrect and potentially destructive way. We haven't solved state management.

- User interface design. Whatever we want to call it—Human-Computer Interaction design, user interface design, user experience design, whatever—we are incapable of either making a usable software interface or of *maintaining* a usable software interface. Any given usable design will inevitably deteriorate into an unusable mess of features.

- We've replaced memory errors with *memory bloat*. Most complex apps today leak memory to some degree and see their memory use grow the longer they run.

Web apps are *especially* bad at these, continuing their trend of being *like native apps; just a tad worse*. Web apps are memory hogs, their user interfaces are confusing messes, and they are so horrible at state management that they keep inventing new ways to be bad at it.

Poor state management means we can't make software that is *correct* and *reliable*. Poor design means we can't make apps that solve the problems they're supposed to. Memory bloat makes our apps unusable on cheaper devices, increasing the 'digital divide.' Improving state management without improving user interface design is a meaningless exercise in going faster in the wrong direction. 'Fixing' interface design with error-prone state management is impossible. A user interface that loses people's work can never, not under any circumstance, be considered 'well-designed'. The time spent fixing memory leaks on an app nobody can constructively use would be better spent on sudoku or crossword puzzles. Brilliant code with a poor interface is a poor app. A thoughtful and effective design backed by trash code is a trash app. Software is a holistic problem.

Many of the problems with software development that can't be traced directly to plain old poor management, such as our inability to correctly estimate the time software development takes, stem from these core problems. We always end up over time, over budget, with

software that doesn't do its job. The task is *always* more complex than we thought. Our organisations are incapable of even trying to consistently make working user interfaces, *even when that interface is nothing more than an API only used in other pieces of software*.

We decide on the problem without checking to make sure it's a *real* problem for our end users. We then design without researching the nature and structure of the problem we're trying to address. We ship without testing to see if it actually does the job it's supposed to.

Only then do we do some actual testing, often A/B tests. We throw two half-baked unfinished designs *into a functional shipping application that people rely on to do their work* and use Data™ to see which unmitigated disaster is marginally less disastrous for the working lives of those held hostage by our applications. We gaslight our customers as a standard practice and think we're doing them a favour.

Twenty years of agile hasn't fixed a thing. After a brief upward arc, we're back where we started, carrying on as we began, in the middle of the second major software crisis in computing history.

The Standish Group has been tracking software project success or failure at a number of companies, small and large, for over 30 years. The results have been depressingly consistent over the years. ***90% of large projects fail***. Over ***70% of medium-sized projects fail***. Only small projects do well: over 60% tend to be successful. (Chaos Report, 2015)

We are still an industry where 60-90% of all software projects fail.

Decades of Agile and software project outcomes have hardly changed.

These results, dire as they are, come from the initial reports made by those responsible for each project, and, as we all know, reports like that rarely show the full grim reality.

12

The many kinds of software failures

There was a human test driver behind the wheel, but video from the car's dash cam published by SF Chronicle shows that they were looking down, not at the road, in the seconds leading up to the crash.

Police say that the car didn't try to avoid hitting the woman.

Uber car software detected woman before fatal crash but failed to stop
(*Lisa Vaas*, 2018)

The Max has been grounded since March 2019, after some badly written software caused two crashes that killed 346 people.

The ancient computers in the Boeing 737 Max are holding up a fix
(*Darryl Campbell*, 2020)

Multiple defects in Toyota's engine software directly caused a September 2007 single vehicle crash that injured the driver and killed her passenger.

An Update on Toyota and Unintended Acceleration
(*Michael Barr*, 2013)

Truly catastrophic software failures rarely get discovered in a sprint review or shipping retrospectives. The serious bugs that induce panic attacks only appear once the software is distributed widely enough to begin to experience edge cases. Most of the time, that only happens long after the software development team has moved on and marked the project as successful or challenged.

Let's imagine for a moment that Toyota, Uber, and Boeing are participants in the Standish Group's research. I have no idea if they are, but let's pretend for a moment.

Which do you think is more likely?

1. The projects that led to these fatalities were marked as failed in the Chaos Report database.

2. After death march sprints, the team shipped code that technically fulfilled the requirements as described. They marked it down as successful or challenged depending on whether they were on time or under budget.

How likely is it that our statistics on software project failures represent reality if software flaws that killed people can be invisible to them?

Studies such as those performed by the Standish Group are vital, but numbers like "90% of all large software projects fail" represent the *lower bound* for any realistic estimate of general software quality and project success. Every one of the projects marked as successful will have bugs, and some of those bugs will have turned out to be catastrophic. To get a sense of just how bad the software crisis is, we need to get a clearer idea of the many different *invisible failures* of software development: the ways that software can *look* successful but actually be broken or substantially flawed.

Delayed impact: conceptual bugs

"Conceptual bug" is just a fancy way of calling something a bad idea. Usually, this happens because of a shortcut you took to ship something on time or to lower costs.

In Boeing's case, they wanted to be able to sell the 737 Max as a 'proper' 737. Meaning pilots trained on earlier 737 models wouldn't have to be recertified for the Max. The problem was that the Max is a very different plane with entirely different handling characteristics. They decided to fix this with software. The plane's controls were already software-mediated; the controls aren't directly connected to the plane's mechanics but connect to a computer, which controls the actual mechanics. Instead of keeping the abstraction relatively simple, they decided to have the software continually adjust the inputs it got from the controls to make it 'feel' more like a traditional 737.

This is what we in the software industry would call a *really bad idea*. There's just so much that can go wrong with this concept, starting with the problem of trying to define and replicate something as subjective as another plane's 'handling' in the first place.

That this software mediation went catastrophically wrong was maybe not inevitable (sometimes you're lucky), but it was always a profoundly bad idea. The idea itself was the root cause of the bug.

This is more common than any of us would like. Web development's fondness for constantly recruiting undertrained and inexperienced coders and throwing them into a coding death march has that effect. Most of the websites you experience on a normal day are half-baked, slow-motion disasters because the developers just didn't know better and management didn't give them the opportunity or space to figure out the right thing.

Another example of a conceptual bug would be passwords stored unencrypted in plain text.

To give those who aren't programmers some context for why this is a bad idea, you need to understand that software generally isn't supposed to store passwords *at all*. You're supposed to store an encrypted *hash* of a password, and that hash needs to be created with a cryptographically secure algorithm. This hash, which is a number calculated based on the original password, can be used to check if any given password matches what you first provided when you signed up without keeping a copy of it. The idea is that if you're unlucky and the server

gets hacked, the hacker will only get a bunch of numbers they can't use to compromise anybody's accounts.

This is standard practice—what everybody is *supposed* to do.

This hasn't prevented companies from shipping software that doesn't bother and stores the passwords directly. Often this is a misguided time-saving measure. "We'll fix it in a later sprint." This fix never happens because the current version is working fine, right?

This also happens because nobody involved knew that *storing passwords properly isn't an optional extra*. It isn't something you can put off. They might have known what they were *supposed* to do, but they didn't understand how important it was. This is more common than anybody in software likes to admit.

What makes conceptual bugs insidious is their delayed impact. The 737 Max bug didn't kick in on every flight—only under specific circumstances. It can take years for a hacker to break into your server and steal the hundreds of thousands of plain text passwords your website has been collecting. Software projects with severe conceptual bugs can ship and be unqualified successes for everybody involved, only to become disasters years later.

It's very hard for broad studies such as the Chaos Report to account for the subtle nature of an atrocious idea.

Delayed impact: legion bugs

In contrast with Boeing's Max 'issues', Toyota's uncontrolled acceleration bug isn't just one bug. As the engineer, Michael Barr, wrote above, "multiple defects" caused the problem, not just one. What happened was that several defects interacted to put the system in an inconsistent state that the code hadn't accounted for, causing uncontrolled acceleration.

All non-trivial software consists of multiple interacting parts or components. A flaw in one component won't be able to put the whole system in a dangerous state, but multiple flaws can cause a cascade of errors that lead to the software behaving in ways it was never supposed to.

Again, this is more common than anybody likes to admit. I'd go so far as to say that this type of failure has been present at one time or another in almost every piece of software that has shipped.

Most security defects that are used to compromise systems are of this nature. An attacker might use a cross-site scripting (XSS) flaw to sneak hostile code into a web app, use another bug in the backend API to get escalated privileges, and then use those to extract sensitive data. That's how you end up with data dumps of entire databases on the web. Browsers and operating systems try to mitigate this from cascading into the rest of your device by isolating components and pages from each other, but bugs cascade easily inside each app.

Many of these defences only prevent *attacks*. They don't prevent a cluster of bugs from interacting to crash apps or lose your work.

By their very nature, these bugs are time delayed: you need a critical mass of flaws for them to kick in. You aren't going to spot one of these in a sprint review until it's too late. The only way to prevent them is to keep the number of defects as low as you can.

Accessibility

It may be controversial to include lump accessibility failures with other software project failures, but any flaw that directly opens you up to legal liability is a deal-breaking flaw.

—*Shipping this could get you sued, and you'd almost certainly lose the case*

Any project where this is true is not a success.

Accessibility requirements are the law in many countries. If you are working in one of these countries, delivering an inaccessible project is like being a builder who doesn't build up to code or an architect whose designs disregard fire safety regulations: *you haven't done your job*. You may not care if a house burns down because of poor fire safety, killing everybody in it, but functioning societies do, and they will punish those that let it happen. The same principle applies to accessible software. You may not care, but society does, and there is a real risk that

those who make inaccessible software will be fined for letting it happen.

That software should be accessible should be about as controversial as the idea that paint on children's toys should be lead-free. Yet it persistently remains the exception rather than the rule. Even if you have a project that is relatively free of defects and delivered on time and under budget, it's still likely to be inaccessible. At least until it causes a PR crisis for you, followed by a legal crisis if the public relations crisis is mishandled.

Builders should build to code. Music venues should have usable emergency exits. Children's toys should not be poisonous. Software should be accessible.

There is another dimension to accessibility beyond the legal requirement. A lot of software that gets shipped and tagged as 'successful' today is not just inaccessible to those with disabilities but also to broad swathes of society. Again, as always, the worst offenders are *web apps* with poorly handled dependencies and code frameworks which lead to ballooning app sizes and load times.

According to the HTTP Archive in 2022, the median JavaScript payload on the web today is 461 KB *compressed*, which on its own noticeably blocks user interaction on load for most devices commonly used if it runs at all.

That's the *median* so that statistic will include content websites that require less code than your average web app, whose JavaScript bundles are over a megabyte.

These apps are unusable on mid- to low-end mobile phones that are the primary computing device for a majority of your software's users. This fact might be tolerable as long as this doesn't completely alienate your core user base. But bundle size is *proportional to project size*. A bigger JavaScript bundle has more code, and more code is strongly correlated with project failure. More code means more bugs, more defects, more complexity, more delays, and higher costs.

Extended loading times and poor performance might be tolerable if the product is free of defects, but a slow app that is also more buggy means your project will be in trouble as soon as it ships to your end

users. A slow and buggy app surrounded by a constellation of other slow and buggy apps can kill the entire ecosystem (Alex Russel, 2021).

