I

WHAT IS SOFTWARE ENGINEERING?

This page intentionally left blank

1

Introduction

Engineering—The Practical Application of Science

Software development is a process of discovery and exploration; therefore, to succeed at it, software engineers need to become experts **at learning**.

Humanity's best approach to learning is science, so we need to adopt the techniques and strategies of science and apply them to our problems. This is often misunderstood to mean that we need to become physicists measuring things to unreasonable, in the context of software, levels of precision. Engineering is more pragmatic than that.

What I mean when I say we should apply the techniques and strategies of science is that we should apply some pretty basic, but nevertheless extremely important, ideas.

The scientific method that most of us learned about in school is described by Wikipedia as:

- **Characterize**: Make an observation of the current state.
- **Hypothesize**: Create a description, a theory that may explain your observation.
- **Predict**: Make a prediction based on your hypothesis.
- **Experiment**: Test your prediction.

When we organize our thinking this way and start to make progress on the basis of many small, informal experiments, we begin to limit our risk of jumping to inappropriate conclusions and end up doing a better job.

If we start to think in terms of controlling the variables in our experiments so that we can achieve more consistency and reliability in our results, this leads us in the direction of more deterministic systems and code. If we start to think in terms of being skeptical about our ideas and explore how we could falsify them, we can identify, and then eliminate, bad ideas more quickly and make progress much faster.

This book is deeply grounded in a practical, pragmatic approach to solving problems in software, based on an informal adoption of basic scientific principles, in other words, **engineering!**

What Is Software Engineering?

My working definition for software engineering that underpins the ideas in this book is this:

> ***Software engineering*** *is the application of an empirical, scientific approach to finding efficient, economic solutions to practical problems in software.*

The adoption of an engineering approach to software development is important for two main reasons. First, software development is always an exercise in discovery and learning, and second, if our aim is to be "efficient" and "economic," then our ability to learn must be sustainable.

This means that we must manage the complexity of the systems that we create in ways that maintain our ability to learn new things and adapt to them.

So, we must become **experts at learning and experts at managing complexity**.

There are five techniques that form the roots of this focus on learning. Specifically, to become ***experts at learning***, we need the following:

- Iteration
- Feedback
- Incrementalism
- Experimentation
- Empiricism

This is an evolutionary approach to the creation of complex systems. Complex systems don't spring fully formed from our imaginations. They are the product of many small steps, where we try out our ideas and react to success and failure along the way. These are the tools that allow us to accomplish that exploration and discovery.

Working this way imposes constraints on how we can safely proceed. We need to be able to work in ways that facilitate the journey of exploration that is at the heart of every software project.

So as well as having a laser-focus on learning, we need to work in ways that allow us to make progress when the answers, and sometimes even the direction, is uncertain.

For that we need to become **experts at managing complexity**. Whatever the nature of the problems that we solve or the technologies that we use to solve them, addressing the complexity of the

problems that face us and the solutions that we apply to them is a central differentiator between bad systems and good.

To become ***experts at managing complexity***, we need the following:

- Modularity
- Cohesion
- Separation of Concerns
- Abstraction
- Loose Coupling

It is easy to look at these ideas and dismiss them as familiar. Yes, you are almost certainly familiar with all of them. The aim of this book is to organize them and place them into a coherent approach to developing software systems that helps you take best advantage of their potential.

This book describes how to use these ten ideas as tools to steer software development. It then goes on to describe a series of ideas that act as practical tools to drive an effective strategy for any software development. These ideas include the following:

- Testability
- Deployability
- Speed
- Controlling the variables
- Continuous delivery

When we apply this thinking, the results are profound. We create software of higher quality, we produce work more quickly, and the people working on the teams that adopt these principles report that they enjoy their work more, feel less stress, and have a better work-life balance.[1]

These are extravagant claims, but again they are backed by the data.

Reclaiming "Software Engineering"

I struggled over the title of this book, not because I didn't know what I wanted to call it, but because our industry has so redefined what *engineering* means in the context of software that the term has become devalued.

In software it is often seen as either simply a synonym for "code" or something that puts people off as being overly bureaucratic and procedural. For true engineering, nothing could be further from the truth.

1. Based on findings from the "State of DevOps" reports as well as reports from Microsoft and Google

In other disciplines, *engineering* simply means the "stuff that works." It is the process and practice that you apply to increase your chances of doing a good job.

If our "software engineering" practices don't allow us to build better software faster, then they aren't really engineering, and we should change them!

That is the fundamental idea at the heart of this book, and its aim is to describe an intellectually consistent model that pulls together some foundational principles that sit at the roots of all great software development.

There is never any guarantee of success, but by adopting these mental tools and organizing principles and applying them to your work, you will certainly increase your chances of success.

How to Make Progress

Software development is a complex, sophisticated activity. It is, in some ways, one of the more complex activities that we, as a species, undertake. It is ridiculous to assume that every individual or even every team can, and should, invent how to approach it, from scratch, every time we begin a new piece of work.

We have learned, and continue to learn, things that work and things that don't. So how can we, as an industry and as teams, make progress and build on the shoulders of giants, as Isaac Newton once said, if everyone has a veto on everything? We need some agreed principles and some discipline that guides our activities.

The danger in this line of thinking is that, if misapplied, it can lead to draconian, overly directive, "decision from authority"–style thinking.

We will fall back on previous bad ideas, where the job of managers and leaders is assumed to be to tell everyone else what to do and how to do it.

The big problem with being "proscriptive" or overly "directive" is, what do we do if some of our ideas are wrong or incomplete? They inevitably will be, so how can we challenge and refute old, but well-established, bad ideas and evaluate novel, potentially great, untried ideas?

We have a very strong example of how to solve these problems. It's an approach that allows us the intellectual freedom to challenge and refute dogma and to differentiate between fashion, plain-old bad ideas and great ones, whatever their source. It allows us to replace the bad ideas with better ideas and to improve on the good ideas. Fundamentally we need some structure that allows us to grow and to evolve improved approaches, strategies, processes, technologies, and solutions. We call this good example *science*!

When we apply this kind of thinking to solving practical problems, we call it *engineering*!

This book is about what it means to apply scientific-style reasoning to our discipline and so achieve something that we can genuinely and accurately refer to as *software engineering*.

The Birth of Software Engineering

Software engineering as a concept was created at the end of the 1960s. The term was first used by Margaret Hamilton who later became the director of the Software Engineering Division of the MIT Instrumentation Lab. Margaret was leading the effort to develop the flight-control software for the Apollo space program.

During the same period, the North Atlantic Treaty Organization (NATO) convened a conference in Garmisch-Partenkirchen, Germany, to try to define the term. This was the first **software engineering** conference.

The earliest computers had been programmed by flipping switches, or even hard-coded as part of their design. It quickly became clear to the pioneers that this was slow and inflexible, and the idea of the "stored program" was born. This is the idea that, for the first time, made a clear distinction between software and hardware.

By the late 1960s, computer programs had become complex enough to make them difficult to create and maintain in their own right. They were involved in solving more complex problems and were rapidly becoming the enabling step that allowed certain classes of problems to be solved at all.

There was perceived to be a significant gap between the rate at which progress was being made in hardware compared to the rate at which it was being made in software. This was referred to, at the time, as the *software crisis*.

The NATO conference was convened, in part, in response to this crisis.

Reading the notes from the conference today, there are many ideas that are clearly durable. They have stood the test of time and are as true today as they were in 1968. That should be interesting to us, if we aspire to identify some fundamental characteristics that define our discipline.

A few years later, looking back, Turing award–winner Fred Brooks compared the progress in software with that in hardware:

> *There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity.*[2]

Brooks was saying this in comparison with the famous Moore's law,[3] which hardware development had been tracking for many years.

---

2. Source: Fred Brooks' 1986 paper called "No Silver Bullet." See https:// bit.ly/2UaIM4T.

3. In 1965, Gordon Moore predicted that transistor densities (not performance) would double every year, later revised to every two years, for the next decade (to 1975). This prediction became a target for semiconductor producers and significantly exceeded Moore's expectations, being met for several more decades. Some observers believe that we are reaching the end of this explosive growth in capacity, because of the limitations of the current approaches and the approach of quantum effects, but at the time of writing, high-density semiconductor development continues to track Moore's law.

