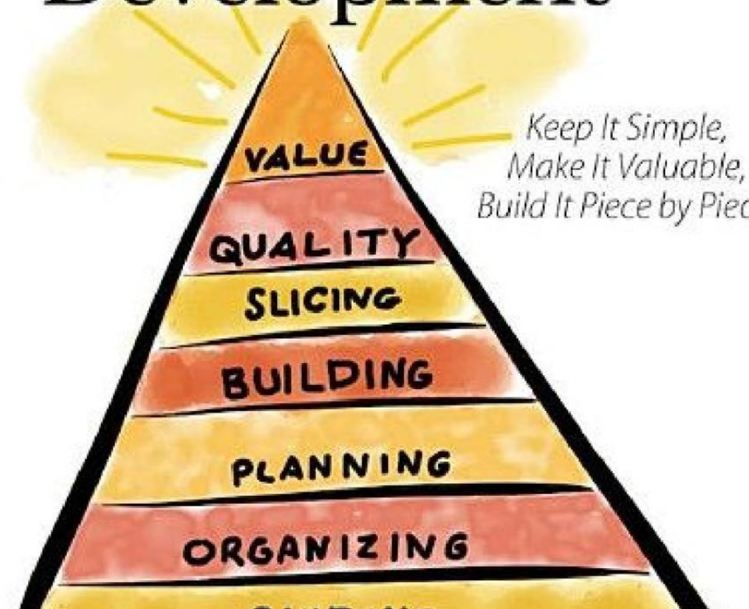
The Pragmatic Programmers

The Nature of Software Development



Keep It Simple, Make It Valuable, Build It Piece by Piece

Ron Jeffries

edited by Michael Swaine

Table of Contents

Early praise for *The Nature of Software Development*

This book should be "The CTO's Guide to Professional Software Development." This is a book every CTO, every VP of engineering, every director of software, and every software team leader should read. In this book they'll find answers to questions that have plagued their peers for decades. The book is simple and direct, and yet it tackles one of the most complicated tasks that humans have ever attempted: managing teams that build high-quality software systems.

→ Robert "Uncle Bob" Martin, founder, Object Mentor

Ditch the buzzword-laden books and read this instead. Ron takes us back to development basics with a great summary of a simple development process that works. Ron shows you just what's important in software development. If you're doing more than this, you're trying too hard.

→ Jeff Langr, author, *Pragmatic Unit Testing in Java 8 with JUnit* and *Modern C++ Programming With Test-Driven Development*

The Nature of Software Development is just like spending a morning with Ron, only you don't have to.

→ Chet Hendrickson

Agile Teacher and Consultant, HendricksonXP

I love this book. Every page has a sketch and a clear explanation of something you can try out right away. It's like sitting down with Ron over a cup of coffee.

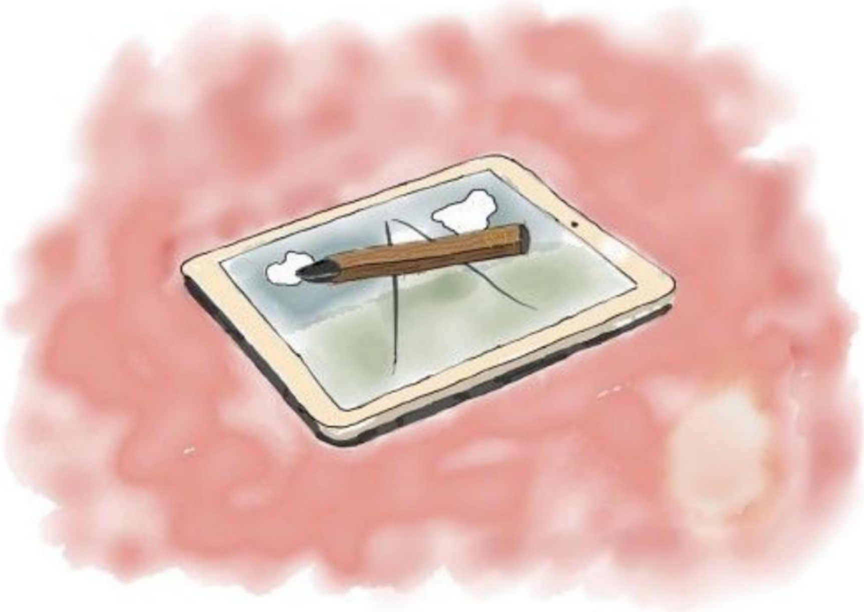
→ Daniel H Steinberg

Dim Sum Thinking

In straightforward prose and sketches, Ron explores the deep question of how to best deliver software. This book is accessible not just to software team members, but to customers and users as well.

→ Bill Wake

Industrial Logic, Inc.



Preface

I've been doing software for over a half century. I've had some great successes and some truly colossal failures.

For all that time, I've been talking with people, coaching, and teaching about software development. And mostly, I've been thinking. I've been trying to figure out how this can all seem so simple and yet be so complex. If you've been involved in software development, you too have probably often felt that all this should be simple, but somehow it gets all complicated.

Thanks to being in the right place at the right time, I've been part of the Agile movement since the very beginning. That has drawn me back toward simplicity.

Like many of the best ideas in software development, modern "Agile" software development offers to make software development more productive and better controlled by making it simpler. Agile is simple. Four values, a dozen principles. How complex could it be? Well, it still seems to get pretty darn complex.

Agile methods like Scrum and XP are also simple. Again a few values, a couple of meetings, a handful of artifacts, how complex could they be? And still it gets so complicated so quickly.

What’s up with that?

I have begun to see a way of looking at the whole process of software development. I’m starting to see a general overview that might help us keep things simple. Inside, there will still be plenty of complexity, but I hope this high-level map will help us pull back and find the simplicity when we find ourselves in the weeds.

Software development has many facets: determining value, managing value flow, organizing around the work, planning, building, and so on. Each of these facets needs to focus on producing value. Value needs to be visible so that it can be guided and managed. For this, we need to step back from the details and find the essential simplicity in this very complex activity.

When I think about things, I draw pictures that focus on some aspect of the topic. I try to think of a few words that will quickly focus my thinking when next I think about the topic. I use pictures to give me a different perspective. Since my drawings are perforce simple—I'm not very skilled—I use them to cut away complexity and look at what's left. I'm giving you a look at that thinking.

This book is an attempt at finding some essential simplicity inside the complex activity of building software products. I believe I have a handle on some good ideas. At best, this is a bit of a clearing along a tangled trail. Please take these thoughts and use them to find your own sense of simplicity amid all the chaos. Good luck!

Acknowledgments

...where to begin...where to end...

My parents, for freedom, trust, and a great library…

Sister Mary Marjorie, for a first taste of science; Mr. Dansky, for a first taste of love for a subject; the Jesuits, for showing the value of thinking and of course for my fashion sense.

Rick Camp, for inviting the kid up the street to be an intern at Strategic Air Command; Bill Rogers, for tossing me into programming and then helping me learn to swim.

Colleagues over the years: Charles Bair, Karen Dueweke, Steve Weiss, Gene Somdahl, Rick Evarts, Mike McConnell, Jean Musinski, Jeanne Hernandez, Dorothy Lieffers, Don Devine…it would take pages to mention everyone who has touched me.

Partners, mentors, colleagues in Agile: Ward Cunningham, Kent Beck, Chet Hendrickson, Ann Anderson, Bob Martin, Alistair Cockburn, Martin Fowler, Michael Feathers, Bob Koss, Brian Button, Brian Marick, Ken Schwaber, Jeff Sutherland, Ken Auer…I can't begin to list all those to whom I'm grateful.

The Internet and Twittersphere, who are surely tired of seeing me try to explain these thoughts so often.

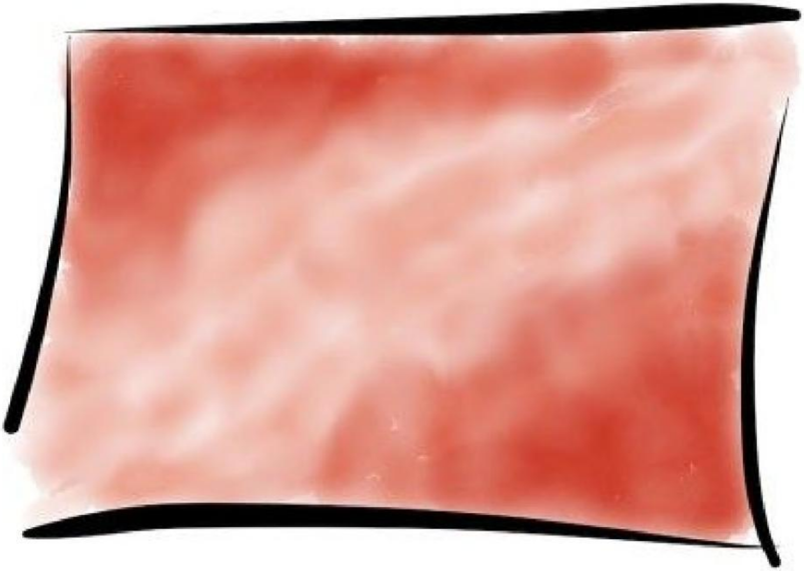
Helpers with this book: Bill Tozier, Laura Fisher, and of course Chet Hendrickson, who has listened to and shaped every word. Any remaining errors are of course his fault.