Beyond legal requirements and poor performance, most projects suffer from an additional accessibility issue that threatens their viability: poor design.

Poor design

> Over 90% of industrial accidents are blamed on human error. You know, if it was 5%, we might believe it. But when it is virtually always, shouldn't we realize that it is something else?
>
> Human Error? No, Bad Design, (Donald Norman, 2018)

User interface design is a common cause of long-term project failures. You might not run into these issues immediately on the project's release. They might not crop up until years after it's been shipped and marked down as 'successful'. But when they appear, the consequences can be dire.

Sometimes it leads to an industrial accident costing life and limbs. Sometimes it is a productivity catastrophe, such as discovering too late that an application doesn't support 'undo' in any way, shape, or form —web apps generally never do—resulting in lost work. Sometimes this is just an annoyance, like a social media site whose layout leads to people accidentally follow or unfollow people. Sometimes it's just frustrating, such as trying to figure out how to do something in Microsoft Word's ribbon menu. But, overall, bad design is the norm for software, not the exception, and worse design is the norm for web apps. The web doesn't have the benefit of falling back on consistent native affordances like integrated context menus or menubars. Even when there are APIs to allow for these features, web apps make sure to use them in ways that diverge unusably from the platform norms. *They have to*, because they need to work the same across OSes.

The advent of the web and smartphones has resulted in user interfaces that are less predictable and less consistent than ever before. Which is bad enough. But companies also don't optimise the end-user

productivity of the user interfaces they ship. Their goal is to maximise *engagement*. They want you to stay in their app for as long as possible. This is generally the opposite of productivity.

There is no real ill will behind these decisions or these bad designs. They point to wishful thinking that dominates software development: the belief that you can't do any harm unless you *intended harm*. This belief sucks away any real sense of urgency that *should* be driving people to improve the projects they work on.

The other belief that maintains this status quo of bad ideas and bad designs is the belief in magic.

I'm sure this sounds ridiculous to most readers, but large parts of the industry are fundamentally driven by a superstition that has no observable foundation in objective reality:

This will be fixed with better technology.

It doesn't matter that we've had seventy-five years of bad software and nothing we've come up with so far seems to have fixed the problem. The software industry keeps praying for something magical to come along and fix everything: *a silver bullet*.

13

Silver bullets

> But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity.
>
> *Frederick Brooks*, No Silver Bullet

Most of the history of computer science has alternated between movements that claim to solve all of the problems inherent in making software and the subsequent realisation that, no, this just moved the problem around.

Even in the rare case when we figure out how to essentially *fix* software in one area, we also *adopt* software in a hundred new industries and fields where that fix does not work. The silver bullets that actually manage to slay the werewolf are ad hoc, specialised, and only work on *that specific werewolf*, not the species of man-turned-beast in general.

We have a ton of great *technical* methods for improving particular qualities of our software. Automated correctness tests work. Which kind of test works depends on the context, but, in general, tests help prevent and fix problems. Various forms of automatic memory management helped prevent many of the mainstay bugs of previous software eras, which is good. Typed code can also help. All of these things together mitigate the problem.

But they do not solve it because the root of the problem isn't technical.

Remember how I wrote earlier that two of the biggest unsolved problems in software are state management and user interface design?

Sure, managing intricate state is incredibly hard and error-prone. That's a technical problem we haven't reliably solved. But the fact that every single one of our software projects inevitably ends up having to manage complex and partially inconsistent state isn't a technical problem. That's a people problem—design problem, to be specific.

User interface design is even more clearly a people problem. You need to figure out a few things:

1. What your audience is.
2. What problem do they have.
3. The nature of that problem.
4. Whether that problem can be solved with software.
5. How it can be solved with software.
6. What user interface designs implement that solution.
7. Write the code that makes it all work.
8. Test it on your actual audience in a controlled, scientific way.
9. And if that doesn't quite work the way you like, loop back to steps 2, 3, 4, 5, or 6, depending on the results.

Most software projects skip this. They replace it with guesswork and patch up the application after shipping, like the Twilight Zone aeroplane gremlin on Opposite Day. They scramble furiously to jam wires back into a burning jet engine, hoping that just a little bit more of the red wire will magically prevent everything from blowing up in your face.

Crash first; fix later; it's the software way.

If we're lucky, our management and budget let us poke a bit at a step or two in that list. They might give us enough space to do a half-assed cosplay of the rest of it. We do what we can and then fill the gaps with theatre and posturing.

We know what parts of software development are the *hard* parts that will cause problems down the road. Yet we consistently end up in situations where we don't get the opportunity to give those problems the attention they need. The problem isn't that we don't know what needs to be done. We do. Something else is at play.

The reason for the state of software is that most of the industry doesn't know the answers to even the most basic questions:

- What is good management?
- How do we prevent escalating complexity?
- How do we prevent *harmful* software? Software that's 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?
- Do we have any capacity to acknowledge that the quality of our software is declining?
- Or, not increasing when it should?
- Are we capable of recognising when the process serves the product's end-user?
- Or, when it's not?

These seem basic. But these questions are also *people* questions. Organisational questions. Social questions. They are about how we operate in groups as a part of society. That's where we're at as an industry: we do not have answers to the most foundational questions of our field. Most companies we work for don't even realise they should be *asking* these questions. Agile hasn't answered any of them, even though its various flavours have made the attempt. Value types and unit tests don't solve them. It can't be solved through technology because technology can only propose silver bullets to social problems. But we keep hoping that this bullet will, at last, slay the werewolf.

This problem has gone on for so long that we don't even know anymore what we don't know.

Most people involved in software development don't know what good software looks like because they've only used mediocre or awful software. They think Google Docs is a great application—that Slack was a brilliant innovation and a revolution in online productivity.

Most of us who make software don't know what a sensible software product development process looks like because they've never seen it happen, never been a part of it, and never experienced it first-hand.

Most of our managers know less about management theory, organisational psychology, or even basic principles of collaboration than an abusive high-school gym teacher. They mean well but are the 'chaos' in the 'Chaos Report' made manifest.

And you never really know where you stand with them.

Part Three:
Frameworks

14

WTF do you mean when you say 'framework'?

One of the most common abstractions in software development is that of a *framework*. They dominate web development and are a growing part of app development as the business need for cross-platform solutions becomes more and more pressing.

They are a key part of any given software development system, even the ones that aren't technically using a framework.

What a framework is, precisely, is notoriously hard to define. They've been defined as 'code that calls your code,' or 'structure you can build your software on,' or 'a third-party platform for developing software applications.' Most definitions leave out entire classes of frameworks, and few offer definitions that are *useful*.

In software, *a framework encodes a particular approach to thinking about and solving problems in software development*.

Those approaches are *paradigms*, also known as worldviews, controlling ideas or mental models, formed through examples of practice. The term originates in Thomas S. Kuhn's *The Structure of Scientific Revolutions* (2012), as a description of how science progresses by switching from paradigm to paradigm as each model progressively better explains reality as we understand it through practice and experimentation.

Kuhn's paradigms differ from software paradigms in that science is constantly driving towards a better, more objective, understanding of reality, whereas software paradigms are generally relative to the problems they are trying to solve. That's why multiple conflicting software paradigms can coexist (relatively) peacefully.

A software paradigm exists outside of software, as a mental model or controlling idea that directs software practice. They differ from a scientific paradigm, but the overall concept remains similar:

- It is a worldview or framing for how to think about an aspect of programming. Sometimes it's holistic and encompasses the entire stack that the developer is likely to interact with. Sometimes it's specialised and applies only to a part of that stack, like storage or rendering.

- It is defined *by example* and by the discourse surrounding the examples.

- As with Kuhn's scientific paradigms, it is open-ended, allowing practitioners to explore applications of and variations on the paradigm as they see fit.

Ultimately, a software paradigm is a blueprint for a software development *system* and a framework is a particular implementation of that system or one of its subsystems. This is the system whose stocks and flows we talk about when we talk about the value of code and designs.

Those frameworks range from the purely *paradigmatic* to the purely *systematic*. Paradigmatic frameworks only exist as documentation and examples of practice and leave the actual system implementation to you. Systematic frameworks exist as implemented systems that your code has to function entirely within.

Programming without a guiding paradigm is just random and haphazard code. You have a process, experienced developers, and great research, but without a controlling paradigm, you are unlikely to be able to integrate them into a functioning software development system. You will need a paradigm if you're going to get anything done. The easiest way to get there is to use a framework.

Paradigmatic versus systematic frameworks

A classic paradigm in software development is the Model-View-Controller (MVC) architecture. It defines how you should structure your application, where the business logic should reside, and how it interacts with the user and platform.

At a high level, the concept is (almost) straightforward:

- The Model controls and manages access to the data store (however that is implemented). It validates data and responds to queries.
- The View controls and manages the user interface. It manages visual state and responds to events.
- The Controller mediates between the two, updates the View with data from the Model, and contains the application's actual business logic.

The details of an MVC system can vary *enormously*. Do you have a lightweight controller and put most of the logic in the Model? Do you let the View fully manage its visual state, or does the Controller take care of that? How do you manage object inheritance? Do you use delegates to mediate between the View and the Controller?

You can implement an MVC-based app without a systematic framework, in a purely paradigmatic way, just with some way of accessing data, a rendering surface, and an object-oriented programming language. MVC is still a framework when you do this because frameworks don't have to be fully systematic. You still have a system based on a paradigm. You just happen to have implemented it yourself.

A purely paradigmatic MVC framework would consist of documentation, with examples, for implementing an MVC architecture in a given programming language on a given platform.

A purely systematic MVC framework would, for example, be when a platform bakes Models, Views, and Controllers into the APIs it provides. Going with a different paradigm would entail working

against the system, or going outside the platform's system entirely. Apple's 'Cocoa' APIs, for example, form a consistent and comprehensive MVC programming framework, which was one of the reasons it was such a productivity boost when it first appeared.

Paradigmatic frameworks are defined informally or through detailed documentation. They tell you which specific *affordances* and *capabilities* of the underlying platform or framework you should use and which you should provide to other parts of the system. They minimise the time spent deciding on approaches. They make your work much easier to understand by others familiar with the same framework. They mean it's likelier that your work and your colleagues' work will fit together without errors.

Systematic frameworks chart your path through the underlying platform or framework by packaging the approved-of capabilities and affordances into a readymade system.

Most frameworks operate in the space between. They encode core ideas from the paradigm into a system or subsystem while leaving the rest up to documentation and example. This is useful because many paradigms in software development are only partial solutions to the problems at hand, which means that there is often an element of mixing and matching subsystems required to get a complete system.

The best example of a partially systematic framework would be *React*, Facebook's Open Source web development framework. It's a reactive programming twist (hence the name) on the originally web-based Single-Page-App paradigm. It's primarily concerned with the view or rendering part of a graphical interface. It can be used with a variety of different state management or business logic subsystems. Its "system", intentionally, doesn't provide everything you need to make a web app. This flexibility is a big reason for React's popularity.

