DAVID FARLEY

MODERN SOFTWARE ENGINEERING

Doing What Works to
Build Better Software Faster

Foreword by **TRISHA GEE**

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for *Modern Software Engineering*

"*Modern Software Engineering* gets it right and describes the ways skilled practitioners actually engineer software today. The techniques Farley presents are not rigid, prescriptive, or linear, but they are disciplined in exactly the ways software requires: empirical, iterative, feedback-driven, economical, and focused on running code."

—Glenn Vanderburg, Director of Engineering at Nubank

"There are lots of books that will tell you how to follow a particular software engineering practice; this book is different. What Dave does here is set out the very essence of what defines software engineering and how that is distinct from simple craft. He explains why and how in order to master software engineering you must become a master of both learning and of managing complexity, how practices that already exist support that, and how to judge other ideas on their software engineering merits. This is a book for anyone serious about treating software development as a true engineering discipline, whether you are just starting out or have been building software for decades."

—Dave Hounslow, Software Engineer

"These are important topics and it's great to have a compendium that brings them together as one package."

—Michael Nygard, author of *Release IT*, professional programmer,
and software architect

"I've been reading the review copy of Dave Farley's book and it's what we need. It should be required reading for anyone aspiring to be a software engineer or who wants to master the craft. Pragmatic, practical advice on professional engineering. It should be required reading in universities and bootcamps."

—Bryan Finster, Distinguished Engineer and
Value Stream Architect at USAF Platform One

This page intentionally left blank

MODERN SOFTWARE ENGINEERING

This page intentionally left blank

MODERN SOFTWARE ENGINEERING

DOING WHAT WORKS TO BUILD BETTER SOFTWARE FASTER

David Farley

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021947543

Cover image: spainter_vfx/Shutterstock

ISBN-13: 978-0-13-731491-1
ISBN-10: 0-13-731491-4

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

I would like to dedicate this book to my wife Kate and to my sons, Tom and Ben.

Kate has been unfailingly supportive of my writing and my work over many years and is always an intellectually stimulating companion as well as my best friend.

Tom and Ben are young men whom I admire as well as love as a parent, and it has been my pleasure, while working on this book, to have also had the privilege to work alongside them on several joint ventures. Thanks for your help and support.

This page intentionally left blank

Contents

Foreword

I studied computer science at university, and of course I completed several modules called "software engineering" or variations on the name.

I was not new to programming when I started my degree and had already implemented a fully working inventory system for my high school's Careers Library. I remember being extremely confused by "software engineering." It all seemed designed to get in the way of actually writing code and delivering an application.

When I graduated in the early years of this century, I worked in the IT department for a large car company. As you'd expect, they were big on software engineering. It's here I saw my first (but certainly not my last!) Gantt chart, and it's where I experienced waterfall development. That is, I saw software teams spending significant amounts of time and effort in the requirements gathering and design stages and much less time in implementation (coding), which of course overran into testing time and then the testing...well, there wasn't much time left for that.

It seemed like what we were told was "software engineering" was actually getting in the way of creating quality applications that were useful to our customers.

Like many developers, I felt there must be a better way.

I read about Extreme Programming and Scrum. I wanted to work in an agile team and moved jobs a few times trying to find one. Plenty said they were agile, but often this boiled down to putting requirements or tasks on index cards, sticking them on the wall, calling a week a *sprint*, and then demanding the development team deliver "x" many cards in each sprint to meet some arbitrary deadline. Getting rid of the traditional "software engineering" approach didn't seem to work either.

Ten years into my career as a developer, I interviewed to work for a financial exchange in London. The head of software told me they did Extreme Programming, including TDD and pair programming. He told me they were doing something called *continuous delivery*, which was like continuous integration but all the way into production.

I'd been working for big investment banks where deployment took a minimum of three hours and was "automated" by the means of a 12-page document of manual steps to follow and commands to type. Continuous delivery seemed like a lovely idea but surely was not possible.

The head of software was Dave Farley, and he was in the process of writing his *Continuous Delivery* book when I joined the company.

