The Addison-Wesley Signature Series

A Vaughn Vernon Signature Book

Strategic Monoliths and Microservices

Driving Innovation Using Purposeful Architecture

Vaughn Vernon
Tomasz Jaskuła

Foreword by
Mary Poppendieck

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for Strategic Monoliths and Microservices

"Most books address either the business of software or the technical details of building software. *Strategic Monoliths and Microservices* provides a comprehensive approach to blending the needs of business and technology in an approachable way. It also dispels many of today's myths while offering practical guidance that any team or organization can apply immediately and with confidence."

—*James Higginbotham, Executive API Consultant, Founder of LaunchAny, and author of* Principles of Web API Design

"Digital Transformation cannot succeed as a 'grass roots' effort. Vaughn and Tomasz offer C-level execs a roadmap to software excellence that includes establishing the culture necessary to foster and sustain software innovation. Written with real-world understanding, Vaughn and Tomasz help the reader to appreciate that moving software development from a cost center to a profit center involves tradeoffs that need not sacrifice innovation. A must-read for decision makers."

—*Tom Stockton, Principal Architect, MAXIMUS*

"In this book, Vaughn Vernon and Tomasz Jaskuła use their extensive experience with DDD to present a comprehensive guide to using the many different aspects of DDD for modern systems development and modernization. It will be a valuable guide for many technical leaders who need to understand how to use DDD to its full potential."

—*Eoin Woods, software architect and author*

"There are common misconceptions and roots of failure around software engineering. One notable example is neglecting the rugged trek towards digital transformation. Such an endeavor comprises breakthrough innovations, failure culture, emphasis on the role of software architecture, as well as on the importance of efficient and effective inter-human communication. Fortunately, the authors offer the necessary help for mastering all hurdles and challenges. What I like most about this book is the holistic view it provides to all stakeholders involved in digital transformation and innovation. Vaughn Vernon and Tomasz Jaskuła introduce a clear path to successful innovation projects. They provide insights, tools, proven best practices, and architecture styles both from the business and engineering viewpoint. Their book sheds light on the implications of digital transformation and how to deal with them successfully. This book deserves to become a must-read for practicing software engineers, executives, as well as senior managers. It will always serve me as a precious source of guidance and as a navigator whenever I am entering unchartered territories."

—*Michael Stal, Certified Senior Software Architect, Siemens Technology*

"Digital transformation is a much used but little understood concept. This book provides valuable insight into this topic and how to leverage your existing assets on the journey. Modern technical and social techniques are combined in the context of a single case study. Compelling reading for both business and technology practitioners."

—*Murat Erder, co-author of* Continuous Architecture in Practice (2021) *and* Continuous Architecture (2015)

"Packed with insightful recommendations for every executive leader seeking clarity on the distinction between when to strategically apply a monolith vs. microservice architectural approach for success. Highly encourage every CEO, CIO, CTO, and (S)VP of Software Development to start here with immersing themselves in Vaughn and Tomasz's succinct distillation of the advantages, disadvantages, and allowance for a hybrid combination, and then go become a visionary thought leader in their respective business domain."

—*Scott P. Murphy, Principal Architect, Maximus, Inc.*

"A 'must-read' for Enterprise leaders and architects who are planning for or executing a digital transformation! The book is a true guide for ensuring your enterprise software innovation program is successful."

—*Chris Verlaine, DHL Express Global Aviation IT DevOps Director, Head of DHL Express Global Aviation IT Software Modernization Program*

"*Strategic Monoliths and Microservices* is a great resource to connect business value to an evolvable enterprise architecture. I am impressed with how the authors use their deep understanding and experience to guide informed decisions on the modularization journey. Along the way every valuable tool and concept is explained and properly brought into context. Definitely a must-read for IT decision makers and architects. For me this book will be an inspiring reference and a constant reminder to seek the purpose in architecture. The Microservices discussion has reached a completely new maturity level."

—*Christian Deger, Head of Architecture and Platform at RIO | The Logistics Flow, organizer of over 60 Microservices Meetups*

"The choice of microservices or monoliths architecture goes far beyond technology. The culture, organization, and communication that exist within a company are all important factors that a CTO must consider carefully in order to successfully build digital systems. The authors explain this extremely well from various perspectives and based on very interesting examples."

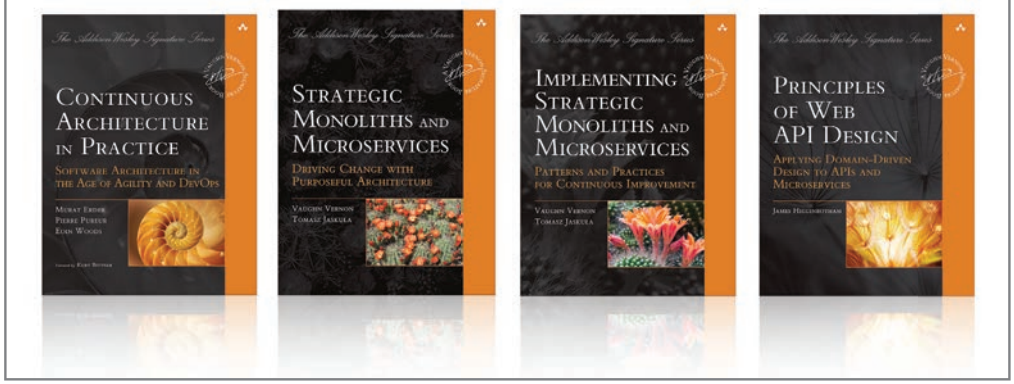
—*Olivier Ulmer, CTO, Groupe La Française*

"Building a technology engine to move quickly, experiment, and learn is a competitive advantage in today's digital world. Will 'de-jour architecture' help with this endeavor? This amazing book by Vaughn and Tomasz fills a void in the market and re-focuses on the core objectives of software architecture: move fast, experiment, focus on the outcomes that bring value. A reader will come away better suited to decide whether microservices architecture and all the complexity with it is right for them."

—*Christian Posta, Global Field CTO, Solo.io*

Strategic Monoliths and Microservices

Pearson Addison-Wesley Signature Series



Visit informit.com/awss/vernon for a complete list of available publications.

The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Vaughn Vernon is a champion of simplifying software architecture and development, with an emphasis on reactive methods. He has a unique ability to teach and lead with Domain-Driven Design using lightweight tools to unveil unimagined value. He helps organizations achieve competitive advantages using enduring tools such as architectures, patterns, and approaches, and through partnerships between business stakeholders and software developers.

Vaughn's Signature Series guides readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

Pearson Addison-Wesley

Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source

Strategic Monoliths and Microservices

Driving Innovation Using Purposeful Architecture

Vaughn Vernon
Tomasz Jaskuła

Addison-Wesley

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

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 the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021943427

Copyright © 2022 Pearson Education, Inc.

Cover image: John I Catlett/Shutterstock

Figures 1.5, 1.6, 1.7: illustration by grop/Shutterstock

Pages 109 and 152: Dictionary definitions from Merriam-Webster. Used with Permission.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions/.

ISBN-13: 978-0-13-735546-4
ISBN-10: 0-13-735546-7

ScoutAutomatedPrintCode

Contents

Foreword

We met the founders of Iterate in April 2007. Three of them had attended our first workshop in Oslo and invited us out to dinner. There, we learned they had just quit their jobs at a consulting firm and founded their own, so they could work in a place they loved using techniques they believed in. I thought to myself, "Good luck with that." After all, they were just a few years out of college and had no experience running a business. But I kept my skepticism to myself as we talked about how to find good customers and negotiate agile contracts.

We visited Iterate many times over the next decade and watched it grow into a successful consulting firm that was routinely listed as one of Norway's best places to work. They had a few dozen consultants and evolved from writing software to coaching companies in Test-Driven Development to helping companies innovate with design sprints. So I should have seen it coming, but when they decided to transform the company in 2016, I was surprised.

We decided to change course, they told us. We want to be a great place to work, where people can reach their full potential, but our best people are limited as consultants. They are always pursuing someone else's dream. We want to create a company where people can follow their own passion and create new companies. We want to nurture startups and fund this with our consulting revenue.

Once again I thought to myself, "Good luck with that." This time I did not keep my skepticism to myself. We talked about the base failure rate of new ventures and the mantra from my 3M days: "Try lots of stuff and keep what works." That's a great motto if you have a lot of time and money, but they had neither. One of the founders was not comfortable with the new approach and left the company. The others did what they had always done—move forward step-by-step and iterate toward their goal.

It was not easy and there were no models to follow. Wary of outside funding, they decided to merge the diametrically opposed business models of consulting and venture funding by limiting to 3% the amount of profit they could make from consulting, pouring the rest back into funding ventures. They had to make sure that consultants did not feel like second-class citizens and those working on new ventures were committed to the success of the consulting business. And they had to learn how to successfully start up new businesses when all they'd ever started was a consulting business.

It's been five years. Every year we visited to brainstorm ideas as the company struggled to make their unique approach work. When the pandemic hit, not only did their consulting business grind to a halt, but the farm-to-restaurant business they had nurtured for three years had no restaurants left to buy local farm goods. But think about it: Iterate had top talent with nothing to do and a venture that was poised to collect and deliver perishable goods. It took two weeks to pivot—they offered the food to consumers for curbside pickup—and the venture took off. While most Oslo consulting firms suffered in 2020, Iterate saw one venture (last-mile delivery) exit through a successful acquisition and three others spin off as separate entities, including a ship-locating system and a three-sided platform for knitters, yarn suppliers, and consumers. As a bonus, Iterate was number 50 on *Fast Company*'s 2020 list of Best Workplaces for Innovators, ahead of Slack and Square and Shopify.

So how did Iterate succeed against all odds? They started by realizing that a consulting approach to software development did not give them the freedom to take a lead role. With software becoming a strategic innovation lever, they felt it was time to claim a seat at the decision-making table. This was scary, because it involved taking responsibility for results—something consultants generally avoid. But they were confident that their experimental approach to solving challenging problems would work for business problems as well as technical problems, so they forged ahead.

You might wonder what Iterate's transformation has to do with the enterprise transformations that are the subject of this book. There is nothing in Iterate's story about Monoliths or Microservices or agile practices—but these are not the essence of a transformation. As this book points out, a transformation begins with the articulation of a new and innovative business strategy, one that provides real, differentiated value to a market. The pursuit of that strategy will be a long and challenging journey, requiring excellent people, deep thinking, and plenty of learning along the way. For those who are starting out on such a transformation, this book provides a lot of thinking tools for your journey.

For example, as you head in a new direction, you probably do not want to blow up the structures that brought your current success, however outmoded they might be. You need that old Big Ball of Mud Monolith (or consulting services) to fund your transition.

Another example: The first thing you want to consider is the right architecture for your new business model, and it probably won't be the same as the old one. Just as Iterate moved from having a pool of consultants to having clearly distinct venture teams, you will probably want to structure your new architecture to fit the domain it operates in. This usually means clarifying the business capabilities that fit the new strategy and structuring complete teams around these capabilities. So instead of

having a layered architecture, you are likely to want one based on the natural components and subcomponents of your product (also known as *Bounded Contexts*).

Think of SpaceX: The architecture of a launch vehicle is determined by its components—first stage (which contains nine Merlin engines, a long fuselage, and some landing legs), interstage, second stage, and payload. Teams are not formed around engineering disciplines (e.g., materials engineering, structural engineering, software engineering), but rather around components and subcomponents. This gives each team a clear responsibility and set of constraints: Teams are expected to understand and accomplish the job their component must do to ensure the next launch is successful.

As you clarify the product architecture in your new strategy, you will probably want to create an organization that matches this architecture because, as the authors point out, you can't violate Conway's Law any more than you can violate the law of gravity. The heart of this book is a large set of thinking tools that will help you design a new architecture (quite possibly a modular Monolith to begin with) and the organization needed to support that architecture. The book then offers ways to gradually move from your existing architecture toward the new one, as well as presents ideas about when and how you might want to spin off appropriate services.

Over time, Iterate learned that successful ventures have three things in common:

- Good market timing
- Team cohesion
- Technical excellence

Market timing requires patience; organizations that think transformations are about new processes or data structures tend to be impatient and generally get this wrong. Transformations are about creating an environment in which innovation can flourish to create new, differentiated offerings and bring them to market at the right time.

The second element of success, team cohesion, comes from allowing the capabilities being developed (the Bounded Contexts) and the relevant team members to evolve over time, until the right combination of people and offering emerges.

The third element, technical excellence, is rooted in a deep respect for the technical complexity of software. This book will help you appreciate the complexity of your existing system and future versions of that system, as well as the challenge of evolving from one to the other.

The Iterate story contains a final caution: Your transition will not be easy. Iterate had to figure out how to meld a consulting pool with venture teams in such a way

that everyone felt valuable and was committed to the organization's overall success. This is something that every organization will struggle with as it goes through a transition. There is no formula for success other than the one offered in this book: highly skilled people, deep thinking, and constant experimentation.