The wonderful folks at The Pragmatic Programmers: Andy Hunt and Dave Thomas; Susannah Pfalzer, who knew when to manage me and when to stand back; Janet Furlow, who pushed the book through production; my patient and long-suffering editor, Mike Swaine—without these people there'd be no book.

My "boys," Ron and Mike, of whom I am most proud and who have filled my life with joy and events of interest.

And more than all of these, Ricia, my wife: without her nothing would seem worth doing. Thank you for taking care of me.

Thanks!



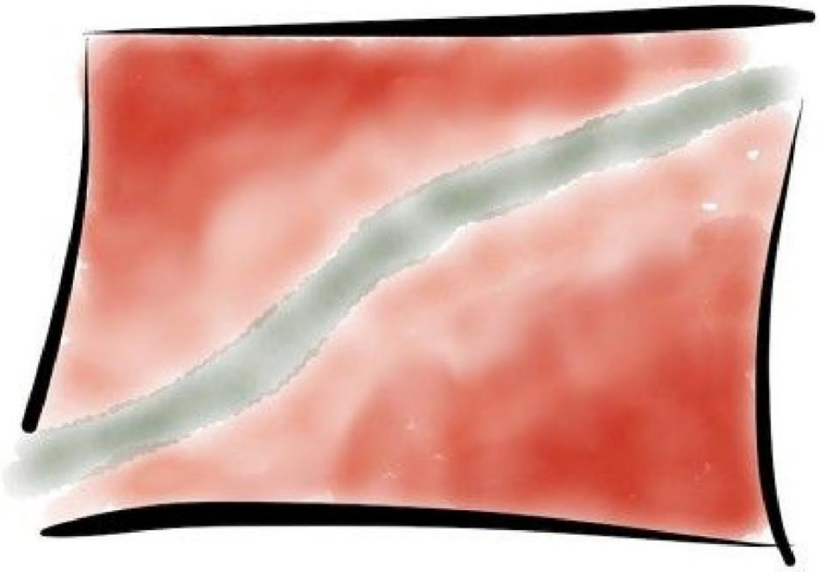
Software is Lava

Introduction

Kids often play a game: *The floor is lava*. In this game, you have to get from one place to another without touching the floor. Because the floor is lava. If you step in lava, you die, horribly, screaming. Don't step in lava. So, in the game, you must jump from the couch to the chair, crawl across the table, and leap to safety in the kitchen, where the floor is not lava.

Software is lava. Often it seems that there's no safe place to step. Worse yet, we're not allowed to jump on the furniture. Mom said. Sorry.

So what are we to do? As we build software, it seems that we're stepping in lava every day. It's complicated, it gets more complicated, and often it seems that we're just doomed.



There has to be a better way.

We all feel it. We're all sure that there must be a way to build software that isn't lava. We didn't get there last time, but next time…*next time*…we'll get it right.

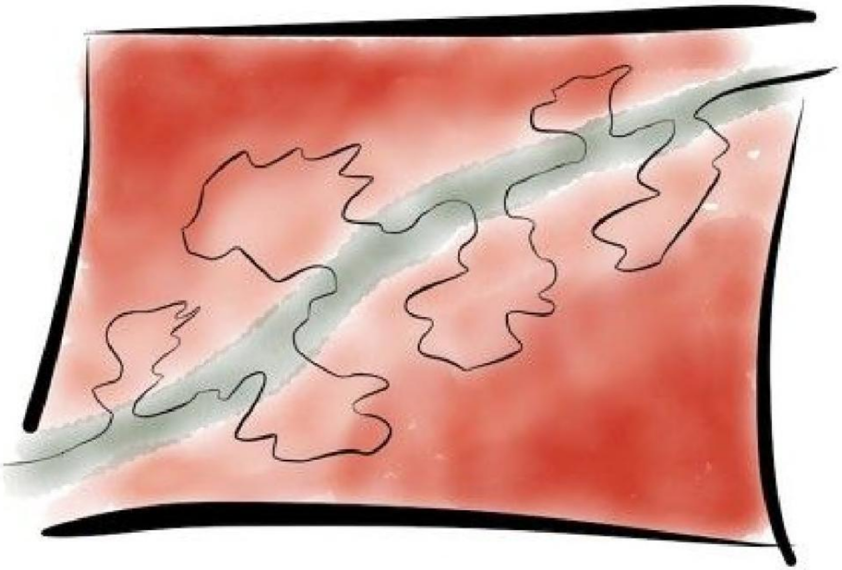
And, sure enough, next time, more lava. Ow! Die screaming.

Yet most of us have had moments when our feet weren't burning.

There seem to be cool, grassy patches amid the lava. Sometimes we find them. It feels so good to be there.

The premise of this book is that there aren’t just patches of grass—there is a cool, green, grassy path. Maybe we can’t be on that path every moment, but understanding the path better is the way to a happier project.

I call that path “the Natural Way,” because I believe that the path is built into a simple notion, a focus on delivering value early and often.

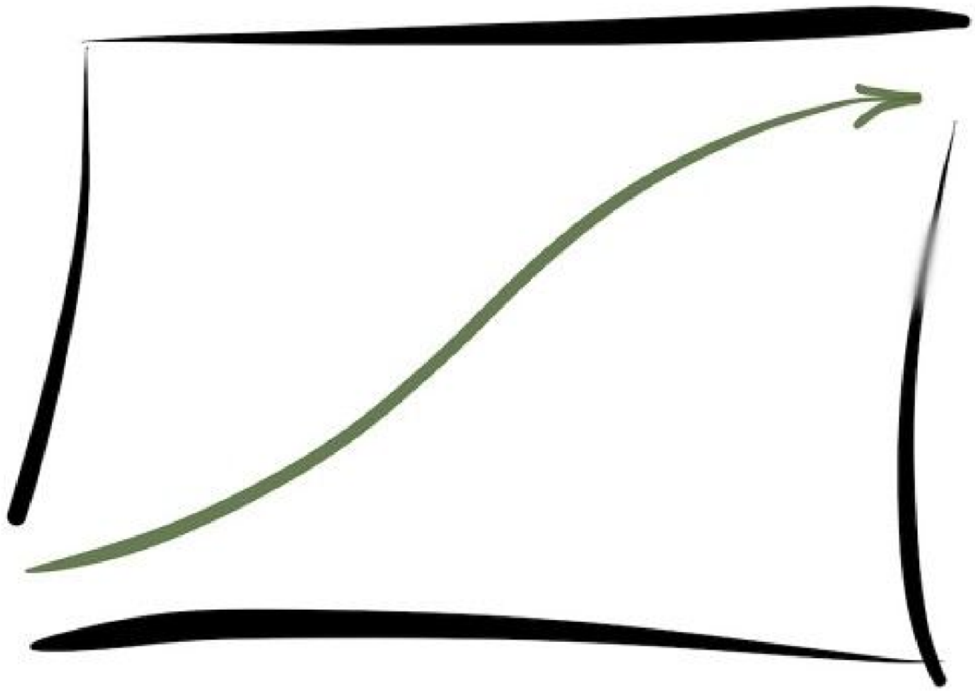


We will wander off the path.

Even though we much prefer to be on the grass than in the lava, it seems that we always get in the lava. (Sometimes lava is spelled differently. Anyway, we're in it.)

If there is a path—and I hope to show you that there is—we will wander off of it. Yes, we will. So as I describe the path to you, don't imagine that I believe we'll all be on the path and live happily ever after with no problems, with our grateful feet caressing the happy grasses of the path. We couldn't be that good, or that lucky.

What we can do is remain aware that there is a path. When we're not on the path, we'll think about value. We'll think about the Natural Way. And quite likely we'll be able to find our way back, if not to the grass, at least to a place where the lava isn't quite so hot.