Many of the purely systematic frameworks in web development today have been called *meta-frameworks* because they take a partially paradigmatic framework, like React, and turn it into an integrated and consistent system— bringing it all the way to the *systematic* side of the spectrum. They take away your options but simplify the process of *using* the framework.

This shows us why the paradigmatic-systematic spectrum helps manage software development. On one end of the spectrum, you have flexibility. On the other end, you have productivity.

The more certain you are of a software project's goals and desired outcomes, the more productivity benefit there is to be had from finding and adopting a fully systematised framework that matches those goals. The more exploratory your project is, the more benefit there is to be had from adopting a partially paradigmatic framework that you can adjust as you discover.

A systematic framework that doesn't match your goals—a heavyweight framework with a rich set of features when what you need is something high-performance and lightweight, for example—will constantly be fighting you and will lead to disaster. A more paradigmatic framework when you need guide rails leading to a specific outcome is similarly a guaranteed disaster.

Choosing a framework, paradigmatic or systematic, is one of the most important actions you can take to manage any given software project.

Frameworks simplify management

They come with many desirable features that make managing software development *easier*. The first is the nature of a paradigm itself: it always exists as a system for solving a specific problem.

- Model-View-Controller, especially in Object-Oriented systems with extreme late binding and state encapsulation, is a well-tested approach to managing state in a dynamic and changing Graphical User Interface using an object-oriented programming language.

- Functional programming makes code easier to test and reason about. It makes the task of proving its correctness more straightforward.

- Functional *reactive* programming tries to bring those benefits to the user interface or event-based code.

- Progressive Enhancement aligns the app structure with the grain of the network and is a way to make partially distributed software a touch less complicated to implement.

Software has a lot of paradigms. Some of which can be used in combination. Others are more holistic and make demands of the entire process. But they all have in common that they are proposed solutions to common problems.

As the frameworks you look at become more systematic, you'll note that they formalise many aspects of how the paradigm is implemented. By providing you with a prefabricated system they are *reducing variability*. Which is ideal. But they are also *systems* and systems generally only do what they want to do. If the system doesn't do what you need, you'll be fighting it every step of the way.

You want a framework whose approach to its paradigm matches your needs. If it does auth, you want a framework that does authentication and authorization the way you want. If it manages state, it should do it in the way you know matches your app's requirements. A systematic framework that fully matches the way your project is supposed to work will improve the odds of your project's success.

If you're wrong about that fully systematic framework, if you didn't actually know it and its idiosyncrasies as well as you thought, or if what your project needs turns out to be different from what you expected, then there's a good chance you will have to abandon that framework and start the project again from scratch.

Or, you deliver a software project where every update, every new feature, and every maintenance cycle becomes a struggle to avoid descending into a death spiral.

Your choice of a framework can make or break your app. Sometimes you're lucky, and the platform leaves you with only one logical choice.

Sometimes you aren't that lucky, and you need to resort to the time-tested tool of the desperate but hopeful: *look for inspiration*.

15

You need to study exemplary software

I wrote earlier about the importance of developing *taste*. Anybody who works in software development *needs to study software*. It's an essential vaccination against the pop culture virus that's endemic in programming. You can't leave the study of exemplary software to the designers. You need your favourite apps, interfaces you think will work well, and code you find impressive, to be able to develop software development as a craft. It doesn't matter if you are a coder or non-coder, designer or developer, project manager or product manager, you *need* to know the field. You are all a part of the craft of making software.

This isn't just because developing *taste* helps you do your research, nor is it just because it gives you an awareness of history that balances against the relentless *now* of pop culture. It's also a valuable decision-making tool. You need to know your software because that's how you choose a framework.

Paradigms are defined through practice. Software paradigms are defined through exemplary software. You don't begin a software project by deciding on a paradigm first, then a framework, before finally moving on to research, design, and development.

That's starting at the wrong end of the process and it will doom the project to failure as a result.

The first step, before you choose a framework, is to find inspiration —the apps whose example you would like to follow. Ideally, this would be done in conjunction with the user research—preexisting apps in your space aren't just competition, but can also be sources of inspiration. The research should be done before you choose a framework and start programming. You need a body of *prior art* whose approaches you like, designs find inspirational, and who embody the kind of software you would like to make with your project.

Software similar to what your project is trying to accomplish is ideal. Inspirational libraries or modules can also be helpful if they represent core parts of your software, like a rich-text editor when you're making a writing app. Frameworks tend to make poor exemplars as they rarely solve meaningful problems for the actual end user. They tend to inspire features for feature's sake. But you do need to be able to recognise a well-made systemic framework, so you still need to study them.

Once you have a body of work that defines the project's aspirations, *then* you go on to figure out what paradigm best represents that body and which framework best systematises that approach.

You may be most comfortable with React-based Single-Page-Apps, but if the project's exemplars, as discovered by research, are all Progressively Enhanced content-oriented websites then staying in your comfort zone will threaten the project's success. If you prefer lightweight frameworks with little client-side code and the exemplars are all offline-first, then going with your preference is likely to cause problems.

The exemplars aren't law. You don't have to use React even though some of the apps you find inspirational use it. But they will point you towards an overall approach—a paradigm. Sometimes that approach is easy to adopt and there is a systemic framework tailor-made for it. Sometimes there isn't and you need to find one you can adjust or coerce into working for your project.

Exemplary apps help you choose a framework, but they also help you decide. They define a common language for your team—a shared lexicon of references and examples that will guide and coordinate your work:

- Examples clarify definitions and intent more accurately than any form of documentation. Instead of trying to define what kind of *popup* the app needs, you instead use an app as a reference point. "Popups like App X has."

- Exemplar apps simplify your decision-making. If an app you and your team like has a rich-text widget that works, then the first step in building a widget of your own is to see how they built theirs. Having app role models can guide your decision-making.

- They help with ideation. Studying and comparing apps *will* generate implementation and design ideas.

An app that represents an ideal role model, operates in a similar domain, uses the same framework as you intend to, and is generally accepted to be of high quality, is one of the greatest gifts you can receive in software development. You don't need to be able to copy any of its code for it to transform your work and your project. Being able to analyse its designs and inspect it in the browser developer tools is productive enough on its own, but if you have the opportunity to read through the source code of a well-made app *seize it immediately*. Reading good code with a direct parallel to your current work is one of the biggest learning opportunities you can get.

Open source exemplars are particularly invaluable.

Exemplars help define your approach both in design and implementation. Attempting to make a web app without any exemplars, either because you haven't looked for them or because the frameworks or languages you're using are too new, guarantees confusion, chaos, delays, and error-filled code.

You need a context to work in—to know your genre. Without that context, you risk getting lost in the wilderness.

That context also gets you started on the next step in choosing a framework—something you really need to do before you fully commit to a framework. Context helps you discover the project's *needs*, of the team, the project, and the end user.

16

Check your needs

Another important step when starting a new project is a list of needs in order of priority for each constituency.

This is another step where software projects commonly fail. The implementation might take a while, but this is often the moment where it went off the rails.

I've been a part of more than one project where the needs being addressed boiled down to: "we need to build something that demos well to make sure the project gets funding; we'll figure the rest out after that." Sometimes this is explicitly stated, though maybe not in writing.

Two years later, the entire product is built, in Jenga fashion, on top of that rickety demo and is constantly on the verge of falling over.

Similarly, there are a lot of software projects where the needs are defined almost exclusively by other parts of the organisation. You need to deliver a component compatible with what the other team is doing; whether it serves the end user is immaterial. Or, because the company just got a new round of funding and is massively increasing the headcount, you need to build everything using React because you have no time or resources for training or acclimation.

Then you have projects with completely unrealistic and unresearched needs or requirements. They either plan on solving too many needs, or they are trying to solve problems nobody has successfully

solved before but are planning the project with the assumption that it's about as hard as a run-of-the-mill database frontend built with Rails.

If you genuinely want to address end-user needs, then you need to narrow the scope. Start simple. No, *simpler than that*. This is the stage where that happens.

You also need to do ***actual*** *research*.

It should be clear by now that user research is *not optional*. Nothing in the software development system works without it.

If you don't do user research, your list of needs is just fiction and theatrics.

Most software projects don't care about and aren't affected by actual end-user needs. Startups are flush with cash and grow by offering half-baked services cheap or free. Internal projects are implemented and shipped with no regard for internal productivity. Big tech companies have a lock on the market and get paid no matter what they do. Google makes its money on ads. Crap versions of Android or ChromeOS have zero impact on its bottom line. Apple's steady decline in software quality does not affect its revenue in any way. Microsoft hasn't shipped a genuinely good app since Microsoft Word for Mac 5.1, and even that wasn't as good as people remember.

This means you have to be honest with yourself when you're outlining the needs that drive the project. If what drives the project is a short-term need to guarantee project funding, then it should be your priority to figure out ways to address that need that don't sabotage other long-term considerations. Mediocre software with funding or organisational backing can be improved.

Common constituencies include end-user, dev teams, customers, sales teams, and recruitment. For many projects, the primary constituencies are the end-user and the development team. Other needs often fall from those. If the end-user is served, the business needs are addressed. If the development team's needs are also addressed, then there's less for management to worry about.

Pitfall number one: managing the project too early
With the needs in hand, it's tempting to dive right into managing the project. Outline the requirements, break them down into steps, and estimate the time and effort required.

But all of that should happen *after* this initial process. Once you have a concrete idea of which tools to use to build the project, backed by research, with several known projects as references or exemplars, estimating and overall project management should be easier.

Pitfall number two: *anti-needs*
You should avoid defining 'anti-needs': needs that are just declarations of not needing something. For example, when you say that an intranet site 'only needs to work on desktops, so bandwidth shouldn't be an issue,' you're describing the inverse of a need. It's a rhetorical trick to persuade yourself that other long-term needs can be disregarded. It would be more accurate to say: 'needs to work on employee devices,' which puts the onus on you to discover what those are likely to be before you make your final decision.

Pitfall number three: *familiarity needs*
The point here is to help you choose a framework and approach. So, instead of putting "needs to use React" in your list, you put something like "needs to be within my comfort zone", as that opens you up to using tech and tools similar to React. It's marginally less limiting.

Or, if the project is of a "test kitchen" type and has a specific purpose of trying out new ideas and approaches, it could instead be "needs to *not* be in my comfort zone".

The need for innovation

Next, you need to figure out whether technical novelty is a genuine organisational and project need. How important are these capabilities? What is your appetite for seeking out and learning new technologies or approaches?

High

The point of the project is to try out new things. Learning is the highest priority need, everything else matters less. A working project using familiar technologies and approaches would be a failure. This is typically something you do in preparation for a larger project.

You've found a tool that could be very important for the main project, but it's unfamiliar to everybody on the team. By testing it out in a software library 'test kitchen', you gain experience with the tool and a better sense of how well it would work for the larger project.

