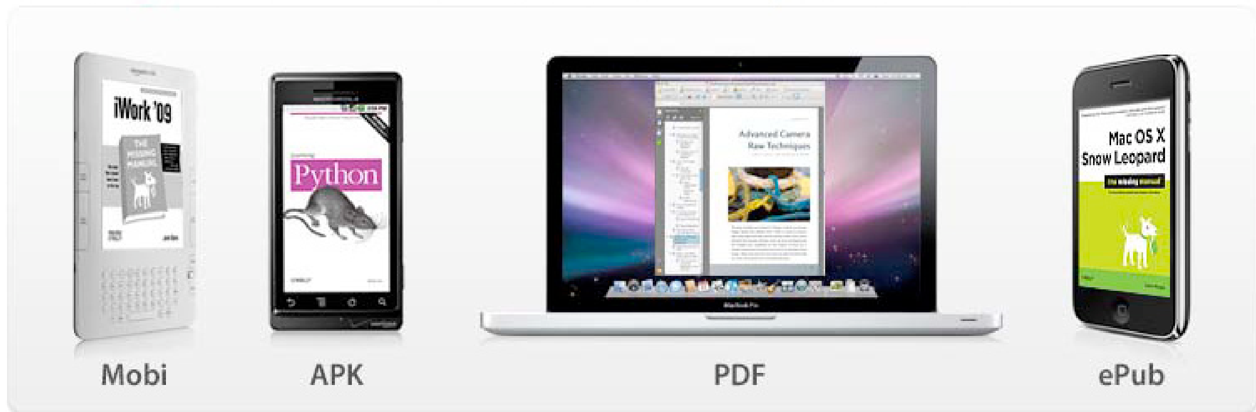
*The Art of Building Great User Experience in Software*

# Effective UI

*Jonathan Anderson,*
*John McRee, Robb Wilson*
*& the EffectiveUI Team*

O'REILLY®

## O'Reilly Ebooks—Your bookshelf on your devices!




When you buy an ebook through oreilly.com, you get lifetime access to the book, and whenever possible we provide it to you in four, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and Android .apk ebook—that you can use on the devices of your choice. Our ebook files are fully searchable and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

**Learn more at http://oreilly.com/ebooks/**

You can also purchase O'Reilly ebooks through iTunes, the Android Marketplace, and Amazon.com.

# Effective UI

*Jonathan Anderson, John McRee, Robb Wilson,*
*and the EffectiveUI Team*

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

**Effective UI**
by Jonathan Anderson, John McRee, Robb Wilson, and the EffectiveUI Team

Copyright © 2010 EffectiveUI. All rights reserved.
Printed in Canada.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

**Editor:** Steve Weiss
**Development Editor:** Jeff Riley
**Production Editor:** Rachel Monaghan
**Copyeditor:** Genevieve d'Entremont
**Proofreader:** Nancy Kotary
**Indexer:** Julie Hawks
**Cover Designer:** Karen Montgomery
**Illustration and Interior Design:** The EffectiveUI Team

**Printing History:**

February 2010: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Effective UI*, the image of a rainbow lorikeet, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-15478-3
[F]

# Contents


**Preface** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **ix**

**1 Building an Effective UI** . . . . . . . . . . . . . . . . . . . **1**

Understanding UX 4

What Good UX Accomplishes 6

Why Engagement and Good UX Matter 10

The Elements of Engaging UX 11

Redefining Two Fundamental Terms 32

Design 32

Development 34

**2 Building the Case for Better UX** . . . . . . . . **37**

Why Now Is the Moment for UX 40

Motive 40

Means 48

Opportunity 50

Winning Support for Better UX 53

Stakeholders 53

Education 57

Quantifying the Business Value 67

Materializing and Proving the Concept 67

Other Strategies for Building Support 73

**3 Effective Planning and Requirements** . . **75**

Uncertainty and the Unknown 77

The Humility of Unknowing 78

The Weakness of Foresight and Planning 79

Friction in a Complex and Peculiar System 81

Subjectivity and Change 87

Lessons from Uncertainty and the Unknown 89

The Further You Are in the Project, the Wiser You Are 89

Start Development As Soon As Possible 90

Written Functional Requirements and Specifications Are Inherently Flawed 90

Commitments to Scope Are Untenable 92

Relish and Respect the Unexpected 92

Intolerance of Uncertainty Is Intolerable 93

Effective Requirements 94

How Framework Requirements Are Built 97

Reexamining the Three-Legged Stool 99

Commitments You Can Live Up To 101

Effective Process 102

Development Methodology 103

## 4 Bringing Together a Team .............. 113

The Project Leader 116
- Relationship to the Product 116
- Relationship to the Stakeholders 117
- Relationship to the Project Team 119
- Who Should Be the Project Leader 119

The Stakeholders 121
- Securing Authority 121
- Collaboration and Decision Making 124

The Characteristics of a Successful Project Team 125
- Getting Professional Help 127
- Insourcing Versus Outsourcing 130

## 5 Getting the Business Perspective ..... 139

Defining Success 141
- Creating a Project Mission Statement 142
- Determining Project Success Criteria 144
- Exercising Restraint 145
- Applying the Pareto Principle 148
- What Not to Restrain 148
- Refocusing Product Objectives 149
- Omissions Aren't Permanent 150

Describing the Product's Users 151
- User Attributes 152
- Exercises to Identify Key User Attributes 153

Creating Business Requirements 160
- Defining "Requirement" 161
- Exercises to Develop Business Requirements 163

Maintaining Stakeholder Buy-in 169

## 6 Getting to Know the User .............. 171

Valuing User Research 173
- Combating Pressure to Skip User Research 175

Key Concepts in User Research 177
- Empathy 177
- User Goals Versus Product Features and Tasks 178
- Qualitative Versus Quantitative Research Methods 180
- Who Should Be Involved in the Research 182
- Finding Research Participants 184
- Determining the Research Sample Size 185
- Making Recordings 188

Research Through Speaking with Users 190
- User Interviews 190
- Structured Interview Techniques 191

Research Through Direct Observation 193

Analyzing the Research Observations 196
- Discovering Personas 196
- Weaving User Stories 198
- Discovering User Priorities 199

Guerilla User Research 200

Stakeholder Buy-in Through User Research 202

## 7 Initial Product Architecture ........... 205

The Initial Product Architecture Team 208
- Contextual Scenarios 210
- Mapping High-Level Workflows 213
- Sketching Low-Fi Visual Representations of Requirements 215
- Examining Key Features and Interactions 216
- Setting a Style Vision 217
- Developing Nomenclature 221

Technical Architecture 222
- Getting a Lay of the Land 223
- Making Platform and Framework Choices 223
- Understanding Data Requirements 224
- Mapping Interactions with Other Systems 225
- Finding Shortcuts Through Third-Party and Open Source Components 228
- Discovering Business Logic 229

Software Architecture in Big Design Up Front (BDUF) 230

Project Infrastructure Needs 232
- Code Source Control 232
- Graphic Asset Management 233
- Testing Infrastructure and Environments 234

**8 The Iterative Development Process . . 235**

Regarding "Process" 239

Iterations and Feedback 239

- The Scope of Iterations 243
- Prioritizing the Subjects of Iterations 245
- Finishing Iterations with Something Complete 246
- Estimating Iterations 247
- Basic Iterative Process 248
- Mapping Progress and Feedback Across Multiple Cycles 252
- Increasing the Amount of Feedback 254

Iteration in Sub-Ideal Project Approaches 256

- Strict Waterfall Process 257
- Iteration in a Big Design Up Front (BDUF) Process 261

**9 Release and Post-Release . . . . . . . . . . . . 263**

Managing Expectations 265

The Alpha and Beta Releases 266

- Receiving Orderly Feedback 268

Last-Minute Housekeeping 269

User Documentation 270

And Champagne Corks Fly... 271

Adoption 272

Post-Release 273

- Review 274
- Measurement and Tracking 277

**Afterword . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 281**

**Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 287**

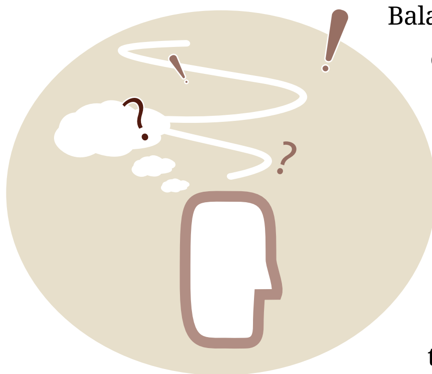
# Chapter 3

# Effective Planning and Requirements

Balancing the realities of how software is built with the need for a sense of security by the business and stakeholders is a challenge that can be met successfully, but it requires a major shift in how managers and stakeholders perceive software development.

**Planning and requirements are** difficult subjects in the world of software development, and are often points of bitter contention between managers and project teams. It seems to be universally recognized that the typical approach to managing software projects hasn't been working particularly well, and projects are perpetually behind schedule and over budget. The development of software—especially innovative, well-designed, user-centered products—simply can't be planned and managed in traditional ways, and failure to recognize this and adapt properly can lead to grinding failures, misconstrued goals, and half-baked products.

Despite the need for unique management approaches, software development projects exist in the context of businesses where strong planning and risk management are critical. Publicized release dates, limited budgets, forces of competition, and other unavoidable pressures push managers to try to get certainty in what will be delivered, when it will be delivered, and what it will cost. That this certainty has never been possible before usually doesn't cause managers to reassess the overall approach. Instead, they tend to resign themselves to the belief that software projects are always dysfunctional, and account for that dysfunction by padding their own estimates and commitments.