The Natural Way

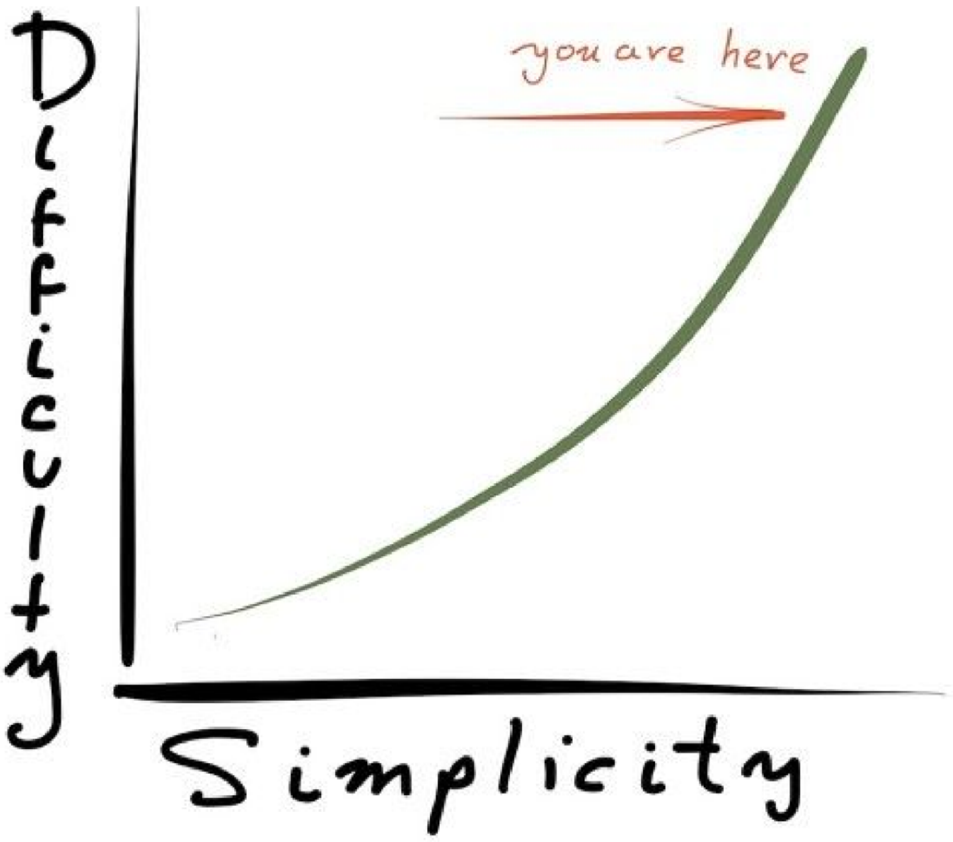
The story in this book is a simple one: there is a *Natural Way* to build software, and it serves everyone well.

The Natural Way serves end users well because it delivers value to them sooner.

The Natural Way serves the business well because it provides a return on investment sooner, because it provides important information quickly, and because it provides the ability to adjust direction as needed.

The Natural Way serves management well too. It lets management see what's really going on inside the project so that when action is needed, there will be time to act. And it reduces management's problems by making information visible so that we don't have to dig for it.

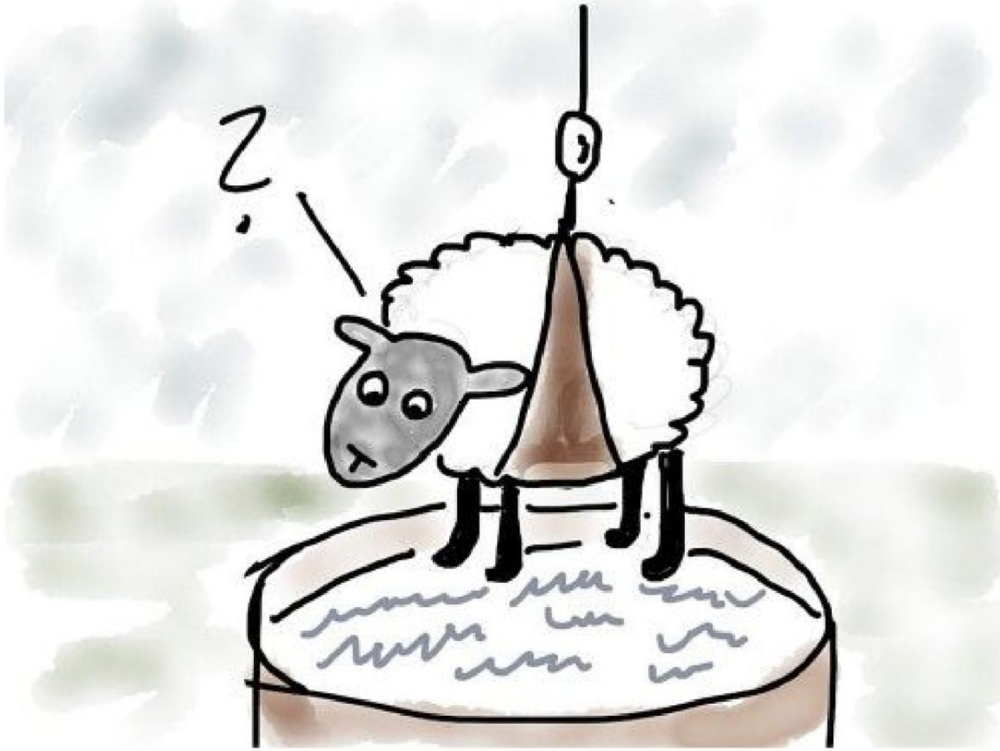
The Natural Way even makes the job easier for developers. It provides them with clear direction and allows them freedom to use their skills to build what the organization needs, when it's needed.



What is described here is simple—but it's not easy. You'll need to think about these ideas, to figure out how they'll be valuable to you, and to learn to do the things we explore here. Keep moving toward simplicity. You'll be glad you did.

The Natural Way does require us to think, to learn, and to change a bit. I think you’ll see here that moving toward the Natural Way need not be traumatic. It can actually be quite a bit of fun.

Come along with me, and explore how we can make software development simpler by focusing on frequent delivery of visible value. We’ll not talk about how things are, but how they might be, if we try.



A final warning before you jump in:

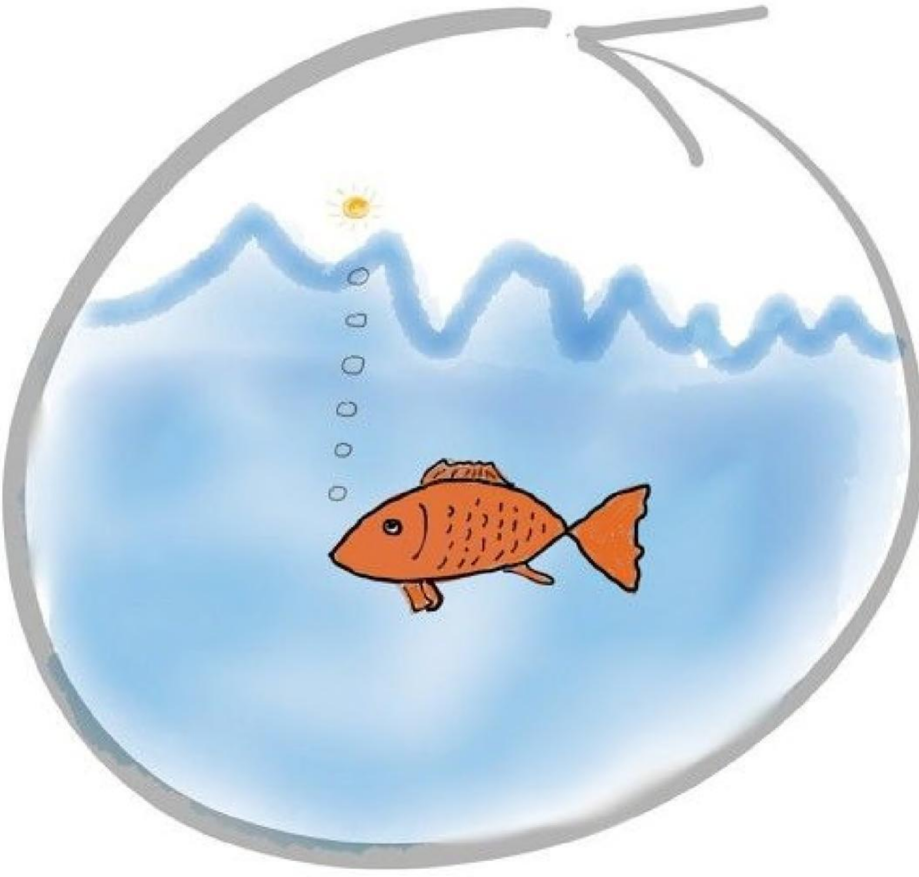
Channeling comedian Eddie Izzard's NSFW "Death Star Canteen" bit:

> This is not a book of what the heck to do!

It’s not a book of recipes. It’s not about one way to do something. That’s not our purpose here. We’re here to think about how things work, to ready ourselves for whatever may happen. There are many ways to accomplish what you need. I trust you to find ways, think of ways, and select among them.