Moderate

This is where you try out a narrow set of new ideas in a project that addresses real needs. This can go very wrong. A single feature like "PDF support using PDF.js" can easily take 10x the time and effort you estimated beforehand. The less familiarity you have with a problem domain, the less ability you have to make accurate estimates.

Low

Where you have a list of needs, a budget, and a low tolerance for failure. You're only going to use familiar tech. Innovation is going to be extrinsic to the system and come from design.

That last point is a big one: familiarity. Most developers do better when projects are built using technology and tools that they are familiar with. Very few of us are in a position where we get paid to do realistic 'test kitchen' style projects specifically to figure out new technology. It's easy in web dev to get stuck in a rut where you only use a single tool like a hammer and see every problem around you as a nail.

I've worked in PHP, Python, and Ruby, but those projects were in the distant past. I am reasonably confident I could get up to speed with Go fairly quickly if I ever need to, but I haven't tried it. I can't know for sure. They're all great languages, each in their own way. But the value they could bring to any of my current or upcoming projects doesn't make up for my lack of familiarity.

The familiarity argument has its limits, as it's easy to sabotage your projects by refusing to learn tools and languages that could have an outsized impact on them. Sometimes solving a problem with CSS is

10x easier than with JS, and vice versa. I've seen web developers undermine their work by refusing to learn standard JS, CSS, HTML, or DOM/browser APIs. Even if most of the work on a web app is done in a framework, familiarity with the web's foundations helps you choose that framework and use the one you choose more effectively.

Once you have your list of needs and a sense of your appetite for novelty, you use that as a gauge to test how appropriate the various tools and tech you encounter are to your problem.

This is the point where things become clearer. Once you have user research, an overview of both user and project needs, and a collection of examples to follow, you can start to cross-reference these to narrow down your options. You won't follow the framework and dependency lead of exemplars that don't fit your or the user's needs. Taking it all together and it will be obvious by now what your overall approach should be in terms of a paradigm and you should have a manageable shortlist of frameworks to choose from, systematic or paradigmatic.

You may even have come to a single obvious choice by now. But, before continuing, there is one last step in choosing a framework: truly digging into and understanding its paradigm.

It isn't enough to know *what* a framework does. You need to know all of the *whys* as well.

17

Understand the paradigms

Even though paradigms are defined through practice—exemplary software—they can usually be explained more easily once they have matured. Once there are a few stable systems based on the paradigm running in the wild, a definition can come from observation and study instead of guesswork and fantasy.

Sometimes the framework offers that explanation as a part of its documentation.

Apple, for example, has a decent explanation of the Model-View-Controller paradigm that underlies its AppKit (née Cocoa) framework in its documentation, including the rationale that drove its original development.

Most frameworks that implement the Model-View-Controller paradigm document in some way what exactly their interpretation of it is. Look at C# Model-View-ViewModel frameworks, and you'll similarly find a thorough explanation of what MVVM means in the context of that particular framework.

These paradigm definitions, even though they are relative to the framework in question, are invaluable aid in helping you confirm your choice of a framework.

Unhelpfully, it's usually the frameworks you are forced to use—they are the platform or vendor default—that best explain their

paradigms. You don't use Apple's AppKit because you like its take on MVC, you use it because you need to make a macOS app.

True to form, it's in web development, where the proliferation of frameworks and meta-frameworks is out of control, that this aspect of each framework is commonly left undocumented. Worse, the concepts and paradigms used for web apps can get quite involved. Even when they are documented, they often leave out core ideas for you to get stuck on when halfway through the project.

I'm not going to lie to you and say that the web's paradigms are simple and easy to follow. We wouldn't need this chapter if they were.

What helps us is that despite the endless number of frameworks, there are only a few controlling paradigms in web development, and you only need to grasp the three most common.

Each comes with a worldview that colours how you see the job and dictates how problems should be solved, and which problems should and shouldn't be solved.

WTF is RESTful Progressive Enhancement?

When people talk about Progressive Enhancement, they leave out the "RESTful" part. This leads to a lot of confusion and misunderstandings as REST is an integral part of how state is managed in Progressive Enhancement, and state is where things go wrong for almost all software.

REST means it's built around *REpresentational State Transfer* (Roy Fielding, 2000). This isn't meant to be a technical book, so I won't go into detail about what REST is and how it works, but the basic idea is that it's a way of structuring an API over the HTTP network protocol.

- The server represents the software's data as *resources*.
- Those resources are exposed as URL endpoints.
- Those endpoints can be accessed, and their resources manipulated using standard HTTP verbs (GET, PUT, DELETE, POST, PATCH).

- The server translates the actions on the resources into modifications of the underlying data.

The client does not get to manipulate the data directly. It manages them indirectly by getting, changing, and then sending back to the server *representations* of the data. The work is done via the resources.

This style of structuring APIs has come in and out of fashion over the past two decades. (Remember, programming is a *pop culture*.) But, because it builds directly on the HTTP protocol, it gets many features for free:

- Fine-grained caching.
- Multi-level caching. On the device, local network, wide network, and at the API server level.
- Transparent support for proxies.
- HTTP servers, caches, and proxies that have seen heavy use and are generally well-tested.

REST is what makes the *Progressive Enhancement* web development paradigm work.

Normally, Progressive Enhancement is explained from the perspective of either the end user or the developer. The Dao of Web Design, written in 2000, captures this well. It presents the practice as built on the assumption that the web developer (called "web designers" back then) does not fully control the page. Browsers, Operating Systems, and end-users have the final say about what happens on the page. They can block JS or CSS. Their environment might not support the features you thought were universal. They might or might not have a screen.

The web developer responds to this by creating adaptable pages that have a functional core and are progressively enhanced based on what their current context supports.

Progressive Enhancement, from this perspective, isn't about making a site that's fully functional without JS but is instead about deliver-

ing a usable core experience that the page then expands upon as additional capabilities in the current context reveal themselves.

But from our perspective, *Progressive Enhancement is a paradigm, based on REST, that's implemented natively as a partially systemic framework by all web browsers*. It's what's baked into the platform and is to web development what the Model-View-Controller paradigm is to Cocoa.

The client only gets *representations* of the entities and data in question. These representations and the methods used to alter their state have native support in the browser—HTML and HTTP. State is handled by the server, using RESTful principles, and the browser only gets access to that state through resource representations. Business logic is implemented on the server because—remember—*this is still just an API server*. It just happens to return API resources that a human can interact with.

Even before the client enters the picture, the app should be fully featured and structured as an API server that works with HTTP verbs, form submissions, and HTML—all built-in features of every browser. RESTful Progressive Enhancement bakes state management right into the foundation of your web app in a way that's natively supported by the platform

The web developer's job in this framework is to take those HTML resources and integrate them into a functional and friendly user experience. This can be a lightweight layer of JavaScript resources. Or, the front end can bootstrap a heavy interactive experience that looks and feels identical to a Single-Page-App. But they all operate through resource manipulation using HTTP.

Front-end state is usually managed by *manipulating the resource itself*: instead of complex state libraries, they tend to save state into the HTML-based Document Object Model in the form of attributes, properties, and nodes.

It isn't the *weight* of the front-end that makes it Progressive Enhancement but that the front-end is built up from an API server that works with HTML representations and that those resources are the

"single source of truth" for state. The server can "speak" other representation formats, but HTML is always its first language.

This model for making web apps has a few advantages.

- All of the core logic for your service or app is consolidated in a single, controlled environment: your server. This can reduce bugs and errors in your code more than testing or type checking combined.
- The framework is natively supported by the browser, which also means less work for you in maintaining the framework.
- You can implement feature-rich meta-frameworks—convenience layers built on top of Progressive Enhancement like *Enhance*—with much less work as the foundation is the platform itself.
- Starting with HTML—forcing you to build a functional baseline using markup—gives you several accessibility features with less work.
- These web apps generally have broad reach as they can often scale down to work well on cheaper devices.
- By starting with functioning HTML resources, most projects don't need as much client-side code as you might expect. The REpresentational State Transfer model with HTML takes care of much of the baseline needs of the project.

The risk with Progressive Enhancement is you could end up needing a feature or experience that is hard or impossible to implement using the approach.

- As state is constructed from independent resources, it becomes hard to manage state *across* resources. It can be done by bootstrapping history management on top of the HTML resources. But breaking the resource-based model of state management will make your app unwieldy in short order.

- Many UI designs do not work well with the resource model of state management. You'd have a hard time building something like Figma with this paradigm. These projects are better served by a paradigm specifically focused on managing how the views are updated in response to changing state. (*See Single-Page-Apps, below.*)

- Because they are structured using independent resources, Progressively Enhanced websites also tend to show their work. Their seams and history can be exhibited as a part of their design and behaviour.

WTF are Single-Page-Apps?

Conversely, the Single-Page-App paradigm doesn't concern itself with state. Or, to be more specific, it only concerns itself with how the user interface *view* gets updated with new state. The paradigm is more interested in the *consequences* of state than in managing state. If you are creating a complex user interface, that is *the* issue to solve. It makes building views—even complex ones—and updating them much simpler. The main concern of the Single-Page-App paradigm and the various frameworks that implement it is how *view state* is synchronised with *application* or *session state*. How that application state is managed isn't defined by the paradigm. It will only ever describe a partial system, requiring you to provide a functioning subsystem of your own that manages state. This is why there are dozens, if not hundreds, of different state management methods and libraries for Single-Page-Apps in existence.

This is great if you are trying to build an app with involved views that have to be responsive to the user's actions. Maintaining a consistent and performant *visual* state for a complex view is a hard problem.

It can be a disaster if you have straightforward views and all of the value your app is supposed to generate comes from managing state with consistent and correct business logic. Most projects overestimate

just how complex the views of their apps are, and when they don't, that's because they made the views more complicated than they need to be.

The added work over the built-in Progressive Enhancement framework is substantial and generally results in big JavaScript payloads and risks making the app non-functional on lower-end devices.

The paradigm has an advantage because of its focus on view update efficiency and performance: it tends to result in much more reliable and efficient network use, *after the app has loaded*. The initial load time is a hurdle where, in extreme situations, low-end devices can stall for seconds *just while parsing the app's code*. But once the app has loaded, the view is stable and persistent—should be cached and saved locally—and further network requests can use minimal representations instead of the bulkier representations used for Progressive Enhancement.

Another tradeoff between Progressive Enhancement and Single-Page-Apps, beyond the fact that the former is built into the platform, is that the former has reliable state management while the latter favours reliable networking, and the former has fast initial loading times while the latter has fast secondary loading times. They each have their strengths.

The Single-Page-App paradigm lends itself to two types of websites: social media and collaborative productivity apps. Both are built around long-lived user sessions, and neither lends itself well to resource-oriented state management. Choose it when it's important to be competitive or have near-feature parity with a native app—when your end-users need for functionality is higher than their need for access or your need for reach. If your revenue correlates with the length of the user's stay, then Single-Page-Apps work.