There is no silver bullet.

—*Mary Poppendieck, co-author of* Lean Software Development

Preface

Chances are good that your organization doesn't make money by selling software in the "traditional sense," and perhaps it never will. That doesn't mean that software can't play a significant role in making money for your organization. Software is at the heart of the wealthiest companies.

Take, for example, the companies represented by the acronym FAANG: Facebook, Apple, Amazon, Netflix, and Google (now held by Alphabet). Few of those companies sell any software at all, or at least they do not count on software sales to generate the greater part of their revenues.

Approximately 98% of Facebook's money is made by selling ads to companies that want access to the members of its social networking site. The ad space has such high value because Facebook's platform provides for enormous engagement between members. Certain members care about what is happening with other members and overall trends, and that keeps them engaged with people, situations, and the social platform. Capturing the attention of Facebook members is worth a lot of money to advertisers.

Apple is for the most part a hardware company, selling smartphones, tablets, wearables, and computers. Software brings out the value of said smartphones and other devices.

Amazon uses a multipronged approach to revenue generation, selling goods as an online retailer; selling subscriptions to unlimited e-books, audio, music, and other services; and selling cloud computing infrastructure as a service.

Netflix earns its revenues by selling multilevel subscriptions to movie and other video streaming services. The company still earns money through DVD subscriptions, but this part of the business has—as expected—fallen off sharply with the rising popularity of on-demand streaming. The video streaming is enhanced for, and controlled by, the experience with user-facing software that runs on TVs and mobile devices. Yet, the real heavy lifting is done by the cloud-based system that serves the videos from Amazon's AWS. These services provide video encoding in more than 50 different formats, serving up content through content delivery networks (CDN) and dealing with chaotic failures in the face of cloud and network outages.

Google also makes its money through ad sales; these ads are served along with query results from its search engine software. In 2020, Google earned approximately $4 billion from direct software usage, such as via Google Workspace. But

the Google Workspace software does not have to be installed on user computers, because it is provided in the cloud using the Software as a Service (SaaS) model. According to recent reports, Google owns nearly 60% of the online office suite market, surpassing even the share claimed by Microsoft.

As you can see from these industry leaders' experiences, your organization doesn't need to sell software to earn market-leading revenues. It will, however, need to use software to excel in business both now and over the years to follow.

What is more, to innovate using software, an organization must recognize that a contingent of software architects and engineers—the best—matter. They matter so much that the demand for the best makes them ridiculously difficult to hire. Think of the significance of landing any one of the top 20 picks in the WNBA or NFL draft. Of course, this description does not apply to every software developer. Many or even most are content to "punch a clock," pay their mortgage, and watch as much of the WNBA and NFL on TV as they possibly can. If those are the prospects you want to recruit, we strongly suggest that you stop reading this book right now. Conversely, if that's where you've been but now you want to make a meaningful change, read on.

For those organizations seeking to excel and accelerate their pace of innovation, it's important to realize that software development achievers are more than just "valuable." If a business is to innovate by means of software to the extent of ruling its industry, it must recognize that software architects and engineers of that ilk are "The New Kingmakers," a term coined by Stephen O'Grady in his 2013 book *The New Kingmakers: How Developers Conquered the World* [New-Kingmakers]. To truly succeed with software, all businesses with audacious goals must understand what drives this ilk of developer to transcend common software creation. The kinds of software that they yearn to create are in no way ordinary or obvious. The most valuable software developers want to make the kind of software that determines the future of the industry, and that's the recruiting message your organization must sound to attract (1) the best and (2) those who care enough to become the best.

This book is meant for C-level and other business executives, as well as every role and level involved in leading software development roles. Everyone responsible for delivering software that either directly results in strategic differentiation, or supports it, must understand how to drive innovation with software.

The authors have found that today's C-level and other executives are a different breed than their predecessors from decades past. Many are tech savvy and might even be considered experts in their business domain. They have a vision for making things better in a specific place, and they attract other executives and deeply technical professionals who grok what the founder or founders are driving to accomplish:

- CEOs who are close to the technology vision, such as startup CEOs, and those who want to be informed about the role of software in their future

- CIOs who are responsible for facilitating and enabling software development as a differentiator
- CTOs who are leading software vision through innovation
- Senior vice presidents, vice presidents, directors, project managers, and others who are charged with carrying the vision to realization
- Chief architects, who will find this book inspiring and a forceful guide to motivate teams of software architects and senior developers to drive change with a business mindset and purposeful architecture
- Software architects and developers of all levels, who are trying to firmly fix a business mentality in themselves—that is, a recognition that software development is not merely a means to a good paycheck, but to prospering beyond the ordinary and obvious through software innovation

This is a vital message that all software professionals must learn from by consuming, ruminating on, and practicing the expert techniques explored in this book.

Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture is not a book on implementation details. We'll provide that kind of information in our next book, *Implementing Strategic Monoliths and Microservices* (Vernon & Jaskuła, Addison-Wesley, forthcoming). This volume is very much a book on software as part of business strategy.

This book is definitely of interest to leaders who lack deep knowledge or experience in the software industry. It informs by showing how every software initiative must discover big ideas, architect with purpose, design strategically, and implement to defeat complexity. At the same time, we vigorously warn readers to resist dragging accidental or intentional complexity into the software. The point of driving change is to deliver software that works even better than users/customers expect. Thus, this book is meant to shake up the thinking of those stuck in a rut of the status quo, defending their jobs rather than pushing forward relentlessly as champions of the next generation of ideas, methods, and devices—and perhaps becoming the creators of the future of industry as a result.

The authors of this book have worked with many different clients and have seen firsthand the negative side of software development, where holding on to job security and defending turf is the aim rather than making the business thrive by driving prosperity. Many of the wealthiest companies are so large, and are engaged in so many initiatives under many layers of management and reporting structure, that their vision-to-implementation-to-acceptance pathway is far from a demonstration of continuity. With that in mind, we're attempting to wake the masses up to the fact that the adage "software is eating the world" is true. Our lessons are served up with

a dollop of realism, demonstrating that innovation can be achieved by means of progressive practical steps rather than requiring instantaneous gigantic leaps.

There is always risk in attempting innovation. That said, not taking any risk at all will likely be even more risky and damaging in the long run. The following simple graph makes this point very clear.

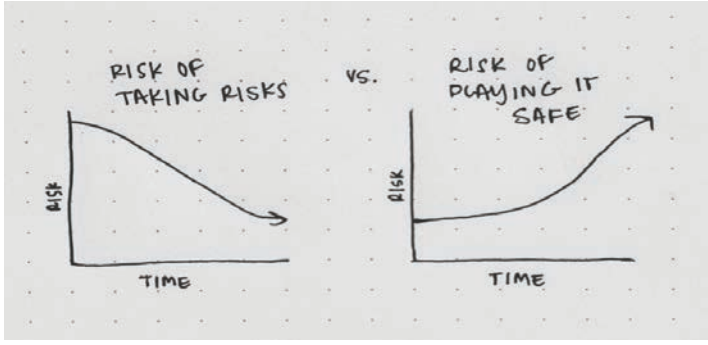


Figure P.1 *There is a risk in taking a risk, but likely even a greater risk in playing it safe.*

As Natalie Fratto [Natalie-Fratto-Risk] suggests, it is generally the case that the risk of taking risks diminishes over time, but the risk of playing it safe increases over time. The venture investor side of Natalie can be seen in her TED Talk [Natalie-Fratto-TED], which explains the kinds of founders in whose businesses she invests. As she explains, many investors seek business founders with a high intelligence quotient (IQ), whereas others look for entrepreneurs with a high emotional quotient (EQ). She looks primarily for those with a high adaptability quotient (AQ). In fact, innovation calls for a great amount of adaptability. You'll find that message repeated in this book in several forms. Everything from experimentation to discovery to architecture, design, and implementation requires adaptability. Risk takers are unlikely to succeed unless they are very adaptable.

As we discuss our primary topic of innovation with software, it's impossible to entirely avoid the highly controversial topic of iterative and incremental development. Indeed, some form of the "A-word"—yes, agile/Agile—cannot be sidestepped. This book stays far away from promoting a specific and ceremonial way to use Agile or to be a lean business. Sadly, the authors have found that most companies and teams creating software claim to *use* Agile, yet don't understand how to *be* agile. *The desire is to emphasize the latter rather than reinforce the former.* The original message of agile is quite simple: It's focused on collaborative delivery. If kept simple, this approach can be highly useful. That said, this is nowhere near our primary message. We attempt only to draw attention to where "un-simple" use

causes damage and how *being agile* helps. For our brief discussion on how we think *being agile* can help, see the section "Don't Blame Agile," in Chapter 1, "Business Goals and Digital Transformation."

Given our background, it might surprise some readers to learn that we do not view *Strategic Monoliths and Microservices* as a Domain-Driven Design (DDD) book. To be sure, we introduce and explain the domain-driven approach and why and how it is helpful—but we haven't limited our range. We also offer ideas above and beyond DDD. This is a "software is eating the world, so be smart and get on board, innovate, and make smart architectural decisions based on real purpose, before you are left behind" book. We are addressing the real needs of the kinds of companies with which we have been engaged for decades, and especially based on our observations over the past five to ten years.

We have been slightly concerned that our drumbeat might sound too loud. Still, when considering the other drums beating all around technology-driven industries, we think a different kind of drumming is in order. When many others are on high mountains, constantly beating the "next over-hyped products as silver bullets" drum, there must be at least an equalizing attempt at promoting our brains as the best tooling. Our goal is to show that thinking and rethinking is the way to innovate, and that generic product acquisition and throwing more technology at hard problems is not a strategic plan. So, think of us as the people on an adjacent mountain beating the other drum to "be scientists and engineers" by advancing beyond the ordinary and obvious, by being innovative and just plain different. And, yes, we definitely broke a sweat doing that. If our intense drumbeat leaves readers with a lasting impression that our drums made that specific brain-stimulating rhythm, then we think we've achieved our goal. That's especially so if the stimulation leads to greater success for our readers.

Legend/Key for Diagrams

Figure P.2 (on page xxii) shows the modeling elements used in most of the architecture diagrams in this book. The elements used range from large- to small-scale, and those in between, depending on the topic of the diagram. Some are taken from EventStorming described on page 87.

In Figure P.2, the top half, from the upper left, are strategic and architectural elements: Business/Bounded Context is a software subsystem and model boundary of a business capability and a sphere of knowledge; Big Ball of Mud is the "unarchitecture" in which most enterprises languish; Ports and Adapters Architecture is both a foundational and versatile style; and Modules are named packages that contain software components.

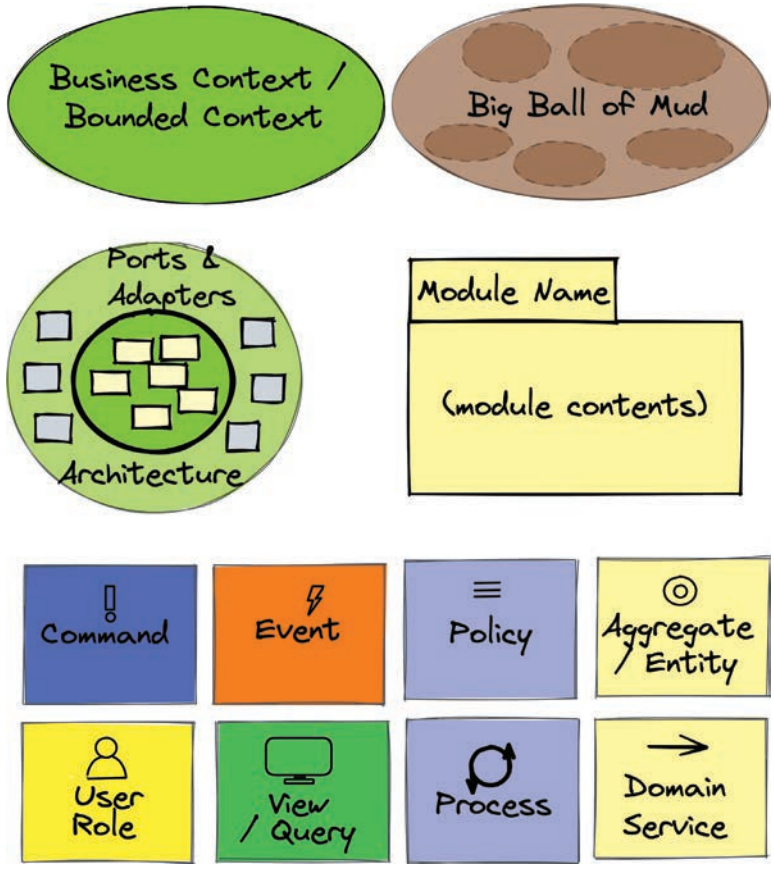


Figure P.2 *Modeling elements used in architecture diagrams throughout this book.*

The bottom half of Figure P.2 depicts eight tactical component types, occurring within a subsystem and that sometimes flow to other subsystems: Commands cause state transitions; Events capture and carry record of state transitions across subsystem boundaries; Policy describes business rules; Aggregate/Entity holds state and offers software behavior; User Role interacts with the system and often represents a persona; View/Query collects and retrieves data that can be rendered on user interfaces; Process manages a multi-step operation through to an eventual completion; and Domain Service provides cross-cutting software behavior.

