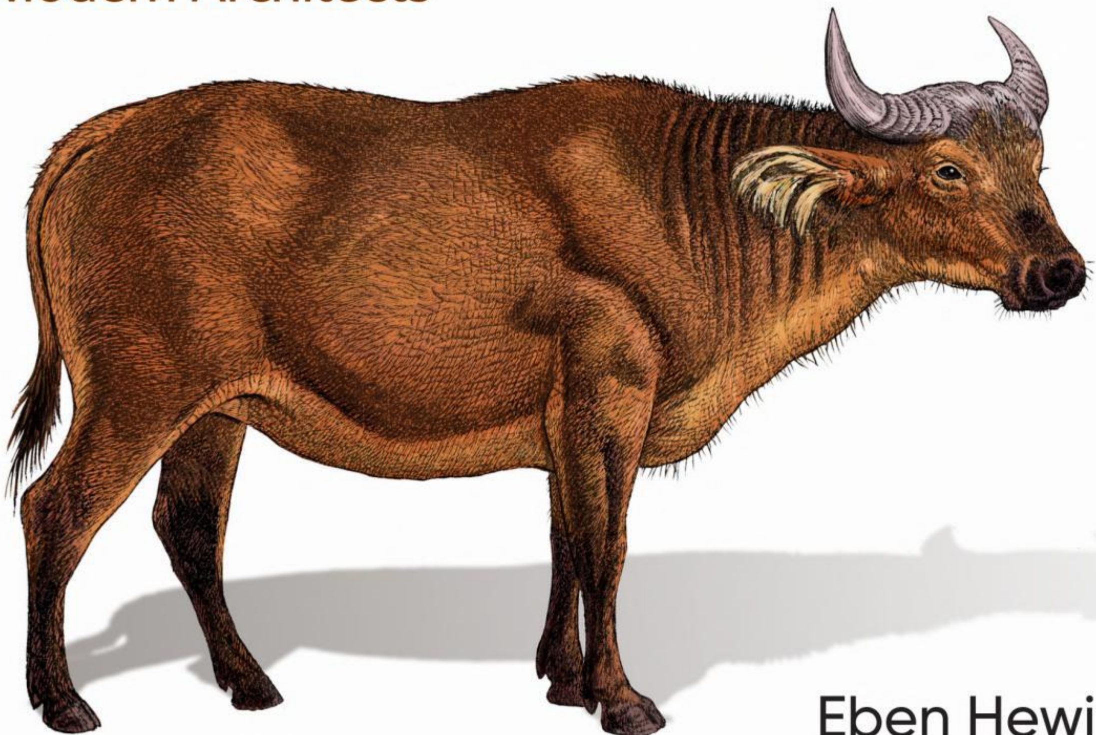
O'REILLY®

Semantic Software Design

A New Theory and Practical Guide for Modern Architects



Eben Hewitt

Copyrighted material

Semantic Software Design

A New Theory and Practical Guide for Modern Architects

Eben Hewitt

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Copyrighted material

Semantic Software Design
by Eben Hewitt

Copyright © 2020 Eben Hewitt. All rights reserved.

Printed in the United States of America.

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

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

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.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Semantic Software Design*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04595-3

[LSI]

Copyrighted material

Table of Contents

Preface. ix

Part I. Episteme: The Philosophy of Design

1. Origins of Software Architecture. 3
   Software's Conceptual Origins 3
   Copies and Creativity 9
   Why Software Projects Fail 10
   The Impact of Failures 13

2. The Production of Concepts. 17
   Semantics and the Software Factory 17
   The Myth of Requirements 19
   Semantics and Software Architecture 20
   The Semantic Field 21
   Designers Are Producers of Concepts 23
     Designing Concepts 24
   What Is a Concept? 25
     Accomplish, Avoid, Fix 26
     Outlining Your Concept on the Concept Canvas 26
   Ideas Are Captured in a Lookbook 30
   Fit to Purpose 32
   The Concept Is Expressed in a Parti 33
     An Example 35
     Adding Aspects to the Parti 36
     The Parti Is Based on a Series of Reveals 36
   Understanding Ideas 38

Copyrighted material

Sense Certainty 38
Metacognition 39
Context 41
Sets 43
Relations 44
Advantages of Semantic Design 45

3. Deconstruction and Design. 49
Introduction to Deconstruction 49
Simplexity 53
(De)composition 55
Affordance 57
Give Intention and Use Value to Negative Space 58
Give Design Decisions at Least Two Justifications 61
Design from Multiple Perspectives 62
Create a Quarantine or Embassy 63
Design for Failure 63
Design Language 64
Naming 64
Start Opposite the User 65
Platforms 66
Disappearing 66

Part II. Semantic Design in Practice

4. Design Thinking. 71
Why Design Thinking? 71
Exploring Design Thinking 72
Principles 73
The Method 74
Implementing the Method 81
Summary 84

5. Semantic Design Practices and Artifacts. 85
Design Principles 86
Pair Designing 88
Murals 89
Vision Box 93
Mind Maps 94
Use Cases 95
Guidelines and Conventions 96

Copyrighted material

Utils 98
Domain 98
service-api 98
service-impl 99
service-client 99
Approaches 99
Design Definition Document 100
Considerations for Composing Your Design Definition 108
Position Papers 111
RAID 112
Presentations and Multiple Viewpoints 113
Summary 115

6. The Business Aspect. 117
Capturing the Business Strategy 120
Provide a Common Understanding 120
Align Strategic Objectives and Tactical Demands 122
Framework Introduction 123
Scope of the Framework 124
Create the Business Glossary 125
Create the Organizational Map 125
Create a Business Capabilities Model 126
Create a Process Map 129
Reengineer Processes 129
Take Inventory of Systems 131
Define the Metrics 131
Institute Appropriate Governance 132
Business Architecture in Applications 133
Summary 136

7. The Application Aspect. 139
Embrace Constraints 140
Decouple User Interfaces 141
UI Packages 142
On Platform Design 142
Service Resources and Representations 144
Domain Language 146
API Guidelines 147
Deconstructed Versioning 148
Cacheability and Idempotence 149
Independently Buildable 151
Strategies and Configurable Services 151

Copyrighted material

Application-Specific Services 153
Communicate Through Services 154
Expect Externalization 154
Design for Resilience 155
Interactive Documentation 157
Anatomy of a Service 158
UI Packages 158
Orchestrations 158
Engines 161
Data Accessors 165
Eventing 165
Structure of an Event Message 168
Contextual Services and Service Mixins 168
Performance Improvement Checklist 170
Separating API from Implementation 171
Languages 172
Radical Immutability 173
Specifications 175
A Comment on Test Automation 178
A Comment on Comments 179
Summary 181