The potential for complexity in a Single-Page-App is much higher than with any other web development paradigm. It's built on top of the Progressive Enhancement framework, which inevitably makes it more complex. Beyond that, the point in reimplementing browser-native features like history and navigation is to not be bound by their limitations. Without those limitations, the ceiling for implementation

complexity is raised because you are no longer bound by the built-in assumptions of the browser vendors. Those reimplementations, even if you are using libraries or frameworks, need to be managed as dependencies and potential sources of bugs. State management will generally be more complicated with a Single-Page-App, if not just because it isn't a predefined part of the system. The freedom to choose how you manage state *will* lead you to choose more complicated or less mature methods.

Under the right circumstances, a Single-Page-App can deliver an amazing experience. Usually at a high cost.

WTF are Transitional Apps?

As Single-Page-Apps have shown their limitations, we've seen a new generation of multi-paradigm frameworks. They build on the idea that the prior worldviews weren't actually that incompatible, that you could take from each paradigm and mix and match their approaches and tactics. These frameworks vary a bit in terms of just how flexible they are, but most of them let you use Progressive Enhancement, client-side rendering and routing, and static rendering, all in the same web app.

These frameworks preserve some of the resource-based model for managing state through REST—each URL corresponds to one resource—even when the app is switching from static rendered, to client-rendered, to server-rendered from view to view. Even when they don't literally use HTTP for state they still let you use HTTP *semantics* to structure your state management. They also tend to let you preserve the Single-Page-App network model as needed: the client-rendered routes can be updated very efficiently as in a regular "SPA".

The poster child for this convergence is SvelteKit. Rich Harris, SvelteKit's creator, was the first to coin the term Transitional App (Rich Harris, 2021). "Transitional" is directly lifted from interior design discourse to indicate that it combines multiple schools of thought: a ***convergence***.

Transitional app frameworks are too numerous to list here, and that list would quickly be out of date anyway.

This explosion in the number of transitional frameworks isn't because everybody ran out to build one after Rich Harris's talk. What happened was, by giving it a label and its first *exemplar*, he turned a broad trend towards convergence into a *movement*. Many developers now had a name for what they were working towards. Existing frameworks changed their approaches; new frameworks popped up; a new paradigm caught the programming pop culture's attention.

These new and adjusted frameworks aren't interoperable in any significant way. They can even differ substantially in their approach and design. They have in common the idea that you shouldn't have to abandon everything about REST and Progressive Enhancement to get the dynamic views of a Single-Page-App. And vice versa.

This could be evidence of a broad trend towards convergence in web development which might indicate that in the long term, the community will settle on a single, overarching paradigm.

Transitional apps are too young for anybody to say with certainty that they are going to work better than the paradigms it aims to replace. It's too young to be coherently defined in words; its meaning is defined through examples. It's just starting out. The future isn't guaranteed to be transitional.

In theory, the benefit to the transitional paradigm is that each part of your web app would only be as complicated as it needs to be and no more. The pages that can be static will be static. Other pages would be server-rendered with minimal client-side JavaScript as needed. The complex interactive views of your app would operate on the principles of a Single-Page-App.

In theory, this should ensure that you get both the user experience benefits of the Single-Page-App but can rely on the state management and distribution benefits of Progressive Enhancement *as needed*.

It's a good theory. It's unproven, but it's a good theory, and I'm mostly inclined to believe it will turn out true.

Using this approach to make a production web app or service is risky. There aren't many exemplary apps or role models to follow. We

aren't certain the model will have a concrete benefit. Maybe transitional apps will degrade into Single-Page-Apps with no meaningful benefit for complexity or effort? Maybe, we need better frameworks for the existing paradigms? It's hard to tell at this point.

The frameworks themselves are stabilising. Many other frameworks were born out of need and were used for production websites from the start.

The safest thing to do is to wait and see, especially if you're currently working with Progressive Enhancement in mind. Why overcomplicate your work, risking your web app's stability, if you don't have to?

However, if your software project requires the behaviours that Single-Page-Apps enable, then a transitional app framework might work well for you. It might let you attain that user experience where you need it, without too much complexity, and with lower costs.

18

Maturity, Complexity, & Layers

You know, broadly, what kind of app you're making. You have a number of exemplary apps that serve as your benchmark and reference. You are aware of your needs, the team's, and the user's. You have a clear understanding of what will be the app's paradigm.

Now you need to narrow your shortlist of frameworks down to one.

Sometimes this part will be straightforward. You're a Ruby shop and need to make a web app: odds are that Rails is your only meaningful option, and anything else would be a pointless risk. But all too often the list remains cluttered and overwhelming.

Need to make a native macOS app? You could go with AppKit. Or, UIKit with Catalyst. Or, SwiftUI with AppKit. C++ with Core* APIs?. Catalyst with SwiftUI? AppKit with SwiftUI components? All-in on Swift or mix in a bit of Objective-C?

C# and Windows? Then you have Windows Forms, WPF, WinUI, and .Net MAUI to choose from. Last I checked. I probably missed some.

Framework proliferation isn't an issue unique to web development. It's become a problem for every software platform available. Narrowing it down by OS support and implementation language still leaves you with half a dozen options, *at least*.

You need to narrow down your choices, somehow.

The first step is to filter by *maturity*.

Don't use immature frameworks

You can't look to pop culture discourse for an answer on which framework is or isn't immature. Whether a framework is considered outdated, obsolete, new, cutting, edge, or mature by software developers has little to no correlation to actual maturity. A framework needs to be unusably immature, with constant errors causing endless frustration, to get a reputation for being 'not quite ready for primetime.'

We need something a little bit more objective.

If the framework regularly changes or deprecates core APIs, then it's immature. It doesn't matter if they use semantic versioning or how much advance notice they give. A framework isn't mature if its APIs keep changing. It means they haven't settled on a solid design (immature framework). Or *their priorities keep changing* (immature owner).

SwiftUI is an immature framework. Core APIs and practices change with every version. They still haven't settled on a solid design. Every version seems to be closer, and one day the changes will slow down to a more natural pace.

React is an immature framework. Core APIs and practices change regularly, even a decade after launch. These changes aren't happening because the React team is incompetent. They happen because React's primary owners (Facebook and a small group of well-funded startups) are immature organisations whose needs and requirements keep changing. React will only mature when its owners stop changing as quickly.

Maturity isn't a question of time but of structural consistency and system stability.

Projects with a low tolerance for risk should not use immature frameworks. They are always a gamble. Will their innovations save you enough work and create enough value in the future to make up for the value your code loses through regular changes?

Prefer simpler systems over complex

Remember Gall's Law?

> 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.
>
> **Galls Law**, *from The Systems Bible (John Gall, 2002)*

At the start of a project, the framework provides the initial system you are extending and building up towards your requirements. If that system is already complex, your odds of project success will crater. Complex systems have multiple elements that interact in ways that are hard for you to foresee.

This is the flip side of the maturity question. Many mature frameworks have evolved *alongside* their exemplars—the apps and software that exemplify how the framework should be used. When you attempt to build something similar using that same framework, you will often fail miserably because you are trying to extend a complex system with complex interactions. The exemplary apps won't have the same problem because the system was simple when their projects began and grew with it. This is a frequent issue with single-vendor frameworks. Whenever you have a framework whose primary purpose is to support a single product or service, you often end up with interwoven complexity that makes using it for a new service much harder.

This is a big reason to try to stick to platform frameworks. They have the incentive to keep it as simple as they can because they need to serve both new and old software projects.

But there is another reason for sticking as close to the platform as possible.

Avoid friction between implementation layers

There is one more filter you can use if you still have too many frameworks to choose from: *layers*.

Software is built up in layers. There's firmware, the kernel, low-level userland code, foundation graphical code, platform frameworks, and, potentially, several layers of convenience frameworks. These layers exist to simplify software development. Without them, we'd still be writing directly in Assembly.

This is only a problem when there is friction or a mismatch between layers.

Desktop Linux suffered from this issue for quite a few years. The X Window layer, which every one of these desktops was built on, wasn't designed with the needs of modern desktop systems in mind. They require visual effects to provide a tolerable user experience. Most desktop Linux distributions have now switched over to another system to provide the rendering layer—Wayland—that more closely matches the needs of a modern desktop.

The problem wasn't the layering—they didn't solve it by dropping the layer entirely—the problem was friction. The affordances the lower layer provided didn't match the needs of the higher layer.

The more layers a framework builds on, the more likely you are to experience friction or a mismatch somewhere. In addition to a needs and affordances mismatch, you can also get a *pace* mismatch. An immature lower layer that keeps changing makes the higher layers less stable. Layer interactions can also get very complicated, making bugs that come from a lower level indecipherable.

As those developing desktop Linux environments discovered, dropping a layer isn't always a solution. The convenience the intervening layer provides can make the difference between a great project and one that doesn't even come close to shipping at all.

But, if you have a choice between two different frameworks, both providing you with similar affordances, and one has more layers, your choice is generally clear: pick the one with the fewest layers.

If both have the same number of layers, pick the one whose design is less likely to have friction with the design of the underlying layer.

And, if you *still* have a number of frameworks to choose from, even after the exemplars, needs list, familiarity, and paradigm analysis, and all of your remaining choices are similar in terms of maturity, complexity, and layering, then you should go with your gut.

If all the remaining frameworks come out the same then they probably *are* effectively the same, and you can pick the one that makes you the happiest.

19

In the end, what else is there but work?

In software development, it's a struggle not to get caught in the pop culture current of the day. Platform developers are anchored by their work. Even if an established framework or platform falls out of fashion, it will still have legacy users for as long as those apps or services are maintained. But the rest of the community doesn't have the same anchor. Developers routinely hop between jobs. Every new workplace and project is an opening for the pop culture tide to grab you with a new trend. You get so enmeshed with the culture that you become one of the drivers constantly pushing for the *new*.

—*Have you tried Bun? It looks amazing. I'm looking for an opportunity to use it somewhere. It seems to run Remix well. At least, I think so. For the most part. I haven't tried it myself...*

We use tool novelty as a proxy to measure progress in the field. To us, a new tool means that software development is now *better*. Old tools represent the past. To use them is to slide back down the curve of progress.

We need to resist this when choosing a framework. We can't let the mob decide for us. We need to choose and be structured in our choice.

But you can only spend so much time on preparation and study. After building a library of exemplary apps and digging into and under-

standing the paradigms in question, you are usually only left with so many choices. Knowing you need to make a certain kind of website, for example, combined with what you know of your own or your team's skills, is likely to narrow down your choices. More experience opens up your choice of frameworks and gives you the skill to assess them better. It takes work.

Having gone through this exercise once—done the full user research, built up a collection of inspiring exemplary software, gone deep into studying the available paradigms, and analysed the maturity and consistency of a number of frameworks—will make it easier for you down the road. It's a skill. You get better with practice. You need to do the work, but you get better each time. The choice will take less and less time with each successive project.

In the long term, you will need to decide whether to specialise in a single framework. Doing so requires a change of perspective. You are no longer a person who looks at a problem or at a specific kind of user need and then searches for an appropriate way to solve it. You become the developer with a particular solution who is looking for the right problem.