Balancing the realities of how software is built with the need for a sense of security by the business and stakeholders is a tremendous challenge. It's a challenge that can be met successfully, but it requires a major shift in how managers and stakeholders perceive software projects. That change in perception is, unfortunately, very difficult and counterintuitive, but it is so critical to succeeding in a software project that we've dedicated the majority of this chapter to trying to drive the point home. These changes in perception and approach, though initially they may be hard fought, can brush aside some of the issues that have historically made building software such a dysfunctional,

brute-force, grinding process. They can improve your company's competency in building software products, dramatically increasing the chance of a successful outcome. In the end, whatever initial challenges arose from changing perceptions is more than made up for in the avoidance of worse difficulties and greater risk of failure.

So, while this book is dedicated to helping you build and apply good practices in your software project, much of this chapter is meant to first break you and your company of bad habits. The deleterious effects of a misguided approach to planning and requirements can easily negate any good practices and hard work. Trying to work with bad planning and requirements is like trying to plant flowers in concrete or ice skate uphill. It's possible, but it'll be very, very hard, and that undue strain will get in the way of doing anything successful or artful.

## Uncertainty and the Unknown

Uncertainty and the unknown are enormous, unavoidable, and fundamental components of every software development project. Being at peace with this reality means you can approach the project in a way that adjusts and flows to account for the unknown. If you fight uncertainty and the unknown—or, even worse, if you suppose they don't exist—it's a path to defeat.

The mistaken belief that uncertainty can be entirely stomped out through upfront planning and everything can be known in advance is the root of many of the worst problems and errors in the management of software projects. This arises from the misapprehension that software development is comparable to and can be managed like other types of large-scale engineering projects—building a bridge across a valley, for example. Bridge building and software development both have components of science and engineering, and of art and craftsmanship. But the role of uncertainty and the unknown, and the way science, art, engineering, and craftsmanship work together throughout the course of the project are very different. Those differences demand a fundamentally different approach to management of the project.

The notion may seem discouraging, but it's much more accurate to compare software development to war than it is to compare it to bridge building. While the battle of software development is fought more with electrons and Mountain Dew than bullets and napalm, the battlefield is a complex, dynamic, unpredictable system of activity residing in shifting political and operational contexts.[1]

## The Humility of Unknowing

> *I am the wisest man alive, for I know one thing, and that is that I know nothing.*
>
> —Socrates

To demonstrate how uncertainty and the unknown are inevitable components of a software development project, we'll examine why the bridge-building analogy fails and the war analogy succeeds. But even with the aid of analogies, it's extremely difficult to explain why uncertainty and the unknown are unavoidable to someone who's never been in the trenches of a software development project. Much of the understanding comes from seeing how design, creativity, and inspiration factor into every aspect of building an application. It also comes from having seen how false certainty, and the demand for it, can cause failure and lead to poorly designed products.

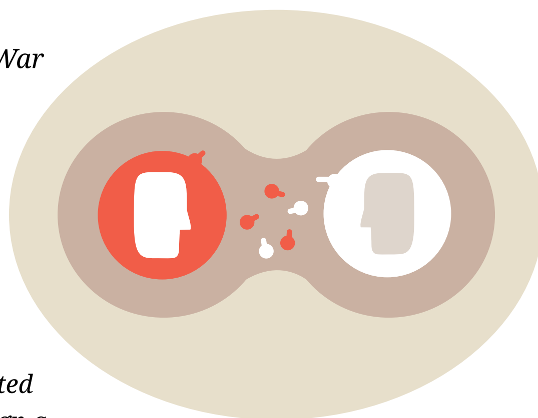
1 In the war of software development, the enemy is failed product design. The enemy is most certainly not the stakeholders and managers, though frustrated project leaders may slip into viewing them as such. Stakeholders and managers are allies, and like all alliances of forces, a certain amount of diplomacy is necessary to ensure that the allies are all pursuing the same goals and working in concert.

It's difficult to explain or prove this fact except to state it this way for now: *you understand your project far less than you think you do*.

And so do your stakeholders, by the way. For your project to be successful, you need to cultivate in yourself and in your stakeholders a certain humility and a recognition that, for as much as you know, you know very little, and that the essence of the project is to investigate and solve a complex problem and not simply to implement a known solution. Embracing this humility of unknowing isn't a resignation to defeat or admission of weakness, but rather is a state of wisdom required to allow you to succeed.

## The Weakness of Foresight and Planning

> *The great uncertainty of all data in war is a peculiar difficulty, because all action must, to a certain extent, be planned in a mere twilight, which in addition not infrequently—like the effect of a fog or moonshine—gives to things exaggerated dimensions and unnatural appearance.*
>
> —Carl von Clausewitz, *On War*



Everything required to design a bridge to a valley is knowable in advance and can be planned to an extremely high level of accuracy before construction begins. All of the important goals, variables, and constraints can be accurately obtained before design begins.

> *Remember that, as we discussed in Chapter 1, design isn't limited to visual and artistic design. Just as an engineer is said to design a bridge or an airplane, a general can design a solution to a battlefield situation, software engineers can design a technical solution, and UX professionals can design interactions and workflows. Design is the application of thought and creativity toward the solution of some challenge or problem, and does not require that the output be of a visual or artistic nature.*

Once those key considerations have been discovered, the design of the project begins and can be entirely completed before construction starts. With accurate and complete designs in hand, construction is then all about ensuring the pieces all come together as designed. Construction is not concerned with any remaining questions about the design and isn't burdened by the risk that the design will change during the course of construction.

By contrast, a general preparing for battle can estimate the strength and disposition of his forces, the resources and capabilities available to him, the attitudes and aptitudes of his commanders in the field, the lay of the battlefield, the strategic goals of the battle, the state of the enemy's forces, and the parameters for success. He also has history and personal experience to help him intuit how events will unfold. Based on this knowledge, he can formulate a plan for the battle.

But this plan, no matter how carefully devised, is inherently incomplete and imprecise. It is wholly premised on estimates of the conditions before the battle and entirely ignorant of the unforeseen conditions that arise during the battle. These unforeseen conditions are based as much on the vagaries of weather, emotion, chance, and uncertainty as they are on even the best-laid plan. This reality is the basis for the famous quote:

> *No battle plan survives first contact with the enemy.*
>
> —Helmuth von Moltke

The same is true of software development. No matter how well you think you understand the domain and no matter how earnestly you've thought through the requirements, there is still great uncertainty in the original facts and premises and a vast depth of the unknown still awaiting you. As with battle, the outcome will be determined at least as much by what comes during the course of the project as by what comes before it.

Not all unknowns are bad, by the way; it's in solving the unforeseen problems that great design and inspiration can take place. Some unknowns may be revelations about your customers and users that fundamentally change how your business interacts with them, or they may be undiscovered opportunities for progress, innovation, efficiency, and improvements to your company's bottom line.

A major reason why uncertainty is unavoidable is that software development, unlike bridge building, requires most of the design to happen at the same time as construction. Construction in the bridge-building business is the application of craftsmanship against the realization of the design plans made prior to construction. Construction in software development is everything from UX design to software engineering to quality assurance.[2] No amount of upfront planning can keep design from being an essential component of the development process. Since design is the process by which problems are identified and solved, it follows that if design can't be completed before development begins, many of the problems and solutions have yet to be identified and cannot be accounted for in any early project plan.

None of this should be taken as an argument for not doing any planning at all. The value and role of planning is still strong, but it should be approached and used differently in light of an understanding of its inherent weaknesses and realistic value.

## Friction in a Complex and Peculiar System

> *Everything is very simple in war, but the simplest thing is difficult. These difficulties accumulate and produce a friction, which no man can imagine exactly who has not seen war.... So in war, through the influence of an infinity of petty circumstances, which cannot properly be described on paper, things disappoint us, and we fall short of the mark. A powerful iron will overcomes this friction, it crushes the obstacles, but certainly the machine along with them.... Friction is the only conception which, in a general way, corresponds to that which distinguishes real war from war on paper. The military machine, the army and all belonging to it, is in fact simple; and appears, on this account, easy to manage. But let us reflect that no part of it is in one piece, that it is composed entirely of individuals, each of which keeps up its own friction in all directions.... This enormous friction, which is not concentrated, as in mechanics, at a few points, is therefore everywhere brought into contact with chance, and thus facts take place upon which it was impossible to calculate, their chief origin being chance.*
>
> —Carl von Clausewitz, *On War*

2 The word "development" is often used to refer to what software engineers do in coding an application, but we'll be using it in the more general sense that constitutes everything that goes into bringing a project to life, which includes design and user research.

The job of the bridge designer is to build a fixed system that can span a certain distance and withstand a variety of forces variably acting on the structure. A bridge, one hopes, is a fixed and solid object. It is composed of bits of metal welded to other bits of metal, cables attached to anchorages, arrangements of trusses, and so on. Though the bridge is a system of individual pieces, it is a simple, static system because once those pieces are properly assembled, they can be viewed reliably as a whole and each piece interacts only with those pieces it is in contact with. When testing a bridge design against external forces, the engineer first tests each piece, then each connection, then each structure formed by each connection, then each larger structure formed by the connection of smaller ones, and so on, until she can test the bridge as a whole system. If the individual component tests are entirely reliable, the whole system tests are also reliable without needing to reexamine the component level.