Refer to Figure P.2 for the legend/key of element types, especially when reading the black-and-white print book, which uses patterns in lieu of colors.

References

[Natalie-Fratto-Risk] https://twitter.com/NatalieFratto/status/1413123064896921602

[Natalie-Fratto-TED] https://www.ted.com/talks/natalie_fratto_3_ways_to_measure_your_adaptability_and_how_to_improve_it

[New-Kingmakers] https://www.amazon.com/New-Kingmakers-Developers-Conquered-World-ebook/dp/B0097E4MEU

Register your copy of *Strategic Monoliths and Microservices* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780137355464) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

Acknowledgments

Writing a book is hard work. Readers might think that the more books written, the better the process is for the author. Multi-book authors would probably agree that the writing flows better as experience grows. Yet, most multi-book authors probably aim higher each time than they knew how to do previously. Knowing what lies ahead before the writing begins can be unnerving. The experienced author knows that each book has a life of its own and requires more mental energy and writing precision than even their own expectations could predict.

It happens every time, at least to one author involved in this effort. In the case of this book, one author knows what to fear and still did it anyway. The second author had translated a book from English to French, but his willingness to sign up for pure writing was based on the experienced author telling him not to worry.

That might be what a shark cage guide says just before the steel around the rank amateur plunges them into the breathtakingly cold waters off of Cape Town, South Africa. Truth is, the spectators who gaze upon great whites in action are fairly safe, at least by statistical accounts, because no one has ever died from that extreme viewing melee. Still, it's a good thing that sharks aren't as attracted to yellow and even brown in water as they are to blood. (We'll leave the close-call research to you.) So, trying to write a book probably won't kill you. Even so, have you ever wondered about the ratio between the people who have said they are going to write a book, but don't, and those who actually do write a book? It's probably similar to those who say they will one day dive with great white sharks and those who actually do.

It might take one, two, or a few people to author a book. But it takes an army to review, edit, edit, edit—add more edits—produce, and publish that book. The first draft manuscript of this book was considered "very clean," but there were still hundreds of additions, deletions, and general corrections made in every single chapter. And don't bring up the illustrations. Please. Even the very best of writers—which these authors would never claim to be—are subject to a daunting battery of "live rounds" before their book is ready for the public. Actually, we'll clarify that. That's the case if you are an author under the prestigious Addison-Wesley brand. (We won't go into the number of obvious errors you can find in the first few pages of books produced by other tech publishers.) The analogy of "live rounds" seems appropriate, because Pearson supports a small army of the best editors with the best aim that can be hired.

We are grateful to Pearson Addison-Wesley for giving us the opportunity to publish under their highly respected label. They have guided us through the process of writing this book until the publication was in sight. Special thanks go to our executive editor, Haze Humbert, for driving the process of acquisition, review, development, and full editorial production so smoothly, and coddling the process when an overly optimistic author didn't deliver all chapters as early as he anticipated. Haze's assistant editor, Menka Mehta, kept correspondence and calendars in sync and flowing. Our development editors Sheri Replin and Chris Cleveland offered high-level edits and prepared our chapters for page layout. Thanks to Rachel Paul for keeping the publication process clipping along. Thanks also to Jill Hobbs for being so kind as she made our "very clean" manuscript read superbly; it's amazing what a fine copy editor can do for a book, and especially a book written by tech authors. When you see things happening steadily but don't know how, it's probably due to a very competent director of product management, and in our case that is Julie Phifer.

In case it is not abundantly clear, the vast majority of editorial professionals with whom we work are women, and we think it is fair to include this team as "women in tech." If you are a woman in tech and want to be a book author, you can't hope for a better team to work with. These authors are not only proud to collaborate with this team, but highly honored that they have trusted us enough to be their extended members. So, future women authors, or future multi-book women authors, please allow me to introduce Haze Humbert, as your gateway to the best experience that book authoring can offer.

This book would not have been the same without the valuable feedback from our reviewers. In particular, we would like to thank Mary Poppendieck, who provided an extensive review of our book and offered rich feedback, and wrote a great foreword. Mary gave us her in-depth perspective on the difference between a software developer and a software engineer. Of course, any company can hire for the position of software engineer, but Mary describes a role that goes far beyond a title. Readers will find many of her viewpoints highlighted by sidebars and boxes, but her gifts to our project are in no way "side anything"—her input is nothing less than pure gold. Pay attention to what she has to say.

Other reviewers who offered particularly valuable reviews have served in such roles as CTO, chief architect, principal architect, and similar, and in a range of companies from very large to nimble startups. They are listed here in order by given (first) name: Benjamin Nitu, Eoin Woods, Frank Grimm, Olaf Zimmermann, Tom Stockton, and Vladik Khononov. There were several others who offered helpful feedback, including C-level executives, vice presidents, and other executives, who shall remain unnamed. We are honored to have gathered a group of highly experienced tech executives who were early readers, and we are thrilled that they were very impressed with our book. We would be remiss if we did not mention the many

people who offered to read and review our manuscript early on. We would have taken pleasure in that, but for various reasons it was not possible to include them. For every bit of help you provided and the confidence that you showed in us, thank you one and all.

Vaughn Vernon

This book would truly not have been possible without Haze Humbert. When Haze took over from my previous executive editor at Addison-Wesley, she actively suggested and discussed ideas for future books that I might write. Haze was very patient with me. After having three books published in roughly five years, I didn't look forward to authoring another one anytime soon. I wasn't burned out, just keenly aware of the commitment necessary to bring a new book to the world. And I was enjoying designing and creating software more than writing books. Being a creative person, during my discussions with Haze I pitched a number of ideas about which she could have laughed out loud. Yet, her kind demeanor and patience covered my audacious and/or ludicrous project pitches.

In early 2020, Haze offered an opportunity that was much more realistic, but completely unexpected and quite difficult to believe and digest, and whose acceptance seemed daunting. Her offer was to become the editor of my own Vaughn Vernon Signature Series. Knowing that my previous books had been successful—even best sellers—and appreciating that I could possibly achieve that feat again with another book, was far less earthshaking than fathoming the inception and delivery of a signature series. It was mind-blowing stuff. After a few weeks and several discussions with my trusted advisor, Nicole, the idea sank in. One thought that solidified the possibility of succeeding was this: If Pearson Addison-Wesley, with its unmatched experience as an elite publisher, thought enough of my work to make that offer, it meant that the company was confident that I would succeed. There's no way that such a publisher would pitch, invest in, and back this effort if it thought anything otherwise.

Based on that alone, not on my own abilities, I accepted. So here I am today, deeply thankful to Haze and the others with whom she works and represents. Thank you all so much.

I am grateful to Tomasz Jaskuła for accepting my offer to co-author this book with me. I hope the sharks didn't get too close for comfort. Tomasz is smart and tenacious, and has also been a worthy business partner in our training and consulting efforts. He's also done nearly all of the heavy lifting for the .NET implementation of our open source reactive platform, VLINGO XOOM.

Both of my parents have been a stabilizing force for me, and have taught me and supported my efforts for as long as I can remember. When I wrote my first published book, *Implementing Domain-Driven Design*, my parents were still full of life and mobile. More than eight years since, and after many months of lockdown have accumulated due to the pandemic, they now face additional challenges. It's a relief that in-person visits with them are once again permitted, and our time together is so enjoyable. Mom still has her witty sense of humor, and her stamina has not entirely abandoned her. I am happy that Dad still yearns for handheld computers, books, and other tools that enable him to remain in touch with engineering. I look forward to seeing his eyes light up when I drop by with a new gadget or book. Mom and Dad, I can't thank you enough.

I can't say enough about the ongoing support from my wife and our son. As crazy as the past 18 months or so have been, we've managed to grow together under continually changing circumstances. Nicole has been incredibly resilient through what seemed like unavoidable damage to our businesses. Despite the challenges, she has led us to new highs in the growth of both our training and consulting company, Kalele, and our software product startup VLINGO. VLINGO XOOM, our open source reactive platform and our initial product, is healthy and its adoption is growing. VLINGO is also building two new SaaS products. Not only have our teams been effective, but Nicole's business savvy has only expanded under greater challenges. It is inconceivable that I could have succeeded with anything at all, let alone a signature series and new book authoring, without her.

Tomasz Jaskuła

Back in 2013, Vaughn Vernon authored an outstanding book, *Implementing Domain-Driven Design*. He followed that with a world tour of workshops under the same name. His was the first book in which Domain-Driven Design was described from a practical point of view, shedding light on many theoretical concepts that were previously misunderstood or unclear for years in the Domain-Driven Design community. When I first learned about Vaughn's IDDD Workshop, I didn't hesitate to attend as soon as possible. It was a time when I was applying Domain-Driven Design on different projects, and I couldn't miss the opportunity to meet one of the most prominent members of the community. So, in 2013 I met Vaughn in Leuven, Belgium, where one of the workshops took place. This was also where I met most of the Domain-Driven Design community influencers, who were there to learn from Vaughn! A few years later, I'm proud to have coauthored this book with Vaughn, who has become a friend. He has been supportive of me through the years and I'm really grateful for the confidence he has in me. Writing this book was a great

learning experience. Thank you, Vaughn, for all your help, your confidence in me, and your support.

I would also like to thank Nicole Andrade, who, with all the kindness in the world, has supported us through the effort of writing this book. She has played an important role in strengthening the friendship between Vaughn and me through the years, and I know she will continue to do so for years to come.

Writing the book without the support from my friend and business partner François Morin of our company, Luteceo, would have been much more difficult. His encouragement of my writing, and his willingness to take care of running the company while I was not available, gave me the space I needed to take on this project.

I would like to thank my parents Barbara and Stefan, who have always believed in me and supported me through my personal challenges. They taught me early the importance of being curious and learning continuously, which is one of the greatest pieces of advice I could have ever received.

Finally, I would not have been able to write this book without the unconditional support and love from my wife Teresa and my lovely daughters Lola and Mila. Their encouragement and support were essential for me to complete this book. Thank you so much.

This page intentionally left blank

About the Authors

Vaughn Vernon is an entrepreneur, software developer, and architect with more than 35 years of experience in a broad range of business domains. Vaughn is a leading expert in Domain-Driven Design and reactive architecture and programming, and champions simplicity. Students of his workshops are consistently impressed by the breadth and depth of what he teaches and his unique approaches, and as a result have become ongoing students attending his other well-known workshops. He consults and trains around Domain-Driven Design, reactive software development, as well as EventStorming and Event-Driven Architecture, helping teams and organizations realize the potential of business-driven and reactive systems as they transform their businesses from technology-driven legacy web implementation approaches. Vaughn is the author of four books, including the one you are now reading. His books and his Vaughn Vernon Signature Series are all published by Addison-Wesley.

Kalele: https://kalele.io

VLINGO: https://vlingo.io

Twitter: @VaughnVernon

LinkedIn: https://linkedin.com/in/vaughnvernon/

Tomasz Jaskuła is CTO and co-founder of Luteceo, a software consulting company in Paris. Tomasz has more than 20 years of professional experience as a developer and software architect, and worked for many companies in the e-commerce, industry, insurance, and financial fields. He has mainly focused on creating software that delivers true business value, aligns with strategic business initiatives, and provides solutions with clearly identifiable competitive advantages. Tomasz is also a main contributor to the OSS project XOOM for the .NET platform. In his free time, Tomasz perfects his guitar playing and spends time with his family.

Twitter: @tjaskula

LinkedIn: https://linkedin.com/in/tomasz-jaskula-16b2823/

This page intentionally left blank

Chapter 1

Business Goals and Digital Transformation

The most outstanding business achievement is to create a product that is needed by a great number of consumers, is completely unique, and is optimally priced. Historically, and in a general sense, the realization of such an accomplishment has depended on the ability to identify what is essential or highly desirable for a key market demographic. This is reflected in a maxim captured by the writings of Plato: "Our need will be the real creator." Today, this statement is better known as "Necessity is the mother of invention."

Yet, the most profound innovators are those who invent an ingenious product even before consumers realize it is needed. Such achievements have occurred serendipitously, but have also been born from those daring enough to ask, "Why not?"[1] Perhaps mathematician and philosopher Alfred North Whitehead struck on this notion when he argued that "the basis of invention is science, and science is almost wholly the outgrowth of pleasurable intellectual curiosity" [ANW].

Of course, the vast majority of businesses face a stark reality: Breakthroughs in product development that lead to far-reaching market impact aren't an everyday happening. Inventing entirely unique products that capture whole markets might seem as likely as aiming at nothing and hitting the center of a pot of gold.