That can be a great career. But, specialising in a particular approach and then approaching problems like a generalist is a recipe for software failure. You'll need to take a page out of Nintendo's books and follow Dr Sarasvathy's principles of looking first at the resources you have and what problems those can solve.

Sometimes we don't *make* the choice, it just falls out of the work we get. We take projects, work our jobs, and a few years later discover that we are a specialist in our field.

I wrote before that paradigms and frameworks are defined by practice—by the work. That applies to us as well. We are what we do. We are defined by our body of work and the skills we developed making it.

It's up to us to make sure we can be proud of it.

Part Four:
System Failure
(Bonus Essays)

20

The Collapse of Complex Companies

> A *society has collapsed when it displays a rapid, significant loss of an established level of sociopolitical complexity*. The term 'established level' is important. To qualify as an instance of collapse a society must have been at, or developing toward, a level of complexity for more than one or two generations.
>
> *The Collapse of Complex Societies, p. 4 (Joseph Tainter, 1988)*

2022 has heralded a season of mass layoffs for most of the big companies in the tech industry. Meta (Facebook) and Intel have cut double-digit percentages of their staff. Other companies are rumoured to be preparing their own cuts. Twitter has cut its staff numbers in half, with more inevitably to come, either through attrition or further layoffs. Investors, none of whom understand how software development systems work, are campaigning for even more bloodletting on a scale not seen since the dot-com crash in the early 2000s.

To be fair to the investors, the executives at these companies don't seem to understand software development systems, either. Remember, most software projects fail, *especially those at large tech companies*. The history of these organisations is filled with unsuccessful research

projects, failed attempts at innovation, and software products that never shipped or were discontinued after failing in the market.

Layoffs, at this scale and in this short period, are extremely risky for companies with currently sustainable software systems.

A company is a complex system

A software company is an intricate complex system composed of numerous elements and complex subsystems. A *large* multinational tech company is an even more complex beast made up of countless subsystems, each with the complexity of a regular software company.

These companies don't become this complex overnight but through a long evolutionary process where the system grows in response to increasing inflows, i.e. rapid growth in revenue, users, and stored data.

These companies experienced cycles where the cost of the increase in systemic complexity was less than the inflow increase the added elements and feedback loops were handling. In Joseph Tainter's terms, the marginal productivity of increasing complexity was positive and increasing (1988, p. 93). These cycles continue for as long as the inflows continue, even when the marginal productivity declines. The systems of companies this size are too complex for any single person to understand. Any given executive will not be able to pinpoint when the benefit to hiring or organisational growth dissipates and becomes negative. This is even harder for software companies.

A zero marginal cost product (i.e. making new copies costs nothing) like software means that most sustainable software systems will have enormous financial margins. This means that these companies can grow *well* beyond the point where that increase in complexity has negative returns.

Add to that:

- Numerous vanity projects.
- Misguided attempts at faux innovation where managers try to force the outcomes desired by the top level.

- Boosting staff numbers for bragging rights and organisational power

Finally, add sheer executive hubris, and you get companies that make an enormous amount of money but are unlikely to be sustainable in the long run.

This lack of sustainability has nothing to do with the peaks and troughs of the surrounding economy and everything to do with the fact that these executives are ignorant, incompetent, and greedy.

They are, not to put too fine a point on it, ***fools*** who stumbled into their positions through happenstance and luck.

They have built toxic systems they do not understand and are now responding to a decline in the worst way possible. Their thoughtless growth risks being followed by equally ill-considered collapse.

Recession, layoffs, and a collapse in complexity

These companies have never experienced meaningful setbacks in the short period of their existence. Next to none of them had been founded during the dot-com collapse. Many even only got their proper start after the 2008 recession. Some have had setbacks similar to what the industry is facing but were much simpler systems at the time.

System complexity is not just a function of revenue but also of the number of users and data stored. Microsoft was a much simpler system back in 2000. At the time, it only stored a small fraction of the data it does today. *Data storage has increased almost exponentially for all major tech companies over the past decade. Their systems have grown with it.* Microsoft also had fewer users—it had a monopoly, but the overall market was smaller. Even with that much simpler system, the layoffs it has experienced over the past two decades have left scars across its software so big they are (figuratively) visible from space. Scratch an interface in a Windows system, and you'll find a component unchanged from the XP or even Windows 2000 era. Microsoft's

software is sedimentary layers of code nobody understands or dares to change *because* of the layoffs it has experienced.

Microsoft was the lucky one. Their biggest rounds of layoffs happened in a different era, before the rise of the "cloud" and the endless collection of user data. Even without the spectre of state regulation, proper handling of user data requires substantial system complexity. Mistakes, leaks, and data loss can have devastating effects on company outcomes. *Because* these mistakes are also devastating to the end user, the existence of these data 'lakes' demands regulation. You can't have one without the other. More regulation means more complexity and more intricate systems.

The collapse that many tech companies are facing today has two causes. The first is a recession: lower revenue has revealed the inadequacies of their systems. The second is regulation. Their carelessness in collecting user data and how they handled it has resulted in regulation, which lowered the overall productivity of their companies as a system. Collecting all that data is now beginning to cost more than it returns. *But they can't stop*. The *purpose* of their systems is to collect data. The only way to stop collecting that data would be to rebuild a new system from scratch: *it would require a new company*. They can tinker with the system, but that won't change what it's designed to do. It will continue to collect data, even as costs increase, even as revenue declines—until the system collapses or is made to collapse.

That's the gamble they are making (or are about to make). They are betting that a controlled collapse will result in a simpler, functioning system that they can grow again in a manner more suited to today's circumstances.

This is unlikely to work. This wave of layoffs will result in zombie software products that either can't be improved or have to be rewritten at a massive cost.

Gall's Inverse

I'm fond of citing Gall's Law:

> 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.
>
> **Gall's Law**, *from The Systems Bible (John Gall, 2002)*

But, there is an inverse of Gall's law that I call *Gall's Inverse*:

A simple system that works is invariably found to have been built from scratch as a simple system. Creating a simple system by hacking away at a complex system never works and cannot be patched up to make it work. You have to start over with a working simple system.

Companies that need major reductions usually downsize by lopping off entire systems. Layoffs are quickly followed by products or services being put out to pasture. You don't try to operate a factory with half staff. You close it. While still painful, this can work. You aren't trying to simplify the system; you are maintaining core system complexity while lopping off systems that feed into it or off it.

The best-case scenario for across-the-board layoffs with deep cuts into functioning software development systems is one of two outcomes:

1. Stagnant products that nobody understands well enough to improve or alter.

2. Substantial rewrites at huge costs by new staff.

The former almost inevitably causes the product to be discontinued and, if it happens to a big enough portion of the company's output, will lead to a steady decline and an inability to compete in the market.

The latter almost always ends up being *more* expensive than not cutting the staff numbers in the first place. But since those expenses come

in a different quarter, further down the line, nobody will connect the dots between the two.

Lucky companies can coast for years with a stagnant product, especially if their product development consisted mostly of tinkering with systems that were already working, annoying users. Some users might experience the stagnation of a mismanaged software company as a *net benefit* to their use of the product.

This decline can often only be counteracted if the company has an opportunity to build a new working system from scratch using its remaining resources, as Microsoft did with Azure.

Those are the best-case scenarios. The worst-case scenario is a company death cycle.

The risk of a death cycle

We have never seen layoffs of this magnitude before in systems as complex as the modern tech company.

We have never had companies as complex as the modern tech company.

We have never had complex systems that gathered as much data as the modern tech company.

If you try to fix an unsustainably complex system by simplifying it, you end up burning it to the ground. If you fix how you handle your data, you increase costs. Tweaking a system designed to collect data to collect less data more selectively will increase costs. Changing your processes to cull unused data will increase costs. Trying to lower costs by laying off staff working on the system will counteract your improvements to data handling, leading you to add more processes and features to the system in response, which increases costs again. Regulatory requirements will only increase because *a system capable of intelligently collecting selective data is more dangerous to society than one that blindly keeps everything*, which leads to higher costs.

Everything they do will increase costs. This was fine for as long as the revenue increased. But spiralling costs, *and* lower revenue *and* a complex system getting less functional with every cut is a recipe for a

death cycle. The cycle will only accelerate if their zombie software products start to haemorrhage users.

The best-case scenario for many of these companies is that they come through it like Microsoft. After years of being unable to compete, with a product catalogue full of zombies and bug-ridden rewrites, they finally rebuild their systems to compete in a new market: cloud hosting services (Azure). But, unlike Microsoft of the early 2000s, we are facing a rapid dissolution of globalisation, an escalation in international tension, and compounding environmental crises.

These companies might not have the luxury of dysfunction over the next few years.

The best way to avoid collapse is to have built a sustainable system with margins high enough to tolerate substantial declines without having to alter the system in response.

This never happens.

Most software companies overshoot their software development system complexity by a substantial margin *because* the margins on software businesses are so good. When times are good, everything you do seems to lead to growth. You begin to associate your random actions with random fluctuations in the market. Managers build mental models for how the market works that have no relation with reality.

Making considered decisions, under those circumstances, is next to impossible.

The first step to try is what many of these companies are doing right now: they are cutting both vanity and research projects because none of them can tell the difference between constructive investment in the future, and wasting money on an executive's ego. This makes it harder to respond to changes in society and the markets. But it can be done without threatening the system as a whole.

Some have already reached *the second step*, which is to cut staff working on secondary products. These are usually products that are either strategic complements to their core or strategic uses of some of the resources built up by the core system. Cutting the complements will lead to a decline in inputs to their core system because that's what the complements were providing. Cutting staff working on the stra-

tegic uses will lower the overall marginal product of the core system because their sole purpose was to increase the returns on their core system.

Both risk leading to a sharp decline in the inflows of the core system, destabilising the whole and making it more vulnerable to shocks.

When that shock happens, they reach *the third step*, which is to cut staff working on core systems. Any complex system that reaches this point risks entering a terminal decline.

Avoiding the Collapse of Complex Companies

Tech companies are, by their nature, fragile systems. Irrespective of reliance on a complex, easily disrupted, globalised supply chain, their software systems are inherently frail.

They build software systems too complex for a human mind to comprehend. Windows has been estimated to be over 50 million lines of code. Google is 2 billion lines of interconnected code. Apple's macOS is rumoured to be over 85 million lines. Facebook is at least 60 million (McFadden, 2021).

There is no way that staff at these organisations know every nook and cranny in their code bases. They are companies built on layers of the unknown, lost, and unexplored. They *don't* know what will happen when the dark corners of their systems begin to break.

The more people they lay off, the more of their software will be covered in shadow. Maybe there isn't anything brittle and load-bearing hiding in these corners, waiting to snap and bring the entire system down. Maybe they're lucky.

But that's all that will separate the survivors from the rest: *luck*.

You can't remove fragility from a brittle system.

What you can do is make your systems robust.

This is hard because fragility concerns go against every instinct in modern programmers and managers. On this, they are united. Complexity pays; complexity is scale; complexity brings wealth.