I worked with him there for four life-changing, career-making years. We really did do pair programming, TDD, and continuous delivery. I also learned about behavior-driven development, automated acceptance testing, domain-driven design, separation of concerns, anti-corruption layers, mechanical sympathy, and levels of indirection.

I learned about how to create high-performance, low-latency applications in Java. I finally understood what big O notation really meant and how it applied to real-world coding. In short, all that stuff I had learned at university and read in books was actually used.

It was applied in a way that made sense, worked, and delivered an extremely high-quality, high-performance application that offered something not previously available. More than that, we were happy in our jobs and satisfied as developers. We didn't work overtime, we didn't have crunch times close to releases, the code did not become more tangled and unmaintainable over those years, and we consistently and regularly delivered new features and "business value."

How did we achieve this? By following the practices Dave outlines in this book. It wasn't formalized like this, and Dave has clearly brought in his experiences from many other organizations to narrow down to the specific concepts that are applicable for a wider range of teams and business domains.

What works for two or three co-located teams on a high-performance financial exchange isn't going to be exactly the same thing that works for a large enterprise project in a manufacturing firm or for a fast-moving startup.

In my current role as a developer advocate, I speak to hundreds of developers from all sorts of companies and business domains, and I hear about their pain points (many of them, even now, not dissimilar to my own experiences 20 years ago) and success stories. The concepts Dave has covered in this book are general enough to work in all these environments and specific enough to be practically helpful.

Funnily enough, it was after I left Dave's team that I started being uncomfortable with the title *software engineer*. I didn't think that what we do as developers is engineering; I didn't think that it was engineering that had made that team successful. I thought engineering was too structured a discipline for what we do when we're developing complex systems. I like the idea of it being a "craft," as that encapsulates the idea of both creativity and productivity, even if it doesn't place enough emphasis on the teamwork that's needed for working on software problems at scale. Reading this book has changed my mind.

Dave clearly explains why we have misconceptions of what "real" engineering is. He shows how engineering is a science-based discipline, but it does not have to be rigid. He walks through how scientific principles and engineering techniques apply to software development and talks about why the production-based techniques we thought were engineering are not appropriate to software development.

What I love about what Dave has done with this book is that he takes concepts that might seem abstract and difficult to apply to the real code we have to work with in our jobs and shows how to use them as tools to think about our specific problems.

The book embraces the messy reality of developing code, or should I say, software engineering: there is no single correct answer. Things will change. What was correct at one point in time is sometimes very wrong even a short time later.

The first half of the book offers practical solutions for not only surviving this reality but thriving in it. The second half takes topics that might be considered abstract or academic by some and shows how to apply them to design better (e.g., more robust or more maintainable or other characteristics of "better") code.

Here, design absolutely does not mean pages and pages of design documents or UML diagrams but may be as simple as "thinking about the code before or during writing it." (One of the things I noticed when I pair programmed with Dave was how little time he spends actually typing the code.

Turns out, thinking about what we write before we write it can actually save us a lot of time and effort.)

Dave doesn't avoid, or try to explain away, any contradictions in using the practices together or potential confusion that can be caused by a single one. Instead, because he takes the time to talk about the trade-offs and common areas of confusion, I found myself understanding for the first time that it is precisely the balance and the tension between these things that creates "better" systems. It's about understanding that these things are guidelines, understanding their costs and benefits, and thinking of them as lenses to use to look at the code/design/architecture, and occasionally dials to twiddle, rather than binary, black-and-white, right-or-wrong rules.

Reading this book made me understand why we were so successful, and satisfied, as "software engineers" during that time I worked with Dave. I hope that by reading this book, you benefit from Dave's experience and advice, without having to hire a Dave Farley for your team.

Happy engineering!

—Trisha Gee, developer advocate and Java champion

This page intentionally left blank

Preface

This book puts the *engineering* back into *software engineering*. In it, I describe a practical approach to software development that applies a consciously rational, scientific style of thinking to solving problems. These ideas stem from consistently applying what we have learned about software development over the last few decades.