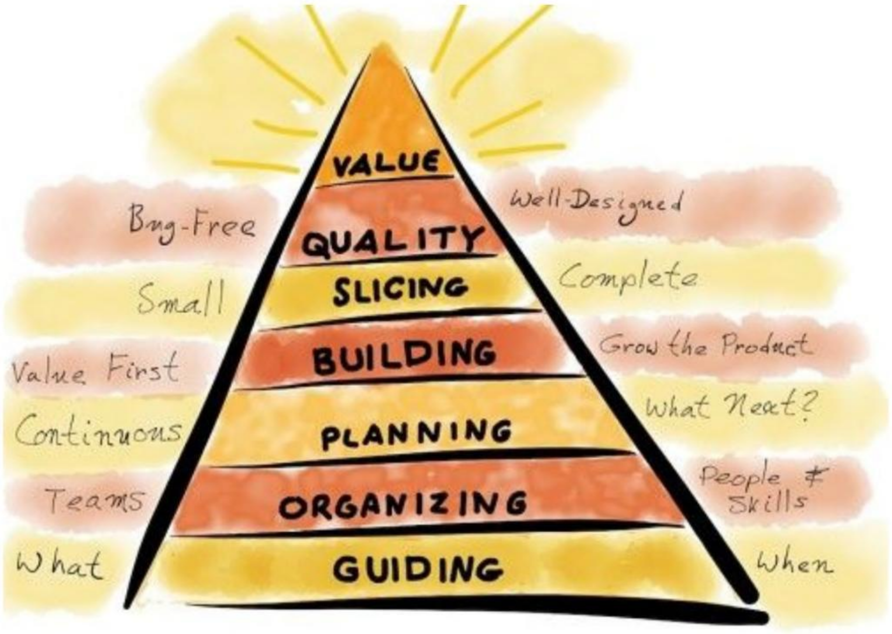
Part 1
The Circle of Value

Sometimes you just have to stop holding on with both hands, both feet, and your tail, to get someplace better. Of course you might plummet to the earth and die, but probably not: you were made for this.



VALUE

Successful software development is hard. It will always be hard. However, doing it smoothly and gracefully has a very *real simplicity. Let's talk about that essential simplicity. As we do, your job is to think a lot, while I write very little.*


VALUE
Bug-Free
QUALITY
Well-Designed
Small
SLICING
Complete
Value First
BUILDING
Grow the Product
Continuous
PLANNING
What Next?
Teams
ORGANIZING
People & Skills
What
GUIDING
When

Chapter 1

The Search for Value

This picture shows the flow of our argument here. Our story begins with value, and value is the point of our work:

Value. Value, we'll see, is "what you want." It can be any kind of value, from money to laughs or lives. We'll explore a bit about what value is.

We'll tell the story by building up from the bottom of the pyramid, describing how to guide, organize, plan, and build our product, in small slices, with a focus on quality. The value we produce is based on these.

Guiding. We produce value by creating teams with responsibility for creating value. We make sure they understand what is needed, and understand the time available. We guide them by observing what they actually build.

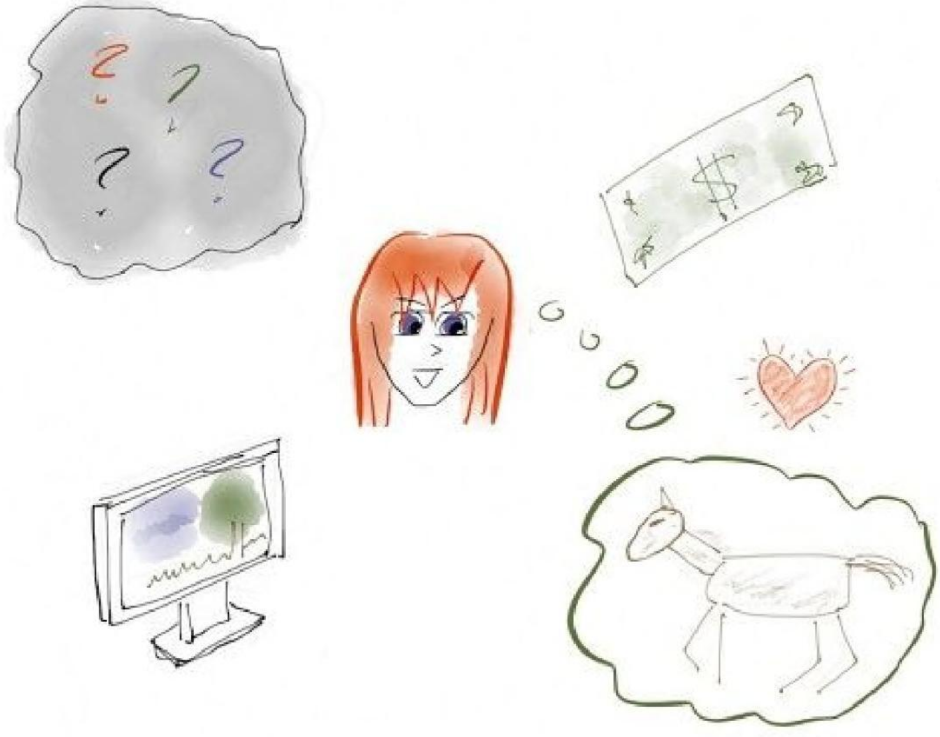
Organizing. We organize teams with the ability to get the job done. We organize around features, because features give us the ability to plan and build value most rapidly. We apply good people and help them build their skills.

Planning. We steer our projects by selecting the features we need, in the order we need them. We produce value in a timely fashion.

Building. We build up our product feature by feature. This provides frequent delivery of value. We can see how things are progressing early and often.

Slicing. We slice features down to the smallest possible value-bearing size. We build a capable product as early as possible, and then enhance and grow it as the deadline approaches. We're always ready to ship.

Quality. We apply the necessary practices to ensure that our product always has a good design and that it is as nearly defect-free as possible. We're able to build value continuously, sustainably, indefinitely.



What is software value?

Chapter 2

Value Is What We Want

We all want value. Value is what we want. Value is—*what we want*. In software, we generally get value by delivering features. Features that have value. Features that we want.

Often it's about money, because software can save time or money. Software can help us earn money. There are other kinds of value: software can make lives more convenient. Software can even save lives.

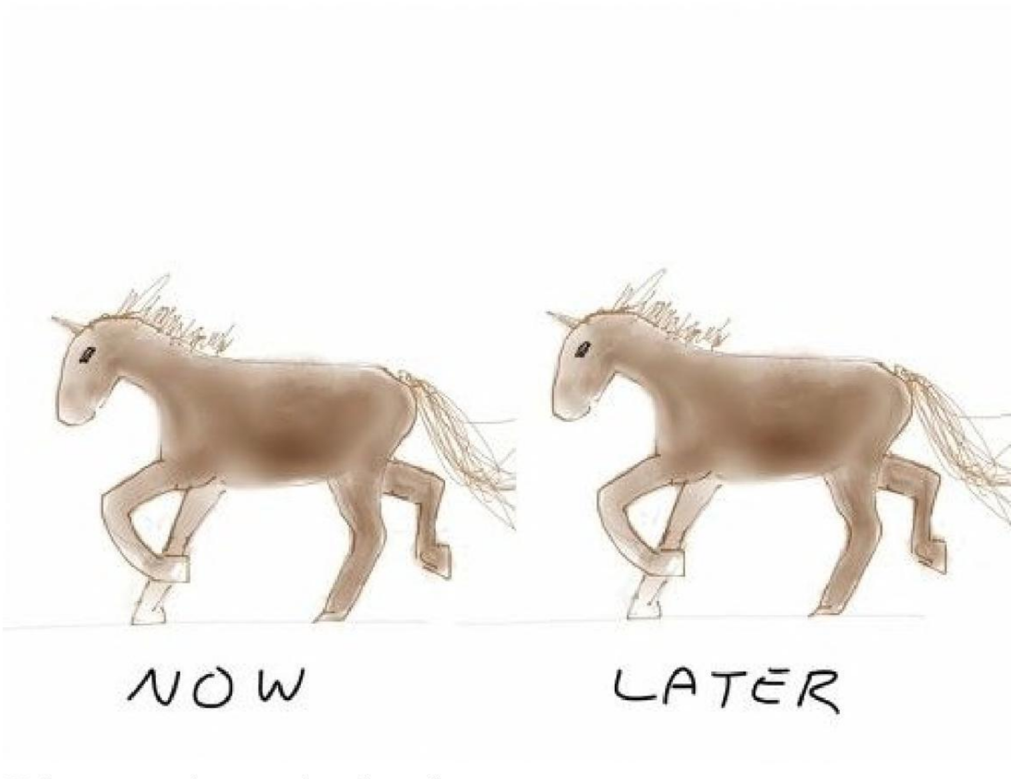
In the end, I think of value as simply *what we want*. We might like to put a number on value, but it's not necessary. As we build the software, we'll make choices. Each choice gives us something we value. We'll choose information, happy users, or saved lives. We'll choose what makes sense. We'll choose *what we want*.

Working incrementally, we'll choose the next thing we want. We'll have our team put it into the software, as quickly and solidly as they can. When they're done, we'll look to be sure we got what we want:

we'll check for the value.

