DAVID FARLEY

MODERN SOFTWARE ENGINEERING

Doing What Works to
Build Better Software Faster

Foreword by TRISHA GEE

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

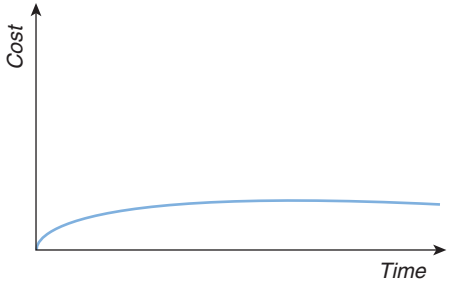

Figure 4.2
The agile cost of change

So, what would it take to achieve a flat Cost of Change curve?

We can't afford to spend lots of time in analysis and design without creating anything, because that means more time not learning what really works. So we need to compress things. We need to work iteratively. We need to do just enough analysis, design, coding, testing, and releasing to get our ideas out into the hands of our customers and users so that we can see what really works. We need to reflect on that and then, given that learning, adapt what we do next to take advantage of it.

This is one of the ideas at the heart of continuous delivery (see Figure 4.3).

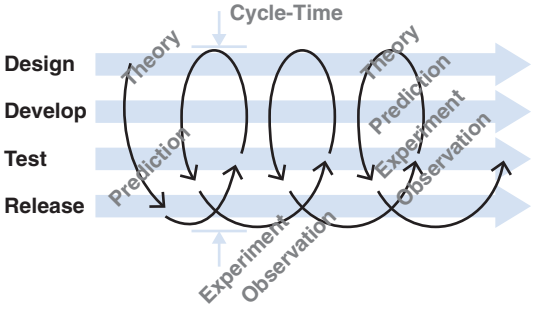

Figure 4.3
Iteration in continuous delivery

The Lure of the Plan

The people who promoted waterfall thinking were well intentioned. They thought that it was the best way forward. Our industry has spent decades trying to make this approach work, and it doesn't.

The difficulty here is that a waterfall approach sounds very sensible: "Think carefully before you start," and "Plan carefully what you are going to do and then execute the plan diligently." Based on our industrial-age experience, these ideas make a lot of sense. If you have a well-defined process, this defined process control approach works extremely well.

When making physical things, the problems of production engineering and the problems of scaling up often outweigh the problems of design. However, this is changing now even in the manufacture of physical things. As manufacturing gets more flexible and some manufacturing plants can change direction, then even in manufacturing this kind of rigid process has been challenged and overturned. This kind of "production-line" thinking dominated most organizations for at least a century, though, and we are somewhat programmed to think about problems this way.

It takes a difficult intellectual leap to recognize that the paradigm in which you are operating is fundamentally the wrong one. This is even more true when the whole world assumes that paradigm to be correct.

Process Wars

If there is no 10x improvement available from language, formalism, or diagramming, where else can we look?

The way in which we organize ourselves and our approach to the skills and techniques of learning and discovery that seem so inherent to our discipline seem like a fruitful avenue to explore.

In the early days of software development, the early programmers were usually highly educated in mathematics, science, or engineering. They worked as individuals or in small groups to develop systems. These people were explorers in a new domain, and like most explorers, they brought their experience and prejudices along with them. Early approaches to software development were often very mathematical.

As the computer revolution kicked in and software development became more ubiquitous, demand rapidly outstripped supply. We needed to produce more, better software faster! So we started looking at other industries to try to copy how they coped with working efficiently at scale.

This is where we made the horrible mistake of misunderstanding the fundamental nature of software development and misapplied techniques from manufacturing and production. We recruited armies of developers and tried to create the software equivalent of mass-production lines.

The people who did this were not stupid, but they did make a big mistake. The problem is multifaceted. Software is complex stuff, and the process of its creation bears no real relationship to a traditional "production problem," which is how most people seem to have thought about it.

Initial attempts at industrializing our discipline were painful, pervasive, and very damaging. It resulted in the creation of a lot of software, but much of it was problematic. It was slow, inefficient, late, did not deliver what our users wanted, and was extremely difficult to maintain. Through the 1980s and 1990s software development exploded as a discipline, and so did the complexity of the processes applied to it in many large organizations.

These failings were despite the fact that many aspects of this problem were well understood by leading thinkers in the discipline.