The only way to win is to stay simple: loosely coupled small systems are less likely to fail. And I don't mean systems here in the 'microservices' sense. I'm talking about software development systems, not just the software systems themselves. A simple system that solves a single valuable problem for many people is less likely to fail than a complex system that solves many problems.

Free software and open-source projects with sustainable communities are more likely to survive staff churn than any other software. They can still collapse—still die from coder burnout—but already have a more robust foundation. No one round of company layoffs can kill off WordPress, Ghost, Linux, or Git, and when that company is ready to grow again, it can draw from a pool of candidates already familiar with the existing code.

Projects that adopt common standards (de jure or de facto) will be more accessible to new programmers, helping with renewal.

Code built on mature, stable platforms, programming languages, and operating systems is more likely to survive stagnant periods intact than code built on the cutting edge.

Some platforms are riskier than others. Web servers are less likely to break as the underlying platform changes than an iOS app. Websites built using progressive enhancement will survive more trauma than those made as React Single-Page-Apps.

None of this is a secret.

All this sounds obvious—everybody agrees and believes in it. But, this serves more as a catechism for coders pursuing personal growth or a shibboleth for mature developers. They are phrases we use to identify each other. They aren't practices we pursue.

That needs to change. We need to be building organisations and software systems that survive crises.

Because, believe me, we have a lot of them coming.

21

Failure demand: when mistakes improve your metrics

You are exhausted. It's shipped. Good work. The new widget does the job. It implements the design beautifully, with nice touches like animation and typographic flair.

It's almost identical to the Operating System's built-in widget.

Almost. The native widget has a few more polished details your version missed. Of course, *your* version matches your organisation's branding needs, so in that way, it's a substantial improvement. But in every other way, it falls short. How could it not? The built-in widget can use APIs and interfaces your app will never be able to use. It doesn't have to worry about ballooning the app's download bundle.

You have just spent four weeks of your life over-engineering a widget that is still under-engineering compared to the built-in alternative.

I've written before about this pattern as it affects web development. It's an issue that plagues most software development.

The more generic term for this pattern is *Failure Demand*, or when failure creates added demand on the system. It was coined by John Seddon to describe how a system failing to serve the customer can

look busier and even *more successful* than a system that works (Vanguard Consulting, 2019).

The streamlined office

Imagine an office that handles permits and licensing for a small city. Its remit is wide-ranging, anything from hunting licenses to building permits. The processes aren't complicated. This city is somewhat sensibly run and has struck a sensible balance between over- and under-regulation. Nevertheless, the system is complicated. The process for handling each application can take anything between thirty to sixty minutes. Management decides to 'streamline' the office by introducing specialisation (experts should be able to handle requests quickly) and by introducing standardisation for the various forms and processes. There isn't any variation in how each permit or license is processed anymore. This lets them consolidate the experts in a centralised office that handle requests from a large group of frontline staff that are the ones who actually talk to the customers. This means that the frontline staff needs less training, which means they can be paid less. This makes it easier to hire more of them.

The number of appointments that the office handles in each given week skyrockets, and the length of each drop to fifteen minutes.

That looks like a massive success. But in similar situations when John Seddon and his team spoke to the office staff, those conversations revealed a different truth.

Demand had increased. People were making more appointments. But that demand was caused by failures to solve the problem on the first visit. Instead of talking directly to a person who could solve their problem, the frontline worker referred it to the specialists. The applicant had to come again and again to sort out multiple stages of the application, find out where the application was, and generally remind the office that they existed. An application that took an hour before streamlining now took several over multiple appointments. The metrics made the streamlining look like a massive improvement but had made the problem worse—much worse.

Failure demand is unproductive work that has a statistical footprint identical to productive work, caused by an ill-advised change to the system.

Failure demand and software development

The concept has an obvious relevance to customer support and is a frequent cause of ballooning support costs for many software companies.

But it's also a near-constant characteristic of modern software development. Components simultaneously over-engineered and under-engineered are instances of *failure demand*. We changed a part of the software, which meant we had to do much more work to stay in place. The custom widget that's almost as good as the native is a perennial in the industry.

- Dropdown widgets that are less accessible than the built-in.
- Custom context menus that only have a fraction of the features and integrations that the native menu gave you for free.
- Buttons that aren't actual buttons.

We keep making the mistake of underestimating the added work that comes from these decisions in the long term. That added work genuinely looks *productive* in all of the team's metrics. One ill-advised custom widget can result in dozens of feature requests and iterative improvements to reach parity with native. All that work looks like progress when you're simply running to say in place. "*We keep knocking down story points!*" for a story that shouldn't have been told in the first place.

This can affect the software development system itself as well. Whenever you add a process or a step to the development system, you risk creating failure demand—added work or micromanagement that only looks like a productive contribution in your metrics but is a waste of everybody's time. Often these are caused by management distrust-

ing the team and the team members' ability to collaborate. Reports and reviews may look productive, but they dominate actual work. The team works with the reviews and your metrics in mind, not the product, and working towards the measure takes over. More work is being done, but less work is happening.

Most of these failures happen because of attempts to optimise a system. An age-old adage in software is that "premature optimisation is the root of all evil" (Knuth, 1974). That applies to all systems. An optimised system is brittle and hard to change. You need to be damn sure that what you're optimising is exactly what you want because you usually don't get an opportunity to correct it afterwards.

22

Disruptive Innovations: Clayton Christensen's mistake

We are conditioned by society to think about the world in terms of being. Our own being—actions, desires, and emotions—or the being of others—the consequences of their movement through the world.

We don't think in terms of ecosystems. We concentrate on the plants and animals, their presence, and their effects on the surroundings.

We don't think in terms of societies. We concentrate on people: ourselves, our friends, the people we know and interact with, and the people we know to avoid.

Systems-thinking is hard because of this, both at work and in a larger cultural sense. We often fail to situate our thoughts and analysis in terms of the system, even though we have a deep *understanding* of it.

This has misled even the smartest of us. It has led clever people to misunderstand the evidence of their own eyes and, just as importantly, it has led many to profoundly misunderstand important analyses and ideas.

That's where Clayton Christensen made his biggest mistake. He introduced his groundbreaking analysis of how established companies get disrupted in terms of the innovations that disrupt them.

Not in terms of how sustained complex systems *get* disrupted, which is what his books are about.

If you've read his book *The Innovator's Dilemma* (Clayton, 2013) you'd know that it's entirely about how established companies get disrupted when another company develops a new product category.

The "disruptive" product has characteristics that are impossible for the established companies to produce with their existing product development systems, requiring them to build a new system from scratch if they want to compete with the new entrant's newly developed system.

The dilemma is obvious once you frame it in terms of systems-thinking:

- The new entrant has developed its system to the point where it's financially viable and sustainable. It might not cover all of the use cases handled by the products made by the systems of the established companies, but it should get there with iteration.

- Changing an existing system is difficult. Changing an optimised, large-scale, existing system is next to impossible. This means that it's *extremely* unlikely that the established companies will be able to effectively compete with their old systems and processes.

- If they start from scratch with a new system, the new entrant gets a massive head start. They already have a system that's successfully producing the new product.

Hence the dilemma. Once an established company is in this position, it has very few options, most unpleasant.

The most common strategy is to flee upmarket. Growing a system takes a long time. It will usually take the disruptive entrant a long time to build up its capabilities. You can make a lot of money for quite a while by catering specifically to the most profitable sectors of your market. You might lose eventually, but that might be years down the road, right?

The problem seems clear enough, but people keep misunderstanding the inherent issue. To use Christensen's own words:

> The problem with conflating a disruptive innovation with any breakthrough that changes an industry's competitive patterns is that different types of innovation require different strategic approaches. To put it another way, the lessons we've learned about succeeding as a disruptive innovator (or defending against a disruptive challenger) will not apply to every company in a shifting market.
>
> What Is Disruptive Innovation? (Clayton, 2015)

People keep thinking of disruption in terms of magic products that destroy incumbent companies—like laser-guided missiles—when it's about companies discovering that their competitors have come up with systems that are sustainable, growing, and they have no answer for. Disruptive innovation might not have all or even any of the characteristics described in the original *Innovator's Dilemma* book. The only required characteristic is that the product is made by a system that the incumbent cannot easily replicate or counter.

This misunderstanding has led countless startups to ruin. They tailormade their products to a simplistic checklist based on a naive misunderstanding of an incomplete theory. They assumed that their half-baked organisation—shoddy system—would automatically win because they had performed the correct magic incantation.

It's management by superstition. It's unfortunately much too common in the software industry.

Even Christensen has made this mistake himself. He repeatedly predicted the imminent disruption and demise of Apple. First with the iPod and then the iPhone—Christensen was an Apple pessimist for years (Thompson, 2013). But as Ben Thompson explained, Christensen had fallen for this exact fumble: he had mistaken the product for the system. Apple's product development systems were in no way threatened by any of the new entrants. They were, in many cases, vastly more capable than the systems of their potential competitors.

Christensen had failed to frame the problem in terms of the organisations as systems. It isn't that he misunderstood the dynamics of what was happening. He understood them better than most of us. He

had fallen victim to his own framing that centred the product over the system.

And, if Clayton Christensen can make that mistake, who was heralded by many as one of the greatest business theorists in history, then you can be damn sure that you and I will be making it as well.

This is hard. It takes practice. All we can do is remind ourselves to step back and look at the ecosystem instead of the animals.

What Next?

23

Further reading

This book is little more than a primer on how to apply systems-thinking to software development. It certainly doesn't have academic rigour, nor does it go deep into any details involved. It is only one step in a journey. The next step is to read more books, some of which are, quite frankly, amazing.

These are, in my opinion, the books that should be at the top of your list.

Thinking in Systems: A Primer by Donella H. Meadows and Diana Wright

This is the first book you should look at to truly understand systems-thinking. It's written in clear language with straightforward examples. It is a brilliant book of a manageable length.

The Systems Bible: The Beginner's Guide to Systems Large and Small: Being the Third Edition of Systemantics by John Gall

This is an essential complement to *Thinking in Systems*. While Donella Meadows' book is all about understanding systems, John Gall's book is all about understanding system *failure*. Funnier than it has any right to be.

Badass: Making Users Awesome by Kathy Sierra

Quite possibly the best book on product design I've ever read. Kathy Sierra is also the reason why this book exists. Back when she was more online, her recommendation was why I first read Meadows' book *and* Systemantics.

Peopleware: Productive Projects and Teams by Tom DeMarco and Timothy R. Lister

Just full of good advice in straightforward language.

Out of the Crisis by W.E. Deming

Everything by Deming is great, but this book should be read after *Thinking in Systems* as it's helpful to have a clear idea of what a "system" means when reading Deming. Reading it can be a bleak experience as you slowly realise how badly mismanaged every single employer you've had really is. It can be tricky to apply some of the ideas to software development, otherwise, it'd be closer to the top of the list.