We'll say, "show us the software," to see the value.

What kinds of value does your project deliver to its users? To your organization? To the team? What value does it deliver to you?

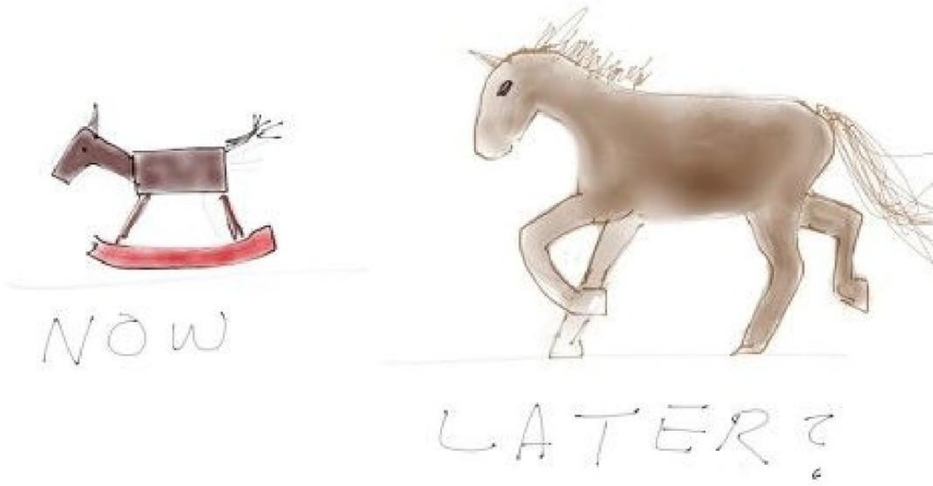


Value starts when we ship the software.

A project delivers value only when we ship the software and put it to use. If we wait until we finish everything, it will be a long time before we get any value. Let’s find a way to deliver value early.

We’d rather have the pony now, not later, but we can’t create everything right now. We have many features in mind, and they’ll take time to build. The more we want, the longer it takes.

What benefits could there be if we could deliver sooner? How might the organization benefit? What about the team? What about you and me?



What if we shipped some valuable part sooner than the rest?

Every product is made up of pieces. Call them features, or minimum marketable features. Call them aspects, functions, or capabilities. Each big piece has smaller pieces. Each is full of details that make that piece more complete, more useful, or just nicer.

Remember, most users of a product don't use every feature. There's some kind of 80/20 rule going on. Everyone may want something different, but no one wants everything. Even in the products you know best and use most, you probably use only a fraction of the features.

Does shipping something small make sense?

Since most users don't use all the features, a smaller set of features can provide real value, and provide it sooner. Sometimes we think we have to have it all. Let’s face it, though: if you’ve done very many software projects, you probably didn’t get everything you wanted by the date you wanted it. We never get it all.

We can stamp our feet and demand a pony, or we can act like managers and steer our software projects to the best possible result. Very likely, there’s a subset of capability that can start providing value sooner than the whole package. Let’s find those features and ship them first. That way, we’ll prosper.

After that first release, we may need to follow up with the rest of the product; otherwise, the final product may be worth less over its lifetime. So we’ll usually plan multiple releases.

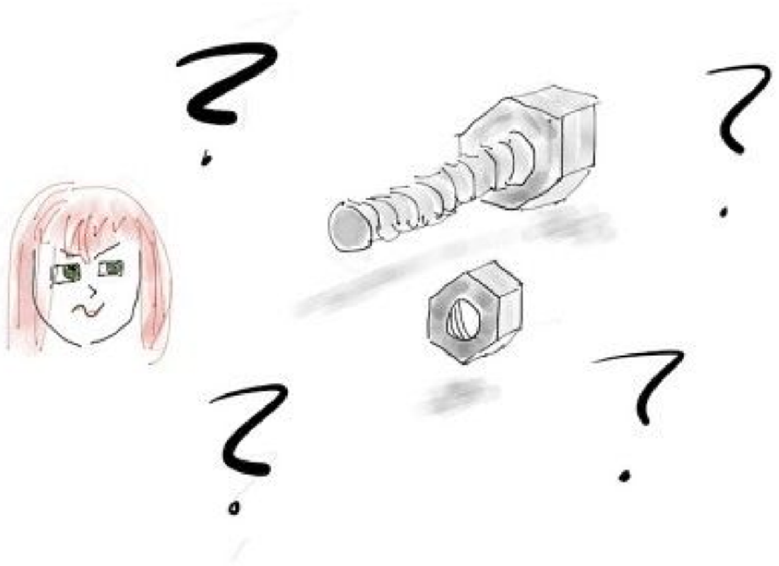
But there are times when we might just ship the first bit and then stop. When could that be the best thing to do? How many different reasons can you think of?

Deliver just one part and then stop?

If we ship just once, we'll get an earlier return, but it will probably be less than if we shipped the whole product, even later on. Or will it?

Information has value as well. Sometimes the most important information we can get is that we're doing the wrong thing. One very good way to find out if we're going in the right direction is to ship a small version of the product early. If it flops, we can change direction at low cost.

Usually, though, we do have a good idea. With a good idea, what kind of pieces should we work on? How would our product best be delivered a bit at a time? What should the pieces look like?



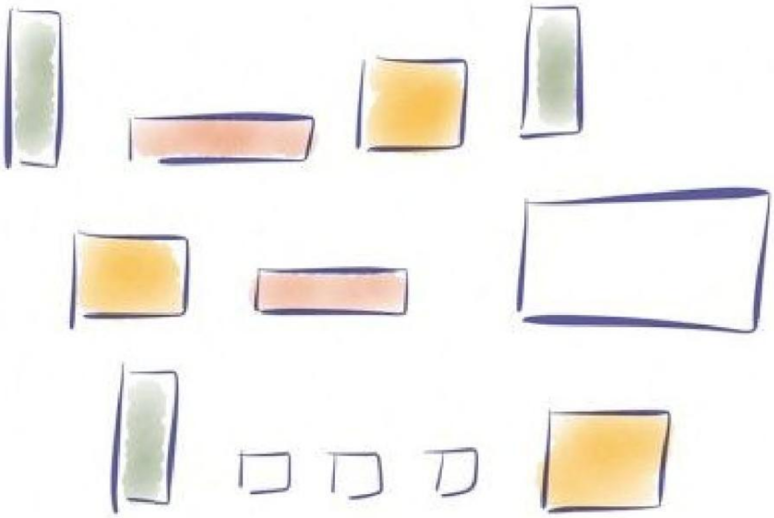
We must see and understand the pieces.

It’s not enough for our teams to work on mysterious technical bits that make sense only to them.

We need to guide our teams to build pieces that make sense to us, and to our users. These are often called *minimal marketable features (MMFs)*. In fact, we’ll often benefit from providing business direction at an even finer grain than the usual MMF.

Here, we’ll call those pieces *features*. When we say “show us the software,” we want to see features that we want and understand.

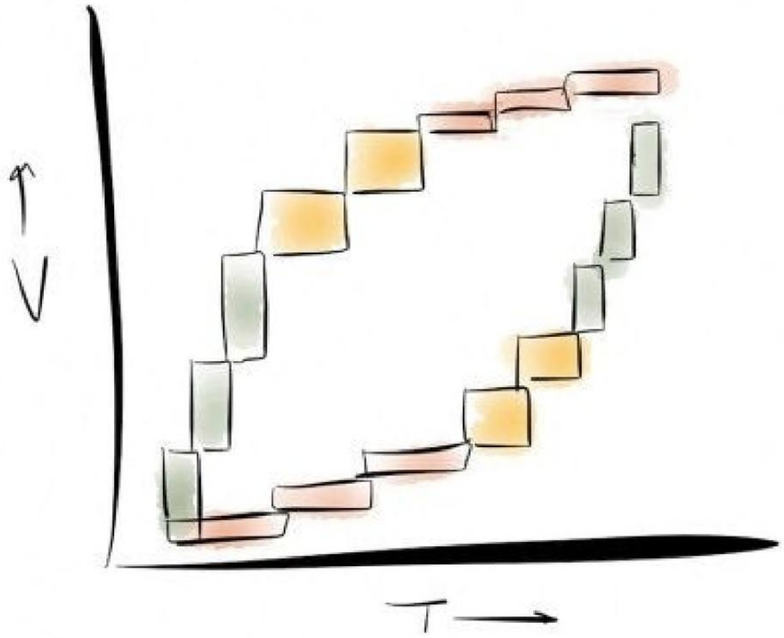
Looking back at some previous projects, what are some features you wish you could have shipped sooner, and why? What are some features that should have been different? Are there some that shouldn’t have been done at all?



Value, by feature

Each feature that we might build adds some value to the product. And each one takes some amount of time. We can't know exactly how valuable or exactly how much time. But we can still get an excellent sense of what to do.