8. The Data Aspect. 183
Business Glossary 183
Strategies for Semantic Data Modeling 184
Polyglot Persistence 187
Persistence Scorecard 188
Multimodeling 189
Data Models for Streams 191
Feature Engineering for Machine Learning 193
Classpath Deployment and Network Proxies 195
Peer-to-Peer Persistent Stores 196
Graph Databases 198
OrientDB and Gremlin 199
Data Pipelines 200
Machine Learning Data Pipelines 203
Metadata and Service Metrics 206
Auditing 207
ADA Compliance 207
Summary 208

Copyrighted material

9. The Infrastructure Aspect. 209
Considerations for Architects 209
DevOps 211
Infrastructure as Code 212
Metrics First 215
Compliance Map 217
Automated Pipelines Also First 217
The Production Multiverse and Feature Toggling 218
Implementing Feature Toggles 219
Multi-Armed Bandits: Machine Learning and Infinite Toggles 221
Infrastructure Design and Documentation Checklist 222
Chaos 224
Stakeholder Diversity and Inside/Out 226
Summary 227

Part III. Operations, Process, and Management

10. The Creative Director. 231
The Semantic Designer's Role 231
Creative Directors Across Industries 234
In Fashion 235
In Film 236
In Video Games 238
In Advertising 238
In Theater 238
In Technology 239
What's In a Name? 241

11. Management, Governance, Operations. 245
Strategy and Tooling 245
Oblique Strategies 247
Lateral Thinking and Working with Concepts 248
Conceptual Tests 252
Code Reviews 254
Demos 255
The Operational Scorecard 255
The Service-Oriented Organization 258
Cross-Functional Teams 262
The Designed Scalable Business Machine 263
Managing Modernization as a Program 266
Change Management 267

Copyrighted material

Governance 270
Goals 270
Metrics 270
Service Portfolio 271
Service Inventory and Metadata 271
Service Design Checklist 273
Service Design 273
Service Operations 274
Business Processes 275
Data 275
Errors 276
Performance 276
Security 276
Quality Assurance 277
Availability and Support 277
Deployment 278
Documentation 278
Further Reading on Organizational Design 279

12. The Semantic Design Manifesto. 281
The Manifesto 281
The Four Ideals 285
The Key Practices 286
Opening 292

A. The Semantic Design Toolbox. 293

B. Further Reading. 297

Index. 303

Copyrighted material

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*.

Copyrighted material

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

Copyrighted material

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.

This element signifies a tip or suggestion.

This element signifies a general note.

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

Copyrighted material

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

O'REILLY®

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*.

Copyrighted material

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.

Copyrighted material

Copyrighted material

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.

Copyrighted material

Copyrighted material

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 crafts-people 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