My ambition for this book is to convince you that engineering is perhaps not what you think it is and that it is completely appropriate and effective when applied to software development. I will then proceed to describe the foundations of such an engineering approach to software and how and why it works.

This is not about the latest fads in process or technology, but rather proven, practical approaches where we have the data that shows us what works and what doesn't.

Working iteratively in small steps works better than not. Organizing our work into a series of small, informal experiments and gathering feedback to inform our learning allows us to proceed more deliberately and to explore the problem and solution spaces that we inhabit. Compartmentalizing our work so that each part is focused, clear, and understandable allows us to evolve our systems safely and deliberately even when we don't understand the destination before we begin.

This approach provides us with guidance on where to focus and what to focus on, even when we don't know the answers. It improves our chances of success, whatever the nature of the challenge that we are presented with.

In this book, I define a model for how we organize ourselves to create great software and how we can do that efficiently, and at any scale, for genuinely complex systems, as well as for simpler ones.

There have always been groups of people who have done excellent work. We have benefitted from innovative pioneers who have shown us what is possible. In recent years, though, our industry has learned how to better explain what really works. We now better understand what ideas are more generic and can be applied more widely, and we have data to back up this learning.

We can more reliably build software better and faster, and we have data to back that up. We can solve world-class, difficult problems, and we have experience with many successful projects, and companies, to back those claims, too.

This approach assembles a collection of important foundational ideas and builds on the work that went before. At one level there is nothing that is new here in terms of novel practices, but the approach that I describe assembles important ideas and practices into a coherent whole and gives us principles on which a software engineering discipline may be built.

This is not a random collection of disparate ideas. These ideas are intimately entwined and mutually reinforcing. When they come together and are applied consistently to how we think about, organize, and undertake our work, they have a significant impact on the efficiency and the quality of that work. This is a fundamentally different way of thinking about what it is that we do, even though each idea in isolation may be familiar. When these things come together and are applied as guiding principles for decision-making in software, it represents a new paradigm for development.

We are learning what software engineering really means, and it is not always what we expected.

Engineering is about adopting a scientific, rationalist approach to solving practical problems within economic constraints, but that doesn't mean that such an approach is either theoretical or bureaucratic. Almost by definition, engineering is pragmatic.

Past attempts at defining *software engineering* have made the mistake of being too proscriptive, defining specific tools or technologies. Software engineering is more than the code that we write and the tools that we use. Software engineering is not production engineering in any form; that is not our problem. If when I say *engineering* it makes you think bureaucracy, please read this book and think again.

Software engineering is not the same thing as computer science, though we often confuse the two. We need both software engineers and computer scientists. This book is about the discipline, process, and ideas that we need to apply to reliably and repeatably create better software.

To be worthy of the name, we would expect an engineering discipline for software to help us solve the problems that face us with higher quality and more efficiency.

Such an engineering approach would also help us solve problems that we haven't thought of yet, using technologies that haven't been invented yet. The ideas of such a discipline would be general, durable, and pervasive.

This book is an attempt to define a collection of such closely related, interlinked ideas. My aim is to assemble them into something coherent that we can treat as an approach that informs nearly all of the decisions that we make as software developers and software development teams.

Software engineering as a concept, if it is to have any meaning at all, must provide us with an advantage, not merely an opportunity to adopt new tools.

All ideas aren't equal. There are good ideas, and there are bad ideas, so how can we tell the difference? What principles could we apply that will allow us to evaluate any new idea in software and software development and decide if it will likely be good or bad?

Anything that can justifiably be classified as an engineering approach to solving problems in software will be generally applicable and foundational in scope. This book is about those ideas. What criteria should you use to choose your tools? How should you organize your work? How should you organize the systems that you build and the code that you write to increase your chances of success in their creation?

A Definition of Software Engineering?

I make the claim in this book that we should think of software engineering in these terms:

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

My aim is an ambitious one. I want to propose an outline, a structure, an approach that we could consider to be a genuine engineering discipline for software. At the root this is based in three key ideas.