This is an interesting observation and one that, I think, would surprise many people, but in essence it has always been true.

Brooks goes on to state that this is not so much a problem of software development; it is much more an observation on the unique, staggering improvement in hardware performance:

> *We must observe that the anomaly is not that software progress is so slow but that computer hardware progress is so fast. No other technology since civilization began has seen six orders of magnitude price-performance gain in 30 years.*

He wrote this in 1986, what we would today think of as the dawn of the computer age. Progress in hardware since then has continued at this pace, and the computers that seemed so powerful to Brooks look like toys compared to the capacity and performance of modern systems. And yet...his observation on the rate of improvement in software development remains true.

Shifting the Paradigm

The idea of *paradigm shift* was created by physicist Thomas Kuhn.

Most learning is a kind of accretion. We build up layers of understanding, with each layer foundationally under-pinned by the previous one.

However, not all learning is like that. Sometimes we fundamentally change our perspective on something, and that allows us to learn new things, but that also means we must discard what went before.

In the 18th century, reputable biologists (they weren't called that then) believed that some animals spontaneously generated themselves. Darwin came along in the middle of the 19th century and described the process of natural selection, and this overturned the idea of spontaneous generation completely.

This change in thinking ultimately led to our modern understanding of genetics and our ability to understand life at a more fundamental level, create technologies that allow us to manipulate these genes, and create COVID-19 vaccines and genetic therapies.

Similarly, Kepler, Copernicus, and Galileo challenged the then conventional wisdom that Earth was at the center of the universe. They instead proposed a heliocentric model for the solar system. This ultimately led to Newton creating laws of gravitation and Einstein creating general relativity, and it allowed us to travel in space and create technologies like GPS.

The idea of paradigm shift implicitly includes the idea that when we make such a shift, we will, as part of that process, discard some other ideas that we now know are no longer correct.

The implications of treating software development as a genuine engineering discipline, rooted in the philosophy of the scientific method and scientific rationalism, are profound.

It is profound not only in its impact and effectiveness, described so eloquently in the *Accelerate Book*,[4] but also in the essential need to discard the ideas that this approach supersedes.

This gives us an approach to learning more effectively and discarding bad ideas more efficiently.

I believe that the approach to software development that I describe in this book represents such a paradigm shift. It provides us with a new perspective on what it is that we do and how we do it.

Summary

Applying this kind of engineering thinking to software does not need to be heavyweight or overly complex. The paradigm shift in thinking differently about what it is that we do, and how we do it, when we create software should help us to see the wood for the trees and make this simpler, more reliable, and more efficient.

This is not about more bureaucracy; it is about enhancing our ability to create high-quality software more sustainably and more reliably.

4. The people behind the "State of DevOps" reports, DORA, described the predictive model that they have created from their research. Source: *Accelerate: The Science of Lean Software and DevOps* by Nicole Fosgren, Jez Humble, and Gene Kim (2018)

This page intentionally left blank

2

What Is Engineering?

I have been talking to people about software engineering for some years now. As a result I regularly get involved in a surprising number of conversations about bridge building. They usually start with the phrase "Yes, but software isn't bridge building" as though this was some kind of revelation.

Of course, software engineering is not the same as bridge building, but what most software developers think of as bridge building isn't like real bridge building, either. This conversation is really a form of confusion between production engineering and design engineering.

Production engineering is a complex problem when the discipline involved is dealing with physical things. You need to get those physical things created to certain levels of precision and quality.

You need your widgets delivered to some specific location in space, at a particular time, to a defined budget, and so on. You need to adapt theoretical ideas to practical reality as your models and designs are found to be lacking.

Digital assets are completely different. Although there are some analogs to these problems, for digital artifacts these problems either don't really exist or can be made trivially simple. The cost of production of digital assets of any kind is essentially free, or at least should be.

Production Is Not Our Problem

For most human endeavor, the production of "things" is the hard part. It may take effort and ingenuity to design a car, an airliner, or a mobile phone, but taking that initial prototype design and idea into mass production is immensely more expensive and complicated.

This is particularly true if we aim to do it with any kind of efficiency. As a result of these difficulties, we, products of the industrial age and industrial age thinking, automatically, almost unthinkingly, worry about this aspect, the production, of any significant task.

The result of this, in software, has been that we have fairly consistently tried to apply "production-style thinking" to our industry. Waterfall[1] processes are production lines for software. They are the tools of mass production. They are not the tools of discovery, learning, and experimentation that are, or at least should be, at the heart of our profession.

Unless we are foolish in our software development choices, for us, production consists of triggering the build!

It is automatic, push-button, immensely scalable and so cheap that it is best considered free. We can still make mistakes and get it wrong, but these are problems that are understood and well addressed by tools and technology.

"Production" is not our problem. This makes our discipline unusual. It also makes it subject to easy misunderstanding and misapplied thinking and practices, because this ease of production is so unusual.

Design Engineering, Not Production Engineering

Even in the real world, what most people think of as "bridge building" is different if the bridge-builders are building the first of a new kind of bridge. In this circumstance you have two problems: one that is relevant to software development and one that is not.

First, the one that is not—when building even the first of a new kind of bridge, because it is physical, you have all of the production problems, and many more, that I mentioned. From a software perspective, these can be ignored.

The second, in the case of bridge-building, is that in addition to those production problems, if you are building the first of a new kind of bridge, the second really difficult part is the design of your new bridge.

This is difficult because you can't iterate quickly when your product is something physical. When building physical things, they are difficult to change.

As a result, engineers in other disciplines adopt modeling techniques. They may choose to build small physical models, and these days probably computer simulations of their design or mathematical models of various kinds.

In this respect, we software developers have an enormous advantage. A bridge-builder may create a computer simulation of their proposed design, but this will only be an approximation of the real thing. Their simulation, their model, will be inaccurate. The models that we create as software, our computer simulations of a problem, are our product.

1. Waterfall, as applied to software development, is a staged, sequential approach to organizing work by breaking it down into a series of distinct phases with well-defined handovers between each phase. The idea is that you tackle each phase in turn, rather than iterate.

We don't need to worry if our models match reality; our models are the reality of our system, so we can verify them. We don't need to worry about the cost of changing them. They are software; thus, they are dramatically easier to change, at least when compared to a bridge.

Ours is a technical discipline. We like to think of ourselves in this context, and my guess is that the majority of people who think of themselves as professional software developers probably have had some science in their education.

Despite this, little software development is practiced with scientific rationalism in mind. In part, this is because we took some missteps in our history. In part this is because we assume that science is hard, expensive, and impossible to achieve within the scope of normal software development schedules.

Part of the mistake here is to assume some level of idealistic precision that is impossible in any field, let alone the field of software development. We have made the mistake of seeking mathematical precision, which is not the same thing as engineering!

Engineering as Math

During the late 1980s and early 1990s there was a lot of talk about more programming-structural ideas. The thinking about the meaning of software engineering moved on to examine the ways in which we work to generate the code. Specifically, how could we work in ways that are more effective at identifying and eliminating problems in our designs and implementations?

Formal methods became a popular idea. Most university courses, at the time, would teach formal methods. A formal method is an approach to building software systems that has, built into it, a mathematical validation of the code written. The idea is that the code is proven to be correct.

The big problem with this is that while it is hard to write code for a complex system, it is even harder to write code that defines the behavior of a complex system and that also proves itself to be correct.

Formal methods are an appealing idea, but pragmatically they haven't gained widespread adoption in general software development practice because at the point of production, they make the code harder to produce, not less.

A more philosophical argument is a little different, though. Software is unusual stuff; it clearly appeals to people who often also enjoy mathematical thinking. So the appeal of taking a mathematical approach to software is obvious, but also somewhat limiting.

Consider a real-world analogy. Modern engineers will use all the tools at their disposal to develop a new system. They will create models and simulations and crunch the numbers to figure out if their system will work. Their work is heavily informed by mathematics, but then they will try it out for real.