As a result, the predominant business plan is to create competition. The uniqueness is seen in pricing the replica rather than in creating the original. Hitting such a large target is entirely ordinary and lacking in imagination and is not even a sure means of success. If creating (more) competition seems to be the best play, consider Steve Jobs's advice: "You can't look at the competition and say you're going to do it better. You have to look at the competition and say you're going to do it differently."

1. (George) Bernard Shaw: "Some men see things as they are and ask why. Others dream things that never were and ask why not."

SpaceX Innovation

Between the years 1970 and 2000, the cost to launch a kilogram to space averaged $18,500 US per kilogram. For a SpaceX Falcon 9, the cost is just $2,720 per kilogram. That's a factor of 7:1 improvement, and so it's no secret why SpaceX has almost all of the space launch business these days. How did they do it? What they did not do was work under contract to the government—that is, the only funding mechanism up until then. Their goal was to dramatically reduce the cost to launch stuff into space. Their main sub-goal under that was to recover and reuse booster rockets. There's a wonderful YouTube video of all the boosters they crashed in order to achieve their goal. The strategy of integrating events (in this case, test booster launches) is how multiple engineering teams rapidly try out their latest version with all the other teams. Government contracts would never have tolerated the crashes that SpaceX suffered. Yet, the crashes speeded up the development of a reliable, cheap booster rocket by perhaps a factor of 5, simply by trying things out to discover the unknown unknowns, instead of trying to think everything through in excruciating detail. That is a pretty classic engineering approach, but would never be allowed in a contracting model. The SpaceX team said it was far cheaper to have crashes and find the problems than to try to wait forever until there was no risk. [Mary Poppendieck]

Imitation is not a strategy. Differentiation is.

Differentiation is the strategic business goal that must be constantly sought after. If pure invention seems nearly impossible, continuous and tenacious improvement toward innovation should not. In this book, we have undertaken the task of helping readers achieve strategic business differentiation through relentless improvement in digital transformation.

Digital Transformation: What Is the Goal?

Understanding that the making of the unordinary is a major feat should not dissuade anyone from taking small, scientific steps with ongoing determination toward actual innovation. No matter the complexity in reaching Z, performing the science of experimentation to arrive at B when starting from A is a realistic expectation. After that, reaching C is doable, which then leads to D. It's a matter of keeping our lab coat and pocket protector on, and acknowledging that unique products that can capture new markets have likely been staring us in the face all along.

Whether Microsoft Office was considered a worker-productivity innovation from the outset, it certainly has been the most successful suite in that market. With Office 365, Microsoft didn't have to reinvent the word processor and the spreadsheet to innovate. It did, however, add a new delivery mechanism and capabilities to enable full teams to collaborate, among other features. Did Microsoft win yet again by innovating through digital transformation?

Digital transformation is left to the eye of the business innovator, but commonly businesses lose sight of the innovation part of transformation. Transformative innovation requires that the business understands the difference between changing infrastructural platforms and building new product value. For example, although taking business digital assets from the on-premises datacenter to the cloud might be an important IT initiative, it is not in itself a business initiative in innovation.

Does migrating your software to the cloud qualify as a digital transformation? Possibly, but more so if the move supports future differentiation. It best qualifies if the cloud delivers new opportunities to innovate or at least to unburden the extremely high cost of digital asset operations and channel those funds to new products. Think of the cloud as creating opportunities by freeing you from most traditional datacenter responsibilities. It won't be transformative, however, if the shift to the cloud amounts to trading one set of costs for a different set of costs. Amazon offering its already successful computing infrastructure to the outside world was a digital transformation for the company that resulted in cloud innovation. Paying a subscription to Amazon to use its cloud is not a transformative innovation to the subscriber. The lesson is clear: Innovate or be innovated on.

Just as migrating to the cloud is not an innovation, neither is creating a new distributed computing architecture. Users don't care about distributed computing, Microservices, or Monoliths, or even features. Users care about outcomes. Improved user outcomes are needed rapidly and without negatively impacting their workflows. For software to stand a chance at meaningful transformation, its architecture and design must support the delivery of better user outcomes as rapidly as possible.

When using the cloud, an improved architecture and design approach (and any additional well-tuned steps that lead to productivity gains) make reaching innovative transformational goals possible. Using infrastructure as a service frees the business to work on innovative business software rather than churning on trying to innovate on its infrastructure. Not only are infrastructure innovations time-consuming and costly, but they might not benefit the business's bottom line, and developing infrastructure in-house might never address infrastructure and operational needs as well as AWS, Google Cloud Platform, and Azure. Yet, this is not always the case. For some businesses, it would be much more cost-effective to bring operations in-house or keep them there [a16z-CloudCostParadox].

Remember, it's A to B, B to C, C to D. . . . Be willing to iterate on any of these steps so that you can learn enough to take the next one. Understanding that going

back from J to G before reaching K is expected, and that Z need not ever happen, is liberating. Teams can innovate, but none of these transformational steps can tolerate lengthy cycles. Chapter 2, "Essential Strategic Learning Tools," shows how experimentation is the friend of innovation and the enemy of indecision.

Software Architecture Quick Glance

This section introduces the term *software architecture*—a term that is referred to often herein. It's a rather broad topic that is covered in more detail throughout this book.

For now, think of software architecture as similar to building architecture. A building has structure, and it reflects the results of communication that has taken place between the architect and the owner regarding the design features, by providing the features as specified. A building forms a whole system of various subsystems, each of which has its own specific purpose and role. These subsystems are all loosely or more tightly connected with other parts of the building, working separately or in conjunction with others to make the building serve its purpose. For example, a building's air conditioning requires electrical power, duct work, a thermostat, insulation, and even a closed area of the building to cool, if that subsystem is to be effective.

Likewise, a software architecture provides structural design—that is, the formulation of many structures, not one. The structural design organizes the system components, affording them the means to communicate as they work together. The structure also serves to segregate clusters of components so they can function independently. The structures must, therefore, help achieve quality attributes rather than functional ones, while the components within implement the functionality specified by teams of system builders.

Figure 1.1 illustrates two subsystems (showing only a fragment of a whole system), each having components that work together internally but in isolation from the other subsystem. The two subsystems exchange information through a communication channel, with the box in between representing the information that is exchanged. Assume that these two subsystems are physically separated into two deployment units, and communicate via a network. This forms a portion of a distributed system.

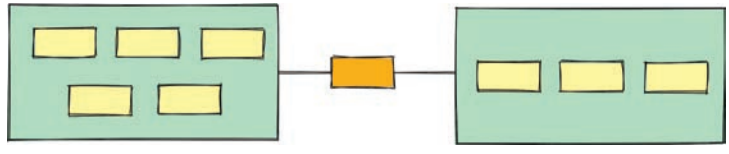


Figure 1.1 *A software architecture provides structure within subsystems and supports communication between them.*

Another important aspect of both building and software architecture is that they must support inevitable change. If existing components fail to meet new demands in either architecture, they must be replaceable without extreme cost or effort. The architecture must also be able to accommodate possible needed expansion, again without major impact to the overall architecture.

Why Software Goes Wrong

We don't want to overstate the seriousness of the poor state of enterprise software development, and we don't think it can be overstated.

When discussing enterprise software system conditions with *Fortune* and Global companies, we quickly learn about their major pain points. These are always related to aged software that has undergone decades of maintenance, long after innovation took place. Most discussions identify that software development is considered a cost center to the business, which makes it that much more difficult to invest in improvements. Today, however, software should be a profit center. Unfortunately, the collective corporate mindset is stuck 30-plus years back when software was meant to make some operations work faster than manual labor.

A specific application (or subsystem) starts with a core business reason to be built. Over time, its core purpose will be enhanced or even altered considerably. Continuous additions of features can become so extensive that the application's original purpose is lost and it likely means different things to different business functions, with the full diversity of those understandings not readily known. This often leads to many hands stirring the pot. Eventually the urgent development transitions from strategic to keeping the software running by fixing urgent bugs and patching data directly in the database in an effort to compensate for failures. New features are generally added slowly and gingerly in an attempt to avoid producing even more bugs. Even so, injecting new bugs is inevitable: With the ever-increasing level of system disorder and lost historical perspective, it's impossible to determine the full impact a single given change will have on the greater body of software.

Teams admit that there is no clear and intentional expression of software architecture, either in individual applications (subsystems) or even overall in any large system. Where some sense of architecture exists, it is generally brittle and obsolete given advances in hardware design and operational environments such as the cloud. Software design is also unintentional, and thus appears to be nonexistent. In consequence, most ideas behind an implementation are implicit, committed to the memories of a few people who worked on it. Both architecture and design are by and large ad hoc and just plain weird. These unrecognized failures make for some really sloppy results due to slipshod work.

Just as dangerous as producing no well-defined architecture at all is introducing architecture for merely technical reasons. A fascination often exists among software architects and developers with regard to a novel development style relative to what they previously employed, or even a newly named software tool that is the subject of a lot of hype and industry buzz. This generally introduces accidental complexity[2] because the IT professionals don't fully understand what impacts their ill-advised decisions will have on the overall system, including its execution environment and operations. Yes, Microservices architecture and tools such as Kubernetes, although duly applicable in the proper context, drive a lot of unqualified adoption. Unfortunately, such adoption is rarely driven by insights into business needs.

The prolonged buildup of software model inaccuracies within the system from failure to perform urgent changes is described as the *debt metaphor*. In contrast, the accumulation from accepting uncontrolled changes to a system is known as *software entropy*. Both are worth a closer look.

Debt Metaphor

Decades ago, a very smart software developer, Ward Cunningham, who was at the time working on financial software, needed to explain to his boss why the current efforts directed toward software change were necessary [Cunningham]. The changes being made were not in any way ad hoc; in fact, they were quite the opposite. The kinds of changes being made would make it look as if the software developers had known all along what they were doing, and serve to make it look like it was easy to do. The specific technique they used is now known as *software refactoring*. In this case, the refactoring was done in the way it was meant to be implemented—that is, to reflect the acquisition of new business knowledge into the software model.

To justify this work, Cunningham needed to explain that if the team didn't make adjustments to the software to match their increased learning about the problem domain, they would continue to stumble over the disagreement between the software that existed and their current, refined understanding. In turn, the continued stumbling would slow down the team's progress on continued development, which is *like paying interest on a loan*. Thus, the *debt metaphor* was born.

Anyone can borrow money to enable them to do things sooner than if they hadn't obtained the money. Of course, as long as the loan exists, the borrower will be

2. Accidental complexity is caused by developers trying to solve problems, and can be fixed. There is also essential complexity inherent in some software, which is caused by the problems being solved. Although essential complexity cannot be avoided, it can often be isolated in subsystems and components specifically designed to tackle them.

paying interest. The primary idea in taking on debt in the software is to be able to release sooner, but with the idea that you must pay the debt sooner rather than later. The debt is paid by refactoring the software to reflect the team's newly acquired knowledge of the business needs. In the industry at that time, just as it is today, software was rushed out to users knowing that debt existed, but too often teams had the idea that you never have to pay off the debt.

Of course, we all know what happens next. If debt continues to stack up and the person borrows more and more, all the borrower's money goes to paying interest and they reach a point where they have zero buying power. Matters work the same way with software debt, because eventually developers deep in debt will be severely compromised. Adding new features will take longer and longer, to the point where the team will make almost no progress.

One of the major problems with the contemporary understanding of the debt metaphor is that many developers think this metaphor supports deliberately delivering poorly designed and implemented software so as to deliver sooner. Yet, the metaphor doesn't support that practice. Attempting that feat is more like borrowing on subprime loans[3] with upward adjustable interest rates, which often results in the borrower becoming financially overextended to the point of defaulting. Debt is useful only as long as it is controlled; otherwise, it creates instability within the entire system.

Software Entropy

Software entropy[4] is a different metaphor but closely related to the debt metaphor in terms of the software system conditions it describes. The word *entropy* is used in statistical mechanics in the field of thermodynamics to measure a system's disorder. Without attempting to go too deep into this topic: "The second law of thermodynamics states that *the entropy of an isolated system never decreases over time*. Isolated systems spontaneously evolve towards thermodynamic equilibrium, the state with maximum entropy" [Entropy]. The software entropy metaphor names the condition of a software system where change is inevitable, and that change will cause increasing uncontrolled complexity unless a vigorous effort is made to prevent it [Jacobson].

3. It's difficult to comprehend that some are unfamiliar with the 2008 financial crisis that extended years into the future. This (ultimately global) crisis was triggered by subprime lending to unqualified borrowers for home purchases. Some early readers of the manuscript for this book asked, "What is a subprime loan?" Learning about that history could save those readers from a lot of financial grief.
4. Other analogs besides entropy also paint a vivid picture of the problem, such as software rot, software erosion, and software decay. The authors mostly use entropy.