- Science and its practical application "engineering" are vital tools in making effective progress in technical disciplines.
- Our discipline is fundamentally one of learning and discovery, so we need to become **experts at learning** to succeed, and science and engineering are how we learn most effectively.
- Finally, the systems that we build are often complex and are increasingly so. Meaning, to cope with their development, we need to become **experts at managing that complexity**.

What Is in This Book?

Part I, "What Is Software Engineering?", begins by looking at what engineering really means in the context of software. This is about the principles and philosophy of engineering and how we can apply these ideas to software. This is a technical philosophy for software development.

Part II, "Optimize for Learning," looks at how we organize our work to allow us to make progress in small steps. How do we evaluate if we are making good progress or merely creating tomorrow's legacy system today?

Part III, "Optimize for Managing Complexity," explores the principles and techniques necessary for managing complexity. This explores each of these principles in more depth and their meaning and applicability in the creation of high-quality software, whatever its nature.

The final section, Part IV, "Tools to Support Engineering in Software," describes the ideas and approaches to work that maximize our opportunities to learn and facilitate our ability to make progress in small steps and to manage the complexity of our systems as they grow.

Sprinkled throughout this book, as sidebars, are reflections on the history and philosophy of software engineering and how thinking has progressed. These inserts provide helpful context to many of the ideas in this book.

This page intentionally left blank

Acknowledgments

Writing a book like this takes a long time, a lot of work, and the exploration of numerous ideas. The people who helped me through that process helped me in all sorts of different ways, sometimes agreeing with me and reinforcing my convictions and sometimes disagreeing and forcing me either to strengthen my arguments or to change my mind.

I'd like to start by thanking my wife, Kate, who has helped me in all sorts of ways. Even though Kate is not a software professional, she read large parts of this book, helping me correct my grammar and hone my message.

I'd like to thank my brother in-law, Bernard McCarty, for bouncing ideas around on the topic of science and making me dig deeper to think about why I wanted to talk about experimentation and empiricism as well as lots of other things.

I'd like to thank Trisha Gee for not only writing such a nice foreword, but also being enthusiastic about this book when I needed a boost.

I'd like to thank Martin Thompson for always being there to bounce around opinions on computer science and for usually responding to my rather random thoughts in minutes.

I'd like to thank Martin Fowler, who despite being over-committed to other projects, gave me advice that helped to strengthen this book.

Many more of my friends have contributed indirectly over the years to help me shape my thinking on these topics, and many more: Dave Hounslow, Steve Smith, Chris Smith, Mark Price, Andy Stewart, Mark Crowther, Mike Barker, and many others.

I'd like to thank the team at Pearson for their help and support through the publication process of this book.

I would also like to thank a whole bunch of people—not all of whom I know—who have been supportive, argumentative, challenging, and thoughtful. I have bounced many of these ideas around on Twitter and on my YouTube channel for some years now and have been involved in some great conversations as a result. Thank you!

This page intentionally left blank

About the Author

David Farley is a pioneer of continuous delivery, thought leader, and expert practitioner in continuous delivery, DevOps, TDD, and software development in general.

Dave has been a programmer, software engineer, systems architect, and leader of successful teams for many years, from the early days of modern computing, taking those fundamental principles of how computers and software work and shaping groundbreaking, innovative approaches that have changed how we approach modern software development. He has challenged conventional thinking and led teams to build world-class software.

Dave is co-author of the Jolt award-winning book *Continuous Delivery*, is a popular conference speaker, and runs the highly successful and popular "Continuous Delivery" YouTube channel on the topic of software engineering. He built one of the world's fastest financial exchanges and is a pioneer of BDD, an author of the Reactive Manifesto, and a winner of the Duke award for open source software with the LMAX Disruptor.

Dave is passionate about helping development teams around the world improve the design, quality, and reliability of their software by sharing his expertise through his consultancy, YouTube channel, and training courses.

Twitter:	@davefarley77
YouTube Channel:	https://bit.ly/CDonYT
Blog:	http://www.davefarley.net
Company Website:	https://www.continuous-delivery.co.uk

This page intentionally left blank

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?

Index

C

D

E

F

G-H

I

J-K-L

M

N

O-P

Q

R

S

T

U-V

W-X-Y-Z