Copyrighted material

```
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.)

Copyrighted material

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,

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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?

Copyrighted material

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

Copyrighted material

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.

Copyrighted material

- 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.

Copyrighted material

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.

Copyrighted material

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."

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

Copyrighted material

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 and zoning laws. Humans have had homes and offices and hotels and formal gowns and luggage and many of the objects of architecture and design for thousands of years. The ideas of going to work in an office with others, or attending a musical performance, or traversing a body of water safe and dry, these are well-understood human functions that have been going on for thousands of years, across all cultures across the entire settled world.

Copyrighted material

We in software and systems have chosen for our conceptual parents "architecture" and "design." These are the words we use to describe our work. We print them on business cards, and they rest in the fields of endless human resources databases to describe our job functions. Our field is prescribed by their inherited language and conceptual models. This is understandable, but perhaps inadequate.

It's understandable because the building architect is concerned primarily with making something that must be sturdy, usable (fit to purpose), and delightful. The conceptual miss comes, I assert, because these fields are not predominated by a concern for something *novel* (an innovation, the expounding of an *idea* that is new). Put bluntly, building architects make rather an object that did not exist before, within a tightly prescribed realm of human interaction.

However, when something is new *as an idea*, and not merely a latest realization of a very old idea, we call it an invention. Or art. In this way, the term *architecture*, the nonnecessary *metaphor* that has been carried over time onto our mental model of how we think of our work, how we talk about it, and what we think our responsibilities are with respect it, has converged into a dead metaphor, perhaps constraining or hampering our work more than it any longer enables and supports it.

What if, in that moment decades ago, as encapsulated at the NATO Conference in 1968, in that moment as fumbling around for how to assign metaphors to ourselves to understand our work, in an effort to bootstrap our field, we had instead adopted the term "composer," or conductor, or play director, or writer? *They were on the table*. It's not *unthinkable*. But our entanglement, our fusion, at the time with the manufacturing processes in hardware, have led us down a path that has created many wonderful programs and advances in software.

But perhaps these advances are in spite of, not because of, these industrial metaphors? Or rather, that they were critical at the time, but no longer as useful?

The world has changed in these many decades since the NATO Conference in 1968. The world is more synthetic. Jobs must move up the value chain. A faculty member of the Arizona State University School of Architecture recently told me that the unemployment rate for architects in the Phoenix area is higher than 50%. In fact, the best way to face the highest possible unemployment rate for yourself is to go to architecture school. It's not a job that creates enough value in the world of physical buildings because computers and civil engineering codes aid lower-level modelers. Such a fate is coming for architects in software who cannot determine how to move up in the chain of value creation for customers.

We have been altogether too inward facing, burdened by thinking the job was to create an enterprise ontology, or fill out the chart of a Zachman framework and think we have done something useful. We have not. We have merely complied with one available method of trying to understand our own place in the world, justifying an

Copyrighted material

existence, the frame of a field grappling with its own identity. This was a necessary stage to move through, yet we cannot remain in stasis there.

To be clear, I am not merely arguing for us to all have a title change and get on with the same practices. But because the name begat practices that don't fit our work, it stands to reason we might learn from having new ones.

The Myth of Requirements

In system design, we speak of the "requirements." This word creates a false center, a supposed constant, which creates problems for our field. These problems come in the form of a binary opposition set up between the product management team and the development team. It supposes, in the extreme form, that the product management team knows what to build and that the development team are passive receptacles into whom we insert this list of what they are required to build. Within an Agile method, some freedom is perhaps allowed to the development team in how to design within that list of requirements.

The requirements, however, do not exist. But the requirements, like everything else of value, are just made up by someone. They are not first known and then told. They are *invented*.

Part of the work of the new architect-creative is to help create those requirements, both functional and nonfunctional. To see what needs to be done, what might work, what structure accounts for what we think we want the system to do, or what we think someone else we've never met might want or need the system to do three years from now when it's harder to change and how to accommodate that.

How do we know that Indiana Jones is the archaeology professor who finds the Lost Ark of the Covenant? Because George Lucas invented a character named Indiana Smith, and Steven Spielberg didn't like the name so he changed it to "Indiana Jones." And all of a sudden there is a world of the 1930s and a man standing in it and he needs to go do something and someone needs to get in his way and how might that work? That's how requirements are made, in the movies and in software. People make stuff up.

When you make stuff up as a software designer, that world, like the world of the movie into which you posit a character with a conflict, is your context. It's the place where you posit signs that have meaning in relation to one another. It's your semantic field.

Copyrighted material

Semantics and Software Architecture

This book has a single primary purpose among many purposes: to help you better design software. To do so, it advances a new model, a new approach, a new set of ideas and tools called *semantic software design*.

Why "semantics"?

Semantics, as a field, is concerned with the production of meaning, and how logic and language are used. It is "the linguistic and philosophical study of meaning, in language, programming languages, formal logic, and semiotics. It is concerned with the relationship between signifiers—like words, phrases, signs, and symbols—and what they stand for in reality."[1] It is about sets. It is about relations, and the possibilities that language itself creates, performs, and cuts off.

This *precisely* describes the role that ~~architects~~ designers should be playing, the kind of work they should be doing. The logic demanded by the compiler and the business requirements remain logical problems, set theoretical problems. Everything the developer does is expressed in language.

Semantics = logic + language.

That sounds *exactly* like the work we do when we are allowed to do our best work as software developers. But we've been trained around these incorrectly conceived metaphors. So we don't have a set of practices to even see where we are making the little mistakes that accrue toward failed projects. We have practices that rather discourage the kind of thinking we must embrace to make successful designs.

The problem with software—a chief reason our projects fail—is a failure of our language. We are *not architects*. Not even close. We do not build buildings with an obvious and known prior purpose, which is an approximate copy of the same kind of building people have been making and using for thousands of years, using tangible commodity materials on a factory line. Quite the opposite.

Our *only* material is that of language and ideas, names and meanings, signifiers and signifieds. Our only material is *semantics*.

When we design software we are designing the semantics of a demarcated field of signifiers and signifieds.

That is our primary activity. It takes its material expression in a collection of classes or functions as syntax in some language. But these languages are interchangeable enough. And the syntax is not the message.

1 *https://en.wikipedia.org/wiki/Semantics*

Copyrighted material