In other engineering disciplines, math is certainly an important tool, but it doesn't replace the need to test and to learn empirically from real-world experience. There is too much variance

in the real world to completely predict an outcome. If math alone was enough to design an airplane, then that is what aerospace companies would do, because it would be cheaper than building real prototypes, but they don't do that. Instead, they use math extensively to inform their thinking, and then they check their thinking by testing a real device. Software is not quite the same as an airplane or a space rocket.

Software is digital and runs on mostly deterministic devices called *computers*. So for some narrow contexts, if the problem is simple enough, constrained enough, deterministic enough, and the variability low enough, then formal methods can prove a case. The problem here is the degree to which the system as a whole is deterministic. If the system is concurrent anywhere, interacts with the "real world" (people) anywhere, or is just working in a sufficiently complex domain, then the "provability" quickly explodes to become impractical.

So, instead, we take the same course as our aerospace colleagues, apply mathematical thinking where we can, and take a data-driven, pragmatic, empirical, experimental approach to learning, allowing us to adapt our systems as we grow them incrementally.

As I write this book, SpaceX is busy blowing up rockets while it works to perfect Starship.[2] It has certainly built mathematical models of nearly every aspect of the design of its rockets, its engines, the fuel delivery systems, launch infrastructure, and everything else, but then it tests them.

Even something seemingly simple, like switching from 4mm stainless steel to 3mm stainless steel, may sound like a pretty controlled change. SpaceX has access to detailed data on the tensile strength of the metal. It has experience and data collected from tests that show exactly how strong pressure vessels constructed from the 4mm steel are.

Yet still, after SpaceX crunched the numbers, it built experimental prototypes to evaluate the difference. It pressurized these test pieces to destruction to see if the calculations were accurate and to gain deeper insight. SpaceX collected data and validated its models because these models will certainly be wrong in some esoteric, difficult-to-predict way.

The remarkable advantage that we have over all other engineering disciplines means that the models that we create in software are the executable result of our work, so when we test them, we are testing our products, not our best guess of the reality of our products.

If we work carefully to isolate the part of the system that we are interested in, we can evaluate it in exactly the same environment that it will be exposed to in production. So our experimental simulation can much more precisely and much more accurately represent the "real world" of our systems than in any other discipline.

2. At the time of writing, SpaceX is developing a new fully reusable spacecraft. SpaceX's intent is to create a system that will allow people to journey to and live on Mars as well as explore other parts of the solar system. It has adopted an intentionally fast, iterative style of engineering to rapidly create and evaluate a series of fast-to-produce prototypes. This is design engineering in extreme form at the limits of engineering knowledge and presents a fascinating example of what it takes to create something new.

In his excellent talk called "Real Software Engineering,"[3] Glenn Vanderburg says that in other disciplines "Engineering means stuff that works" and that almost the opposite has become true for software.

Vanderburg goes on to explore why this is the case. He describes an academic approach to software engineering that was so onerous that almost no one who had practiced it would recommend it for future projects.

It was heavyweight and added no significant value to the process of software development at all. In a telling phrase, Vanderburg says:

> *[Academic software engineering] only worked because sharp people, who cared, were willing to circumvent the process.*

That is not engineering by any sensible definition.

Vanderburg's description of "engineering as the stuff that works" is important. If the practices that we choose to identify as "engineering" don't allow us to make better software faster, then they don't qualify as engineering!

Software development, unlike all physical production processes, is wholly an exercise in discovery, learning, and design. Our problem is one of exploration, and so we, even more than the spaceship designers, should be applying the techniques of exploration rather than the techniques of production engineering. Ours is solely a discipline of design engineering.

So if our understanding of engineering is often confused, what is engineering really about?

The First Software Engineer

During the period when Margaret Hamilton was leading the development of the Apollo flight control systems, there were no "rules of the game" to follow. She said, "We evolved our 'software engineering' rules with each new relevant discovery, while top management rules from NASA went from "'complete freedom'" to "'bureaucratic overkill.'"

There was very little experience of such complex projects to call on at this time. So the team was often breaking new ground. The challenges facing Hamilton and her team were profound, and there was no looking up the answers on Stack Overflow in the 1960s.

Hamilton described some of the challenges:

> *The space mission software had to be man-rated. Not only did it have to work, it had to work the first time. Not only did the software itself have to be ultra-reliable, it needed to be able to perform error detection and recovery in real time. Our languages dared us to make the most subtle of errors. We were on our own to come up with rules for building software. What we learned from the errors was full of surprises.*

3. https://youtu.be/RhdlBHHimeM

At the same time, software in general was looked down on as a kind of "poor relation" compared to other, more "grown-up" forms of engineering. One of the reasons that Hamilton coined the term *software engineering* was to try to get people in other disciplines to take the software more seriously.

One of the driving forces behind Hamilton's approach was the focus on how things fail—the ways in which we get things wrong.

> *There was a fascination on my part with errors, a never ending pass-time of mine was what made a particular error, or class of errors, happen and how to prevent it in the future.*

This focus was grounded in a scientifically rational approach to problem-solving. The assumption was not that you could plan and get it right the first time, rather that you treated all ideas, solutions, and designs with skepticism until you ran out of ideas about how things could go wrong. Occasionally, reality is still going to surprise you, but this is engineering empiricism at work.

The other engineering principle that is embodied in Hamilton's early work is the idea of "failing safely." The assumption is that we can never code for every scenario, so how do we code in ways that allow our systems to cope with the unexpected and still make progress? Famously it was Hamilton's unasked-for implementation of this idea that saved the Apollo 11 mission and allowed the Lunar Module Eagle to successfully land on the moon, despite the computer becoming overloaded during the descent.

As Neil Armstrong and Buzz Aldrin descended in the Lunar Excursion Module (LEM) toward the moon, there was an exchange between the astronauts and mission control. As the LEM neared the surface of the moon, the computer reported 1201 and 1202 alarms. The astronauts asked whether they should proceed or abort the mission.

NASA hesitated until one of the engineers shouted "Go!" because he understood what had happened to the software.

> *On Apollo 11, each time a 1201 or 1202 alarm appeared, the computer rebooted, restarted the important stuff, like steering the descent engine and running the DSKY to let the crew know what was going on, but did not restart all the erroneously-scheduled rendezvous radar jobs. The NASA guys in the MOCR knew—because MIT had extensively tested the restart capability—that the mission could go forward.*[4]

This "fail safe" behavior was coded into the system, without any specific prediction of when or how it would be useful.

So Hamilton and her team introduced two key attributes of a more engineering-led style of thinking, with empirical learning and discovery and the habit of imagining how things could possibly go wrong.

4. Source: "Peter Adler" (https://go.nasa.gov/1AKbDei)

A Working Definition of Engineering

Most dictionary definitions of the word *engineering* include common words and phrases: "application of math," "empirical evidence," "scientific reasoning," "within economic constraints."

I propose the following working definition:

> *Engineering is the application of an empirical, scientific approach to finding efficient, economic solutions to practical problems.*

All of the words here matter. Engineering is applied science. It is practical. Using "empirical" means to learn and advance understanding and solutions toward the resolution of a problem.

The solutions that engineering creates are not abstract ivory-tower things; they are practical and applicable to the problem and the context.

They are efficient, and they are created with an understanding of, and constrained by, the economics of the situation.

Engineering != Code

Another common misperception of what *engineering* means when it comes to software development is that engineering is only the output—the code or perhaps its design.

This is too narrow an interpretation. What does engineering mean to SpaceX? It is not the rockets; they are the products of engineering. Engineering is the process of creating them. There is certainly engineering in the rockets, and they are certainly "engineered structures," but we don't see only the act of welding the metal as engineering unless we have a weirdly narrow view of the topic.

If my definition works, then engineering is about applying scientific rationalism to solving problems. It is the "solving of the problems" where the engineering really comes to play, not just the solutions themselves. It is the processes, tools, and techniques. It is the ideas, philosophy, and approach that together make up an engineering discipline.

I had an unusual experience while writing this book: I published a video about the failure of a game on my YouTube channel, which was dramatically more popular than most of my videos.

The most common negative feedback I got, in saying that this was a "failure of software engineering," was that I was blaming programmers and not their managers. I meant that it was a failure in the whole approach to producing software. The planning was bad, the culture was bad, the code was bad (lots of bugs apparently).