Big Ball of Mud

An application or system like the one previously described has become known as a *Big Ball of Mud*. In terms of architecture, it has been further described as haphazardly structured; sprawling; sloppy; duct-taped-and-baling-wired; jungle; unregulated growth; repeated, expedient repair. "Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition" [BBoM].

It seems appropriate to describe the Big Ball of Mud "architecture" as the *un*architecture.

Throughout the remainder of this chapter, as well as in this book in general, we will key in on a few of these characteristics: haphazardly structured; unregulated growth; repeated, expedient repair; information shared promiscuously; all important information global or duplicated.

An enterprise norm of the Big Ball of Mud results in organizations experiencing competitive paralysis, which has spread across business industries. It is quite common for large enterprises, which once enjoyed competitive distinction, to become hamstrung by systems with deep debt and nearly complete entropy.

You can easily contrast the Big Ball of Mud system in Figure 1.2 to that depicted in Figure 1.1. Of course, the segment of the system in Figure 1.1 doesn't represent the number of features that are supported by the system in Figure 1.2, but clearly the architecture of the first system brings order, whereas the lack thereof in the second offers chaos.

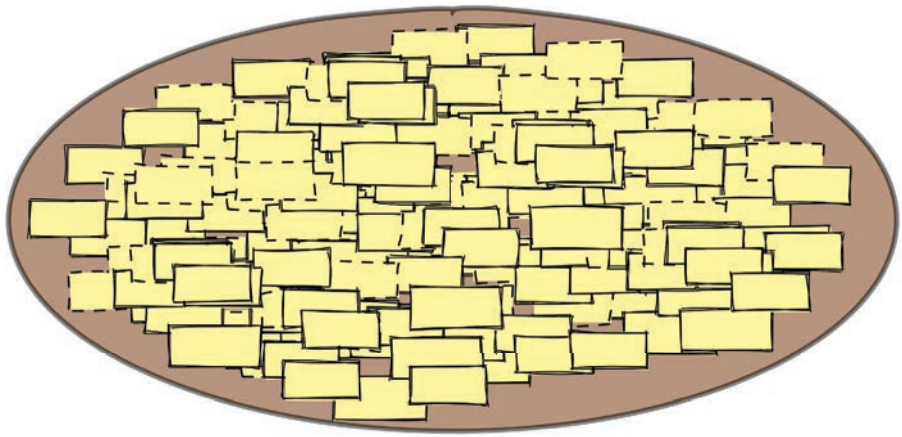


Figure 1.2 *The Big Ball of Mud might be classified as the unarchitecture.*

These chaotic conditions prevent more than a few software releases per year, which result in even worse problems than the software releases of previous years. Individuals and the teams to which they belong tend to become indifferent and complacent because they know they can't produce the change they see as necessary to turn things around. The next level from there is becoming disillusioned and demoralized. Businesses facing this situation cannot innovate in software and continue to compete under such conditions. Eventually they fall victim to a nimble startup that can make significant strides forward, to the point where within a few months to a few years, it can displace previous market leaders.

Running Example

From this point forward, we present a case study using an existing Big Ball of Mud and describe a situation where the affected business struggles to innovate as it faces the realities of the associated deep debt and entropy. Because you might already be tired of reading bad news, here's a spoiler: The situation improves over time.

There is no better way to explain the issues every company has to face with software development than with examples borrowed from the real world. The example offered here as a case study in dealing with an existing Big Ball of Mud comes from the insurance industry.

At some point in life, just about everyone has to deal with an insurance company. There are various reasons why people seek to obtain diverse insurance policies. Some are to address legal requirements, and some provide security measures for the future. These policies include protection for health, personal lines such as life, automobile, home, mortgage, financial products investments, international travel, and even the loss of a favorite set of golf clubs. Policy product innovation in the field of insurance seems endless, where almost any risk imaginable can be covered. If there is a potential risk, you're likely to find an insurance company that will provide coverage for it.

The basic idea behind insurance is that some person or thing is at risk of loss, and for a fee, the calculated financial value of the insured person or thing may be recovered when such loss occurs. Insurance is a successful business proposition due to the law of large numbers. This law says that given a large number of persons and things being covered for risks, the overall risk of loss among all of those covered persons and things is quite small, so the fees paid by all will be far greater than the actual payments made for losses. Also, the greater the probability of loss, the greater the fee that the insurance company will receive to provide coverage.

Imagine the complexity of the insurance domain. Is coverage for automobiles and homes the same? Does adjusting a few business rules that apply to automobiles make covering homes possible? Even if automobile and home policies might

be considered "close enough" to hold a lot in common, think of the different risks involved in these two policy types.

Consider some example scenarios. There is a much higher possibility of an automobile striking another automobile than there is of a part of a house striking another house and causing damage. The likelihood of a fire occurring in a kitchen due to normal everyday use is greater than the likelihood of the car's engine catching fire due to normal everyday use. As we can see, the difference between the two kinds of insurance isn't subtle. When considering the variety of possible kinds of coverage, it requires substantial investment to provide policies that have value to those facing risk and that won't be a losing proposition to the insurance company.

Thus, it's understandable that the complexity among insurance firms in terms of business strategy, operations, and software development is considerable. That is why insurance companies tend to specialize in a small subset of insurable products. It's not that they wouldn't want to be a larger player in the market, but rather that costs could easily outweigh the benefits of competing in all possible segments. It's not surprising, then, that insurance companies more often attempt to lead in insurance products in which they have already earned expertise. Even so, adjusting business strategies, accepting unfamiliar yet measured risks, and developing new products might be too lucrative an opportunity to pass up.

It is time to introduce NuCoverage Insurance. This fictitious company is based on real-world scenarios previously experienced by the authors. NuCoverage has become the leader in low-cost auto insurance in the United States. The company was founded in 2001 with a business plan to focus on providing lower-cost premiums for safe drivers. It saw a clear opportunity in focusing on this specific market, and it succeeded. The success came from the company's ability to assess risks and premiums very accurately and offer the lowest-cost policies on the market. Almost 20 years later, the company is insuring 23% of the overall US market, but nearly 70% in the specialized lower-cost safe-driver market.

Current Business Context

Although NuCoverage is a leader in auto insurance, it would like to expand its business to other kinds of insurance products. The company has recently added home insurance and is working on adding personal lines of insurance. However, adding new insurance products became more complex than was originally perceived.

While the development process of personal lines of insurance was ongoing, management had an opportunity to sign a partnership with one of the largest US banks, WellBank. The deal involves enabling WellBank to sell auto insurance under its own brand. WellBank sees great potential in selling auto insurance along with its familiar auto loans. Behind the WellBank auto insurance policies is NuCoverage.

Of course, there are differences between NuCoverage auto insurance products and the ones to be sold by WellBank. The most prominent differences relate to the following areas:

- Premiums and coverages
- Rules and premium price calculation
- Risk assessment

Although NuCoverage has never before experienced this kind of partnership, the business leaders immediately saw the potential to expand their reach, and possibly even introduce a completely new and innovative business strategy. But in what form?

Business Opportunity

NuCoverage's board of directors and executives recognized an even larger strategic opportunity than the WellBank partnership: They could introduce a *white-label*[5] insurance platform that would support any number of fledgling insurers. Many types of businesses might potentially support selling insurance products under the business's own brand. Each business best knows its customers and grasps what insurance products could be offered. The recently inked partnership with WellBank is just one example. NuCoverage can certainly identify other forward-thinking partners that would share the vision of selling insurance products under a white label.

For example, NuCoverage could establish partnerships with car makers that offer their own financing. When a customer purchases a car, the dealer could offer both financing and manufacturer-branded insurance. The possibilities are endless, due to the fact that any random company cannot easily become an insurance company, but can still benefit from the margins gained through insurance sales. In the long run, NuCoverage considered diversifying with new insurance products such as motorcycle, yacht, and even pet insurance.

This possibility seems very exciting to the board and executives, but when the software management team learned about the plans, a few of them had to swallow hard. The original auto insurance application was built quickly under a lot of pressure to deliver, which quickly led to a Big Ball of Mud Monolith. As Figure 1.3 illustrates, with more than 20 years of changes and deep unpaid debt, and the ongoing development of the system for the personal lines of insurance, the teams have reached a point of stifling unplanned complexity. The existing software will absolutely not support current business goals. All the same, development needs to answer the call.

5. A white-label product is a product or service produced by one company (the producer) that other companies (the marketers) rebrand to make it appear as if they had made it.

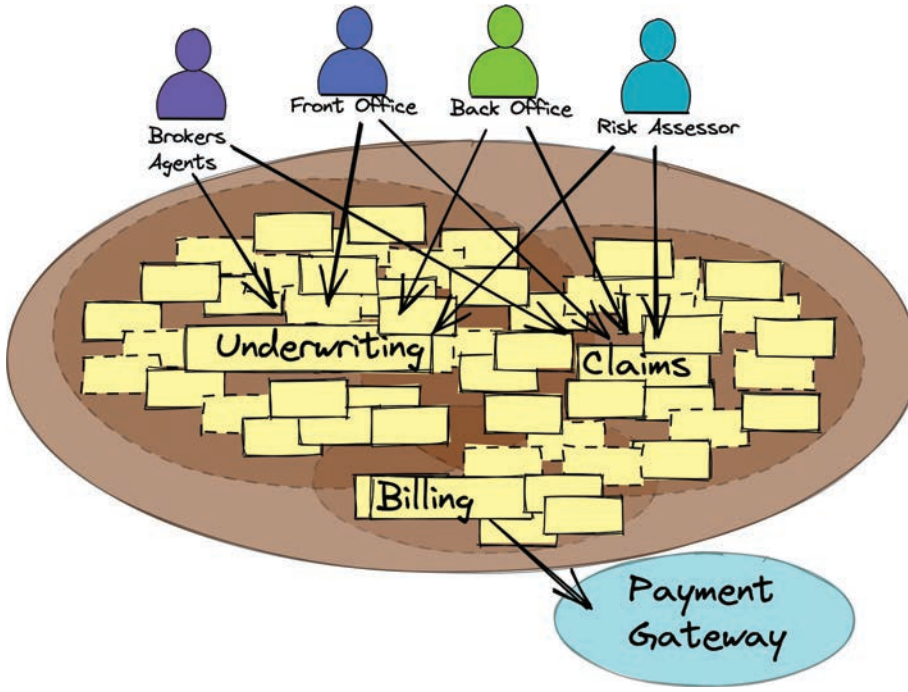


Figure 1.3 *The NuCoverage Big Ball of Mud. All business activities are intertwined with tangled software components that are in deep debt and near maximum entropy.*

What NuCoverage must understand is that its business is no longer insurance alone. It was always a product company, but its products were insurance policies. Its digital transformation is leading the firm to become a technology company, and its products now include software. To that end, NuCoverage must start thinking like a technology product company and making decisions that support that positioning—not only as a quick patch, but for the long term. This is a very important shift in the company mindset. NuCoverage's digital transformation cannot be successful if it is driven only by technology choices. Company executives will need to focus on changing the mindset of the organization's members as well as the organizational culture and processes before they decide what digital tools to use and how to use them.

Your Enterprise and Conway's Law

A long time ago (well, in 1967), in a galaxy not far away (our own), another really smart software developer presented an unavoidable reality of system development.

It's so unavoidable that it has become known as a law. The really smart developer is Mel Conway, and the unavoidable reality is known as Conway's Law.

> Conway's Law: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" [Conway].

The correlation to the preceding description of Big Ball of Mud is fairly obvious. It's generally a matter of broken communication that causes the "haphazardly structured; unregulated growth; repeated, expedient repair."

Still, there is another big communication component that's almost always missing: the business stakeholders and the technical stakeholders having productive communication that leads to deep learning, which in turn leads to innovation.

> Assertion: Those who want to build good software that innovates must get this communication–learning–innovation pathway right before trying anything else.

Funny things, these laws. Is it possible to "get better" at a law? For example, humans can't really "get better" at the law of gravity. We know that if we jump, we will land. The law and our earth's gravitational influence even enable us to calculate how much hang time anyone who jumps can possibly have. Some people can jump higher and farther, but they are still subject to the same law of gravity as everyone else on earth.

Just as we don't get better at the law of gravity, we don't really get better at Conway's Law. We are subject to it. So how do we get Conway's Law, right? By training ourselves to be better at dealing with the unavoidable realities of this law. Consider the challenges and the possibilities.

Communication Is about Knowledge

Knowledge is the most important asset in every company. An organization cannot excel at everything, so it must choose its core competencies. The specific knowledge a company acquires within its domain of expertise enables building competitive advantage.

Although a company's knowledge can be materialized in physical artifacts such as documentation, and in models and algorithms by means of source code implementations, these are not comparable to the collective knowledge of its workers. The greater part of the knowledge is carried by individuals in their minds. The knowledge that has not been externalized is known as *tacit knowledge*. It can be collective, such as the routines of unwritten procedures within the business, or

the personal preferred ways of working that every individual possesses. Personal knowledge is about skills and crafts—the undocumented trade secrets and historical and contextual knowledge that a company has collected since its founding.