The Mythical Man Month by Fred Brooks, again, described these problems and how to avoid them in some detail in 1970. If you have never read this seminal work in our discipline, you would probably be surprised at how accurately it describes the problems that you, most likely, face nearly every day in your work as a software developer. This despite the fact that it is based on Brooks's experience of developing the operating system for the IBM 360 mainframe computer in the late 1960s using the comparatively crude technology and tools of the day. Brooks was, yet again, touching on something more important and more fundamental than language, tools, or technology.

During this period many teams produced great software, often completely ignoring the then current "wisdom" of how projects should be planned and managed. There were some common themes in these teams. They tended to be small. The developers were close to the users of their software. They tried ideas quickly and changed tack when things didn't work as they expected. This was revolutionary stuff during this period—so revolutionary in fact that many of these teams essentially operated in stealth mode, where the organization where they worked applied heavy-weight processes that slowed them down.

By the late 1990s, in reaction to these heavyweight processes, some people began to try to define strategies that were more effective. Several different competing approaches to software development were gaining in popularity. Crystal, Scrum, Extreme Programming, and several others tried to capture this very different approach. These viewpoints were formalized into the Agile Manifesto.

In software, it took the agile revolution to overthrow that norm, but even today many, perhaps even most, organizations at heart remain plan/waterfall-driven.

In addition to the difficulty of recognizing the problem, there remains a fair bit of wishful thinking in organizations that cling to waterfall-style planning. It would be lovely if an organization could:

- Correctly identify its users' needs
- Accurately assess the value to the organization if those needs were met
- Accurately estimate how much it would cost to fulfill those needs
- Make a rational decision on whether the benefit outweighed the cost
- Make an accurate plan
- Execute the plan without deviation
- Count the money at the end

The trouble is that this is not credible either at the business level or at a technical level. The real world, and software development within it, just doesn't work like this.

Industry data says that for the best software companies in the world, two-thirds of their ideas produce zero or negative value.[3] We are terrible at guessing what our users want. Even when we ask our users, they don't know what they want either. The most effective approach is to iterate. It is accepting that some, maybe even many, of our ideas will be wrong and work in a way that allows us to try them out as quickly, cheaply, and efficiently as possible.

Assessing the business value of an idea is notoriously difficult, too. There is a famous quote from IBM president Thomas J. Watson, who once predicted that the world demand for computers would one day get as high as five!

This is not a technology problem; this is a human-limitation problem. To make progress we must take a chance, make a guess, be willing to take a risk. We are very bad at guessing, though. So to make progress most efficiently, we must organize ourselves so that our guesses won't destroy us. We need to work more carefully, more defensively. We need to proceed in small steps and limit the scope, or blast radius, of our guesses and learn from them. We need to work iteratively!

Once we have an idea that we would like to execute on, we need to find a way to decide when to stop. How do we call a halt on a bad idea? Once we have decided that the idea is worth the risk of attempting it, how do we limit that blast radius in a way that means that we don't lose everything on a terrible idea? We need to be able to spot the bad ideas as soon as we can. If we can eliminate the bad ideas just by thinking about it, great. However, many ideas aren't that obviously bad. Success is a slippery concept. An idea may even be a good idea, but may be let down by bad timing or poor execution.

We need to find a way to try our ideas with minimum cost, so that if it is bad, we can find that out quickly and at relatively low cost. A 2012 survey of software projects carried out by the McKinsey Group in association with Oxford University found that 17% of large projects (budgets over $15M) went so badly that they threatened the existence of the company that undertook them. How can we identify these bad ideas? If we work in small steps, get real reaction to the progress or otherwise, and constantly validate and review our ideas, we can see soonest, with lowest investment, when things start to work differently to our hopes and plans. If we work iteratively in small steps, the cost of any single step going wrong is inevitably lower; therefore, the level of this risk is reduced.

In "The Beginning of Infinity," David Deutsch describes the profound difference between ideas that are limited in scope and ideas that are not. The comparison of a planned, waterfall, defined-process approach and an iterative, exploratory, experimental approach is a comparison between two such fundamentally different ideas. Defined process control models[4] require a "defined process." By definition this is finite in scope. At the limit of such an approach, there is, at some level, the capacity of a human brain to hold the detail of the entire process. We can be smart and use ideas like abstraction

3. Source: "Online Controlled Experiments at Large Scale," https://stanford.io/2LdjvmC

4. Ken Schwaber described waterfall as a "defined process control model" that he defined as: "The defined process control model requires that every piece of work be completely understood. Given a well-defined set of inputs, the same outputs are generated every time. A defined process can be started and allowed to run until completion, with the same results every time." Schwaber compares this to the "empirical process control model" represented by an agile approach. See https://bit.ly/2UiaZdS.

and concepts like modularity to hide some of the detail, but ultimately defining the process end to end in some kind of plan requires us to have covered everything that will happen. This is an inherently limited approach to solving problems. We can only solve the problems that we can understand up front.