Managing the Design Factory: A Product Developer's Toolkit by Donald G. Reinertsen

Reinertsens's books are all truly interesting if a bit mercenary in their focus on economic value. Some concepts can be tough to apply to software, but it makes up for that difficulty by being more rigorous than anything found in software development literature.

24

References

- ‘95% of Performance Is Governed by the System’. Vanguard Consulting Ltd, 18 June 2015, https://beyondcommandandcontrol.com/library/dr-demings-aphorisms/95-of-performance-is-governed-by-the-system/.
- A Modern Sequel | Babbage Engine | Computer History Museum. https://www.computerhistory.org/babbage/modernsequel/. Accessed 22 Oct. 2022.
- A Quote by William Gibson. https://www.goodreads.com/quotes/681-the-future-is-already-here-it-s-just-not-evenly. Accessed 22 Oct. 2022.
- Alan Kay - Programming and Scaling. www.youtube.com, https://www.youtube.com/watch?v=YyIQKBzIuBY. Accessed 22 Oct. 2022.
- Allsopp, John. ‘A Dao of Web Design’. A List Apart, 7 Apr. 2000, https://alistapart.com/article/dao/.
- An Update on Toyota and Unintended Acceleration « Barr Code. https://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/. Accessed 22 Oct. 2022.

- Architectural Styles and the Design of Network-Based Software Architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Accessed 22 Oct. 2022.
- 'Augmentation Research Center'. Wikipedia, 21 Jan. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Augmentation_Research_Center&oldid=1066994471.
- Bakke, Dennis W. Joy at Work. P V G, 2005.
- Basics of the Unix Philosophy. http://www.catb.org/~esr/writings/taoup/html/ch01s06.html. Accessed 22 Oct. 2022.
- Bjarnason, Baldur. 'Over-Engineering Is under-Engineering'. Baldur Bjarnason, https://www.baldurbjarnason.com/notes/under-engineering-websites/. Accessed 2 Dec. 2022.
- Bjarnason, Baldur. 'Software Crisis 2.0'. Baldur Bjarnason, https://www.baldurbjarnason.com/2021/software-crisis-2/. Accessed 30 Oct. 2022.
- Bjarnason, Baldur. 'You Are What You Do, Not What You Say or Write'. Baldur Bjarnason, https://www.baldurbjarnason.com/2021/you-are-what-you-do/. Accessed 22 Oct. 2022.
- Brooks. 'No Silver Bullet Essence and Accidents of Software Engineering'. Computer, vol. 20, no. 4, Apr. 1987, pp. 10–19. DOI.org (Crossref), https://doi.org/10.1109/MC.1987.1663532.
- Cagan, Marty. Empowered Product Teams. https://www.svpg.com/empowered-product-teams/.
- Cagan, Marty. The Most Important Thing. https://www.svpg.com/the-most-important-thing/.
- Campbell, Darryl. 'The Ancient Computers in the Boeing 737 Max Are Holding up a Fix'. The Verge, 9 Apr. 2020, https://

www.theverge.com/2020/4/9/21197162/boeing-737-max-software-hardware-computer-fcc-crash.

- Chaos Report 2015. 2015, https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.
- Christensen, Clayton M. The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail. Harvard Business Review Press, 2013.
- Christensen, Clayton M. 'What Is Disruptive Innovation?' Harvard Business Review, 1 Dec. 2015. hbr.org, https://hbr.org/2015/12/what-is-disruptive-innovation.
- Cooper, Alan, et al. About Face: The Essentials of Interaction Design. Fourth edition, John Wiley and Sons, 2014.
- Cooper, Alan. The Inmates Are Running the Asylum. Sams, 2004.
- Creating Passionate Users. https://headrush.typepad.com/. Accessed 22 Oct. 2022.
- Curious Cat Deming Management Concepts - Dr. W. Edwards Deming on Tampering. https://curiouscat.com/management/deming/tampering. Accessed 22 Oct. 2022.
- Cutler, John. 'TBM 41a/52: Slow Down on Your Own Terms'. The Beautiful Mess, 7 Oct. 2021, https://cutlefish.substack.com/p/tbm-41a52-slow-down-on-your-own-terms.
- DeMarco, Tom, and Timothy R. Lister. Peopleware: Productive Projects and Teams. Third edition, Addison-Wesley, 2013.
- Deming, William Edwards. Out of the Crisis. 1. MIT Press ed, The MIT Press, 2000.
- Dubner, Stephen J. 'The Cobra Effect'. Freakonomics, https://freakonomics.com/podcast/the-cobra-effect-2/. Accessed 22 Oct. 2022.

- 'Failure Demand'. Vanguard Consulting Ltd, 25 Oct. 2019, https://beyondcommandandcontrol.com/failure-demand/.
- 'Fast Path to a Great UX - Increased Exposure Hours'. UX Articles by UIE, 30 Mar. 2011, https://articles.uie.com/user_exposure_hours/.
- Gall, John. The Systems Bible: The Beginner's Guide to Systems Large and Small: Being the Third Edition of Systemantics. General Systemantics Press, 2002.
- Gries, David, and A. G. Fraser. Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. 1968.
- Have Single-Page Apps Ruined the Web? | Transitional Apps with Rich Harris, NYTimes. www.youtube.com, https://www.youtube.com/watch?v=860d8usGC0o. Accessed 22 Oct. 2022.
- How to Upgrade to React 18 – React Blog. https://reactjs.org/blog/2022/03/08/react-18-upgrade-guide.html. Accessed 22 Oct. 2022.
- 'Human Error? No, Bad Design'. Jnd.Org, 13 Apr. 2014, https://jnd.org/stop_blaming_people_blame_inept_design/.
- If Builders Built Buildings the Way Programmers Wrote Programs, Then the First Woodpecker That Came Along Would Destroy Civilization – Quote Investigator. https://quoteinvestigator.com/2019/09/19/woodpecker/. Accessed 22 Oct. 2022.
- Kay, Alan. Dr. Dobb's Interview with Alan Kay. https://link.springer.com/content/pdf/bbm:978-3-319-90008-7/1.pdf.
- Kim, Beaumie, et al. 'The Affordances of Informant Design in Educational Game Development'. International Journal of Arts and Technology, vol. 6, no. 3, 2013, p. 215. DOI.org (Crossref), https://doi.org/10.1504/IJART.2013.055388.

- Knuth, Donald E. 'Structured Programming with Go to Statements'. ACM Computing Surveys, vol. 6, no. 4, Dec. 1974, pp. 261–301. DOI.org (Crossref), https://doi.org/10.1145/356635.356640.
- Kuhn, Thomas S., and Ian Hacking. The Structure of Scientific Revolutions. Fourth edition, The University of Chicago Press, 2012.
- Model-View-Controller. https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html. Accessed 1 Nov. 2022.
- Manifesto for Agile Software Development. https://agilemanifesto.org/. Accessed 22 Oct. 2022.
- McFadden, Christopher. These Are Some of the Biggest Software Programs by Lines of Code. 15 July 2021, https://interestingengineering.com/innovation/whats-the-biggest-software-package-by-lines-of-code.
- Meadows, Donella H., and Diana Wright. Thinking in Systems: A Primer. Chelsea Green Pub, 2008.
- Naur, Peter. 'Programming as Theory Building'. Microprocessing and Microprogramming, vol. 15, no. 5, May 1985, pp. 253–61. DOI.org (Crossref), https://doi.org/10.1016/0165-6074(85)90032-8.
- 'Nobody Gives a Hoot About Profit - The W. Edwards Deming Institute'. Https://Deming.Org/, https://deming.org/nobody-gives-a-hoot-about-profit/. Accessed 22 Oct. 2022.
- 'Productivity Paradox'. Wikipedia, 20 Sept. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Productivity_paradox&oldid=1111403831.

- Reinertsen, Donald G. Managing the Design Factory: A Product Developer's Toolkit. Free Press, 1997.
- Royce, W. W. 'Managing the Development of Large Software Systems: Concepts and Techniques'. Proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society Press, 1987, pp. 328–38.
- Sarasvathy, Saras D. 'What Makes Entrepreneurs Entrepreneurial?' SSRN Electronic Journal, 2006. DOI.org (Crossref), https://doi.org/10.2139/ssrn.909038.
- 'Second-System Effect'. Wikipedia, 24 Aug. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Second-system_effect&oldid=1106324339.
- Sierra, Kathy. Badass: Making Users Awesome. First edition, O'Reilly, 2015.
- Stuart, Keith, and Keza MacDonald. 'Inside Nintendo's Secretive Creative Process'. The Guardian, 25 Apr. 2018. The Guardian, https://www.theguardian.com/games/2018/apr/25/nintendo-interview-secret-innovation-lab-ideas-working.
- 'Study: 68 Percent of IT Projects Fail'. ZDNET, https://www.zdnet.com/article/study-68-percent-of-it-projects-fail-6103001175/. Accessed 22 Oct. 2022.
-
- Tainter, Joseph A. The Collapse of Complex Societies. Cambridge University Press, 1988.
- Taylor, Dorian. 'Agile As Trauma'. Dorian Taylor, https://doriantaylor.com/agile-as-trauma.
- The Betting Table | Shape Up. https://basecamp.com/shapeup/2.2-chapter-08#cool-down. Accessed 22 Oct. 2022.

- The CADT Model. https://www.jwz.org/doc/cadt.html. Accessed 22 Oct. 2022.
- 'The Curious Case of the CHAOS Report 2009'. Project Smart, https://www.projectsmart.co.uk/it-project-management/the-curious-case-of-the-chaos-report-2009.php. Accessed 22 Oct. 2022.
- 'The Mobile Performance Inequality Gap, 2021'. Infrequently Noted, https://infrequently.org/2021/03/the-performance-inequality-gap/. Accessed 22 Oct. 2022.
- Thompson, Ben. 'What Clayton Christensen Got Wrong.' Stratechery by Ben Thompson, 22 Sept. 2013, https://stratechery.com/2013/clayton-christensen-got-wrong/.
- Tribus, Myron. The Germ Theory of Management. SPC Press, 1992.
- 'Uber Car Software Detected Woman before Fatal Crash but Failed to Stop'. Naked Security, 9 May 2018, https://nakedsecurity.sophos.com/2018/05/09/uber-car-software-detected-woman-before-fatal-crash-but-failed-to-stop/.
- Unix Philosophy Description by The Linux Information Project (LINFO). http://www.linfo.org/unix_philosophy.html. Accessed 22 Oct. 2022.
- Values, Leadership, and Implementing the Deming Philosophy. www.youtube.com, https://www.youtube.com/watch?v=bu7JMAQggfo. Accessed 22 Oct. 2022.
- 'Wabi-Sabi'. Wikipedia, 20 Sept. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Wabi-sabi&oldid=1111346372.
- Wagner, Jeremy. The 2022 Web Almanac: JavaScript. HTTP Archive, 26 Sept. 2022. almanac.httparchive.org, https://almanac.httparchive.org/en/2022/javascript.

- ‘Wayland (Display Server Protocol)’. Wikipedia, 27 Aug. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Wayland_(display_server_protocol)&oldid=1106893281.