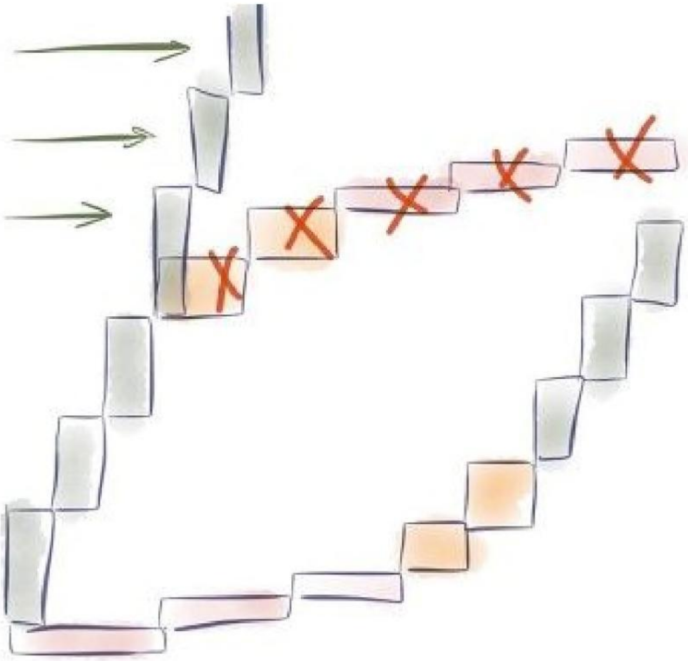
Suppose the height of the features is their value, and the width is their cost. Which ones should we build first, and which ones should we defer until later? Pretty clear, isn't it?



Value growth depends on what we choose to do.

Look at the difference in the growth of value if we choose the higher-value, inexpensive features first and defer lower-value, costly features until later. And these features only vary by about a factor of three to one. In most products the best ideas are tens of times better than the worst—or more. The results would hardly fit on the page!

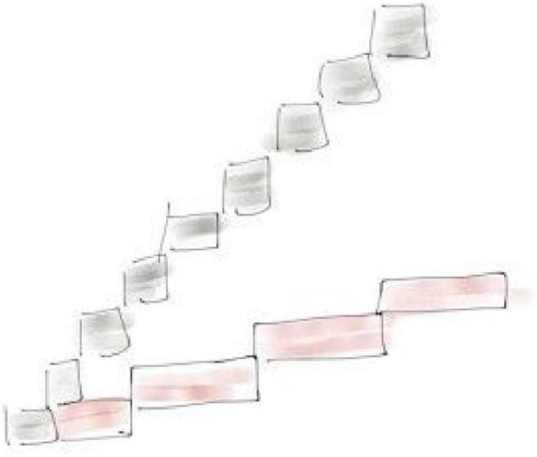
Some of those later features look pretty boring. What would happen if we did different, more valuable features, even for some other product?



We might even switch our investment to a new product!

When we begin to ship frequently, with highest value first, the time soon comes when the next features aren't worth the time and money to create them. This is a good thing. We can often do far better by investing in a new product.

What's the next product we'd like to do? Who might feel negatively impacted by a product shift? How might we make that shift a good thing for everyone? Can we focus on a portfolio rather than separate products with diminishing returns? Can we show more software, with more value?



Best value comes from small, value-focused features, delivered frequently.

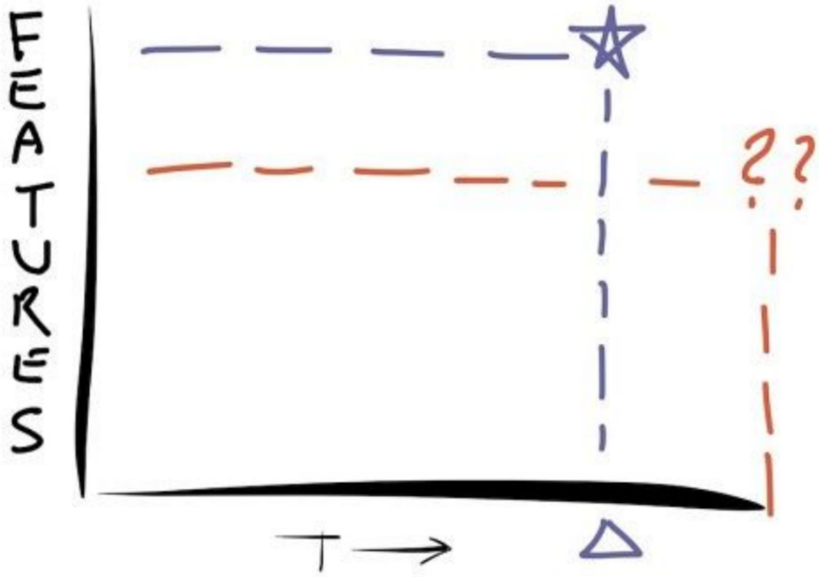
OK, we can see that small features could deliver value sooner if we can do them. Let's think next about managing our project. Will smaller visible results help us manage? How might they get in the way?

What about our teams? Are they organized to work this way? Do they have the people they need, the skills they need, and the help they need? Read on—we'll talk about all those things.

The main thing to remember is that we get the best results from delivering the software, feature by feature.

Further reading:

- Chapter 10, *Value—What Is It?*
- Chapter 11, *Value—How Can We Measure It?*


FEATURES
T →
??

Chapter 3

Guiding Goes Better “Feature by Feature”

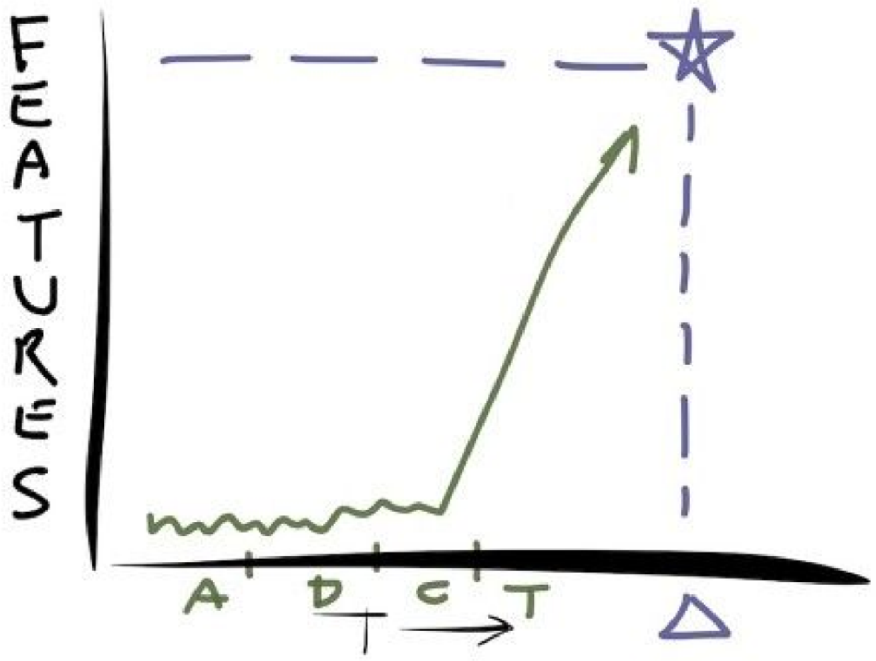
The first thing we know about any project is the deadline—at least it always seems that way. That’s the vertical blue line with the triangle at the bottom.

And what do we want by the deadline? Why, everything, of course. That’s the horizontal line. The star is our plan: have everything by the deadline. No problem!

Somehow it doesn’t turn out that way. We usually wind up shipping less, or later, or both: the red lines with the question marks. Heck, we’re sure to get less than we want. After all, we asked for *everything!*

We really can’t have it all. Let’s manage that reality, not just let things happen. Let’s steer our project, not just ride it wherever it takes us.

In a recent project of yours, what important things didn't get done? What got done that turned out to be wasted? What did you find out about too late, or nearly so?