An iterative approach is very different. We can begin when we know almost nothing and yet still make useful progress. We can start with some simple, understandable facet of the system. Use this to explore how our team should work on it, try out our first thoughts on the architecture of our system, try out some technologies that we think might be promising, and so on. None of these things is necessarily fixed. We have still made progress even if we found that the tech was a bad idea and our first concept of the architecture was wrong. We now know better than we did before. This is an inherently open-ended, infinite process. As long as we have some kind of "fitness function," a way of telling if we are heading toward our goal or away from it, we can continue in this vein forever, refining, enhancing, and improving our understanding, our ideas, our skills, and our products. We can even decide to change our "fitness function" along the way if we decide that there are better goals to aim for.

A Beginning of Infinity

In his mind-expanding book *The Beginning of Infinity*, physicist David Deutsch describes science and the enlightenment as the quest for "good explanations" and explains how various ideas in human history represent a "beginning of infinity" that allow us to cope with any conceivable relevant application of these good explanations.

A good example of this is the difference between an alphabet and a pictographic form of writing.

Humans began with the pictographic forms of writing, and Chinese and Japanese writing still take this form (for some writing styles). These are beautiful to look at, but they have a serious flaw. If you come across a word that is new to you, you hear it spoken; you can't write it down until you get someone else to tell you how. Pictographic forms of writing are not really incremental; you have to know the correct symbol for each word. (There are approximately 50,000 characters in Chinese writing.)

An alphabet works in a profoundly different way. Alphabets encode sounds, not words. You can spell any word, maybe incorrectly, in a way that anyone can, at least phonetically, understand what you wrote.

This is true even if you have never heard the word spoken or seen it written before.

Equally you can read a word that you don't know. You can even read words that you don't understand or don't know how to pronounce. You can't do either of these things with pictographic writing. This means that the range of an alphabetic approach to writing is infinite, and a pictographic one is not. One is a scalable approach to representing ideas; the other is not.

This idea of infinite reach or scope is true of an agile approach to development and not true of a waterfall-based approach.

A waterfall approach is sequential. You must answer the questions of the stage that you are in before proceeding to the next stage. This means that however clever we are, there must, at some point, be a limit at which the complexity of the system as a whole goes beyond human understanding.

Human mental capacity is finite, but our capacity to understand is not necessarily so. We can address the physiological limits of our brains by using techniques that we have evolved and developed. We can abstract things, and we can compartmentalize (modularize) our thinking and so scale our understanding to a remarkable degree.

An agile approach to software development actively encourages us to start work on solving problems in smaller pieces. It encourages us to begin work before we know the answer to everything. This approach allows us to make progress, maybe sometimes in suboptimal or even bad directions, but nevertheless, after each step, we learn something new.

This allows us to refine our thinking, identify the next small step, and then take that step. Agile development is an unbounded, infinite approach because we work on small pieces of the problem before moving forward from a known and understood position. This is a profoundly more organic, evolutionary, unbounded approach to problem-solving.

This is a profound difference and explains why agile thinking represents an important and significant step forward in our ability to make progress in solving, ideally, harder and harder problems.

This doesn't mean that agile thinking is perfect or the final answer. Rather, it is an important, significant, enabling step in the direction of better performance.

The lure of the plan is a false one. This is not a more diligent, more controlled, more professional approach. Rather, it is more limited and more based on hunch and guesswork and can, realistically, work only for small, simple, well-understood, well-defined systems.

The implications of this are significant. It means that we must, as Kent Beck famously said in the subtitle to his seminal work *Extreme Programming Explained*, "Embrace change"!

We must learn to have the confidence to begin work precisely when we don't yet know the answers and when we don't know how much work will be involved. This is disquieting for some people and for some organizations, but it is only the same as the reality of much of the human experience. When a business starts out on a new venture, they don't really know when, or even whether, it will be a success. They don't know how many people will like their ideas and whether they will be willing to pay for them.