The *semantic field* comprises the set of sets of interplaying linguistic terms that form the idea our software represents from a comprehensive systems view. It's the nouns and verbs in your domain, how they relate, and how in your software system design that complete set of ideas acts as an overlay representing the "real" world.

We are haunted by our inherited language. It's the air we breathe: it's ubiquitous and invisible. It has both shaped and deformed our thinking, and our software suffers.

Semantics is the missing step. This is the piece that we skip because we did not know it was required. Because our inherited conception of our field took us to the factory lines, away from language and epistemology (the study of what is knowable, and how we can know what we know), and philosophical categories.

To perform semantic software design, you perform these steps:

1. Define its semantic field.
2. Produce your concept within it.
3. Deconstruct the concept to improve it.
4. Design the system according to the deconstructed concept and its semantic field.
5. Write the software and realize the attendant systems and processes.

Where we fall short is in rigorously creating a concept of our software as above. When we do this, our software succeeds. When we do not, we endure a thousand minor missteps, many of which we don't even see, that over time add up to larger failures of our projects and systems.

The rest of this book unpacks these ideas and illustrates how to apply them to make more successful software systems and projects.

The Semantic Field

A *proposition* is a declaration about what is the case. It represents the set of possible worlds or states of affairs in which it obtains truth-value, in which it is true.

The universe is an infinite conjunct of propositions.

As an infinite conjunct of propositions, the universe is a (very long) list of all of the statements that result in a truth-value. Because time keeps passing, that list is infinitely long.

The conjunctive is just the logical connector "and." We could say "this is true and this is true and this is true..." If we said only true things, and said all the true things, we would have a complete image of the entire universe across time and space. If we could iterate every proposition across space and time, we would have an exhaustive representation of the universe.

Copyrighted material

Representing some aspect of the actual world in its true propositions is the work of the software designer.

If the scope of our software was to represent the entire universe, we would translate the infinite list of propositions into executable statements. This would be straightforward because computers understand the true/false binary.

But someone has to pay for this project. And they don't have infinite time and they don't need all that scope. Just some of it. We use logic and language to form a concept. Our concept is the collection of our propositions. We carve out a space from that infinite conjunct of propositions representing the world. We create a boundary separating the scope of our software, its domain, from the rest of the universe. There are things we represent and things we will not. This is how we define that semantic field.

Because we do not have time and scope and budget or need to represent the entire universe, we carve off the scope of our domain. All software for certain and by necessity will have this boundary. This is the edge of your semantic field, that place where your software stops representing the world. At this boundary, you will suffer border skirmishes between your representation and what you've cast out or left out beyond the horizon. We are forced to round our thought off in a not-entirely-consistent way.

If we did not draw such a line around the domain, our work would be to represent All The Things, our scope would be infinite, our representation would be of the entire universe in eternity, and our software would be the actual lived world and we would be God. Because this is not the case, we have to stop making representations, and that's our semantic boundary, and that makes inconsistencies in our logic and language, our semantics. But if we consider that boundary consciously, because we're aware of it, because we understand that our work is actually semantics and not engineering or architecture, we will make the logic and language better. And because they are only building blocks in software, our software will be better.

The main thesis of this book is that software fails because of improper understanding of the world, because of an improper understanding of our role—we have thought we were engineers and architects instead of philosophers and semanticists—and this results in unclear objectives, undue complexity, incorrect and changing requirements, lack of alignment, lack of focus, wasted effort, churn, and disarray—many of the top reasons the McKinsey report states that software projects fail.

Software is a linguistic and logical endeavor. If we think we are the semanticists or philosophers of our systems, we will make better language and use better logic. And because those are the only tools of software design, our software will be better.

The semantic field allows for the possibility of concepts.

Copyrighted material

Designers Are Producers of Concepts

> *To be engaged with architecture is to be engaged with almost everything else as well: culture, society, politics, business, history, family, religion.*
>
> —Paul Goldberger

Vitruvius is the first Roman architect of record, working in the first century BC.

He wrote *de Architectura*, now known as *Ten Books on Architecture*, which is still taught to this day at university. It would be nearly 1,500 years before another book on architecture was written. Vitruvius declares that the architect should be versed in drawing, geometry, optics, history, philosophy, astronomy, music, theater, medicine, law, and other fields.

Building architects are told this sort of thing all the time: that they must engage with all of culture, all schools of thought and academic disciplines, and understand many disparate fields in order to do their work. The lineage of this assertion comes from *de Architectura*.

Yet we in software somehow find ourselves exempt. As the world in general becomes more and more specialized, we frequently find ourselves satisfied to recount the variants of Big O notation and argue the virtues of MergeSort over QuickSort, or (heaven forbid) this JavaScript framework over that one.

This should not be the case.

Thinking only from our own perspective as computing practitioners leaves our design tepid, derivative, inefficient, incomplete, untrustworthy, unstable, and costly to expand and maintain.

We must begin with the concept.

The concept must support integrity and harmony. It must provide for, as Vitruvius asserts, the three critical components: stability, utility, and beauty.

Technology Strategy Patterns

Please see this book's companion volume, *Technology Strategy Patterns*, for a more in-depth discussion of an architect's attributes and how architecture and strategy best work together in a tech organization.

Copyrighted material

Designing Concepts

Good designs do not merely execute the stated requirements.

The creative architect will first create a coherence and an integrity to the concept.

First, we design the concepts. The concepts inform, provoke, and support the local designs that they encompass. For the effective enterprise architect, these might be designs of software systems, integrations, infrastructure, organizations, the use of data, and business processes.

Proceeding from the concept, all the elements can work together in a coherent system of signs.

We are not merely drawing deployment diagrams. We ask ourselves, what is your theme, your point of view? What design principles can a user intuit from your work without being told them?

Thinking in systems means that you observe the entire system. Step back far enough to see all of it, the whole thing. You need to see all the parts to form an understanding of the relations between all the parts, both within the bounds of the system and the universe of systems that it touches and in which it participates. Then, in a double-action, use that knowledge to understand each part on its own. Considering each part as its own integral system, without a view of those relations, what new light does it emit? What new understanding can you find in the observance?

Now strip it down further: consider the object of the system as a thing-in-itself, relieved of our assumptions about what it is and why it is.

Now build the system up again, suspending your prior knowledge, reaching each object itself, and see how the relations reveal themselves anew. Reexamine how the relationships could be improved, augmented, destroyed, and rearranged based on this violent investigation.

Only now can you proceed with confidence that you have considered for your client the forces at work, the justifications for their presence in the system, their organization, and the context in which this system will operate and others within which it possibly could.