So, for this book, when I talk about engineering, unless I qualify it specifically, I mean **everything that it takes to make software**. Process, tools, culture—all are part of the whole.

The Evolution of Programming Languages

Early efforts in software engineering were focused primarily on creating better languages in which to program things. The first computers made little or no separation between hardware and software. They were programmed by plugging wires into patch boards or flipping switches.

Interestingly, this job was often given to "computers," often women, who had previously done the computation (math) before the computer (as a machine) arrived.

This underplays their role, though. The "program" at this point, specified by someone "more important" in the organization, was often of the form "we'd like to solve this mathematical problem." The organization of the work, and later the specifics of how to translate that into appropriate machine-settings, was left to these human "computers." These were the real pioneers of our discipline!

We would use a different language to describe these activities today. We would describe the description passed to the people doing the work as *requirements*, the act of forming a plan to solve the problem as *programming*, and the "computers" as the first real *programmers* of these early electronic computer systems.

The next big step was to move to "stored programs" and their encoding. This was the era of paper tape and punched cards. The first steps on the adoption of this storage media for programs was still pretty hardcore. Programs were written in machine code and stored on tape, or card, before being fed into the machines.

High-level languages that could capture ideas at a higher level of abstraction were the next major advance. This allowed programmers to make progress much more quickly.

By the early 1980s, nearly all the foundational concepts in language design had been covered. That doesn't mean there was no progress after this, but most of the big ideas had been covered. Nevertheless, software development's focus on language as a core idea in our discipline has continued.

There were several significant steps that certainly affected the productivity of programmers, but probably only one step gave, or came close to giving, Fred Brooks 10x improvement. That was the step from machine code to high-level languages.

Other steps along this evolutionary path were significant, such as procedural programming, object orientation, and functional programming, but all of these ideas have been around for a very long time.

Our industry's obsession with languages and tools has been damaging to our profession. This doesn't mean that there are no advances to be had in language design, but most work in language design seems to concentrate on the wrong kinds of things, such as syntactic advances rather that structural advances.

In the early days, certainly, we needed to learn and explore what is possible and what made sense. Since then, though, a lot of effort has been expended for relatively little progress. When Fred Brooks said there were no 10x improvements, the rest of his paper was focused on what we could do to overcome this limitation:

> *The first step toward the management of disease was replacement of demon theories, and humors theories, by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions.*
>
> *…the system should first be made to run, even though it does nothing useful except call the proper set of dummy subprograms. Then, bit-by-bit it is fleshed out, with the subprograms in turn being developed into actions or calls to empty stubs in the level below.*

These ideas were based on deeper, more profound ideas than trivial details of language implementation.

These were issues more to do with the philosophy of our discipline and the application of some foundational principles that hold true whatever the nature of the technology.

Why Does Engineering Matter?

Another way to think of this is to consider how we go about the production of the things that help us. For the vast majority of human history, everything that we created was the product of craft. Craft is an effective approach to creating things, but it has its limits.

Craft is very good at creating "one-off" items. In a craft-based production system, each item will, inevitably, be unique. In its purest sense this is true of any production system, but in craft-based approaches this is more true because the precision, and so the repeatability, of the production process is generally low.

This means that the amount of variance between individually crafted artifacts is higher. Even the most masterful of craftspeople will create items with only human levels of precision and tolerance. This seriously impacts the ability of craft-based systems to reproduce things reliably. Grace Hopper said:

> *To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge.*

The Limits of "Craft"

We often have an emotional reaction to craft-based production. As human beings we like the variance; we like the feeling that our treasured, hand-crafted thing embodies the skill, love, and care of the craftsperson who created it.

However, at the root, craft-based production is fundamentally low-quality. A human being, however talented, is not as accurate as a machine.

We can build machines that can manipulate individual atoms, even subatomic particles, but a human being is extraordinarily talented if they can produce something, manually, with the accuracy of 1/10 of a millimeter.[5]

How does this precision matter in software? Let us think about what happens when our programs are executed. A human being can perceive change, any change, at the limit of approximately 13 milliseconds. To process an image or to react to something takes hundreds of milliseconds.[6]

At the time of writing, most modern consumer-level computers operate on a clock cycle of around 3GHz. That is 3 billion cycles per second. Modern computers are multicore and operate on instructions in parallel, so often they process more than one instruction per cycle, but let us ignore that and imagine, for simplicity, that each machine instruction that moves values between registers, adds them or references some in-cache piece of memory, takes a single clock cycle.

That is 3 billion operations per second. If we do the math and calculate how many instructions a modern computer can crunch through in the absolute minimum time that a human being could perceive any external event, that number is 39,000,000 instructions!

If we limit the quality of our work to human-scale perception and accuracy, we are, at the very best, sampling what is going on at a rate of 1:(39 million). So, what are our chances of us missing something?

Precision and Scalability

This difference between craft and engineering highlights two aspects of engineering that are important in the context of software: precision and scalability.

Precision is obvious: we can manipulate things at a much higher resolution of detail, through the application of engineering techniques, than by hand. Scalability is perhaps less immediately obvious but is even more important. An engineering approach is not limited in the same way that a craft-based approach is.

The limits of any approach that relies on human capability is, ultimately, limited by human capability. If I dedicate myself to achieving something extraordinary, I may learn to paint a line, file a piece of metal, or stitch leather car seats to within tiny fractions of a millimeter, but however hard I try, however gifted I may be, there are hard limits to how accurate human muscles and senses can be.

An engineer, though, can create a machine to make something smaller and more precise. We can build machines (tools) to make smaller machines.

This technique is scalable all the way down to the limits of quantum physics and all the way up to the limits of cosmology. There is nothing, at least in theory, to prevent us, via the application of

5. Atoms vary in size but are usually measured in tens of picometers (1 x 10^-12m). So, the best of human handcraft is 10 million times less accurate than a good machine.

6. "How Fast is Real-time? Human Perception and Technology," https://bit.ly/2Lb7pL1

engineering, to manipulate atoms and electrons (as we already do) or stars and blackholes (as we may do one day).

To put this more clearly into the context of software, if we are very skilled and train very hard, we could perhaps enter text and click buttons quickly enough to test our software at a rate where we could imagine being able to carry out a test of our software in a few minutes. Let's imagine for the sake of comparison that we can carry out one test of our software every minute (not a pace that I can imagine myself being able to sustain for very long).

If we can run a test per minute, we are under-testing compared to a computer by hundreds of thousands, probably millions, of times.

I have built systems that ran around 30,000 test cases in about 2 minutes. We could have scaled that up considerably further but had no reason to do so. Google claims to run 150 million test executions per day. That works out to 104,166 tests per minute.[7]

Not only can we use our computers to test hundreds of thousands of times more quickly than a human being, we can sustain that pace for as long as we have electricity for our computers. That is scalable!

Managing Complexity

There is another way in which engineering scales, where craft does not. Engineering thinking tends to lead us to compartmentalize problems. Before the American Civil War in the 1860s, if you wanted a gun, you went to a gunsmith. The gunsmith was a craftsman, and he was usually a man!

The gunsmith would create a whole gun for you. He would understand every aspect of that gun, and it would be unique to you. He would probably give you a mold for your bullets, because your bullets would be different from everyone else's and specific to your gun. If your gun had screws, each one was almost certainly different from all of the others, because it would have been hand-made.

The American Civil War was unique in its day. It was the first war where arms were mass-produced.

There is a story of the man who wanted to sell rifles to the northern states. He was an innovator and, it seems, a bit of a showman. He went to Congress to make his case to get the contract to make the rifles for the armies of the northern states.

He took with him a sack full of rifle components. As part of his presentation to the Congressmen, he emptied the bag of components onto the floor of Congress and asked the Congressmen to select components from the pile. From these components he assembled a rifle, won the contract, and invented mass production.

This was the first time that this kind of standardization was possible. A lot of things had to happen to make it possible; machines (tools) had to be engineered to make components that were

7. "The State of Continuous Integration Testing at Google,"https://bit.ly/3eLbAgB

repeatably identical to one another, within some defined tolerance. The design had to be modular so that the components could be assembled, and so on.