In addition, the process of building the bridge is a strongly centrally organized system. Although there is great complexity to how the pieces come together and systemic ripple effects can be caused by a breakdown in one part of the construction process, the entire system is perpetually reorganizing itself to the same static, central goal: building the bridge explicitly defined in the designs.

Software systems and the development of them are, on the other hand, complex systems. Specifically, they're Complex Adaptive Systems (CAS):

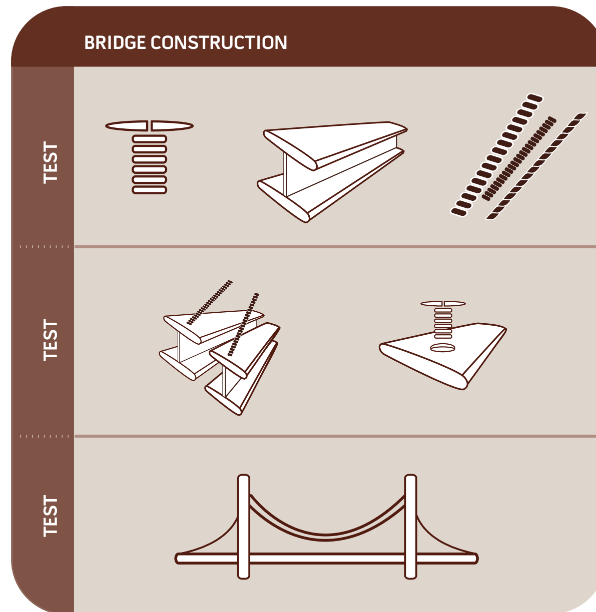
> *A Complex Adaptive System (CAS) is a dynamic network of many agents...acting in parallel, constantly acting and reacting to what the other agents are doing. The control of a CAS tends to be highly dispersed and decentralized. If there is to be any coherent behavior in the system, it has to arise from competition and cooperation among the agents themselves. The overall behavior of the system is the result of a huge number of decisions made every moment by many individual agents.*[3]

Understanding why this is the case in software helps to further the understanding of the role of uncertainty and the unknown in software development.

3 John H. Holland, *Complexity: The Emerging Science at the Edge of Order and Chaos* (Penguin Books, 1994).

## The CAS of software

In object-oriented programming (OOP), every element of code that goes into a product—every class, component, library, data connection, and so on—is a discrete "agent." That's what makes OOP an effective approach to software development: it allows a complex software system to be built out of individual, smaller, comprehensible pieces with their own instructions and behaviors. Because each of these pieces—these agents, like individual soldiers on the battlefield—acts and reacts according to its own situation and instructions, and according to the state of the system, the result is a complex system that's far greater than the sum of its parts. Whereas a bridge is the sum of its parts—the pieces of metal and welds and everything else all add up to a single, static bridge—a software system is the behavior created by the dynamic interaction of its parts. The complexity further multiplies when you consider that the human user is an agent in the system, and the system must accommodate a wide diversity of users who don't behave in predictable ways. Unlike with a bridge—where, no matter what variable forces are in play, it's always the same bridge—a single software system can produce a near-infinite number of possible different behaviors and experiences.




*Software doesn't lend itself to discrete phases for testing in the same way that construction projects often do.*

Returning to Clausewitz's thoughts on the complexity of the military enterprise, if you recognize that software is composed not of large, static units, but rather a multitude of individual agents, "each of which keeps up its own friction in all directions," you can begin to understand why even the simplest thing can be difficult in unforeseen ways. The fact that the agents in the software system have the potential to act, react, and interact in unexpected ways is also the reason why bugs become an increasingly difficult problem as a product grows, because the complexity of the system and the profusion of potential interactions and behaviors are increasing nearly exponentially. In construction, as the staging area where you keep all the pieces empties as the project progresses, you're left with fewer and fewer questions; things become simple the closer you get to completion. The opposite is true in software. As you begin building, you start to realize how dynamic the system is and to see ways it might behave and ways people might use it that you hadn't considered before, and as you continue to build, the complexity and possibilities multiply.

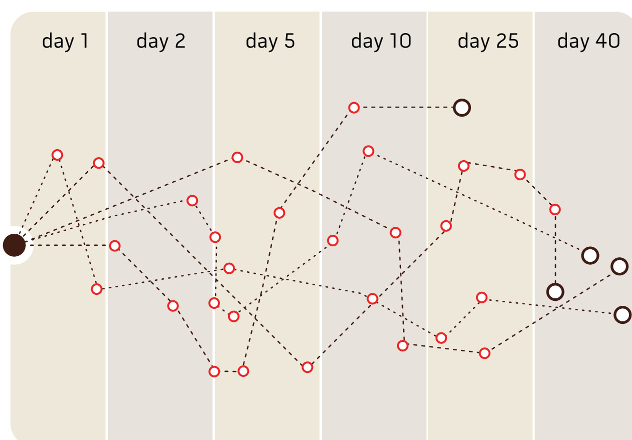
The vast potential friction in the complex system of software is a key reason why the unknown bears so heavily on software development. Something that seems simple on paper is, in fact, difficult, and the scale and effect of that difficulty can't be accurately estimated or known in advance because chance, the unintended, and the unexpected are such strong factors, and a complex system can't be fully comprehended by any person.

### The CAS of software development

It's fascinating, though perhaps discouraging, to consider that the process of developing software is itself a CAS. In this case, the agents in the system are not only the agents in the software itself, but also the members of the project team, the stakeholders, the development infrastructure, and even the office environment. With a team building a bridge, no matter what unforeseen complications arise (delays in materials, weather, poor workmanship, and so on), the entire system is constantly reorganizing itself around executing on the design since that design is entirely accurate, comprehensive, and stable. So, notwithstanding any happenstance during the construction phase, the outcome will always be the same: one predefined bridge.

The course of a software development project, on the other hand, is highly dependent on the idiosyncrasies of every agent in the system interacting with each other and the effect of happenstance, because there is no accurate, comprehensive, and stable plan around which to constantly reorganize. The conditions and progress of yesterday become the basis of what happens today, which determines what happens tomorrow, and ultimately shapes the end result.

This is easy to imagine if you consider assigning the same project to two different, equally qualified project teams. The initial conditions are essentially the same, but the agents—the people, their office environments, their infrastructure, and so on—are different. From the very first minute of the commencement of the two projects, they become divergent. Though equally qualified, the teams will nonetheless approach and solve problems differently, rely on different experience in decision making, and have different internal politics. They will also be subject to different happenstance events, such as having certain team members absent at certain times, getting different answers to the same questions by asking different stakeholders, and getting bogged down by different types of problems and bugs.




*The conditions and progress of yesterday become the basis of what happens today.*

Both teams will hopefully produce a working piece of software, but the two products will be very different from each other. This fact is not problematic in itself, but is a further indication of how strong a role the inestimable effects of chance and friction play in the course of a project. In other words, this is yet another reason why the unknown is such a huge and unavoidable component of a software project.

## The peculiarity of the system

> *Further, every war is rich in particular facts; while, at the same time, each is an unexplored sea, full of rocks, which the general may have a suspicion of, but which he has never seen with his eye, and round which, moreover, he must steer in the night.*
>
> —Carl von Clausewitz, *On War*

The most reliable way to avoid uncertainty is to build products and solve problems precisely like others you've done before. This works strongly in favor of the bridge builder, who has the opportunity to use essentially the same bridge design to span a great number of different valleys. Though they may be different lengths and be subject to different forces of nature, the scope of the problem and the design is a lot less than it was the first time that type of bridge was designed. This allows a much greater predictability in the project and helps the bridge designer make accurate estimates of cost and schedule. It also makes it possible for her to point to the last bridge she built and say, "I'll make you one just like that one," which gives her client a much clearer picture than a written proposal, blueprints, or artist's renderings ever could.

Not so with software. Odds are, the product you're trying to build is very different from any other product that's been built before and certainly different from any you and your team have ever built, especially considering the role of emerging technologies, platforms, devices, and media. But even if you're trying to rebuild an existing product that your team is very familiar with, this go-around is nevertheless certain to be "rich in particular facts." If the original product were perfect, you wouldn't be rebuilding it, so the new version probably requires some significant improvements and changes or is operating under differing constraints and priorities.

Since software and software development are complex systems, a changed starting position means everything that follows will be different. Problems that have been solved before will have to be solved again under entirely new circumstances, and chance and friction will play out in new and unexpected ways. This is particularly the case with UX-focused products, because the mandate for better UX changes how every problem is approached and brings in the contributions of people and domains of thought that didn't exist for the previous version.

So, in short, every software project is unique and tremendously different from any other. Each project will have its own wealth of peculiar details, problems, solutions, and inspirations. This is what makes anticipating any approximations of scope and any corresponding estimates of cost and timeline impossible and unreliable, even for the most experienced and professional companies and teams.

In short, every software project is unique and tremendously different from any other. Each project will have its own wealth of peculiar details, problems, solutions, and inspirations.

But never fear; this doesn't mean that all commitments are impossible or inherently not credible. Solid commitments to schedule and cost are, within the right context, entirely possible. However, commitments to a certain scope are not.