The behavior your system exhibits reveals the web of all of these interrelated and interdependent subsystems. There are many decisions to be made, whether by you, your team, or the participating team (the application developers or those working in the process).

The ~~architect~~ is the chief philosopher of their system.

The work and the joy of the architect is to create a concept, then clarify it, then communicate it for realization.

Copyrighted material

What Is a Concept?

> *So architecture is art and it is not art; it is art and it is something more, or less. This is the paradox and its glory, and always has been.*
>
> —Paul Goldberger

A *concept* is a complex idea consisting of compounded abstractions over a variety of related ideas. A concept is an interpreted representation of some aspect of the world.

Concepts are *not facts*. They are attempts to explain something. Your software might not appear to be an attempt to explain something about the world. But it is in fact the result of a concept. That concept might be very poor: it might be logically unsound, ethically problematic, or aesthetically challenged. One of the arguments of this book is to foreground the concept given that you have no material to carve, no plot of land to build on with concrete and steel. You are defining concepts. That's the job of the software designer.

A concept is always a concept *of something*: it is a representation. As such, you are necessarily *interpreting* what is important about the world, what requires independence, what merits refining, what earns a place at the table of competing representations, who gets a voice and a name and a fully rounded character and who doesn't. You are making value judgments, ethical judgments, aesthetic judgments, telling and participating in a story about the world, whether you're doing so consciously or not.

A concept is *nonobvious*. It's a complex of ideas and abstractions mixed with judgments. It is the product of thinking. A simple and direct referent is not a concept. Saying "My software system is an ecommerce website" is not a concept. That is obvious, understood, undistinguished from any of the other millions of ecommerce websites. Saying "My software system is an ecommerce website that lets people barter (trade goods and services) with each other instead of paying with money" is one step closer. It's more distinct, refined, and complected.

A concept can be *argued against*. A reasonable person could argue that your concept is incorrect, that your representation is incomplete, shoddy, or misguided. This is an easy test to see whether your concept is forming. If no one would argue the opposite of your statement, you haven't done anything but cheer a marketing slogan.

If I were to ask you to draw a picture of a "pet," what would you draw? Perhaps a big, fat snuggly kitty. Or a skittish and playful kitten. Or a bird, an iguana, a dog, a ferret. There are many different ideas that complect into a concept. Foregrounding metacognition, or thinking about how you think, helps you recognize these kinds of differences, including your own biases. It's an important step to doing these more consciously. That, in turn, is an important step in creating compelling concepts that are the hallmark of truly innovative software.

Copyrighted material

Accomplish, Avoid, Fix

To be useful in a typical software project, your concept will generally be about one of three things: accomplishing something, avoiding something, or fixing something:

Accomplish
: This might mean that your user can make a contribution, or can take advantage of a new opportunity in an emerging market. Projects involving accomplishing are about doing something new, different, exciting. They're about making more cakes.

Avoid
: Your project might be about helping you avoid something negative, like fraud or noncompliance, or averting risk. They're about more fairly dividing up the cake you already have.

Fix
: Software projects often arise in order to address some sins of the past, and "simplify" or "streamline" some particularly messy process. They're not really about cake.

Your new software project probably is not about all three, or even two of these. If it seems that way, your concept might be too sprawling, unruly, and too poorly constrained. You should refine it.

Outlining Your Concept on the Concept Canvas

To start to work with your concept in a more practical way, you can outline it.

Consider something that you do know about the project. Think in terms of something your customer might want to accomplish, avoid, or fix. In a sentence, answer this question:

> *Who wants what by when and for what reason?*

These are basically the aspects of the "reporter" questions, and are very similar to the structure of a user story. Your organization might have a "one pager" or "Business Requirements Document" that is intended to answer these kinds of questions. Your design concept is most immediately informed by the business idea: some application or major update that product management or other executives want to make. It is informed too by the overall business strategy, your technology strategy, and the creative work you perform in designing the concept.

These are interrelated, and shown in a cluster of associated ideas, as illustrated in Figure 2-1. They should inform one another in a continuous cycle, and not unidirectional or only top down. The design concept for your local application can be robust

Copyrighted material

and rich and innovative enough to reinform and at times even reinvent the technology strategy and business strategy.

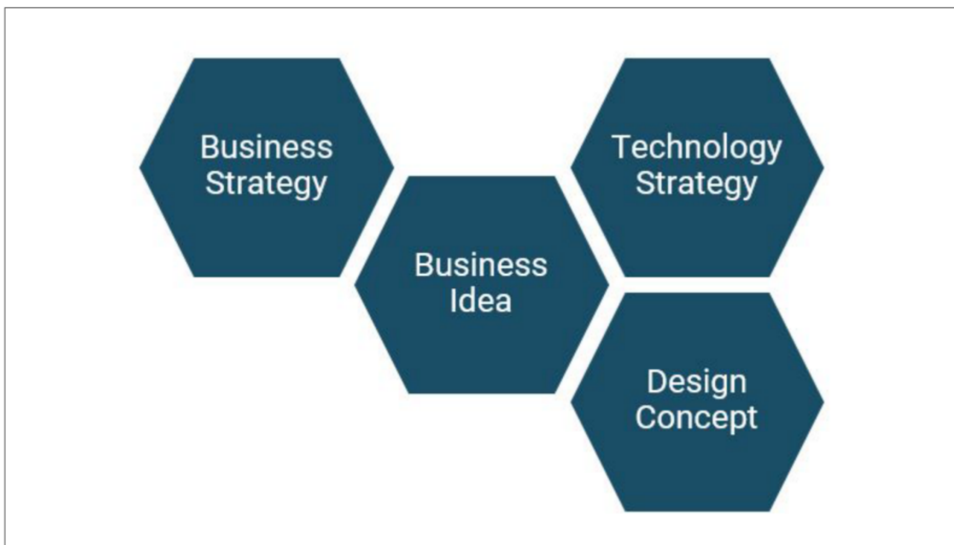


Figure 2-1. The relations between these elements are not hierarchical

To support your concept, and this richer cycle, consider what the need behind the need is. Consider how they would like to accomplish this. Typically the "business" will come up with what needs to be done, and expect architecture to describe how it should be done. This is fine. There is greater value in the designer who can shape the technology concept such that it informs and changes and perhaps even reimagines the business idea.

What are the salient bullet points across People, Process, and Technology? Consider the strengths your organization can build on, and challenges to overcome.

Constraints are often found to be frustratingly constricting in other models of software design. They are welcome in our world, however, because they give us an anchor, something real to help orient us.

Divergent and convergent thinking

As you work through your concept, you should go through two stages, *divergent thinking*, followed by *convergent thinking*.

With divergent thinking, you generate a list of candidate solutions that should be very different from one another, and very different from what exists today. Then, in a second, distinct stage, use convergent thinking to conflate these ideas, throw out the ones that won't work, and come up with your concept based on this refinement:

Copyrighted material

Divergent thinking

Generate a wide variety of possible solutions. They should have variety and be distinct across the array of candidate solutions. What solutions do not neatly conform with your current application or business landscape? How can you follow your curiosity? How can you imagine a solution that is prompted outside the field of the local software problem, such as by a bit of music, art, an opera, a toy, a game, something entirely outside the domain? Are you taking a risk? You should be clear on what the risk is. If you are not sure what it is, you might not be doing something sufficiently interesting. Capture your solution candidates in a list that becomes part of your lookbook or scrapbook.

Convergent thinking

After your divergent thinking exercise has generated a list of candidate solutions, it's time to narrow this field to a coherent single concept. Here, you are creating a set of filters or lenses by which to view your related ideas so that you can clarify and refine these scattered lists into what will become your working concept. To do so, ask yourself and your team the following types of questions for each candidate solution:

1. What absolute constraints are known?
2. How might these candidates fit within a budget, if known?
3. How might these candidates fit within a timeline?
4. What known elements of the business or technology strategy do these candidates support?
5. What new opportunities does this create?
6. What positive and negative elements of our current landscape of People, Process, and Technology does this enhance or aggravate?
7. What people or roles would need to approve or work together with these candidates?

There are many questions and conversations your team will have that might be more relevant to this process for your situation. These are just to get you started.

The convergent thinking exercise will result in a few key components. There might be three to seven of them. These are the main ideas that together form the concept. Later in executive briefings, marketing slides, customer-facing product decks, interviews, and other forms of communication, you'll use the statement and then these main bullet points as the "elevator pitch" to quickly and concisely express the concept—what this system is about, why it exists, and whom it benefits.