The result was devastating. The American Civil War was, in essence, the first modern war. Hundreds of thousands of people were killed because of the mass production of armaments. These arms were cheaper, easier to maintain and repair, and more accurate than those that had gone before.

All this was because they were engineered with more precision, but also because there were lots more of them. The process of production could be de-skilled and scaled up. Instead of needing an expert master craftsperson for each weapon, the machinery in the factory could allow less-skilled people to create rifles of comparable precision to a master.

Later, as tooling, production techniques, and engineering understanding and discipline increased, these mass-produced weapons exceeded the quality, as well as the productivity, of even the greatest master craftsmen, and at a price that anyone could afford.

A simplistic view may interpret this as a "need to standardize," or a need to adopt "mass production for software," but this is, once again, confusing the fundamental nature our problem. This is not about production—it is about design.

If we design a gun that is modular and componentized in the way that the arms manufacturers of the American Civil War did, then we can design parts of that gun more independently. Viewing this from a design perspective rather than from a production engineering or manufacturing perspective, we have improved our management of the complexity of building guns.

Before this step, the gunsmith master-craftsmen would need to think of the whole gun if they wanted to change some aspect of its design. By componentizing the design, the Civil War manufacturers could explore changes incrementally to improve the quality of their products step-by-step. Edsger Dijkstra said:

> *The art of programming is the art of organizing complexity.*

Repeatability and Accuracy of Measurement

The other aspect of engineering that is commonly seen, and is sometimes used to reject engineering as an idea applicable to software, is that of repeatability.

If we can build a machine to reliably and accurately reproduce a nut and bolt, we can churn them out, and all of the copies of bolts will work with any of the copies of nuts that are produced.

This is a production problem and not really applicable to software. However, the more fundamental idea that underpins this kind of capability is applicable to software.

To make nuts and bolts, or anything else, that needs to reliably work together, we need to be able to measure things with a certain level of precision. Accuracy in measurement is an enabling aspect of engineering in any discipline.

Let us for a moment imagine a complex software system. After a few weeks of operation, let's say the system fails. The system is restarted, and two weeks later it fails again in much the same way; there is a pattern. How would a craft-focused team cope with this compared to an engineering-focused team?

The crafty team will probably decide that what they need is to test the software more thoroughly. Because they are thinking in craft terms, what they want is to clearly observe the failure.

This isn't stupid; it makes sense in this context, but how to do it? The commonest solution that I have seen to this kind of problem is to create something called a *soak test*. The soak test will run for a bit longer than the normal time between failure, let's say three weeks for our example. Sometimes people will try to speed up time so that the soak will simulate the problem period in a shorter time, but usually not.

The test runs, the system fails the test after two weeks, and the bug is, eventually, identified and fixed.

Is there any alternative to this strategy? Well, yes!

Soak tests detect resource leaks of one form or another. There are two ways to detect leaks; you can wait for the leak to become obvious, or you can increase the precision of your measurement so you catch the leak early before it becomes catastrophic.

I had a leak in my kitchen recently. It was in a pipe, buried in concrete. We detected the leak once it had soaked the concrete sufficiently for water to start to puddle on the surface. This is the "obvious" detection strategy.

We got a professional in to help us fix the leak. He brought a tool, an engineered solution. It was a highly sensitive microphone that "listened" for the sound of the leak underground.

Using this tool, he could detect the faint hiss of leaking water buried in concrete with sufficient, super-human precision to allow him to identify the location within a few inches and dig a small trench to get at the defective piece of pipe.

So back to our example: the engineering-focused team will use accurate measurement rather than waiting for something bad to happen. They will measure the performance of their software to detect leaks before they become a problem.

This approach has multiple benefits; it means that catastrophic failure, in production, is largely avoided, but it also means that they can get an indication of a problem and valuable feedback on the health of their system much, much sooner. Instead of running a soak test for weeks, the engineering-focused team can detect leaks during regular testing of the system and get a result in a matter of minutes. David Parnas said:

> *Software engineering is often treated as a branch of computer science. This is akin to regarding chemical engineering as a branch of chemistry. We need both chemists and chemical engineers, but they are different.*

Engineering, Creativity, and Craft

To think about engineering in general and software engineering specifically, I have been exploring some of these ideas for a few years. I have spoken on this topic at software conferences and occasionally written on this topic in blog posts.

I sometimes get feedback from people who are adherents to the ideas of software craftsmanship. This feedback is usually of the form "You are missing something important in dismissing craftsmanship."

The ideas of software craftsmanship were important. They represented an important step away from the big-ceremony, production-centered approaches to software development that preceded them. It is not my contention that software craftsmanship is wrong, but rather that it is not enough.

In part, these debates begin from an incorrect premise, one that I have already mentioned. Many of these software craftspeople make the common mistake of assuming that all engineering is about solving production problems. I have already covered that issue; if our problem is "design engineering," then this is a very different, much more exploratory, creative discipline compared to "production engineering."

In addition, though, my software craftspeople interlocutors are concerned about the dangers of throwing away the gains that software craftsmanship has brought—namely, a focus on the following:

- Skill
- Creativity
- Freedom to innovate
- Apprentice schemes

These things are important to any effective, professional approach to software development. However, they are not limited to craft-based approaches. Software craftsmanship movement was an important step in improving software development by refocusing on things that were important, with the things in the previous list being some of those important things.

These ideas had become lost, or at least subsumed, by attempts through the 1980s and 1990s to force-fit some kind of command-and-control, production-centered approach onto software development. This was a terrible idea because although waterfall-style processes and thinking have a place in problems where the steps are well understood, repeatable, and predictable, this bears little or no relationship to the reality of software development.

Software craftsmanship was a much better fit for the type of problem that software development really is.

The problem with craft-based solutions to problems is that they are not scalable in the way that engineering-based solutions are.

Craft can produce good things, but only within certain bounds.

Engineering discipline in virtually all human endeavors increases quality, reduces costs, and generally provides more robust, resilient, and flexible solutions.

It is a big mistake to associate ideas like skill, creativity, and innovation only with craft. Engineers in general, but certainly design engineers, exhibit all of these qualities in abundance all of the time. These attributes are central to the process of design engineering.

So taking an engineering approach to solving problems does not, in any way, reduce the importance of skill, creativity, and innovation. If anything, it amplifies the need for these attributes.

As for training, I wonder if my software crafty friends believe that a new graduate engineer leaving university is immediately given responsibility to design a new bridge or a space shuttle? Of course not!

An engineer at the beginning of their career will work alongside more experienced engineers. They will learn the practicalities of their discipline, their craft, maybe even more so than a craftsperson would.

I see no tension here between craft and engineering. If you take the reasonably formal view of craftsmanship, with guilds, apprentices, journeymen, and master craftsmen, then engineering really was the next step on from that. As scientific rationalism took hold, following on from the enlightenment thinking of the 17th and 18th centuries, engineering was really craft enhanced with a bit more accuracy and measurement. Engineering is the more scalable, more effective offspring of craft.

If you take the more colloquial definitions of craft—think craft fair here—then there are no real standards for quality or progress, so engineering is, perhaps, more of a jump.

Engineering, specifically the application of engineering thinking to design, is really the difference between our high-tech civilization and the agrarian civilizations that preceded us. Engineering is a discipline that allows us to undertake staggeringly complex problems and find elegant, efficient solutions to them.

When we apply the principles of engineering thinking to software development, we see measurable, dramatic improvements in quality, productivity, and the applicability of our solutions.[8]

Why What We Do Is Not Software Engineering

In 2019, Elon Musk's company SpaceX made a big decision; it was working on creating spacecraft that will one day allow humans to live and work on Mars and explore other parts of the solar system. In 2019, it switched from building its Starships out of carbon fiber to building them from stainless steel instead. Carbon fiber was a pretty radical idea; they had done a lot of work, including building prototype fuel tanks from the material. Stainless steel was also a radical choice; most rockets are built from aluminum because of its lightness and strength.

The SpaceX choice of stainless steel over carbon fiber was based on three things: the cost per kilogram was dramatically lower for steel; the high-temperature performance, to cope with re-entry temperatures, was better than aluminum; the low-temperature, cryogenic performance was dramatically better than both of the alternatives.