People inside the organization exchange knowledge through effective communication. The better their communication is, the better the company's knowledge sharing will be. Yet, knowledge is not just shared statically as if feeding encyclopedic input with no other gain. Sharing knowledge with an achievement goal in mind results in learning, and the experience of collective learning can result in breakthrough innovation.

Knowledge Is Not an Artifact

Because knowledge is not something that one person passes to another in the same way that a physical object is exchanged, the knowledge transfer takes place as a combination of *sense-giving* and *sense-reading*, as illustrated in Figure 1.4 [Polanyi].

Sense-giving occurs when a person communicates knowledge. The knowledge is structured into information and externalized [LAMSADE]. The person on the receiving side undergoes the process of sense-reading. This individual extracts data from the information received, creating personal knowledge and internalizing it. The probability that two people will give the same meaning to the same information is determined not just by the accuracy of the communication that has occurred between those individuals, but also by past experiences and the specific contexts in which the receiver places it.

It is not guaranteed that the piece of information that someone receives is exactly what another person wants to communicate. This is illustrated with a concrete example.

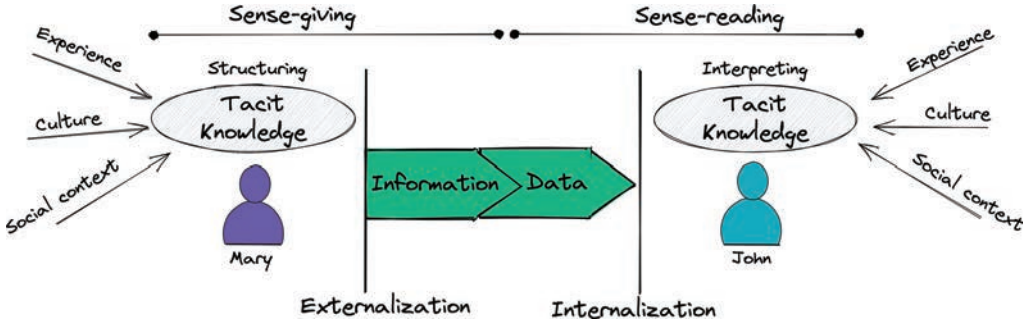


Figure 1.4 *Tacit knowledge transfer through the process of sense-giving and sense-reading.*

The Telephone Game

The Telephone Game illustrates the trouble with certain communication structures. You might know this game by another name, but the rules are the same. People form a line, and at one end of the line a person whispers a message to the next person in line, which is then repeated to the next person, and so forth, until the message reaches the last person in the line. Finally, the last message receiver tells everyone the message that they received, and the person at the beginning of the line discloses the original message. Of course, the fun comes from the repeated message becoming badly distorted by the time it reaches the end.

What's most interesting about this game and the effects on communication is that the distortion occurs at every separate point of communication. Everyone in the line, even those closest to the message's origin, will be told something that they can't repeat accurately. The more points of relay, the more distorted the message becomes.

In essence, every point of relayed communication creates a new translation. This highlights the reality that even communication between just two people is difficult. It's not unfeasible to reach clarity and agreement, but it can be challenging to get there.

When this happens in business, it's not a game, and it isn't fun. And, of course, the more complex the message, the more likely it is for greater degrees of inaccuracy to occur. As Figure 1.5 shows, there are often many points of relay. In very large

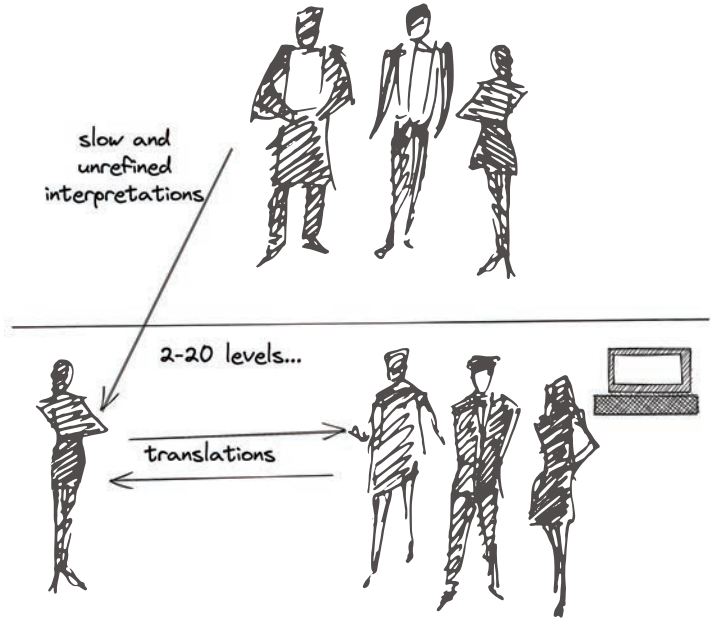


Figure 1.5 *Typical communication structure from C-level to project managers to developers.*

organizations, there might be even more than 20 levels. The authors often hear of so much hierarchy that it seems insuperably difficult for anything in the organization to be accomplished with any degree of accuracy, and the software developers near the end of the line marvel at it.

Reaching Agreement Is Hard

The negative feelings of team members, such as indifference, complacency, disillusionment, and demoralization, can be overcome. It's done by helping the team create reachable goals and providing new, lightweight techniques, such as shaping the team for better communication and engaging in stepwise, value-driven restructuring of the software.

Yet, the separations between points of communication and even the style of communication at each level of hierarchy can cause a widening gap in business and technical stakeholders. When the communication gap is broad in the face of big changes, agreement is hard to achieve.

A noxious problem exists when technical leadership see themselves and their teams as threatened by criticism of their work and hints that big change is imminent. After all, the distorted message being heard intimates that what has existed for a long time isn't sustainable. As has been noted more than a few times throughout history, humans have egos and more often than not are heavily invested in what they have produced as a result of hard work. This strong attachment is often referred to as being "married." When an institution as tightly connected as marriage seems breakable, the involved parties often adopt a defensive posture that not only tightly grips *what* has been done, but also clings to *how* things have been done. Moving beyond that hardened stance isn't easy.

There are also those from outside who strongly recommend the kind of changes that are incompatible with business as usual. This apparent adversary hasn't gone through the decades of hard work under conflicting time forces that are blamed for the deep software debt and entropy that today throbs like two sore thumbs. All of these uncomfortable perceptions accumulate into a pressure cooker of emotions and shouts of "Executive betrayal!" in the conscious thoughts of technical leadership. It's obvious that the responsible parties have been singled out and will now be repaid for ongoing delivery under unrelenting impossible circumstances with a swift shove under a speeding bus.

When technical leadership has these misgivings, they typically multiply their doubts by confiding in at least a few members of their teams who will support their concerns. Naturally those supportive team members themselves confide in others, and the fear leads to widespread resistance.

But Not Impossible

This whole problem is most often perpetuated by a company culture known as "us versus them." This happens, once again, because of deficient communication structures. Glancing back at Figure 1.5, do you see a big problem? It's the hierarchy, which breeds an "us versus them" mentality. Edicts come down from on high and subordinates carry out the orders. If this hierarchy is retained, executives shouldn't expect outcomes that lead to cooperative change.

Cooperative change must emanate from leadership, which begins at the executive level. When executive leadership can see the untenable result of hierarchical command and control, the answer is not to replace the old controlled with the newly controlled.

In every endeavor, teams are by far more successful at large undertakings than are individuals. Mature sports teams succeed by crafting innovative playbooks and communicating each play to the whole team with tedious precision.

Acting like a team requires being a team. In teams, communication is not one way. Any one team member can have enough experience to suggest that something was overlooked in the playbook, or that a given play could be better with this or that additional move or removal of an inefficiency. When every team member is respected for their competency and experienced viewpoint, it serves to make communication that much more effective (Figure 1.6).

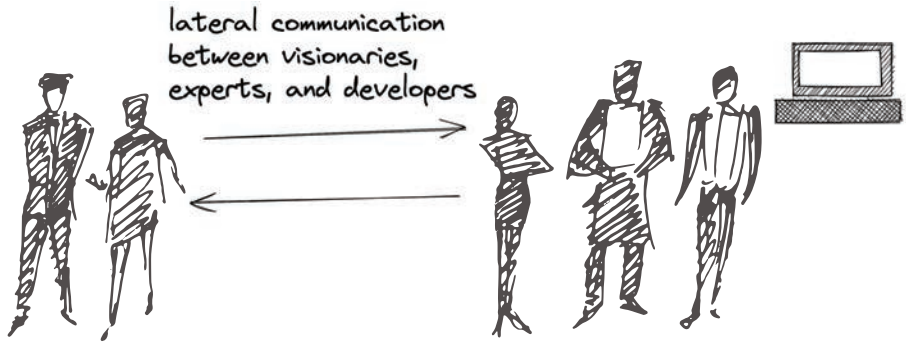


Figure 1.6 *Optimal communication structures are the result of team play.*

Consider these keys to optimal communication:

- It's us, not us and them.
- Servant leadership must not be beneath anyone.
- Realize the power in building strategic organizational structures.

- No one should feel threatened for communicating their constructive viewpoints.
- Positive influence is critical in motivating people toward constructive action.
- Forming business–technical partnerships based on mutual respect is essential.
- Deep communication, critical thinking, and cooperation are indispensable to achieve disruptive, transformational, software systems.

These strategic behavioral patterns are not new and novel. They are centuries old and are the practices of successful organizations.

Conway's Law doesn't leave anyone guessing about how to make organizational communication structures work for the greater good. As the conclusion of Conway's paper states:

> We have found a criterion for the structuring of design organizations: a design effort should be *organized according to the need for communication*.
>
> Because the design which occurs first is almost never the best possible solution, the prevailing system concept may need to change. Therefore, *flexibility of organization is important to effective design*.
>
> Ways must be found to reward design managers for keeping their organizations lean and flexible. [Conway]

These ideas are reflected in Figure 1.5 and are woven throughout this book.

(Re)Thinking Software Strategy

Focusing on thinking and rethinking before the more technical bits is advisable. Until we understand what strategic business goals must be pursued, we shouldn't try to specify system technical characteristics. After some thoughts on thinking and rethinking, introducing system-level planning will have meaning and purpose.

Thinking

Known as the source of many quotable quotes, (George) Bernard Shaw made this statement with regard to thinking:

> I suppose that you seldom think. Few people think more than two or three times a year. I have made an international reputation for myself by thinking once or twice a week.

Of course, we all think every day. Life would be impossible without thought. Yet, Shaw's entertaining statement exposes an interesting fact about people in general. Much of life is conducted in routine function and regularly employs a kind of autopilot. The less people need to think about specifics, the less they will tend to think consciously about what they do. This is why older people tend to lose cognition unless they remain mentally engaged into their later years. Shaw shows that *deep thought* by even the most notable among thinkers may not occur that often. Rightly, then, a lack of deep thought is a concern even among knowledge workers.

The problem with knowledge workers going on autopilot is that software has little tolerance for mistakes, and especially mistakes left unaddressed for long periods of time. If individual software developers aren't careful, they will become lax in paying debt in their software and transition into the mode of allowing unregulated growth and using repeated, expedient repair.

There is also a concern that developers will begin to increasingly rely on what product companies want to sell them, rather than thinking for themselves in the context of their business focus. New technology buzz and hype are pumped in from outside with much more frequency than is possible from internal business channels. However, the constant drumbeats are missing the context of what matters most locally. Developers who have become complacent may wish for technology to solve the problems. Others will simply long for new toys to engage their minds. Likewise, the dynamic underlying the Fear of Missing Out (FOMO) is not driven by deep, critical thought.

As Figure 1.7 highlights, thinking a lot about everything involved in system specification is essential in making proper business decisions, and later the necessary supporting technical decisions. Here are some motivation checkers:

- ***What are we doing?*** Perhaps poor-quality software is being released as a means to meet a deadline. This doesn't put the teams in a good position to refactor later, and chances are strong that refactoring is not planned. It's possible that the team is pushing for a big reimplementation using a newer, more popular architecture as a solution. Don't overlook the fact that those involved have already failed to introduce a helpful architecture into existing systems or allowed a lapse in maintaining the architecture that was already in place.
- ***Why are we doing it?*** External messages based on selling products as solutions may sound more attractive than taking sensible steps to shore up existing software that has lost its reason for existing in exchange for simply keeping it operational. FOMO and CV-driven development can become a stronger motivator rather than practicing good development techniques. Be certain that a particular architecture or technology is justified by actual business and technical needs.

- ***Think about all the things.*** Every single learning must be examined with critical thought, both for and against. Having a strong opinion and talking the loudest is proof of nothing. Thinking in an informed manner, clearly, broadly, deeply, and critically are all extremely important. These can lead to deep learning.