Your ideation work at this stage can be captured in this template, which I call the "Concept Canvas." Figure 2-2 depicts this.

Copyrighted material

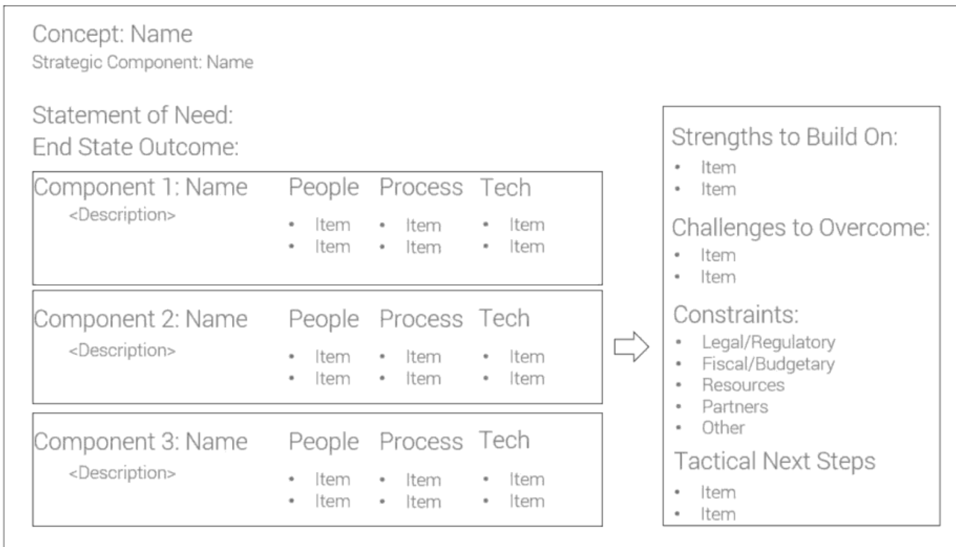


Figure 2-2. Capturing your concept in the Concept Canvas

Of course, companies don't have concepts: people do. Get your team together for a morning and work through the Concept Canvas. This can then serve how you put together the project plan and create the detailed design.

In our practice, we don't do "architecture," for reasons we discussed. Rather, in Semantic Software Design we are producers of concepts, designers of concepts. We express them in a way that allows others to be inspired and participate and understand the boundaries.

In summary, a rough guiding outline for how to work with your concept at this early stage is as follows:

1. **Concept statement**: A single sentence or phrase. This is like the melody of a tune you can hum. It's not the whole song; it's a memorable image that helps you communicate the basic subject.
2. **Statement of need**: Captures who wants what by when for what reason. This ensures that you are striking the right balance between being creative and curious and not going off on a tangent that has no business value. Who are the customers, end users, business partners, and internal executives who stand to benefit—or could stand in the way?
3. **Alignment with strategy**: You will have a greater chance of relevance, impact, and support if it is very clear that your concept relates and advances at least one element of the business and technology strategies. You should identify this explicitly.

Copyrighted material

4. **Idea components**: These are the highly cohesive idea components that work together to form the concept. Consider them each through the lenses of People, Process, and Technology.
5. **Path forward**: After you have your basic concept, you want to consider how you will bring it into the world as a real system. There is of course considerable work to be done yet. At this point, you have only a complex set of ideas that together form your concept. The remainder of this book is devoted to showing how to turn that concept into a designed system that can be implemented as fantastic software. But you need a bridge to help cross this gap between concept and designed system. The *path forward* captures circumstances in the real world and tactical next steps that you want to take in order to advance your concept into a system design and working software.

You capture these in your single Concept Canvas. You can then add this to your parti, as we discuss shortly.

Ideas Are Captured in a Lookbook

In the fashion and design world, there's something called a *lookbook*. This is a collection of photographs that a designer will use to showcase their work for a particular line or season or campaign. It gives viewers possible suggestions on how to pull together a few components from the new season's line, such as these jeans, that sweater, and these boots to form a look or a personal style.

John Malkovich Lookbook

Venerable actor John Malkovich has turned his talents to designing his own fashion line, and you can find an example of his lookbook here.

In fashion, this is a collection of images illustrating the concept. At first, you can use it that way, too. Eventually, your *lookbook* will become a compilation of design sources, inspirations, and otherwise random-seeming documents. It's your idea diary, and it helps you to recall all the aspects of the concept you're working with as you form it. It helps you as a concise compendium to show others so that you can collaborate on the design.

Copyrighted material

Your lookbook might have many of the following items:

- Informal sketches
- UML-type diagrams, but nothing formal or definitive-looking
- Images
- Mind maps
- Snippets of thoughts
- Key customers
- Relevant quotes
- Stories
- Links
- Videos
- Colors
- Materials

Your lookbook is like an active journal in collage form. There will be many sources of inspiration along the way that might have informed your concept. Simply capture them in this single place so that you have them to refer back to. This single place might just be a growing Word document, a special page on the wiki, a OneNote file, a web page, or whatever you like.

You might be working with a set of themes, the way a composer would have a set of themes for different characters or events. One might be "craftsmanship." How would you express that to your team or think of it yourself? You might consider some of the following:

- The Mercedes-Benz AMG "one man, one engine" philosophy, as shown in this video. Every AMG engine bears the signature of the one man who made it.
- A master seamstress making a tiny replica of the Miss Dior Dress from the 1950s in this video.
- A master cobbler making a pair of Prada shoes in this video.

If one of your themes was about radically rethinking historical approaches, you might include the Google X Moonshot Thinking video, and so forth.

Initially, the audience for your lookbook will be the other folks on your team, but it's probably not useful outside that at first. It should feel a bit personal, as if to share it, you'd be revealing something, a bit of your attitude, tastes, inspirations, understanding, limits of that understanding, some part of yourself. You might feel a slight pang

Copyrighted material

of nerves to do so. That's good. This means that you're doing something that matters to you, something you're truly engaged with.

As it becomes more refined, you can use it as a catalog from which to pull particular views that help you communicate the design to the variety of diverse collaborators who might include UI/UX folks, developers, executives, managers, and customers.

Fit to Purpose

> *As an artist, yes, I have constraints. Gravity is one of them.*
>
> —Frank Gehry

The Walt Disney Concert Hall opened its doors in Los Angeles in 2003 to become the new home of the Los Angeles Philharmonic. After being designed by architect Frank Gehry, it was constructed over the course of four years.

At the time of its opening, the following story was told by *Los Angeles Times* music critic Mark Swed:

> When the orchestra finally got its next [practice] in Disney, it was to rehearse Ravel's lusciously orchestrated ballet, *Daphnis and Chloé*. … This time, the hall miraculously came to life. Earlier, the orchestra's sound, wonderful as it was, had felt confined to the stage. Now a new sonic dimension had been added, and every square inch of air in Disney vibrated merrily. Toyota says that he had never experienced such an acoustical difference between a first and second rehearsal in any of the halls he designed in his native Japan. Salonen could hardly believe his ears. To his amazement, he discovered that there were wrong notes in the printed parts of the Ravel that sit on the players' stands. The orchestra has owned these scores for decades, but in the Chandler no conductor had ever heard the inner details well enough to notice the errors.

Figure 2-3 shows this fantastically expressive building.

This is architecture at its best: inventive, coherent, clear in concept, expressive, in conversation with its context, multivariate, improvisational, alive. The building appears as moving music itself. To support the quality of sound that it does, and the comfort and clarity it affords patrons, is astonishing. Gehry's building is brilliant, beautiful. Moreover, as the story about the misprinted music sheets reveals, the building is incredibly well fit to purpose. So must our concepts be.

Copyrighted material

Figure 2-3. The Walt Disney Concert Hall in Los Angeles by architect Frank Gehry (photo: Wikipedia)

In an interview, Gehry states that in architecture, you must ask, "Then what?" You can love the clients, love the city, hit the budget, be polite, be good to work with. These things are merely the table stakes. So you must ask yourself, "Then what?" to get the real value out of your work.

We must push ourselves to deliver something truly special, something of such wonderful function that we help our users hear notes they never heard before. We can astonish and delight.

The Concept Is Expressed in a Parti

> *It is better to enter a turn slow and come out fast than to enter a turn fast and come out dead.*
>
> —Dr. Ferry Porsche

Building architects have space, a neighborhood, and a building to build. They can start with physical objects, like a sculptor: a block of marble.

We in technology cannot do this. We have no space, no material but our logic, our language, and how we employ semantic signs to produce a concept.

Copyrighted material

The concept is the first moment of our work, and the one most often skipped and ignored because we did not even know it should be part of our work. Because we started with the "architect" metaphor. This causes us to make many other local category mistakes that accrue toward the failure of our projects.