8. *Accelerate Book* describes how teams that take a more disciplined approach to development spend "44% more time on new work" than teams that don't. See https://amzn.to/2YYf5Z8.

Carbon fiber and aluminum are significantly weaker than steel at very low and high temperatures.

When was the last time you heard anyone make a justification for a decision associated with software creation that sounded even vaguely like that?

This is what engineering decisions look like. They are based on rational criteria, strength at a certain temperature, or economic impact. It is still experimental, it is still iterative, it is still empirical.

You make a decision based on the evidence before you and your theory of what that will mean, and then you test your ideas to see if they work. It is not some perfectly predictable process.

SpaceX built test structures and then pressurized them, first with water and then with liquid nitrogen, so that they could test the cryogenic performance of the materials (steel) and of their manufacturing process. Design engineering is a deeply exploratory approach to gaining knowledge.

Trade-Offs

All engineering is a game of optimization and trade-offs. We are trying to attempt to solve some problem, and, inevitably, we will be faced with choices. In building their rockets, one of the biggest trade-offs for SpaceX is between strength and weight. This is a common problem for flying machines, and actually for most vehicles.

Understanding the trade-offs that we face is a vital, fundamental aspect of engineering decision-making.

If we make our system more secure, it will be more difficult to use; if we make it more distributed, we will spend more time integrating the information that it gathers. If we add more people to speed up development, we will increase the communication overhead, coupling, and complexity, all of which will slow us down.

One of the key trade-offs that is vital to consider in the production of software, at every level of granularity from whole enterprise systems to single functions, is coupling. (We will explore that in much more detail in Chapter 13.)

The Illusion of Progress

The level of change in our industry is impressive, but my thesis is that much of this change is not really significant.

As I write this, I am at a conference on the topic of serverless computing.[9] The move to serverless systems is an interesting one; however, the difference between the toolkits provided by AWS, Azure, Google, or anyone else doesn't really matter.

9. Serverless computing is a cloud-based approach to providing "functions as a service." Functions form the only unit of computing, and the code to run them is started up on demand.

The decision to adopt a serverless approach is going to have some implications for the design of your system. Where do you store state? Where do you manipulate it? How do you divide up the functions of your system? How do you organize and navigate complex systems when the unit of design is a function?

These questions are much more interesting and much more important to the success of your endeavor, whatever it may be, than the detail of how you specify a function or how you use the storage or security features of the platform. Yet nearly all of the presentations that I see on this topic are about the tools, not the design of systems.

This is as if I was a carpenter and was being told the important differences between a slot-headed screw and a cross-headed screw, but I was not being told what screws are useful for, when to use them, and when to choose nails.

Serverless computing does represent a step forward as a computing model. I don't question that. This book is about the ideas that allow us to judge which ideas are important and which are not.

Serverless is important for several reasons, but principally because it encourages a more **modular approach** to design with a better **separation of concerns**, particularly with respect to data.

Serverless computing changes the economics of systems by moving the calculation from "cost per byte" to "cost per CPU cycle." This means, or should mean, that we need to consider very different kinds of optimizations.

Instead of optimizing our systems to minimize storage, by having normalized data stores, we should probably be accepting a more genuinely distributed model of computing using non-normalized stores and eventual-consistency patterns. These things matter because of their impact on the modularity of the systems that we create.

The tools matter only to the degree to which they "move the dial" on some more fundamental things.

The Journey from Craft to Engineering

It is important not to dismiss the value of craft. The care and attention to detail are necessary to create work of high quality. It is also important not to dismiss the importance of engineering to amplifying the quality and effectiveness of the products of craft.

The first people to build a controllable, heavier-than-air, powered flying machine were the Wright Brothers. They were excellent craftsmen and excellent engineers. Much of their work was based on empirical discovery, but they also did real research into the effectiveness of their designs. As well as being the first people to construct a flying machine, they were the first people to build a wind tunnel to allow them to measure the effectiveness of their wing designs.

An airplane wing is a remarkable structure. The Wright brothers construction is a beautiful, though by modern standards incredibly crude, device. It is built of wood and wire and covered in cloth taughtened and made wind-proof by banana oil.

It and the wind tunnel were used to evolve their understanding of the basics of a theory of aerodynamics, building on the work of earlier pioneers. However, primarily, the Wright Brothers' flying machine in general, and wing in particular, was built through a process of trial and error more than pure theoretical design.

To modern eyes it looks like the product of craft more than engineering. This is partly, though not wholly, true. Many people had tried craft-based approaches to building a "flying machine" and failed. One of the important reasons for the success of the Wright Brothers was that they employed engineering. They did the calculations and created and used the tools of measurement and research. They controlled the variables so that they could deepen their understanding and refine their model of flight. Then they created models and gliders and wind-tunnel pieces to test and then grow their understanding. The principles that they established weren't perfect, but they improved on not just the practicalities, but also the theory.

By the time the Wright Brothers had achieved heavier-than-air-controllable-flight, their aerodynamic research allowed them to build flying machines with an 8.3:1 glide ratio.[10]

To compare this with a modern airplane wing, say the wing of a modern sailplane: The wing of the Wright Flyer was under-cambered (a slow high-lift airfoil), and it was heavy by modern standards, though of light construction in its day. It used simple natural materials and achieved this 8.3:1.

Through engineering, empirical discovery, and experimentation, as well as materials science, refining of aerodynamic theory, computer modeling, and so on, a modern sailplane will have a carbon fiber, high-aspect-ratio wing. It is optimized to be light and strong to the degree that you can clearly see it bend and flex as it generates lift. It can achieve glide ratios of more than 70:1, nearly nine times better than the Wright Flyer.

Craft Is Not Enough

Craft is important, particularly so if by *craft* you really mean creativity. Our discipline is a deeply creative endeavor, but so is engineering. I believe that engineering is actually the height of human creativity and ingenuity. That is the kind of thinking that we need if we aim to create great works in software.

Time for a Rethink?

The evolution of **software engineering** as a discipline has not really achieved what many people hoped for. Software has changed, and is changing, the world. There have been some wonderful pieces of work and innovative, interesting, and exciting systems built, but for many teams, organizations, and individual developers, it is not always clear how to succeed, or even how to make progress.

10. The glide ratio is one measure of the efficiency of a flying machine. The ratio is between distance traveled and height lost. For example, for every foot (or meter) that the plane descends in a (unpowered) glide, it will move forward 8.3 feet (or meters). See https://en.wikipedia.org/wiki/Lift-to-drag_ratio.

Our industry is awash with philosophies, practices, processes, and technologies. There are religious wars among technologists over the best programming languages, architectural approaches, development processes, and tools. There often seems to be only a loose sense of what the objectives and strategies of our profession are or should be.

Modern teams fight with schedule pressure, quality, and maintainability of their designs. They often struggle to identify the ideas that really land with users, and they fail to allow themselves the time to learn about the problem domain, the technology, and the opportunities to get something great into production.

Organizations often struggle to get what they want out of software development. They often complain about the quality and efficiency of development teams. They often misunderstand the things that they can do to help overcome these difficulties.

Meanwhile, I perceive a fairly deep level of agreement among the experts, whose opinions I value, about some fundamental ideas that are not often, or at least not clearly enough, stated.

Perhaps it is time to think again about what some of those fundamentals are. What are the principles that are common to our discipline? What are the ideas that will be true for decades, not just for the current generation of technical tools?

Software development is not a simple task, and it is not a homogeneous task. However, there are some practices that are generic. There are ways of thinking about, managing, organizing, and practicing software development that have a significant, even dramatic, impact on all of these problematic aspects of the endeavor.

The rest of this book is intended to explore some of these generic ideas and to provide a list of foundational principles that should be common to all software development, whatever the problem domain, whatever the tools, whatever the commercial or quality demands.

The ideas in this book seem to me to represent something deep, something fundamental, about the nature of our endeavor.

When we get these things right, and many teams do, we see greater productivity, less stress and burnout in team members, higher quality in design, and more resilience in the systems that we create.