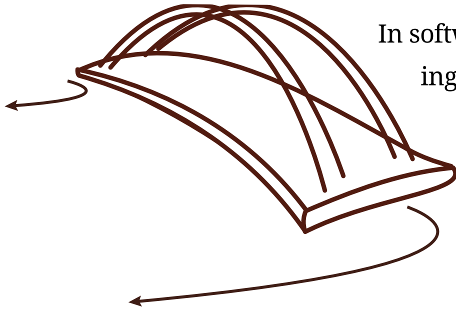
## Subjectivity and Change

Change is the inevitable consequence of uncertainty and the unknown. The more that's specified from the outset, the more change there's likely to be as discoveries are made. This change can come from within the project as opportunities, risks, and issues are encountered. It can also come from outside the project as the priorities of your company and stakeholders shift, changing the context and priorities for the project.

Change demanded by stakeholders, however well intentioned or valuable, can be especially pernicious. Here are a number of requests that our happy bridge builder never has to worry about getting midway through the construction process:

- *Can we move the bridge 17 feet to the left? It's only 17 feet, so that's not a big deal, right?*
- *Our CEO just read an article about how a cantilever bridge collapsed in Quebec in 1907, so he's worried about risks you failed to tell us about. Please change the bridge design from cantilever to suspension.*
- *We aren't very happy with how the bridge looks so far. Can you propose a change in the kind of materials you're using to make it more attractive?*
- *The natural gas pipeline that we ran under the bridge without telling you just exploded and partially destroyed the bridge. Why didn't you build it to withstand explosions? Please fix the damage and give it proper reinforcement, coordinate with our gas pipeline vendor, and have everything done within the original timeline and budget.*
- *Remember when you asked us whether the bridge would ever need to support vehicle traffic and not just pedestrian traffic, and we weren't sure, so we just settled on the cheaper pedestrian version? Well, we were wrong. What can you do to make this work for our needs?*

- *My nephew is majoring in civil engineering and he says the best, most advanced kind of bridge is a side-spar cable-stayed suspension bridge. I don't really know what that means, but why aren't we getting the most advanced bridge possible?*
- *We've hired an offshore company to start building from the other side of the valley so we can cut the construction time in half. They're making some improvements on your design, so please coordinate with them to make sure everything comes together ahead of schedule.*



In software, scarcely a week goes by without some comparable request coming down from stakeholders. Because most people don't understand how software is built and because it has no material, tangible presence, they don't have any basis for understanding what is hard and what is easy. They tend to think everything is easy. It's obvious that moving a bridge 17 feet to the left is an enormous undertaking, but a comparable change in software can be appreciated only by those building it.

It's also extremely important to remember that success in building software is defined in the minds of your stakeholders, not by the objective value of the product itself. A bridge is a success if it fulfills the original design, spans the valley, and withstands reasonable stressors. But software is often judged with much more subjective criteria by stakeholders. If the final product accomplishes 99 out of 100 goals, but the one missing goal was the pet feature of a key stakeholder, you may have failed. If the product exceeds expectations in every functional dimension but fails to integrate well with the visual standards of your company's brand, you may have failed.

This is why taking control of and maintaining your stakeholders' expectations is pivotal to your success. The one mitigating factor at your disposal, though, is that positive feedback from users can usually trump any stakeholder's misgivings and objections. Usually. Even this factor depends on your ability to cultivate a sense of deference to users in your stakeholders. Educating your stakeholders on the value of UX and user adoption doesn't end once you've gotten them to sign off on a budget for the project; their understanding must be continually maintained. Much of your project's success will depend on managing stakeholder perceptions and expectations, so each chapter in this book offers advice about how to work with stakeholders. The business planning and user research stages discussed in Chapters 5 and 6 are crucial to getting your stakeholders into the right mindset.

# Lessons from Uncertainty and the Unknown

> *The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable.*
>
> —Sun Tzu, *The Art of War*

Having accepted that uncertainty, the unknown, and change are unavoidable, you can bring your project to a position of strength in its ability to accommodate them. In fact, you can turn them from threats into a source of value and strength. There are a lot of corollary lessons that become apparent with this new understanding. These lessons strongly underlie how we approach planning, requirements, and process.

## The Further You Are in the Project, the Wiser You Are

The entire duration of the project involves the discovery of problems and their solutions and the constant contributions of design and inspiration. As progress moves forward, you gain greater and greater understanding of your users' needs and the possibilities, goals, constraints, and scope of the project as the window of uncertainty closes toward the completion of the project. That greater understanding includes a more complete understanding of and respect for the overwhelming complexity of the project that helps you put things in the proper perspective.

It follows, then, that the later you make a decision in a project, the more likely it is to be the best one by virtue of having been made from a position of greater experience. This is one of the reasons that attempting to comprehensively define the functional requirements of a product on day zero is absurd and futile; that's the day when you know the absolute least about the product, and any decisions you make on that day are very likely to be incorrect and eventually (hopefully) changed. Decisions taken too early and stated as fact rather than conjecture risk preventing informed thought and design from taking place for the betterment of the product.

This means that decisions that can be made later generally should be made later. We'll get into this more as we discuss how to approach realistic requirements, but this understanding demands that the initial requirements for the project be specific and concrete with respect to only what is actually known. They should be silent or permissive on any question that can be answered in the future from a more informed position.

## Start Development As Soon As Possible

Development is where the majority of design happens, and design is the activity that discovers unknown problems and their solutions, so development should begin as soon as possible. Remember that, as we discussed in Chapter 1, development isn't the exclusive domain of software engineers. It's the stage where everyone on the project team collaborates to develop a solution. The sooner you begin, the faster the learning comes and the sooner unknown challenges and opportunities are discovered. This is an additional argument against spending a lot of time specifying and planning the project up front, because it means you're spending time and money guessing at the solution when you should be investigating and discovering it.

## Written Functional Requirements and Specifications Are Inherently Flawed

Functional requirements and specifications are written before development begins, so they're immediately handicapped by having been made from the least informed perspective. It's also extremely difficult to make an effective language-based description of an experience, interaction, or visual design. And by virtue of their written form, they're never as fluid and dynamic as the project itself. It's pointless to try to keep the written requirements and specifications up to date (which no one ever does, anyway), because then you're just updating them to match what's already happened in development. That defeats the ostensible purpose of specifications guiding development, and there's no point in maintaining a written history of your product as it develops.

The production of detailed written functional requirements and specifications poses a number of other problems:

- *They're usually extraordinarily long and take a very long time to develop, so they delay the start of development and take budget and resources away from the building of the product.*
- *Being so long and so focused on the details and minutiae, they're rarely read in their entirety and make it difficult to see the forest for the trees.*
- *The focus on specificity and detail can in fact cause everyone on the project to lose sight of the big picture (if they had sight of it in the first place) and get caught up in focusing on the details.*
- *They tend to stifle thought and innovation by the development team because they can just follow the letter of the specifications without questioning whether they reflect good decisions and thoughtful approaches.*
- *They usually wind up being an unrealistic laundry list of every possible feature, rather than a studied, thoughtfully scoped product framed by reasonable constraints. This also means that every pet idea and feature of each of your stakeholders will be in the document and will therefore be apparently committed to, leading to potential conflict as unnecessary or infeasible ideas and features fall by the wayside.*
- *They tend to cause stakeholders to think, having spent weeks exhaustively laying out the specifications, that their work is done and all the questions have been answered. This leaves them with a false sense of certainty that you'll later inevitably have to defy, and also means they won't be around to participate during the development phase.*

Our new clients often come to us with a phonebook-size binder of documentation, or ask us to help develop one to satisfy a bureaucratic requirement in their organization. We've even seen clients spend more money on building the specifications than on developing the product itself. Invariably, when we finish a project and look back on the early documentation for it, the documentation and the product bear no resemblance to each other.

Putting together written requirements and specifications is not entirely without value, though, so long as it's perceived in the right way. The goal should not be to build a definitive description of the product, but rather to do a dry run of the product design, to get the team to start thinking through the problems that lie ahead. To quote another famous war strategist:

> *In preparing for battle, I have always found that plans are useless, but planning is indispensable.*
>
> —Dwight Eisenhower

Rather than thinking of early documents and planning as strict requirements, it's more correct and useful to view them as guidelines. They're the encapsulation of the best understanding that existed at the time. This understanding cannot stagnate at such an early stage; it must deepen and improve through the whole course of the project. Setting out initial guidelines from the perspective of the business and the user are the subjects of Chapters 5 and 6, which cover business planning and user research, respectively.

## Commitments to Scope Are Untenable

Any estimation of scope, having been defined in the "mere twilight" of the project kickoff or contract negotiation, cannot possibly be accurate. And a comprehensive description of the scope of a project is so enormously complex that it simply can't be done; the only perfect description of a product is the product itself.

If the estimated scope is incorrect and incomplete, any subsequent estimate or commitment will be incorrect and incomplete. This is the root problem of the mistrust that often exists between managers and project teams, or between clients and their software services vendors. Managers look for certainty of scope, schedule, and cost—the so-called three-legged stool—and press for firm commitments to all of them. When scope inevitably changes, it makes the stool teeter off balance. More often than not, you find yourself either face-first on the floor or sitting on a much shorter stool than the one you thought you were building. Neither position is particularly dignified in the eyes of your stakeholders. In our experience, investing in UX yields such tremendous benefits that the period of uncertainty and the flexibility required with respect to the three-legged stool prove to be well worth it when the project ends.

## Relish and Respect the Unexpected

Everyone carries a lot of preconceptions and assumptions into and through a project. The application of creativity and intelligence against the challenges, opportunities, and unknowns through the course of the project is bound to take the project down unexpected paths, and to challenge preconceptions about the product's users, its requirements, and the best solutions to key goals and problems. Those who put excessive faith in their own preconceptions or who are averse to the unexpected will find the progress of the project constantly clashing against their expectations and sense of security.