Our work is to *produce a concept*. That concept produces a *system design*. That system design is comprehensive to create the best context for writing valid and sound requirements, both functional and nonfunctional, and for allowing them to be viewed together. The concept also informs a *designed project model*. Because our view is comprehensive, we design the project plan every bit as much as the software system. Because they go together in symbiosis. Taken together, our projects then have a far higher chance of succeeding than software projects have over the past 25 years or so. Such a program model produces working *software* that is innovative, delights customers, and features outstanding support for nonfunctional requirements. It also offers the most rewarding opportunity for the people on these teams to have fun and make a meaningful contribution that they are excited and delighted to do. With our approach, we stand a better chance to light a fire *within* people instead of under people.

The advent of the microprocessor meant that we had to conceive of how to create sturdiness, and fitness to purpose, and beauty, in a nonphysical realm. This is the realm of the philosopher more than of the architect.

As we have discussed, one reason so little software is properly functional or pleasant to use is that when we were busy borrowing metaphors, perhaps we picked the wrong one. And after we did, even then we skipped a part, and an important one: the *parti*.

The parti is short for "parti pris," meaning a "decision taken" in French. It is an image expressing the general organization of a design. The parti takes the Concept Canvas, the lookbook, and the ongoing changes and reveals of the project over time as inputs and refines them over the course of the project into a decision log of the key components. The parti is the first representation of high-level executable system components that can be built as software modules.

Partis are never reused because they are particular to this design challenge, these constraints, this context.

A straightforward, simple example comes from NASA (though they don't call it a parti), which you can see in Figure 2-4.

Copyrighted material

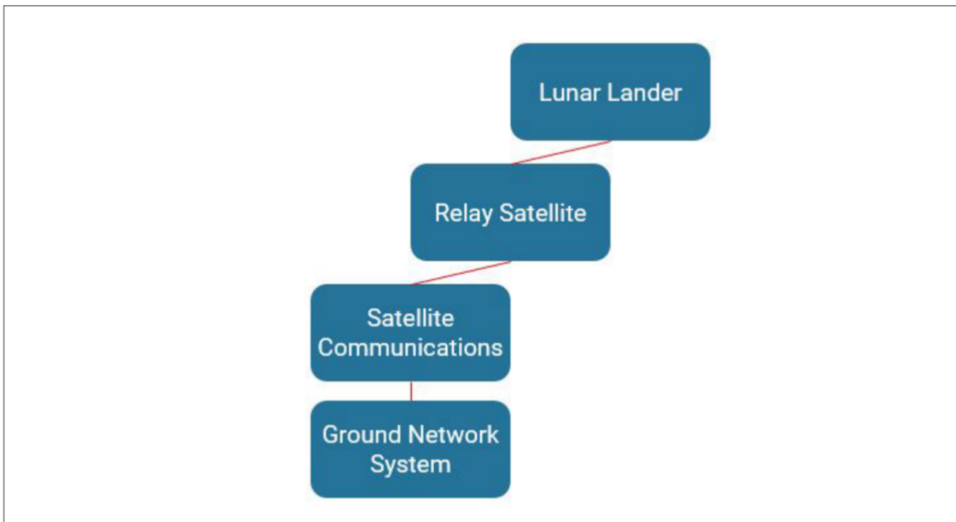


Figure 2-4. A concept sketch for a lunar landing system

This is enough to have hard discussions with as you focus your concept. It is at a high level. It focuses on the comprehensive system context, not one subsystem.

An Example

Imagine that we're to begin work on a new machine learning–based software project for the travel industry. We might create a parti for this software based on Athena. She is the Greek goddess of wisdom, strategy, craft, the harvest, and war, and advisor to travelers.

We ask, what possibilities does this suggest? Where does it direct our focus and attention? How can this create a theme for our design that supports coherence? Many things come to mind:

- The machine learning must not be tacked on to one aspect, but must be natively relevant in the entire scope.
- The Strategy pattern can be used to inject ever more implementations of stated algorithms. The system must create a new context for the business to pivot and support alternate growth.
- A focus must remain on craftsmanship and careful adherence to resilience.
- The system will bring the harvest, the new capabilities in retailing and offers.
- The system should offer exceptional user support through its interfaces, offering creative and just-in-time advice to travelers.

Copyrighted material

Now you have the basis, a grounding, for thinking about ways in which such ideas might be realized in the architecture. Pulling together these high-level contours under a single unifying personage as "Athena" makes sense. Capturing your concept in a unifying character, figure, name, or readily expressible idea will help you communicate your ideas with others who can help to refine the concept.

Define one supporting pole around which your idea can find another idea to enter a dialogue. Where do these ideas argue? On what basis? What do they try to persuade each other of? Where can they agree? Use that tension to create a space for the circulation of ideas.

Pick one pole, and design that entire pole. Now you have something to hang other ideas on, something that has survived the first round of interrogation. This will help prepare other aspects of the system.

Adding Aspects to the Parti

At this point, you've explored how one aspect of the system might work, how it can be useful, and powerful. Now change dimensions and design across the whole field, but only one inch deep so that you can see where the boundaries might be. You don't need to define them all firmly yet, but you've put a line on the horizon. You have one aspect thought through, and many others as points identified on the field.

You do not need to express the parti directly or map all of these elements to something concrete. It acts as an organizing principle and should be useful to you as you continue to mentally process and further explore and imagine the system. Eventually, your parti will find its way into a variety of concrete documents with design decisions, and the trick is to keep it in mind as you create these:

- Use case diagrams
- A deck outlining the design
- Class and component diagrams for key areas
- A complete architecture definition document

We address these in further depth in Chapter 5. The parti should not burden, but ignite.

The Parti Is Based on a Series of Reveals

> *I have always felt that if you know what you're going to do in advance, then you won't do it. Your creativity starts with whether you're curious or not.*
>
> —Frank Gehry

The parti must reveal, moment by moment, the key aspects of the story. It is nothing but a silly flight of fancy without a concrete realization. The parti is a disposable

Copyrighted material

bridge toward human use. It can lend an organizing principle to your design that allows people to intuit it better and support you in providing more ways to serve the customer, the human user, as they want to be, and as they might not yet have imagined they want to be.

A *reveal* is the careful dosage to the implementing teams of what they can understand. It is your job, not theirs, to provide the concrete links to the parti within aspects of the design. Eventually the parti will fall away altogether, having blossomed from abstraction into design diagrams into a working system.

Make the system for the extreme users: both the experienced power user who is able to do everything, like make their own macros, and the novice user who only cares about 10% of the functions should be able to easily and readily do the obvious jobs. Consider the extremes up front and play them against each other to provide something that works for both of them. Consider other spectra for extreme users: old and young, native speakers and nonnative speakers, women and men, short and tall, those who need the deep details and those who need the quick summary.

Know what and where your reveals are. Consider the people on your project and how you will implant the parti into everyday life.

Look for opportunities to express the concept in every aspect, across the templates you make, the hiring practices, the culture of the project team, the development life cycle, the milestones, the management, the ordering and prioritizing.

Do not expect too much of the parti. It has its moment of real value in capturing the concept, and then will fade away. New requirements, laws, and constraints will emerge. Change it or abandon it if and when necessary and reconceive based on new things you've learned. You must do this in order to retain the holistic integrity of the concept, not the original concept or your parti.

Let the system begin to speak to you. Enter into a dialog with your concept that hourly gains greater embodiment, through ever more avenues: the system diagrams, the use cases, the goals, and the ways to achieve them.

Let it change your course as it takes on more life of its own. You make the child, name it, teach it. Then, as the child grows, they show you that they're not a tiny version of you, but have their own values, desires, and methods; the child becomes your teacher.

As Eisenhower said, "Planning is indispensable. Plans are useless."

Copyrighted material

Understanding Ideas

> *Every block of stone has a statue inside it and it is the task of the sculptor to discover it. I saw the angel in the marble and carved until I set him free.*
>
> —Michelangelo

We do not understand the idea that represents our system. That is because it is incomprehensible. But also because it has not been our aim.

Michelangelo might have viewed his work as revealing the angel already within the marble. But the marble existed, and the only work was to chisel. The creative architect starts with emptiness, with nothing. And before him, a world of infinite conjuncts, a field, in which to assert some object anew. We have no marble.

When we approach system design in attempting to understand, we subvert our best efforts because we cannot understand what we have yet to invent.

We therefore seek instead to understand the idea of ideas, not the idea of our system or the solution we think we're making, but ideas themselves. Are we quite sure we know what an idea is?

Sense Certainty