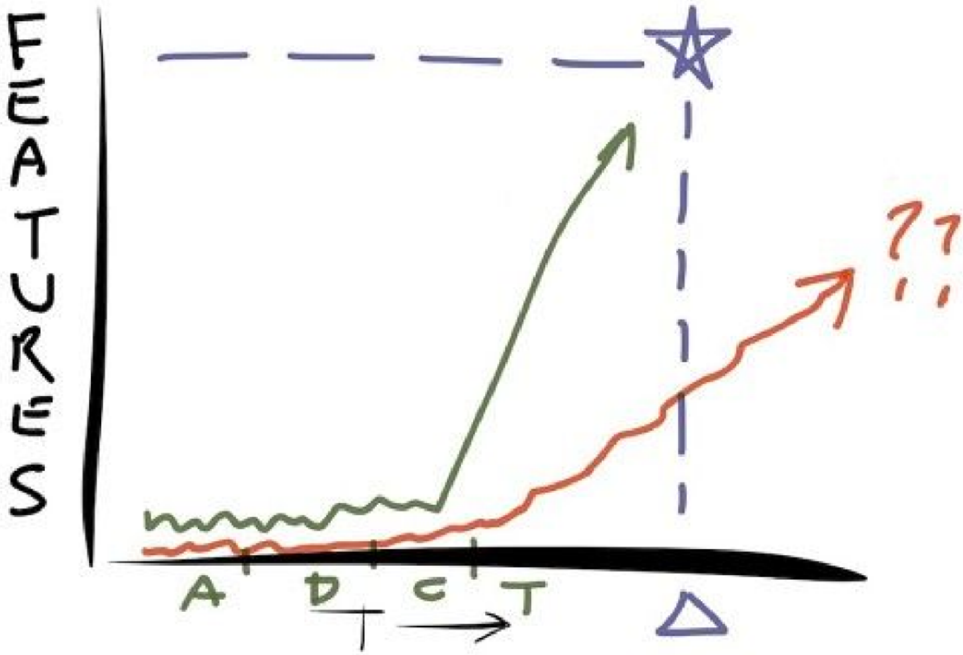


Conventional software projects proceed in phases.

Many projects plan with *activity-based* phases: Analysis, Design, Coding, and finally Testing. The green line is our plan for such a project, and it may look good. But even if we get Analysis done on time, that doesn't tell us how well we'll do on Design or Coding.

Until we begin to see the software, we can't really tell how well we're doing. And when we start getting and testing that code, what happens? Generally nothing good!

Have any of your projects given you too little time to react when trouble arose? Would there be value to knowing sooner what's really going on? Did you ever wish you could get at least some value out of all that effort?



Worse yet, things rarely go according to plan.

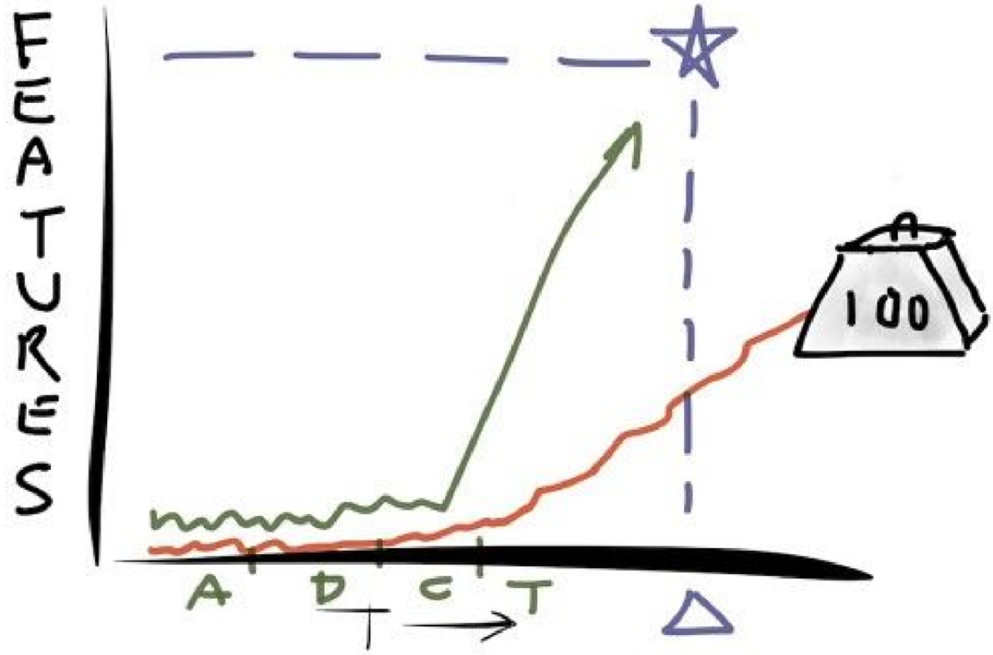
Finally, we begin to see and test the code. And the facts aren't good. Inevitably we're later than we thought. We have less done than we thought. What we have done doesn't work very well.

We knew we had asked for more than we could do: that's the nature of goal setting. But by the time we find out where we are, it's too late to do much about it.

With more warning, maybe we could have shipped a subset on time. Now we have few choices. We could write the project off, but that would be career suicide. Or we can trudge gamely on, hoping to ship something before they give up on us.

Either way, we look bad. Either way, it *is* bad!

Have you ever had to ship in bad condition? Were there too many defects still in the software? Was the software too hard to change? Were important features missing? Important new ideas that it was too late to add?



The activity-based product is a monolith.

With a monolithic project, late in the game we can't do much to cut costs. We have already written requirements for things we'll never get. We've designed and even written code for things that we'll never complete. All that work is wasted.

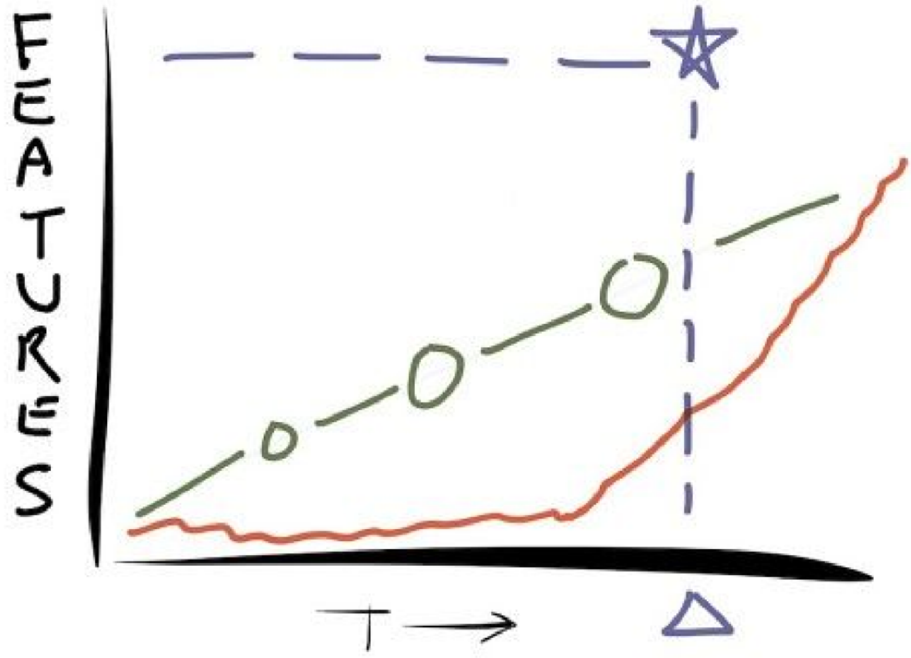
If only we had known the truth. We could have deferred some of that work.

We laid out this project with an all-or-nothing mentality. We analyzed it all. We designed it all. We tried to code it all. We discovered, too late, that we can’t have it all.

Trying to plan and build it all has hurt us. We have no time to change, and even if we had time, we’d never untangle all the things we shouldn’t have done from the things we should have.

Instead, let’s plan for multiple releases from the very beginning. Multiple releases are easier to manage and deliver value sooner. It’s even easier to build the software that way. Everyone wins.

Do those plan, analyze, design-code-test phases really help you manage your project? Wouldn’t it be easier to manage things if you could just get the features, a few at a time, in the order you wanted them, starting right at the beginning? Let’s look at that.



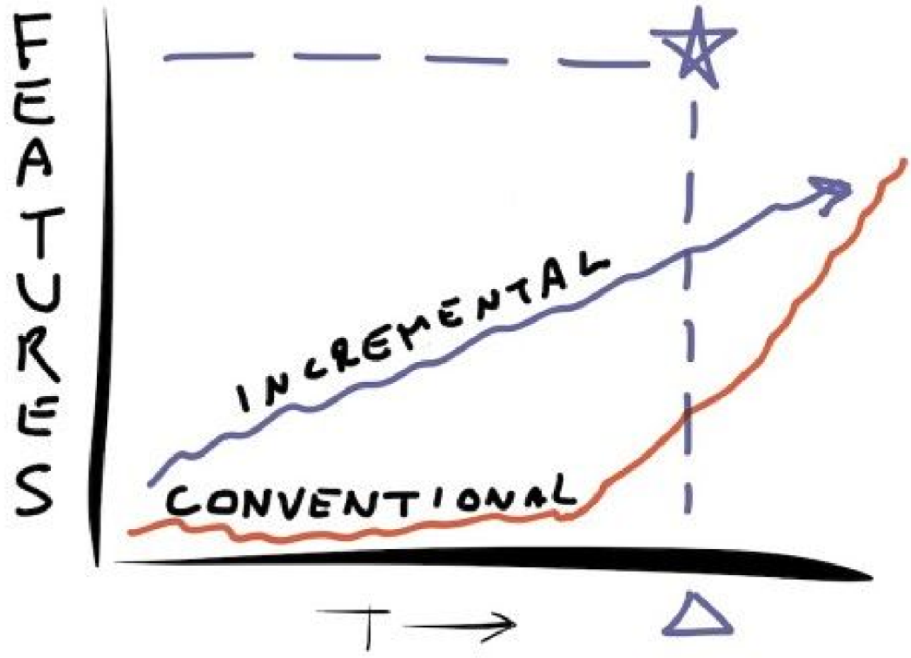
A project that delivers feature by feature is more predictable.