What you get back from the project team is often going to be significantly different from what you expected because the team has gone through an in-depth study and design process. If you have an intelligent, creative, professional team operating in the proper context, its discoveries and solutions should be much more solid than anyone's preconceptions and should therefore be respected and trusted. People involved in the project who are at peace with uncertainty and the unknown will actually come to enjoy unexpected turns and discoveries, because these offer lessons that are valuable beyond the product itself. They are evidence of innovation and effective, creative design at work. And in order to respond effectively to the unexpected without getting bogged down by new questions, the team must learn to respond to unknowns quickly and intelligently. This builds a strength and nimbleness that benefits the whole project.

## Intolerance of Uncertainty Is Intolerable

People who oversee software projects have an entirely reasonable need to be able to plan for them in the context of the larger organization and to meet commitments of their own—and that need typically manifests as pressure for certainty. Some certainty can be offered, but some things are impervious to certainty. This is an immovable fact, as we've gone to great lengths to explain so far in this chapter. Unfortunately, it's extremely difficult to convince anyone who hasn't been in the trenches of software development of this fact. Too often, those people mistakenly view a person who humbly and wisely recognizes reality as being mealy-mouthed and resistant to accountability.

Intolerance of uncertainty can cause some serious problems. It pressures project leaders to present things as certain when they should know they're not.

Intolerance of uncertainty causes very serious problems. It pressures project leaders to present things as certain when they should know they're not, setting up future conflict and injuries to credibility. It also tends to cause project leaders and team members to be overly optimistic in their projections as they try to offer pleasing answers to their stakeholders. As the project progresses and the weaknesses of their projections become apparent, the project team will often hold the stakeholder at a distance, in the hope they can scramble to pull off a last-minute miracle. This means that challenges to the project that should have been identified and disclosed as soon as they happened accumulate until the end. At that point, it's too late for the stakeholder to help or make adjustments, and they're blindsided by failure and disappointment.

It's certainly the responsibility of the project leader and team to act more responsibly than this, but it's very difficult to build a product in an environment of intolerance of uncertainty. Anything you can do to help stakeholders understand how to create the right climate of accountability and realistic expectations (giving them a copy of this book, perhaps?) will go a long way to ensure a successful project. The project's focus on delivering superior UX quality provides a helpful star by which to help people navigate. Ultimately, the project and your company are accountable to the needs of the user as an objective point of reference. Continually reorienting the team and the stakeholders to the UX goals of the project can help you slough off unreasonable expectations, focus on what's important, and take the best advantage of the unknowns as they arise.

## Effective Requirements

Requirements need to be of a nature and in a form that allow them to adapt and remain useful and relevant through the winding course of the project. This is why the most useful approach is to think of the requirements as a framework for answering questions rather than a catalog of answers. The framework, if devised properly, will be stable because it will be composed only of knowable goals and constraints and not of solutions or designs that will be subject to future design and change. It should also be wide enough to allow room for a variety of successful outcomes (success is a range, not a single point) but narrow enough to fence out most unsuccessful outcomes. The framework requirements can be pictured as a frame describing the bounds of a successful solution.

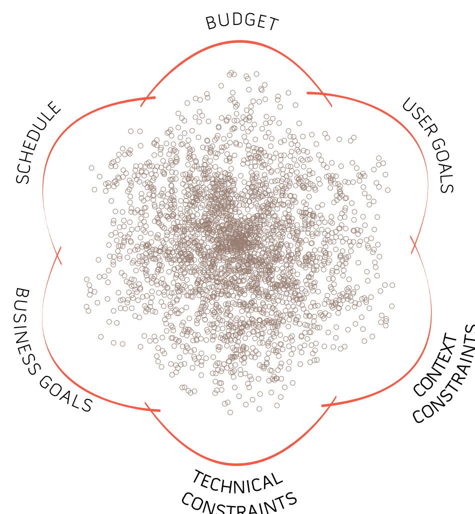
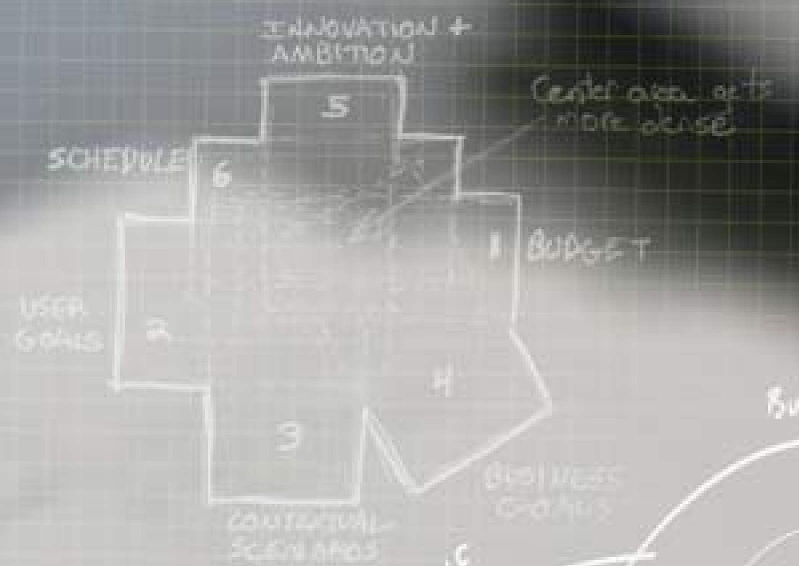



*Figure 3-1. Framework requirements describing the bounds of a successful solution*


FRAMEWORK/2
INNOVATION +
AMBITION
Center area gets
more dense
5
SCHEDULE
6
1
BUDGET
USER
GOALS
2
4
3
BUSINESS
GOALS
CONTEXTUAL
SCENARIOS

BUDGET
USER GOALS
CONTEXTUAL SCENARIOS
INNOVATION & AMBITION
BUSINESS GOALS
SCHEDULE

In this visualization, the dense area represents the successfulness of the solution, while the constraining boundaries represent the framework requirements. This portrays a very effective set of framework requirements because the framework encompasses barely more than the most successful solutions, is wide enough that it encompasses all possible successful solutions, and is dead on center so that the tendency will be for answers to questions to also find their way to the center.

Throughout the course of the project, thousands of little questions and decisions will need to be made for which there won't yet be an explicit answer. These questions are like rubber balls tossed into the framework; they bounce from wall to wall and tend to arrive near the center. Thus a successful framework provides not the answers to every question, but the design parameters for how team members can themselves discover answers and make decisions, large and small, through the course of the project.

The framework requirements—being a set of constraining parameters rather than a list of answers—are a description of the problem and not of the solution. In our experience, most companies planning a new product haven't had a chance to develop a solid understanding of the problem they're trying to solve, let alone how they'll solve it. A phonebook-size binder of requirements documentation represents an exhaustive attempt to accurately define a solution. But, as you've learned in this chapter, that view of the solution is guaranteed to be inaccurate. This approach is an attempt to answer every question before the real work of design has had a chance to begin.

Recognizing that the purpose of requirements is to define the problem and not the solution, all efforts should be made to ensure that guesses at the solution don't wind up becoming parameters in the framework requirements. The framework parameters need to be entirely reasonable, accurate, and stable, but they also need to be flexible and restrained. When guesses at the solution are built into the framework, they risk being wrong and falsely limiting or misleading design decisions, undermining the value of the framework itself.

It's a hallmark of good framework requirements that they remain stable and unchanging through the project, because it means they haven't crossed the

line into areas reserved for the design of the product. The closest analog for good framework requirements is the U.S. Constitution, because it's general and flexible enough to provide a framework for answering questions that the founding fathers could never have possibly foreseen, and yet remains a resilient foundation for democracy, stability, and the rights of citizens.

## How Framework Requirements Are Built

The great news about framework requirements is that they don't require 12 weeks of Sisyphean planning efforts and documentation the length of *War and Peace*. While clients often come to us with their own first attempts at requirements, the first part of a project is still spent building the framework requirements.

The process of building framework requirements involves investigating each of its key parameters, and then distilling the findings of those investigations into a form that can be easily used and understood by everyone on the project. The parameters that go into a project's framework vary from case to case, but they generally fall into three categories:

- *The needs of the business*
- *The needs of the user*
- *The technical and infrastructural constraints*

The framework shown earlier in Figure 3-1, for example, was composed of six example parameters:

- *Business goals*
- *Schedule*
- *Budget*
- *User goals*
- *Context constraints*
- *Technical constraints*

In the next three chapters, you will learn how the business, user needs, platform and context constraints, and technical constraints are investigated, distilled, understood, and communicated to form a usable, realistic framework of requirements.

## Extending the requirements

The essence of building a software product is an ongoing evolution and deepening of the team's understanding of the requirements through ongoing design, with engineering following close behind. The framework requirements are a starting place and become, as the project progresses, the exterior frame within which tighter and tighter frameworks of understanding are developed.