Seeking deep thought kicks off our real mission, which is rethinking our approach to software development that leads to strategic differentiation.

Figure 1.7 *Be a leader in thought. Think a lot and discuss.*

Rethinking

The ancient Hippocratic Oath[6] is said to have included the statement "First do no harm." This seems relevant not only in medicine but in other fields, including software engineering. A legacy system is just that—a legacy. Legacy implies value,

6. Whether the Hippocratic Oath is still considered relevant and applicable today, or at what point the specific statement "First do no harm" originated, is beside the point being made. Many physicians still perceive the oath and the highlighted statement as important.

something that is inherited. After all, if it didn't have value, it wouldn't be a legacy; it would be unplugged. The system's continued and broad use is what makes it irreplaceable at present. As much as software professionals often think that the business doesn't get it, the business totally gets that decades of investment into a system that has supported and still supports revenues must not be harmed.

Of course, the deep debt and entropy might not be the fault of those currently responsible for keeping a system operational, or those who highly recommend its ultimate replacement. Frankly, many legacy systems do need some help into retirement. This is especially the case when these systems are implemented with one or more archaic programming languages, technologies, and hardware created by people with great-grandchildren, or who are no longer with us. If this sounds like COBOL using an old database and running on mainframe computers, the authors won't deny the similarities.

Still, there are other systems matching this description, such as the many business systems built on C/C++. At the time the work was done, C/C++ was admittedly a better choice than COBOL. One big advantage was the low memory footprint required by C programs, and the fact that a lot of software was being built for PCs and their 256K–640K RAM limits. There are also systems built on completely obsolete and unsupported languages and technologies such as FoxPro, marginalized Delphi, and the only-mostly-dead Visual Basic language.

The major problem with replacing a legacy system is related to losing features in the replacement process or just plain breaking things that previously worked. Replacement also happens in the face of continued legacy change—perhaps slow change, but change nonetheless. Change doesn't necessarily mean new features. It can mean daily patches of code and persisted data. Trying to replace a moving target is like, well, trying to replace a moving target. It's hard. This is not to mention the fact that the software is already in the condition it's in because it hasn't received the care it has both deserved and needed. So suddenly introducing great care as the target moves and as people are actively firing rounds at it seems iffy at best.

That a system is to be justifiably replaced using modern architectures, programming languages, and technologies doesn't make the task any less precarious. Many conclude that jumping in and getting it over with by ripping apart the current implementation and coding up a new one is the only way to go. It is common for those championing such efforts to request several months to accomplish this feat, undisturbed by change. That request translates into halting the moving target for a number of months, and as has already been noted, the system will very likely require patches to code and data. Shall those be put on hold for an unknown length of time?

When There's No Choice

One of the authors was involved in such an effort when platforms shifted out from under an implementation. For example, think of moving a large system with a graphical user interface implemented on MS-DOS to the Microsoft Windows API. One really tricky thing, among many that you never think about until you are deep into the problem, is that two APIs may transpose parameters. For example, in the different APIs, the GUI X,Y coordinate system changed. Missing even one of those translations can cause untold problems that are extremely difficult to track down. In this case, "extremely difficult" involved months of investigation. The overarching reason for the complexity was the unsafe memory space with C/C++ programs, where incorrect memory references not only overwrite memory in invalid ways, but sometimes end up overwriting memory in different ways every time. Thus, the bizarre memory access violations occurred in many mysterious ways.

Of course, that's not a typical problem faced in today's modernizations—modern programming languages mostly prevent that specific kind of error. In any case, there are potential gotchas that are completely unpredictable. Dealing with these kinds of unplanned complications can eat up much of the "number of months" that were presumably reserved for "jumping in, ripping apart, and coding up a new one." It's always harder and takes longer than you think.

Where is the rethinking in all this? In blowing the common legacy escape hatch, it appears likely that a lot of harm will be done. It's a knee-jerk reaction to vast problems that leads to a high probability of replacing them with immense problems or ending up with two sets of enormous problems. The Big Ball of Mud being the enterprise norm leads to competitive paralysis—but to first cause no harm, the patient must still be able to breathe if there can be any hope to perform health-restoring treatments. We need to find a way to jump into a reimplementation, but not by doing one of those cannonball dives that makes a huge splash. This requires some special anti-splash measures and maneuvers.

What hasn't been considered is the creation of new learning opportunities. If we merely rewrite in C# a large system that was originally implemented in Visual Basic, from a strategic point of view nothing at all has been learned. One client, for example, observed in a replacement effort of a COBOL legacy system that 70% of the business rules developed over 40 years had become obsolete. These still lived in the COBOL code and required cognitive load to deal with them. Now, imagine not learning this information, but instead spending the time and the effort to translate all of these business rules from COBOL to a modern architecture, programming language, and technology set. The transformation was already a complex multiyear program without including a very large body of unnecessary rework.

Expanding our previous motivation checkers, the following questions highlight the need for vital strategic learnings:

- ***What are the business goals and strategies?*** Every software feature within a strategic initiative should have direct traceability to a core business goal. To accomplish this, state (1) the business goal, (2) the target market segment (persons and/or groups) that must be influenced to reach that goal, and (3) the impact that must be made on the target market segment. Until the necessary impacts are understood, there is no way to identify the software functionality that is needed or a range of specific requirements. The tools for uncovering the strategic goals and impacts are described later in this book.
- ***Why aren't we doing it?*** There's another important term in technology that needs to be taken into account when making strategic decisions: You Aren't Gonna Need It (YAGNI). This term was meant to help teams avoid the development of currently unnecessary business features, and there are good reasons to do so. Spending time and money, and taking risks, on delivering unnecessary software is a poor choice. Unfortunately, declaring YAGNI has become a general way to cast any opposing viewpoint in a bad light. Using YAGNI as a trump card won't win team loyalty or create breakthrough learning opportunities. Sometimes not implementing some features that "aren't needed" is a mistake of enormous proportions. If a breakthrough that can lead to innovative differentiation is shot down immediately, it's likely more a problem with the shooters' ability to think deeply and recognize an opportunity or the loss thereof. In fact, absolutely refusing to make room for subsequent discussions will reveal the weakest thinkers in the mix.
- ***Can we try new things?*** Teams might agree or disagree about what might work in their target market. It is mostly impossible to absolutely foresee the market's reaction to a given expression of strategy. Gauging the market's response accurately requires giving the market the opportunity to try out the business ideas. Using the science of experimentation may provide the only true way to understand the real possibilities and limitations of strategies; however, to try new things, it's not always easy to think outside of the established mental model. "It is tremendously difficult for people to realize when they are chained to a model, especially if it is subconscious or so woven into the culture or their expectations that they can no longer see how much it is holding them back" [Brabandère].
- ***What are the service-level requirements?*** Once a reasonable set of strategic business goals is understood, the teams involved can start to identify the necessary architectural decisions that must be made. The candidate architectural

> decisions will depend on the service-level requirements. Teams should not settle on solutions too quickly because there are often advantages in delaying decisions about some details of the architecture. For example, even if the teams are convinced that a Microservices architecture is necessary, delaying the introduction of services separated by the computing network can help the team focus on actual business drivers rather than trying to cope with the distributed computing overhead too early. (See the section "Deployment Last," in Chapter 2, "Essential Strategic Learning Tools.")

Rethinking is a critical step, and it feels right. There is a benefit from thinking multidimensionally and critically, and rethinking from a position of ordinary to a fresh strategic vantage point.

We need not conclude, however, that all legacy Monoliths are necessarily the Big Ball of Mud variety. While the vast majority are definitely Big Ball of Mud systems, we must think carefully before making this judgment. The point being made follows next.

Are Monoliths Bad?

Over the past several years, the words *Monolith* and *Monolithic* as applied to software have come to have very negative connotations. Even so, just because the vast majority of Monolithic legacy systems have arrived at the Big Ball of Mud zone, that doesn't mean it is a necessary destination. It's not the Monolith that's the problem—it's the mud.

The term *Monolith* can simply mean that the software of an entire application or whole system is housed in a container that is designed to hold more than one subsystem. The Monolith container often holds all or most of the subsystems of an entire application or system. Because every part of the system is held in one container, it is described as *self-contained*.

The internal architecture of a Monolith can be designed to keep the components of different subsystems isolated from each other, but can also provide the means for communication and information exchange between subsystems. Figure 1.8 shows the same two subsystems from Figure 1.1, but with both subsystems inside a Monolithic container.

In Figure 1.1, we assumed that the two subsystems were physically separated from each other in two processes, and that they communicated via a network. That diagram implies a distributed system. In Figure 1.8, the same two subsystems are physically together in the same process and perform their information exchange through simple in-process mechanisms such as programming language methods or functions.

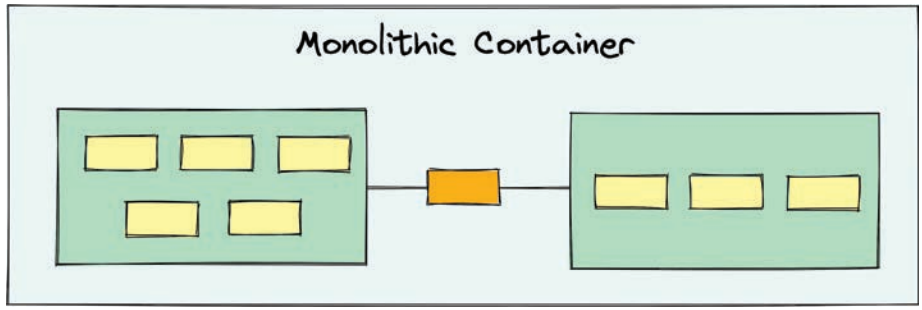


Figure 1.8 *A Monolithic container showing a portion of a whole system. Only two of possibly several subsystems that make up the whole are shown here.*

Even if the ultimate system architecture is to be Microservices, there are advantages to the system starting out life as a Monolith. Not having a network between subsystems can prevent a lot of problems that are unnecessary to contend with early on and are quite counterproductive. Also, using a Monolith is a good way to demonstrate commitment to loose coupling between subsystems when it is easier to allow tight coupling. If transforming to a Microservices architecture is the plan, you will ultimately find out how loose the coupling actually is.

Although some oppose the approach of using a Monolithic architecture for early development where a distributed Microservices architecture is the target, please reserve judgment until this topic is discussed in Parts II and III of this book.

Are Microservices Good?

The term *Microservice* has come to mean many different things. One definition is that a Microservice should be no more than 100 lines of code. Another claims that it's not 100 lines, but 400. Yet another asserts that it's 1,000 lines. There are at least a few problems with all of these attempted definitions, which probably reflect more on the name itself. The term "micro" is often seen to imply size—but what does "micro" mean?

When using "micro" to describe computer CPUs, the full term is *microprocessors*. The basic idea behind a microprocessor is that it packs all the functionality of a CPU onto one or a few integrated circuits. Before the design and availability of microprocessors, computers typically relied on a number of circuit boards with many integrated circuits.

Note, however, that the term *microprocessor* doesn't carry the idea of size, as if some arbitrary specific number of integrated circuits or transistors is either appropriate or not. Figure 1.9 shows a case in point—that one of the most powerful CPUs

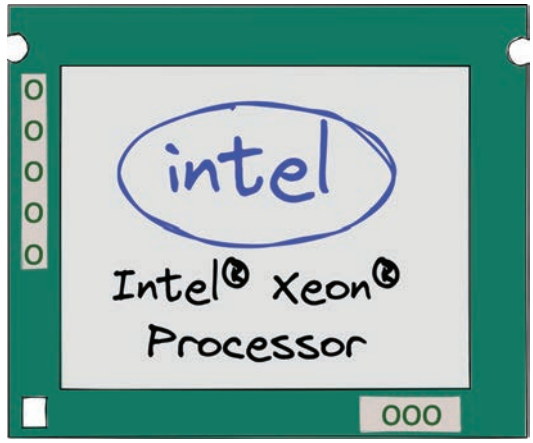


Figure 1.9 *The Intel Xeon is one of the most powerful modern microprocessors. No one has said it is not a microprocessor because it has "too many circuits and transistors."*

available is still a microprocessor. For example, the 28-core Xeon Platinum 8180 sports 8 billion transistors. The Intel 4004 had 2,250 transistors (year 1971). Both are microprocessors.

Microprocessor limits are generally set based on purpose; that is, some microprocessor types simply don't need to provide the power of others. They can be used for small devices that have limited power resources, and thus should require less draws on power. Also, when the power of a single microprocessor—even one of outstanding proportions—is not enough for the computing circumstances, computers are supplied with multiple microprocessors.

Another problem with putting some arbitrary limit on the number of lines of code in a Microservice is the fact that a programming language has a lot to do with the number of lines of code required to support some specific system functionality. Saying that this limit is 100 lines of code for Java is different than saying it is 100 lines of code for Ruby, because Java tends to require 33% more code than Ruby. In a matchup of Java versus Clojure, Java requires around 360% more lines of code.