The systems that we build please their users more. We see dramatically fewer bugs in production, and teams that employ these ideas find it significantly easier to change almost any aspect of the systems that they work on as their learning evolves. The bottom-line result of this is usually greater commercial success for the organizations that practice in this way. These attributes are the hallmarks of **engineering**.

Engineering amplifies our ability to be creative, to make useful things, to proceed with confidence and quality. It allows us to explore ideas and ultimately to scale our ability to create things so that we can build ever bigger, more complex systems.

We are at the birth of a genuine engineering discipline for software. We could, if we grasp this opportunity, begin to change the way in which software development is practiced, organized, and taught.

This may well be a generational change, but it is of such enormous value to the organizations that employ us, and to the world in general, that we must try. What if we could build software more quickly and more cost-effectively? What if that software was also higher quality, easier to maintain, more adaptable, more resilient, and a better fit for the needs of its users?

Summary

In software we have somewhat redefined what *engineering* means. Certainly in some circles we have come to see engineering as an unnecessary, onerous, and burdensome thing that gets in the way of "real software development." Real engineering in other disciplines is none of these things. Engineers in other disciplines make progress more quickly, not less. They create work of higher quality, not lower.

When we begin to adopt a practical, rational, lightweight, scientific approach to software development, we see similar benefits. Software engineering will be specific to software, but it will also help us to build better software faster, not get in the way of us doing that.

3

Fundamentals of an Engineering Approach

Engineering in different disciplines varies. Bridge building is not the same as aerospace engineering, and neither is it the same as electrical engineering or chemical engineering, but all of these disciplines share some common ideas. They are all firmly grounded in scientific rationalism and take a pragmatic, empirical approach to making progress.

If we are to achieve our goal of trying to define a collection of long-lasting thoughts, ideas, practices, and behaviors that we could collectively group together under the name *software engineering*, these ideas must be fairly fundamental to the reality of software development and robust in the face of change.

An Industry of Change?

We talk a lot about change in our industry. We get excited about new technologies and new products, but do these changes really "move the dial" on software development? Many of the changes that exercise us don't seem to make as much difference as we sometimes seem to think that they will.

My favorite example of this was demonstrated in a lovely conference presentation by "Christin Gorman."[1] In it, Christin demonstrates that when using the then popular open source object relational mapping library Hibernate, it was actually more code to write than the equivalent behavior written in SQL, subjectively at least; the SQL was also easier to understand. Christin goes on to amusingly contrast software development with making cakes. Do you make your cake with a cake mix or choose fresh ingredients and make it from scratch?

1. Source: "Gordon Ramsay Doesn't Use Cake Mixes" by Christin Gorman,https://bit.ly/3g02cWO

Much of the change in our industry is ephemeral and does not improve things. Some, like in the Hibernate example, actually make things worse.

My impression is that our industry struggles to learn and struggles to make progress. This relative lack of advancement has been masked by the incredible progress that has been made in the hardware on which our code runs.

I don't mean to imply that there has been no progress in software—far from it—but I do believe that the pace of progress is much slower than many of us think. Consider, for a moment, what changes in your career have had a significant impact on the way in which you think about and practice software development. What ideas made a difference to the quality, scale, or complexity of the problems that you can solve?

The list is shorter than we usually assume.

For example, I have employed something like 15 or 20 different programming languages during my professional career. Although I have preferences, only two changes in language have radically changed how I think about software and design.

Those steps were the step from Assembler to C and the step from procedural to OO programming. The individual languages are less important than the programming paradigm to my mind. Those steps represented significant changes in the level of abstraction that I could deal with in writing code. Each represented a step-change in the complexity of the systems that we could build.

When Fred Brooks wrote that there were no order-of-magnitude gains, he missed something. There may not be any 10x gains, but there are certainly 10x losses.

I have seen organizations that were hamstrung by their approach to software development, sometimes by technology, more often by process. I once consulted in a large organization that hadn't released any software into production for more than five years.

We not only seem to find it difficult to learn new ideas; we seem to find it almost impossible to discard old ideas, however discredited they may have become.

The Importance of Measurement

One of the reasons that we find it difficult to discard bad ideas is that we don't really measure our performance in software development very effectively.

Most metrics applied to software development are either irrelevant (velocity) or sometimes positively harmful (lines of code or test coverage).

In agile development circles it has been a long-held view that measurement of software team, or project performance, is not possible. Martin Fowler wrote about one aspect of this in his widely read Bliki in 2003.[2]

2. Source: "Cannot Measure Productivity" by Martin Fowler,https://bit.ly/3mDO2fB

Fowler's point is correct; we don't have a defensible measure for productivity, but that is not the same as saying that we can't measure anything useful.

The valuable work carried out by Nicole Fosgren, Jez Humble, and Gene Kim in the "State of DevOps" reports[3] and in their book *Accelerate: The Science of Lean Software & DevOps*[4] represents an important step forward in being able to make stronger, more evidence-based decisions. They present an interesting and compelling model for the useful measurement of the performance of software teams.

Interestingly, they don't attempt to measure productivity; rather, they evaluate the effectiveness of software development teams based on two key attributes. The measures are then used as a part of a predictive model. They cannot prove that these measures have a causal relationship with the performance of software development teams, but they can demonstrate a statistical correlation.

The measures are **stability** and **throughput**. Teams with high stability and high throughput are classified as "high performers," while teams with low scores against these measures are "low performers."

The interesting part is that if you analyze the activities of these high- and low-performing groups, they are consistently correlated. High-performing teams share common behaviors. Equally, if we look at the activities and behaviors of a team, we can predict their score, against these measures, and it too is correlated. Some activities can be used to predict performance on this scale.

For example, if your team employs test automation, trunk-based development, deployment automation, and about ten other practices, their model predicts that you will be practicing **continuous delivery**. If you practice continuous delivery, the model predicts that you will be "high performing" in terms of software delivery performance and organizational performance.

Alternatively, if we look at organizations that are seen as high performers, then there are common behaviors, such as continuous delivery and being organized into small teams, that they share.

Measures of stability and throughput, then, give us a model that we can use to predict team outcomes.

Stability and throughput are each tracked by two measures.

Stability is tracked by the following:

- **Change Failure Rate**: The rate at which a change introduces a defect at a particular point in the process
- **Recovery Failure Time**: How long to recover from a failure at a particular point in the process

3. Source: Nicole Fosgren, Jez Humble, Gene Kim, https://bit.ly/2PWyjw7

4. The *Accelerate Book* describes how teams that take a more disciplined approach to development spend "44% more time on new work" than teams that don't. See https://amzn.to/2YYf5Z8.

Measuring stability is important because it is really a measure of the quality of work done. It doesn't say anything about whether the team is building the right things, but it does measure that their effectiveness in delivering software with measurable quality.

Throughput is tracked by the following:

- **Lead Time**: A measure of the efficiency of the development process. How long for a single-line change to go from "idea" to "working software"?
- **Frequency**: A measure of speed. How often are changes deployed into production?

Throughput is a measure of a team's efficiency at delivering ideas, in the form of working software.

How long does it take to get a change into the hands of users, and how often is that achieved? This is, among other things, an indication of a team's opportunities to learn. A team may not take those opportunities, but without a good score in throughput, any team's chance of learning is reduced.

These are technical measures of our development approach. They answer the questions "what is the quality of our work?" and "how efficiently can we produce work of that quality?"

These are meaningful ideas, but they leave some gaps. They don't say anything about whether we are building the right things, only if we are building them right, but just because they aren't perfect does not diminish their utility.

Interestingly, the correlative model that I described goes further than predicting team size and whether you are applying continuous delivery. The *Accelerate* authors have data that shows significant correlations with much more important things.

For example, organizations made up of high-performing teams, based on this model, make more money than orgs that don't. Here is data that says that there is a correlation between a development approach and the commercial outcome for the company that practices it.

It also goes on to dispel a commonly held belief that "you can have either speed or quality but not both." This is simply not true. Speed and quality are clearly correlated in the data from this research. The route to speed is high-quality software, the route to high-quality software is speed of feedback, and the route to both is great engineering.