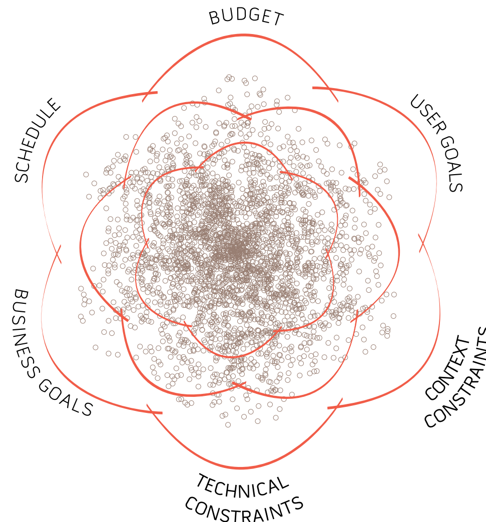


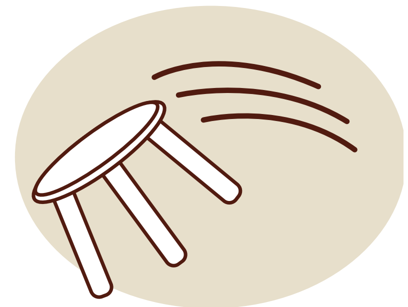

*Figure 3-2. Extending the framework requirements*

Each step of the project, including the design and engineering work that go into building it, are elements of an ongoing investigation of what the product should be. Every document that's produced, every meeting that's conducted, and every bit of design that's done is oriented at honing the collective understanding of the product. The result should be an increasingly narrow, multifaceted, and accurate view of the requirements with an ever-decreasing area of uncertainty. The final product represents the moment when all is known and all questions have been answered, and so would be a single, perfectly round point at the center.

## Reexamining the Three-Legged Stool

The fact that commitments to scope are untenable seems to fly in the face of the managers' need to have a reasonable degree of certainty of what they'll be getting. But not basing commitments on a specific, early guess at scope actually gives them a greater degree of certainty, so long as they trust you and the project team.

Errors in and changes to a scope commitment can have a wide effect on a project. The scope may prove to be overambitious given the budget and schedule available, forcing the project leader to go back to his stakeholders to revise scope or get more money and time. Ambiguous or largely inaccurate initial guesses at scope can cause a project to run far off in the wrong direction, requiring scope to be cut or money and time to be added to bring it back on course. Overwhelmingly long and specific scope documents can fail to carry through the company's and the stakeholders' overall vision for the product, leading to a product that disappoints and may require more time and money to bring up to suitability.



So, in short, by forcing a commitment to an early guess at scope, managers are, in fact, contributing to the peril that all of commitments they were relying on will be challenged and changed. The reason for this becomes very obvious if viewed in the form a pseudo-equation for the traditional three-legged stool of committed scope, schedule, and cost:[4]

```
product = f( scope & schedule & cost )
```

The problem here is that whereas schedule and cost are known, scope is not. This means this equation is inherently unsolvable until scope is known. As we've said, the only true and full definition of the scope of a product is the product itself, a fact that would make this the case:

```
product = scope
```

And invoking the rule of equivalence, we end up with:

```
product = f( product & schedule & cost )
```

4 The ampersand (&) is a logical symbol, but is being applied very loosely here—as a placeholder, essentially—because the way the variables are related isn't knowable and isn't important to the argument being made.

The only way this equation is meaningful is if schedule and cost have no effect whatsoever. But schedule and cost are certainly key factors that will go to define and limit the product. In fact, it should be uncontroversial to leave scope out of the picture and say that the product will be some function of schedule and cost:

```
product = f( schedule & cost )
```

Then we follow a simple chain of logic that leads to a happier place:

```
product = f( schedule & cost )
    &
product = scope
    therefore
product = scope = f( schedule & cost )
```

In the end, the fact that scope is unknowable until the completion of the project means that the notion of scope is unusable as a defining parameter for the product. The project's requirements must be a function only of knowable variables, of which scope is not one.

This is a fact that's acknowledged and successfully addressed by the framework approach to requirements. The unknown value of product is ultimately solvable by a study of the parameters that represent its constraints, which are variables that all have known values. The example framework presented earlier in the chapter would, for example, have a pseudo-equation like this:

```
product = scope =  f( schedule & cost & business_goals & user_goals
& platform_and_context_constraints & technical_constraints )
```

Every variable of which the product and scope are a function is knowable, and therefore the equation is solvable. Schedule and cost can be dictated by managers; business goals are fixed during business planning (see Chapter 5); user goals and context constraints are discovered during user research (Chapter 6); and the technical constraints are found during the initial product architecture stage (Chapter 7).

It's also somewhat of a change to view schedule and cost not as flexible factors influenced by scope, but rather as fixed, constraining parameters. Software projects can generally be made to fit within or expand to nearly any reasonable budget size; it's just a question of how ambitious you want to make them, how detailed you allow things to get, how richly designed

you can allow them to be, how you choose and allocate resources, and so on. Clients often come to us with requirements documentation and ask us to prepare an estimate based on it, but everything in the documentation except the high-level business requirements is frankly irrelevant, because we know it will change. The more important questions are: how much are you willing to spend on this product, and when do you need it by? The answers to those questions give a much, much clearer picture of what the true scope of the project will ultimately be.

Approaching projects in this way requires a big leap of trust on the part of the client or stakeholders, though, so often it's not an option. We typically make cost and schedule estimates as best we can based on whatever constraining variables we've been permitted to know about, and then work with our client to hone the estimates to fit their actual constraints. Though these cost and schedule estimates may be based on early requirements documentation, once these estimates have been approved they supplant the requirement documentation as part of the framework requirements.

So what does this all mean for the three-legged stool? In the stool metaphor, the stool is the product, which is, as we've just demonstrated, also the scope. So to say the stool rests on scope, schedule, and cost is to say the stool is resting, in part, on itself. This is a paradox worthy of a mind like M.C. Escher's, but is hardly proper territory for a software product. The stool sits not on scope, schedule, and cost, but rather on schedule, cost, and any other constraining parameter.

## Commitments You Can Live Up To

All of this may require an enormous mental shift for you, but once you do it, you'll find you can make commitments with a much greater degree of confidence and reliability. What you should be committing to is fidelity to the constraints—the framework parameters—for the product. Luckily, two of those constraints are cost and schedule, and being able to make confident commitments to those two will go a long way toward reassuring managers and stakeholders. The remainder of your commitment is not that the product will conform to some preordained scope, but rather that it will satisfy the needs of the business and its high-level criteria for success and will satisfy the needs of the user. Who could object to that?

The trick then becomes making sure that the needs of the business and the user have been well understood and are reasonably construed. Once they are, they become an essential part of the basis of your commitment. Rather than requiring certainty of scope, your stakeholders should hold you accountable to the project's fidelity to its business and user constraints. This makes it important that stakeholders are in agreement with and have signed off on those constraints, which is why we spend a lot of time discussing stakeholder buy-in throughout this book.

## Effective Process

The process by which software gets developed is just as much guided by uncertainty and the unknown as the requirements are. This is for two principal reasons:

- *Design happens in the context of the unknown through the whole course of the project, so the project's process must support successful design that leads to correct decisions and outcomes.*
- *The actual destination of the project (the final product) is unknown up to the end, so the process must ensure that there is a minimum of off-course meandering before you arrive.*

We should be clear at this point to explain what we *don't* mean when we say "process." Some people seem to believe that the software development process is like an instruction manual; if you follow all of the instructions to the letter, you'll end up with a successful product. In our experience, that kind of "process" is a dangerous myth. Remembering that software and software development are complex and peculiar systems, no instruction manual could possibly exist that would cover every possible project. There's also a risk with this type of thinking that project teams will view their success not in terms of the success and quality of the product itself, but rather in terms of how well they followed the process and whether they did a good job of producing process-mandated documentation on time.

People are also often under the misapprehension that, as with an instruction manual, the software development process is a serial progression of discrete steps. This view is very appealing to managers and stakeholders because it gives them a sort of timetable for what progress and deliverables to expect and also lets them know when they need to pay attention. Unfortunately, it's just not that simple.

Good software development process addresses the effects of uncertainty and the unknown that we just identified. It supports successful continuous design to ensure good decisions and outcomes, keeps the project headed in the right direction with a minimum of course deviations, and keeps all of the key contributors participating in design through the full course of the project. These three goals are supported by combining proper methodology with effective tools and techniquess.

## Development Methodology

It's perilous for us to tread into the realm of software development methodology, but there's no way around it. Software professionals often have very strong opinions on what constitutes good versus bad methodology, in some cases exhibiting cult-like adulation of some specific approach and complete intolerance of differing views.

In our experience, no one methodology suits every possible project. The infinite variety and peculiarity of projects makes this conclusion rather obvious. Every different, specific methodology has well-reasoned underpinnings. When you're acquainted with that reasoning, you can figure out what's best for a given project.

### Waterfall

The waterfall methodology is the most familiar to people because it's the most widely employed and also seems to make the most intuitive sense. As shown in Figure 3-3, it proposes that software be built in a sequence of major steps—usually business requirements, design requirements, development, then deployment—each of which is entirely completed before the next one begins.

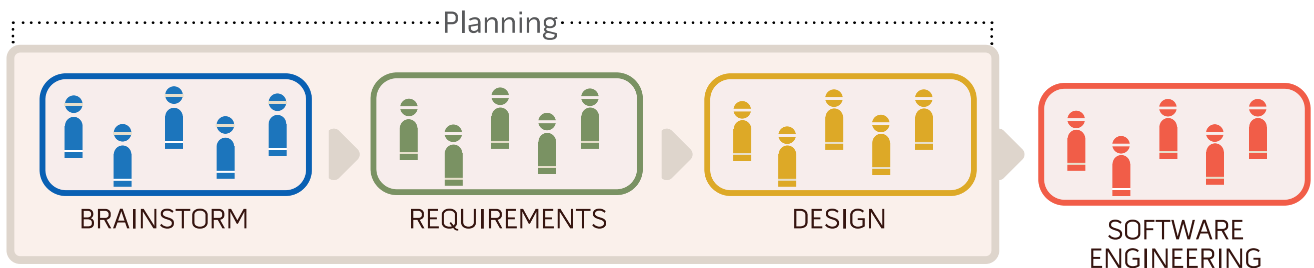



*Figure 3-3. A basic waterfall process*

The supposed strength of waterfall is that it seems to provide a great deal of clarity and certainty through each step of the project. It's appealing to managers because it suggests that once the brainstorming is done and the requirements have been built, everything else follows naturally.