Furthermore, and even more germane to the point, creating tiny-tiny Microservices has been shown to result in a number of disadvantages:

- The sheer number of Microservices can grow beyond hundreds, to thousands, and even tens of thousands.
- Lack of dependency comprehension leads to unpredictability of changes.
- Unpredictability of change results in no changes or decommissioning.

- More Microservices are created with similar (copy-paste) functionality.
- The expense of ever-growing yet often obsolete Microservices increases.

These major disadvantages suggest that the result is unlikely to be the hoped-for cleaner distributed system solution and autonomy. With the background given in this chapter, the problem should be apparent. In essence, the many tiny-tiny Microservices have created the same situation as the Monolithic Big Ball of Mud, as seen in Figure 1.10. No one understands the system, which falls prey to the same issues that are experienced with a Monolithic Big Ball of Mud: haphazard structure; unregulated growth; repeated, expedient repair; information shared promiscuously; all important information global or duplicated.

Some solutions and additional efforts may be made to move beyond these outcomes, but they generally do not use a completely separate container deployment per Microservice.

The best way to think about a Microservice is not by defining the size, but rather by determining the purpose. A Microservice is smallish in comparison to a Monolith, but the guidance we provide is to avoid implementing tiny-tiny Microservices. These points are discussed in detail later, in Part II of this book.

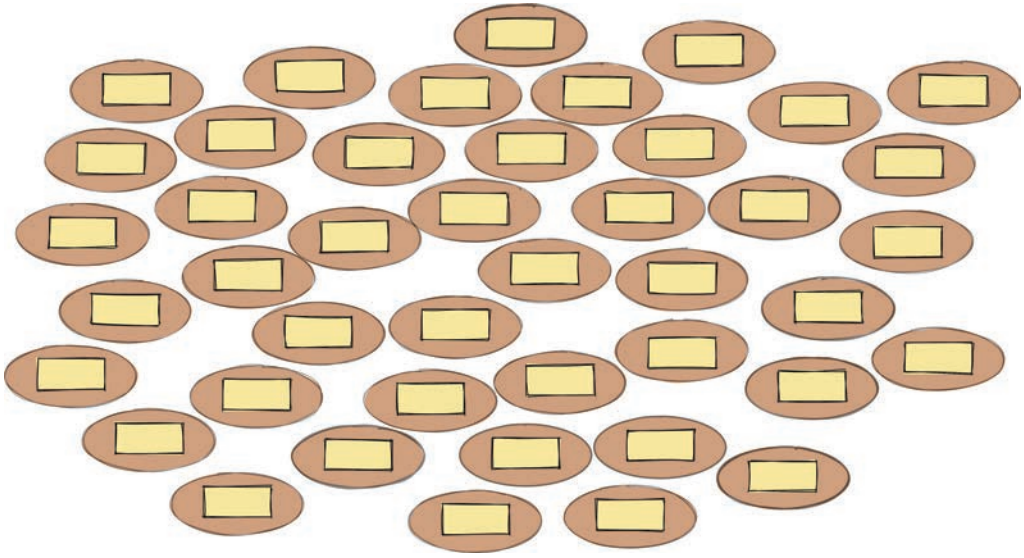


Figure 1.10 *Many tiny-tiny Microservices result in a distributed Big Ball of Mud.*

Don't Blame Agile

In the 1969 film *If It's Tuesday, This Must Be Belgium*, a tour guide leads groups of Americans on fast-paced sightseeing tours through Europe. As far as touring is concerned, this is a classic example of travelers going through the motions.

The same kind of tour can happen with agile software development. "It's 10:00 a.m. and we're standing. We must be doing agile." With reference to daily standups, going through the motions turns otherwise valuable project communications into mere ceremony. Reading Scrum.org on the topic "Agile: Methodology or Framework or Philosophy" is a real eye-opener. At the time of writing, there were approximately 20 replies to this post, and about as many different answers.[7] A reasonable question regarding this concern is, why should that matter?

Recently, agile software development has drawn a lot of criticism. Perhaps most of the criticism is directed toward a specific project management methodology for which mastery certifications can be obtained after a few days of training, because the two are often conflated. This is a sad state, given what agile software development should represent. Much of the industry claiming to use agile methodology or to be agile can't really define it, as noted in the previous Scrum.org experience.

The original agile philosophy never promised to turn poor software developers into good software developers. Has agile made any promises at all? There's a mindset to developing software in an agile way (the Agile Manifesto), and there is a history behind that [Cockburn]. As already observed, agile can't even cause developers to embrace that mindset.

Consider one problem. The ideas of agile software development have been reduced to arguments over whether to refer to this approach as Agile, agile, or "Agile." In fact, we could even draw fire for referring to it as an "approach." So, from here on, we will refer to "it" no longer as "it" but as #agile. This terminology is intended to represent every possible use. However each individual chooses to spell #agile, software developers must demand to get more from #agile than #agile takes from them.

Consider a second problem. A vast complexity has become wrapped around some rather straightforward concepts. For example, the terms and steps of #agile are actually presented as subway maps. At least one high-end consultancy represents its #agile approach in the form of a map of the New York City or London subway/tube system. Although traveling anywhere on an extensive subway network is not extremely complicated, most wouldn't choose to take every route in a large system on a daily or weekly basis just to reach a necessary destination, such as the workplace in the morning and home in the evening.

7. Given that the number of permutations in three options is six, somehow ending with 20 answers seems odder than the fact that there are even 20 answers at all.

This is all very unfortunate. Many have hijacked #agile and moved it far away from its origins, or simply travel in naivety. Usage should be far simpler. Working in #agile should boil down to these four things: collaborate, deliver, reflect, and improve [Cockburn-Forgiveness].

Before accepting any extraneous and elaborate routes, teams should learn how to get to work and back home in these four basic steps:

1. ***Identify goals.*** Goals are larger than individual work tasks. They require collaboration to find the impacts that your software must make on the consumer. These kinds of impacts change the consumers' behaviors in positive ways. They are recognized by consumers as what they need, but before they even realized their need.
2. ***Define short iterations and implement.*** An iteration is a procedure in which repetition of a sequence of operations yields results successively closer to a desired result. A team collaborates to identify these. Given project pressures, unavoidable interruptions, and other distractions, including the end of day and week, teams should limit work items to a number that can be readily remembered and grasped.
3. ***Deploy increments when legitimate value is achieved.*** An increment is the action or process of increasing, especially in quantity or value; something gained or added; or the amount or degree by which something changes. If delivery of value is not possible by means of one day's work, then at least an increment toward value can be reached. Teams should be able to string together one or two additional days of iterations and reach value delivery.
4. ***Review the outcome, record debt, goto 1.*** Now reflect on what was accomplished (or not), with the intention of improving. Has the increment achieved an intended and vital impact? If not, the team may shift toward another set of iterations with increments of different value. If so, note what the team sees as reasons for success. Even when reaching delivery of some critical value, it is normal that an incremental result doesn't leave the team feeling entirely successful. Implementation leads to increased learning and a clearer understanding of the problem space. The iteration is time boxed and doesn't leave enough time to immediately remodel or refactor current results. Record it as debt. The debt can be addressed in the next iterations, leading to an increment of improved value that gets delivered.

Planning too far ahead will lead to conflicts in goals and execution. Going too far too fast can lead to purposely overlooking debt or forgetting to record it. When under heavy pressure, the team might fail to care for debt sooner than later.

The few steps identified in this brief overview of essential #agile can take teams a long way forward. This is the mindset that experimentation affords, and what #agile should primarily be about. It's possible to get more out of #agile than #agile takes.

Getting Unstuck

Any company that has gotten stuck in a Big Ball of Mud and taken complex detours with technologies and techniques needs to get unstuck and find its way out. There isn't one single answer; there are no silver bullets. At the same time, there are means that can serve companies well.

A software system that has become deeply in debt and possibly reached the maximum entropy level took years or even decades for its sterling qualities to erode and digress that far. It's going to take time to make progress out of this mess. Even so, effecting big change is not a waste of time or money. Consider two reasons for this assertion, both based on the poor decision to continue to invest in a losing proposition:

- ***Escalation of commitment.*** This is a human behavior pattern in which an individual or group facing increasingly negative outcomes from a decision, action, or investment nevertheless continues the behavior instead of altering course. The actor maintains behaviors that are irrational, but align with previous decisions and actions [EoC].
- ***Sunk cost fallacy.*** A sunk cost is a sum paid in the past that is no longer relevant to decisions about the future. "Sunk costs do, in fact, influence people's decisions, with people believing that investments (e.g., sunk costs) justify further expenditures. People demonstrate 'a greater tendency to continue an endeavor once an investment in money, effort, or time has been made.' Such behavior may be described as 'throwing good money after bad,' while refusing to succumb to what may be described as 'cutting one's losses'" [SunkCost].

This does not mean that saving any part of the preexisting system always equates to chasing the sunk cost fallacy. The point is that continuing to maintain the existing system as is, with its deep debt and near maximum entropy, is a losing proposition from both an emotional standpoint and a financial position.

Time won't stand still and change won't cease while teams heroically defeat the great brown blob. Moving onward as time ticks away while surrounded by mud and inevitable change, and without sinking deeper, is an absolute necessity. Succeeding

under those conditions depends more on attitude than on distributed computing. Positive attitude is developed through confidence, and the remainder of this book delivers a number of tools and techniques to build the confidence needed to make strides to achieve strategic innovation.

Summary

This chapter discussed the importance of *innovation* as a means to achieve software differentiation as a primary business goal. To aim for relentless improvement in digital transformation is the strongest play in the age of "software is eating the world." Software architecture was introduced, along with the role that it plays inside every company. The chapter explored how the effects of Conway's Law shape the communication paths inside organizations and teams, and the ways it impacts the software produced by organizations and teams. Discussing the importance of communication brought the topic of *knowledge* to the fore. Knowledge is one of the most important assets in every company. To obtain the best results with software, knowledge must be raised from tacit to shared. Knowledge cannot be shared without proper communication paths, and competitive advantage can't be achieved without either. Ultimately, bad communication leads to incomplete knowledge and poorly modeled software—and then to the Big Ball of Mud as the best possible outcome. Finally, we focused on the original #agile mindset and how it can help teams to get unstuck and focus on the right goals.

The principal points made in this chapter are as follows:

- Innovation is the most important aspect of digital transformation. Innovation leads to profitable differentiation from competitors, and should therefore be a strategic goal of every company.
- Software architecture must support the inevitable change without extreme cost and effort. Without good architecture, the best alternative is bad architecture, which eventually leads to unarchitecture.
- The Big Bull of Mud is often the outcome of broken communication that can't possibly lead to deep learning and shared knowledge.
- The people in organizations must exchange knowledge through open and nuanced communication that can lead to breakthroughs in innovation.
- Monoliths are not necessarily bad and Microservices are not necessarily good. Choosing one over the other based on purpose is a result of an informed decision.

Chapter 2 introduces strategic learning tools that can help mitigate poor communication and set the bar higher for enterprise culture. By applying these tools, you can learn how to make informed decisions based on experimentation, and how they can affect the resulting software and its architecture.

References

[ANW] A. N. Whitehead. "Technical Education and Its Relation to Science and Literature." *Mathematical Gazette* 9, no. 128 (1917): 20–33.

[a16z-CloudCostParadox] https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/

[BBoM] https://en.wikipedia.org/wiki/Big_ball_of_mud

[Brabandère] Luc de Brabandère and Alan Iny. *Thinking in New Boxes: A New Paradigm for Business Creativity*. New York: Random House, 2013.

[Cockburn] https://web.archive.org/web/20170626102447/http://alistair.cockburn.us/How+I+saved+Agile+and+the+Rest+of+the+World

[Cockburn-Forgiveness] https://www.youtube.com/watch?v=pq1EXK_yL04 (presented in French, English, and Spanish)

[Conway] http://melconway.com/Home/Committees_Paper.html

[Cunningham] http://wiki.c2.com/?WardExplainsDebtMetaphor

[Entropy] https://en.wikipedia.org/wiki/Entropy

[EoC] https://en.wikipedia.org/wiki/Escalation_of_commitment

[Jacobson] https://en.wikipedia.org/wiki/Software_entropy

[LAMSADE] Pierre-Emmanuel Arduin, Michel Grundstein, Elsa Negre, and Camille Rosenthal-Sabroux. "Formalizing an Empirical Model: A Way to Enhance the Communication between Users and Designers." *IEEE 7th International Conference on Research Challenges in Information Science* (2013): 1–10. doi: 10.1109/RCIS.2013.6577697.

[Manifesto] https://agilemanifesto.org/

[Polanyi] M. Polanyi. "Sense-Giving and Sense-Reading." *Philosophy: Journal of the Royal Institute of Philosophy* 42, no. 162 (1967): 301–323.

[SunkCost] https://en.wikipedia.org/wiki/Sunk_cost

Index

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U