We've seen that delivering release by release, feature by feature, lets us ship value sooner. What about our ability to manage and guide the effort?

Our old red conventional project drones on and on, delivering too little information, too late. But the green project shows us real, valuable features at frequent intervals. We can see what is happening. We can see the software!

Can you see how a flow of visible features would be easier to manage? Can you see how you could maximize project value as you go?

What about risk? Can you see how to evaluate or reduce a project risk by building something visible? Can you see how to deal with a marketing risk with a small test feature?



Feature by feature gives better information, better guidance, better results.

When we build our software projects feature by feature, things go better. We can see how much is done and how rapidly the project is progressing. We get a good sense of how much will be done by any given date.

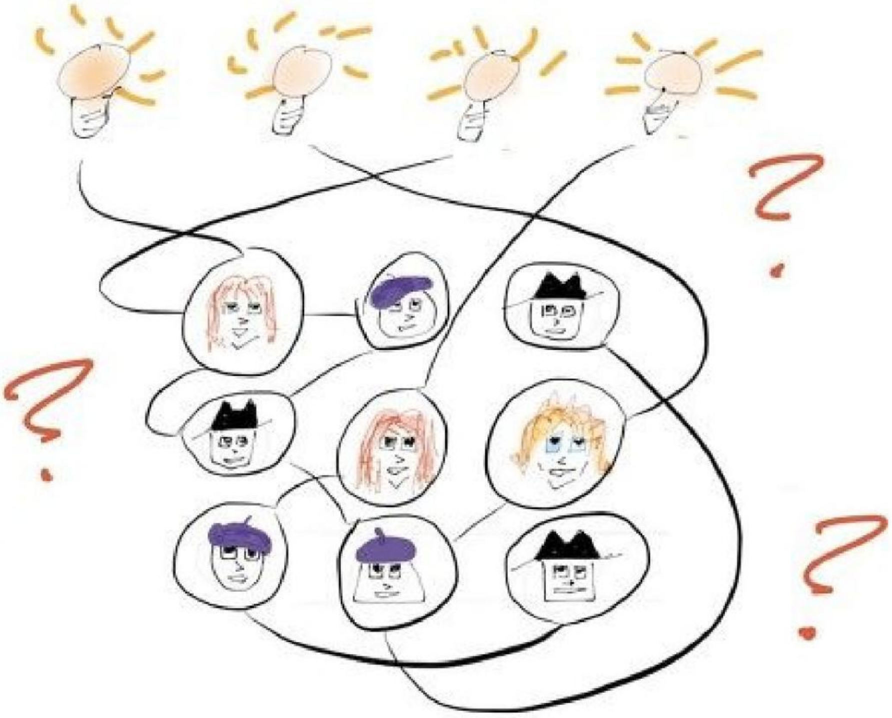
We choose the most important features to do next. We build the best possible combination of features for any desired shipment date—even one earlier than our original desired date. We can even change features, adding new ones in response to better ideas or changing user needs.

When our projects grow feature by feature, we can respond to what's really happening. We can respond to the changing needs and inputs of the business and of management.

What would it take to make this way of working possible? How can we plan a project when we don't even know what we'll wind up wanting to do?

Further reading:

- Chapter 14, *Creating Teams That Thrive*
- Chapter 16, *Managing Natural Software Development*



Building features requires multiple skills.

Chapter 4

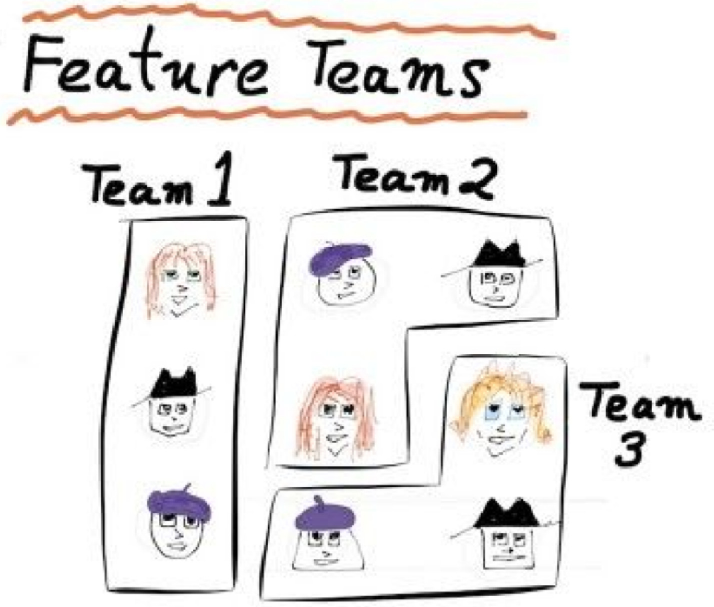
Organizing by Feature

We want to get value in small bites: features. We prosper when we manage in terms of value, in terms of features.

How can we organize our work, and ourselves, for the best and most rapid flow of value?

To get the work done, different parts require different skills. The work won't be done—or at least not done well—until it has had the attention of people with each needed skill.

If we organize teams by skill-set, each piece of work will need to be passed around among teams. Each handoff will require scheduling and cause delays. Quite likely, problems will arise from each handoff.

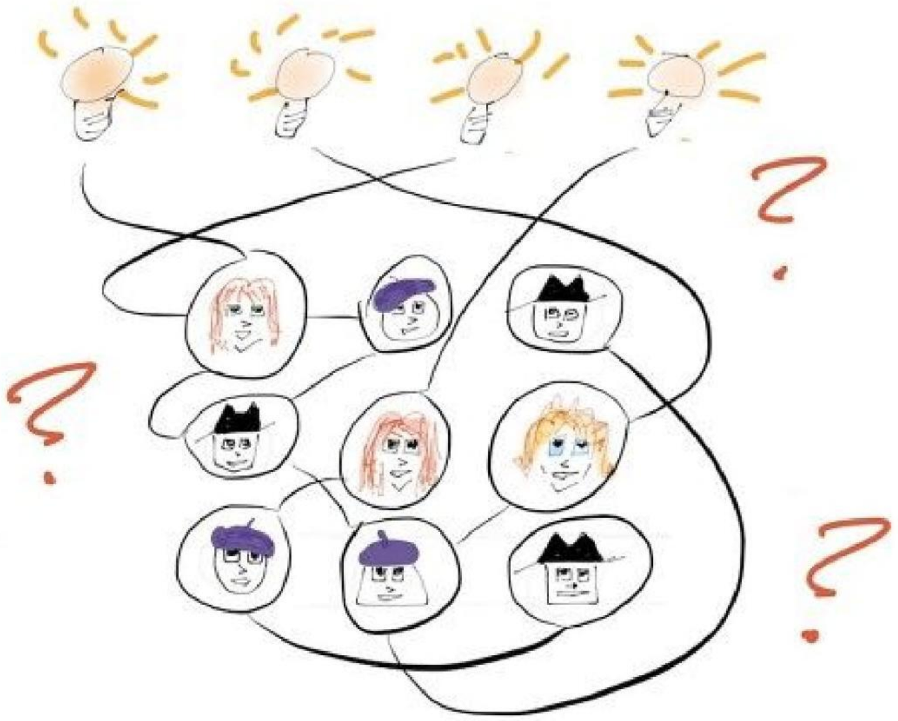


Teams build features.

The answer is simple: organize into small teams, each of which builds features that the Product Champions can understand. Make sure that each team has all the people and all the skills necessary to build the entire feature, not just part of it.

The advantages of this should be clear: we can allocate work across teams easily. We can see where everything is. Each feature gets dedicated attention. Responsibility and authority are aligned.

It’s simple and works well. But it’s not that easy, is it?



But…but…but we’re not organized that way.

I know. And I’m here to say that you probably should be.

If each feature that you want must be passed through multiple teams, it takes longer and results in lower quality. Why? Because the teams need to be coordinated somehow, and each item has to be passed from one team to the next. After it's passed, it needs to sit in the queue for the next team to wait its turn. And often—very often—the feature needs to go back to the first team to fix something they didn't understand. Often it will go back and forth several times.

This slows you down.

Yes, feature teams may be a change for you. But if you want to go rapidly and smoothly, you'll very likely benefit from moving in that direction. Take your time, don't panic, but give it a try. Create a Feature Team, a darn good one. See whether fewer handoffs speed delivery and improve quality. I'm betting things will go better. If so, rinse, repeat.