The flaws with this approach should already be obvious, but before we get into that, it's worthwhile to point out that waterfall is actually an effective approach for some types of projects. Waterfall is efficient and effective for products that represent minimal design and engineering complexity, or that are cookie-cutter implementations of well-understood solutions. We would employ this approach if, for example, we were building a calculator application. A calculator is very simple to build, and there are very few questions that need to be answered about the calculator's features (what it should look like, whether it should be basic or scientific, whether it should include memory functions, and so on) before development begins. The answers to any design questions are readily obtained in advance and are highly certain.

But no one ever asks us to build calculators. If that's what you're working on, you should quit reading this book and just go build the darn thing.

For any other project, waterfall's fatal flaw is its total failure to account for uncertainty and the unknown. It presumes that each step can be entirely and perfectly completed before the next step begins. We've devoted a great deal of this chapter teaching you what a huge mistake this is; running down the list of problems with this approach would be beating an already quite dead horse.

There are, however, two other serious problems with this approach that we've touched on only briefly so far:

- *Because each step is entirely separate, each group of contributors is entirely siloed from the others. The people brainstorming and writing requirements for the product never collaborate with the people architecting and designing it. The software engineers never have the opportunity to collaborate with the architects and designers, let alone the business managers and stakeholders. Collaboration across all disciplines is absolutely critical to the building of great software, as we'll discuss shortly.*

- *This approach forces the engineering and quality assurance (QA) stages to absorb almost all of the effects of the risks and unknowns that arise during the project. Since the planning, architecture, and design of the product are already ostensibly complete, there's no option for changes to them because the money for them has already been spent and the resources have been allocated to other things. This leaves it to the engineers to figure out how to account for the inevitable unforeseen problems and unknowns—and to do it within the budget and timeline they were allocated before the problems and unknowns were identified.*

This is why it so often seems like a beautifully conceived and designed product gets hacked and compromised into severe mediocrity by the engineers. They aren't being lazy or incompetent; they're simply delivering what they can despite being left to absorb the full brunt of risk and the unknown on their own. That they are likely to have made compromises and hacks that the stakeholders and designers disagree with is just one more reason why waterfall's tendency to silo resources is such a terrible problem.

## Big Design Up Front

The term Big Design Up Front (BDUF) is shorthand, often used as a pejorative for a sort of methodology that's similar to waterfall but takes a meaningful step in the right direction. As the name suggests, BDUF essentially involves large upfront design efforts before engineering and QA begin. It differs from waterfall, however, in acknowledging that not all design occurs up front and some design (in the form of resources, budget, and prerogative) must be reserved for the engineering and QA phase.

BDUF accounts for waterfall's tragic central fallacy that each step can be made perfect before the next begins, but it does so rather weakly: it suggests that the design step can be made nearly perfect before development begins.

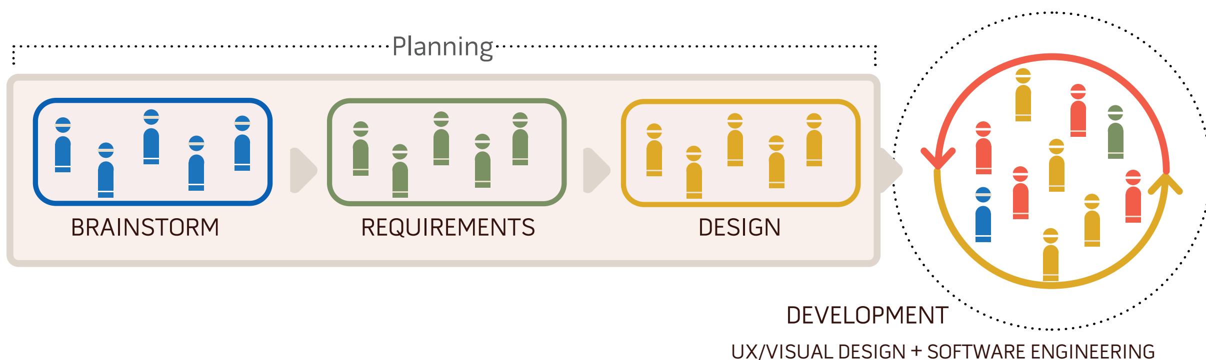



*Figure 3-4. Big Design Up Front*

Proponents of BDUF suggest that planning a product on paper and on whiteboards before engineering begins saves time and cost throughout the rest of the project because it's easier to change requirements and sketches than it is to change actual code. This is absolutely true; only proponents of the most anarchic, cowboy methodologies would ever argue that No Design Up Front is a sensible approach. The problem with BDUF, though, is in the "Big."

BDUF tends to require that too much design be done up front. It still treats design largely as a discrete phase that begins and ends before the actual development of the product begins. Upfront design efforts suffer from the absence of any of the understanding about risk, opportunity, and the unknown that come through the engineering effort, and through development more broadly. Upfront design is also typically done without the assistance of software engineers, who should be present to assess the cost and feasibility of certain ideas and to contribute ideas from their unique perspective. The early days of the project are when the least is understood about the project, so the more design work done during that period, the more likely that work will be off base. This all means that significant portions of the big upfront investment in design will be wasted, depriving the rest of the duration of the project of valuable design resources and budget, and delaying the commencement of actually building the product.

Like waterfall, this approach also promotes a false sense of certainty in stakeholders and siloes resources from each other. Stakeholders tend to participate only in the upfront design process, and UX designers participate in the development stage only to the extent that their budget and time weren't expended up front.

But the fact that experience designers participate in the development stage at all is a huge step in the right direction. It's an acknowledgment that unknowns and problems will be uncovered for which collaboration with experience designers will be beneficial, and acknowledges to a degree that the initial designs will need to be adapted and modified.

Truth be told, BDUF is a methodology EffectiveUI is frequently compelled to employ, despite our preference to do otherwise. This is because we are a professional services company that builds products for other companies, and those companies have a very reasonable need to understand what to expect from us and to be reassured that we understand their needs and know what we're doing.

In the absence of a trusting relationship built on a long history of partnership and successes, we can't ask a client to just give us a budget, timeline, and a high-level understanding of their goals and trust us to build them something they'll love. That approach can work very well—that's essentially how we developed an extraordinarily successful partnership with eBay to build eBay Desktop[5]—but it requires a degree of trust and latitude that's rarely available in any project, let alone a first engagement with a new vendor. Whether you're building your product for a client or for your own company, the same considerations of credibility and trust will probably pressure your process toward BDUF.

What BDUF offers in this circumstance is the opportunity to work intensively with stakeholders to translate their business and user needs into a comprehensive set of visual experience design requirements. At the end of the upfront design effort, the stakeholders are given a thorough stack of visually rich documentation that demonstrates that their needs have been heard and understood, that they're working with a professional and qualified team, and that the team has a strong understanding of the product's requirements and the road ahead. This stack of documentation is often what's used to unlock the remainder of the budget for the project or to seek buy-in from higher levels of the management ladder. BDUF projects are also easier to sell, because they allow stakeholders to sign off on a smaller design project before committing to the larger full project.

The problem with BDUF is that it generally keeps the engineers on the bench, in the dark, and out of the conversation for too long. A great number of unknowns, problems, and opportunities can be identified and solved through an exhaustive upfront design process, but until engineering begins, a vast, rocky sea of the unknown remains unexplored. Additionally, without the benefit of the engineers' input, promises and estimates can be made that will later prove to be impossible or unrealistic, leading to disappointments and increased tensions.

Like waterfall, BDUF also has the tendency to suggest a higher degree of certainty than can actually be obtained through early design efforts. In some

5 With an eBay account, you can play with this application by downloading it from *http://desktop.ebay.com*.

cases, managers and stakeholders try to use this apparent certainty as a means of "exposing" their engineering team—that is, of putting the engineering team in a position of being isolated and solely responsible for the completion of the project. This tactic is calculated to increase accountability for the engineering team, but its true effect is to make all teams less effective and put an undue burden of risk and strain on the engineering team.

As EffectiveUI's stature and commensurate credibility in our market have grown, we've been able to reduce the amount of upfront design required to reassure our clients. Your success in moving in this direction will also be dependent on your credibility, which is why so much of this book is dedicated to the subjects of maintaining enthusiastic buy-in and building credibility with your stakeholders.

## Agile with a capital "A"

Agile is the name of a broad set of methodologies that arose from frustrated software engineers who were trying to find more effective approaches to their work than the traditional, failing ones. Despite its origins in software engineering, the concepts of Agile are very applicable to the entire product development process. The integration of UX design into Agile processes is a somewhat new frontier of thought in software methodology.