Applying Stability and Throughput

The correlation of good scores in these measures with high-quality results is important. It offers us an opportunity to use them to evaluate changes to our process, organization, culture, or technology.

Imagine, for example, that we are concerned with the quality of our software. How could we improve it? We could decide to make a change to our process. Let us add a change approval board (CAB).

Clearly the addition of extra review and sign-offs are going to adversely impact on throughput, and such changes will inevitably slow down the process. However, do they increase stability?

For this particular example the data is in. Perhaps surprisingly, change approval boards don't improve stability. However, the slowing down of the process does impact stability adversely.

> *We found that external approvals were negatively correlated with lead-time, deployment frequency, and restore-time, and had no correlation with change fail rate. In short, approval by an external body (such as a manager or CAB) simply doesn't work to increase the stability of production systems, measured by time to restore service and change fail rate. However, it certainly slows things down. It is, in fact, worse than having no change approval process at all.*[5]

My real point here is not to poke fun at change approval boards, but rather to show the importance of making decisions based on evidence rather than guesswork.

It is not obvious that CABs are a bad idea. They sound sensible, and in reality that is how many, probably most, organizations try to manage quality. The trouble is that it doesn't work.

Without effective measurement, we can't tell that it doesn't work; we can only make guesses.

If we are to start applying a more evidence-based, scientifically rational approach to decision-making, you shouldn't take my word, or the word of Forsgren and her co-authors, on this or anything else.

Instead, you could make this measurement for yourself, in your team. Measure the throughput and stability of your existing approach, whatever that may be. Make a change, whatever that may be. Does the change move the dial on either of these measures?

You can read more about this correlative model in the excellent *Accelerate* book. It describes the approach to measurement and the model that is evolving as research continues. My point here is not to duplicate those ideas, but to point out the important, maybe even profound, impact that this should have on our industry. **We finally have a useful measuring stick.**

We can use this model of stability and throughput to measure the effect of any change.

We can see the impact of changes in organization, process, culture, and technology. "If I adopt this new language, does it increase my throughput or stability?"

We can also use these measures to evaluate different parts of our process. "If I have a significant amount of manual testing, it is certainly going to be slower than automated testing, but does it improve stability?"

We still have to think carefully. We need to consider the meaning of the results. What does it mean if something reduces throughput but increases stability?

Nevertheless, having meaningful measures that allow us to evaluate actions is important, even vital, to taking a more evidence-based approach to decision-making.

---

5. *Accelerate* by Nicole Forsgren, Jez Humble, and Gene Kim, 2018

The Foundations of a Software Engineering Discipline

So, what are some of these foundational ideas? What are the ideas that we could expect to be correct in 100 years' time and applicable whatever our problem and whatever our technology?

There are two categories: process, or maybe even philosophical approach, and technique or design.

More simply, our discipline should focus on two core competencies.

We should become **experts at learning**. We should recognize and accept that our discipline is a creative design discipline and has no meaningful relationship to production-engineering and instead focus on mastery of the skills of exploration, discovery, and learning. This is a practical application of a scientific style of reasoning.

We also need to focus on improving our skills in managing complexity. We build systems that don't fit into our heads. We build systems on a large scale with large groups of people working on them. We need to become **expert at managing complexity** to cope with this, both at the technical level and at the organizational level.

Experts at Learning

Science is humanity's best problem-solving technique. If we are to become experts at learning, we need to adopt and become skilled at the kind of practical science-informed approach to problem-solving that is the essence of other engineering disciplines.

It must be tailored to our problems. Software engineering will be different from other forms of engineering, specific to software, in the same way that aerospace engineering is different from chemical engineering. It needs to be practical, light weight, and pervasive in our approach to solving problems in software.

There is considerable consensus among people who many of us consider to be thought leaders in our industry on this topic. Despite being well known, these ideas are not currently universally or even widely practiced as the foundations of how we approach much of software development.

There are five linked behaviors in this category:

- Working iteratively
- Employing fast, high-quality feedback
- Working incrementally
- Being experimental
- Being empirical

If you have not thought about this before, these five practices may seem abstract and rather divorced from the day-to-day activities of software development, let alone software engineering.

Software development is an exercise in exploration and discovery. We are always trying to learn more about what our customers or users want from the system, how to better solve the problems presented to us, and how to better apply the tools and techniques at our disposal.

We learn that we have missed something and have to fix things. We learn how to organize ourselves to work better, and we learn to more deeply understand the problems that we are working on.

Learning is at the heart of everything that we do. These practices are the foundations of any effective approach to software development, but they also rule out some less effective approaches.

Waterfall development approaches don't exhibit these properties, for example. Nevertheless, these behaviors are all correlated with high performance in software development teams and have been the hallmarks of successful teams for decades.

Part II explores each of these ideas in more depth from a practical perspective: How do we become experts at learning, and how do we apply that to our daily work?

Experts at Managing Complexity

As a software developer, I see the world through the lens of software development. As a result, my perception of the failures in software development and the culture that surrounds it can largely be thought of in terms of two information science ideas: concurrency and coupling.

These are difficult in general, not just in software design. So, these ideas leak out from the design of our systems and affect the ways in which the organizations in which we work operate.

You can explain this with ideas like Conway's law,[6] but Conway's law is more like an emergent property of these deeper truths.

You can profitably think of this in more technical terms. A human organization is just as much an information system as any computer system. It is almost certainly more complex, but the same fundamental ideas apply. Things that are fundamentally difficult, like concurrency and coupling, are difficult in the real world of people, too.

If we want to build systems any more complex than the simplest of toy programming exercises, we need to take these ideas seriously. We need to manage the complexity of the systems that we create as we create them, and if we want to do this at any kind of scale beyond the scope of a single, small team, we need to manage the complexity of the organizational information systems as well as the more technical software information systems.

As an industry, it is my impression that we pay too little attention to these ideas, so much so that all of us who have spent any time around software are familiar with the results: big-ball-of-mud systems, out-of-control technical debt, crippling bug counts, and organizations afraid to make changes to the systems that they own.

---

6. In 1967, Mervin Conway observed that "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." See https://bit.ly/3s2KZP2.

I perceive all of these as a symptom of teams that have lost control of the complexity of the systems that they are working on.

If you are working on a simple, throwaway software system, then the quality of its design matters little. If you want to build something more complex, then you must divide the problem up so that you can think about parts of it without becoming overwhelmed by the complexity.

Where you draw those lines depends on a lot of variables: the nature of the problem that you are solving, the technologies that you are employing, and probably even how smart you are, to some extent, but you must draw the lines if you want to solve harder problems.

Immediately as you buy in to this idea, we are talking about ideas that have a big impact in terms of the design and architecture of the systems that we create. I was a little wary, in the previous paragraph, of mentioning "smartness" as a parameter, but it is one. The problem that I was wary of is that most of us overestimate our abilities to solve a problem in code.

This is one of the many lessons that we can learn from an informal take on science. It's best to start off assuming that our ideas are wrong and work to that assumption. So we should be much more wary about the potential explosion of complexity in the systems that we create and work to manage it diligently and with care as we make progress.

There are five ideas in this category, too. These ideas are closely related to one another and linked to the ideas involved in becoming experts at learning. Nevertheless, these five ideas are worth thinking about if we are to manage complexity in a structured way for any information system:

- Modularity
- Cohesion
- Separation of concerns
- Information hiding/abstraction
- Coupling

We will explore each of these ideas in much more depth in Part III.

Summary

The tools of our trade are often not really what we think they are. The languages, tools, and frameworks that we use change over time and from project to project. The ideas that facilitate our learning and allow us to deal with the complexity of the systems that we create are the real tools of our trade. By focusing on these things, it will help us to better choose the languages, wield the tools, and apply the frameworks in ways that help us do a more effective job of solving problems with software.

Having a "yardstick" that allows us to evaluate these things is an enormous advantage if we want to make decisions based on evidence and data, rather than fashion or guesswork. When making a choice, we should ask ourselves, "does this increase the quality of the software that we create?" measured by the metrics of **stability**. Or "does this increase the efficiency with which we create software of that quality" measured by **throughput**. If it doesn't make either of these things worse, we can pick what we prefer; otherwise, why would we choose to do something that makes either of these things worse?