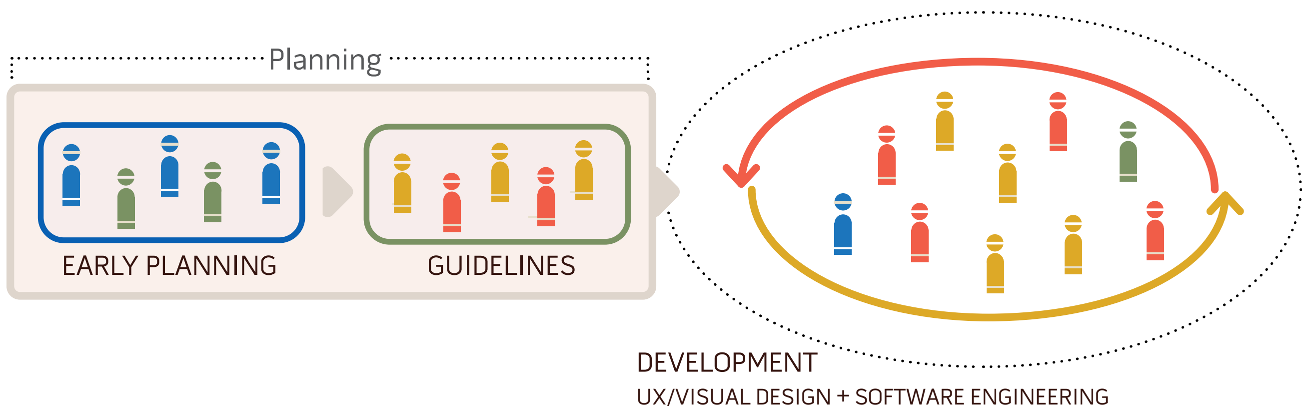



*Figure 3-5. Agile processes in UX design*

Unfortunately, some people already have a bad taste in their mouths left over from previous encounters with Agile devotees. The problem with Agile is that it has some very overzealous supporters who insist that Agile

concepts are the panacea for every possible software woe. These people may dogmatically enforce some particular purist submethod of Agile to the great detriment of the project and the sanity of its team. There is no single right answer, no perfect methodology that can address the full range of projects and problems in the world. Just as software projects have no room for false certainty in features, they have no room for false faith and dogma in methodologies.

But notwithstanding its outspoken supporters, Agile concepts contain a great deal of wisdom born of a long history of experience. Many people have a passing familiarity with some of the offspring of the Agile movement, such as Extreme Programming (XP) and Scrum, but not with the Agile Manifesto itself. The fact that it's called a manifesto may seem to bode poorly (think Unibomber), but bear with it.

### MANIFESTO FOR AGILE SOFTWARE DEVELOPMENT

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.[6]

This aligns very tidily with everything we've discussed so far in this chapter. At the heart of Agile is the acknowledgment of uncertainty and the unknown, which requires that flexibility, collaboration, and thoughtfulness be favored over rigid commitments and the stunting and segregation of design and thought.

6 *http://www.agilemanifesto.org/*

## Effective development methodology

What, then, is EffectiveUI's solution to the methodology question? We've been successful in building superlative products using a broad range of methodologies, and that experience has brought about a lot of ideas from many of our team members. The difficulty in proposing an EffectiveUI methodology is the fact that no single methodology will work in every circumstance. We're also living very much on the cutting edge of this sort of thinking, and anything concrete we propose at the time of writing this book is bound to be outdated by the time the book goes to press.

We espouse, therefore, not a specific, patent-pending methodology, but rather a set of principles and best practices. Like the framework approach to requirements, good principles can guide successful thought and progress and stay relevant as the domain of thought progresses. The rest of this book is dedicated to sharing those principles and showing them at work, with a specific emphasis on development methodology (coming up in Chapter 8). If it seems like we've spent most of this chapter breaking you down without building you back up, please bear with us.

If you return to our discussion of the definitions of "design" and "development" at the end of Chapter 1, you'll note that we mean development to be inclusive of every professional discipline, including stakeholders, the project leader, UX architects, visual designers, and software engineers. A huge part of what makes taking a restrained approach to upfront planning and minimizing wasteful upfront design efforts so important is that it frees up room for a larger, more inclusive development stage. The ideal setting for building great UX is one where the business leaders, designers, UX architects, and software engineers are all working in tandem and actively collaborating to build the product. This can't happen if each group's contributions are segregated into discrete phases. Working closely together as part of one large development team allows everyone to benefit from the learning that occurs during its course, and to contribute to the decision making that responds to unknowns, problems, and opportunities.

For the sake of avoiding redundancy, we're leaving the greater part of the discussion of the development cycle for Chapter 8. But that chapter is very much a sister chapter to this one, since all of the concepts we discuss regarding how to handle the approach, methodology, and planning for the product are mostly aimed at creating a fertile ground in which development can occur. So, if you're curious and want to continue exploring this line of thinking in more depth, you may consider jumping ahead to that chapter.

### Efficiency and the unknown

It may seem at first blush that a project at the mercy of uncertainty and the unknown will be inefficient to produce and, therefore, more expensive. Compared to the nonexistent project where all things are known and there is no uncertainty, a real-world project will certainly be less efficient. If you know exactly where you're going, you'll naturally take the straightest path there. But since that sort of certainty is never available, the efficiency of a project will be a function of how well you account for uncertainty and the unknown. Clinging to false certainty is a surefire recipe for enormous waste.

Consider again a project assigned to two equally qualified teams. One team is managed using a waterfall process and the other using a more nimble framework requirements–driven process. The course of each team's progress toward the same destination might look like what's shown in Figure 3-6.

Team waterfall's first step is to start executing an in-depth plan that, by virtue of having been developed in a "mere twilight," has them heading in the wrong direction. They don't discover this until they finish, present the results, and fail to please. They then do more extensive planning to identify how the product needs to be modified to reach success. That plan leads them closer to success, but not quite all the way, resulting in two more planning and building cycles.

Team agile, on the other hand, zigzags along the course to success. Each time they pause and assess the situation, they see that they are off course and make a correction. Further, each time they make a correction they're further along in the project and are therefore able to make better decisions about course adjustments, so the distance of each deviation gets smaller each time.

And this doesn't even account for the time team waterfall spends planning. Nor does this reflect the fact that it's much more time-consuming to refactor large volumes of code that were written over a long period of time than it is to make adjustments to small bits of code that were recently written. When you add this all up, the efficiency of an agile approach in the context of the unknown is clear.

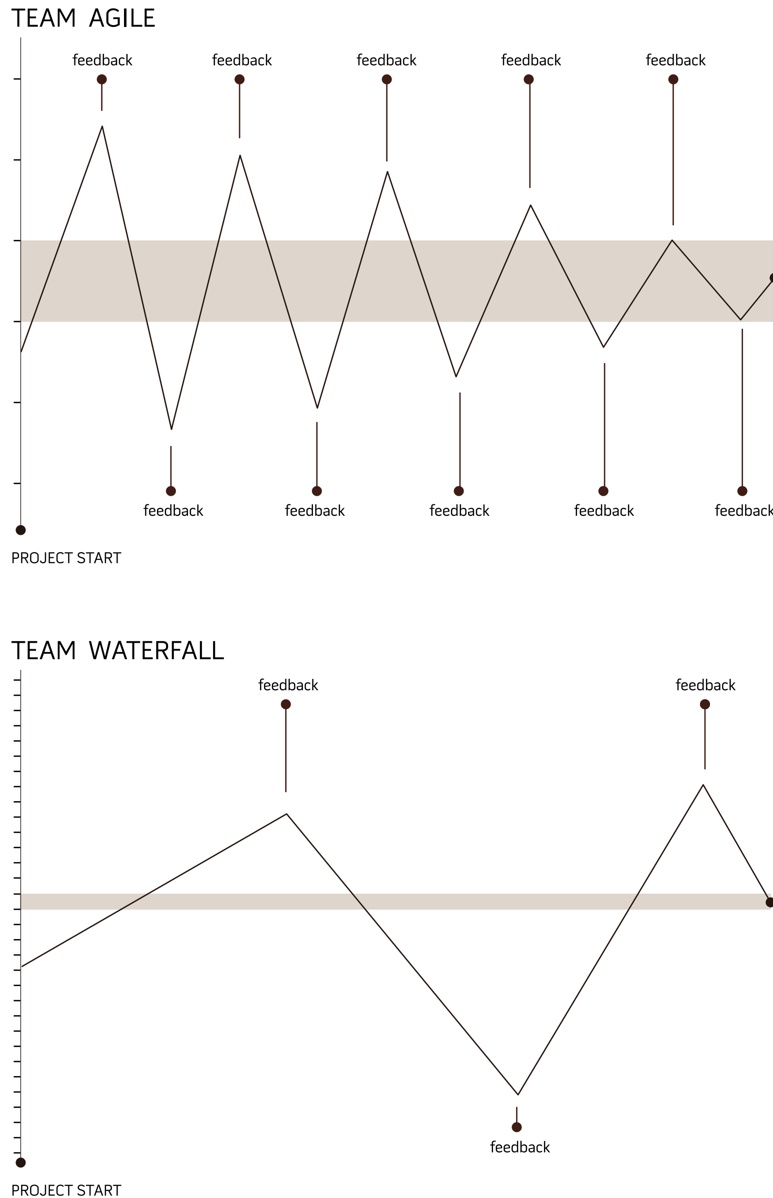



*Figure 3-6. Comparing methodology development pathways*

# Want to read more?

You can find this book at oreilly.com
in print or ebook format.

It's also available at your favorite book retailer,
including iTunes, the Android Market, Amazon,
and Barnes & Noble.

O'REILLY®

Spreading the knowledge of innovators

oreilly.com