> *See this? This is this. This ain't something else. This is this.*
>
> —Robert DeNiro, *The Deer Hunter*

We receive sensory data, a multiplicity of inputs, constantly. A filmed motion picture typically runs at 24 frames per second. The pictures are all still photographs. But as with a flip book, our minds fill in the transitions that are not truly there to give us the illusion of motion and continuity.

This is not thinking, but sensing. We do not have an idea. We have not mixed this stream of sense data with our own apprehension and conclusions. We have only complected sense. Nineteenth-century German philosopher Hegel calls this "sense-certainty," and it's sometimes called "picture-thinking."

We can, in this mode, believe that they understand utterances like "here," "now," and "this," concretely, as if they were direct referents—as if we think there is a fixed, understood definition of "here" or "now" or "this."

To be blunt, when we say these words, we believe we are saying something meaningful and that we know what we are talking about, when in fact we do not. Parsing these very commonly used words is almost impossible.

The distinction is critically important because our software projects are filled with the words of the requirements, the words of the design, and the words of the code. We must be crystal clear (as much as possible) that we are saying what we mean. When

Copyrighted material

we start to try to express what we have observed about the world in language, mixed with our ideas about their coherence, we begin to form concepts. These are the basis of strong designs.

Metacognition

One of the most important skills you can have as a designer is to cultivate your metacognitive ability. You notice yourself thinking about *how* you think, as you do it. You see not only your concepts, but you form more complex concepts and notice the manner in which you constructed them.

When you think about how you think, you call into question a variety of things:

- The sensory data you take in, respond to, recall, and retain, and how you respond to it, what you pick out, prioritize, conjoin, and disjoin.
- How you synthesize this data to represent it back to yourself as interpreted ideas.
- Your own understanding of yourself as a stable identity that can perform this apprehension consistently, with clarity.

Foregrounding your metacognition puts you in a dialog with yourself. Being in a dialog with yourself as if you were two people, perhaps arguing, will help you to quickly shape nothing into something. And that "something" will be better, more interesting, higher performing because you are considering it more carefully, more richly, with fewer assumptions and biases.

You can practice this by batting your concepts back and forth between seemingly disparate characteristics. Consider the following:

- Sturdiness and flexibility
- Distribution and performance
- Security and ease of use
- Simplicity and complexity
- Tall and short
- Wide and narrow
- Bright and dark
- Solid and void
- Stasis and circulation
- Presence and absence
- Software and hardware

Copyrighted material

- Business and philosophy
- Architecture and art

How are you privileging one term in the binary pair? What sense data, history, ideas, subliminal suggestions, constraints, laws, cultural norms, biases, stereotypes, and viewpoint led you to this privileging? How can you find the concept that unifies both terms in each pair, such that the trade-offs you make become no apparent trade-off at all?

Then, after you have incorporated the competing concerns and satisfied the constituent members of the British parliament arguing in your head to the point where you feel there are no longer opposites, you have a concept with integrity, harmony, and sturdiness, and one that is closer to bringing the design to its truth of the matter.

Criticize your own mental processes. Stand back and observe how you intake data, from where, and why. You are always absorbing data; this data continually shapes your mental space, the field which harvests thought. What can you observe about what you're taking in, to perform a habitual act of synthesis?

How can you then subvert or overturn that synthesis with a new perspective of apparently disparate or seemingly unrelated things? How is a raven like a writing desk?

Go shopping or to the park or to see a movie or listen to music or a lecture on something entirely unrelated to your design challenge. Not as a field trip with a stated aim, but as a quotidian act of noticing how your daily commute informs your design, how a crumpled paper might beget Gehry's Disney Concert Hall. All of these will inform your thinking, what you see as possibilities of relations, and give you raw material and metaphors to work with as you hone your concepts about the design and light a path toward what concepts your design in turn affords the world.

It is an act of pattern recognition, synthesis, and subversion.

Software is often broken, and often broken from the start, in its conceptual understanding of the world. As we have discussed, a software design represents our conception of a portion of the real world. Yet we cannot design and make the software that represents the world of infinite conjuncts; we would never be finished and go to market. So we must draw a line, a border, create a margin around some subset of this world as we conceive it to limit our scope to have something to build. And that we will call the domain. This is the set, the scope of the software, and it is at this horizon, the gap between our concept and our created field imposed on top of the phenomenological world, that computers must act rationally, decidably, given their inputs. Their inputs are only those within the field we demarcate, and their outputs only those that we allow. Despite our best efforts, at some point, the point of this horizon, we must stop and ship the software. And there is ambiguity at these borders, the

Copyrighted material

meeting points of the phenomenological world and our artificially superimposed field.

For example, we might be called upon to make a system to predict the price of homes. So, naturally, we define, among others, the class "House." We spend a million dollars on sophisticated machine learning projects to make better predictions. We do not understand why our prediction so often fail us. We included the attributes of age, square feet, acreage. But, fatally, did not include the attribute "proximity-to-the-beach": because we curtailed our semantic field there.

We cannot conceive of all the things. We cannot include all that we can conceive of. At some point, we must stop and make a compromise. Make these moments of compromise *conscious*, and this will mitigate the blow of the lie we're telling our system about its origins and context. This is the key aspect to better concepts, which are the supporting substructure of better software.

Context

> *Always design a thing by considering its next largest context: a chair in a room, a room in a house, a house in an environment, and environment in a city plan.*
>
> —Finnish architect Eliel Saarinen

There are only two kinds of problems in the world: trivial and nontrivial.

A trivial problem is straightforward. Its cause is direct, simple, and obvious. Its span of influence is small. Examples include pricking your thumb, or running out of paper towels. Its solution is similarly clear, direct, and simple. These are simple systems and the behavior of the constituent elements of simple systems is predictable.

We are not interested in those here.

A nontrivial problem is almost always more complex than at first it seems. Trendy practitioners will tell you to "Keep It Simple, Stupid." This is a useless and empty phrase. The problem is not simplicity versus complexity, and developers "making things complex." Sometimes things are in fact complex.

Imagine you are designing an ecommerce system. You have a database of Products, wherein you assign an ID and name and description. We know when we add products to our cart, we are asked for a quantity. So we add a column to the Products table for "quantity." That's the simplest thing to do. But this is absurd.

We learn from our quick trip into sets that here there are two concepts at work: the product, and the product-as-object-during-shopping-by-a-particular-customer. And that is a related, but different matter.

This thing has certain properties that are its essence, and then there are other new properties that are obtained only in the process of shopping; those cannot be

Copyrighted material

separated from that idea. There is no abstract quantity. So you must create something new. You might invent the *InventoryItem* or *CartProduct* to express this new relation: you have the user. The pencil doesn't have quantity=3; that decorating idea must exist to capture ideas that are not metadata about the product but are first-order properties of the shopped item.

This is the purpose of item *variants*. We think there is a "shirt." But a shirt is an abstraction. You can't sell it until you know its size and maybe its color and maybe its intended gender. Are we to make three rows for small, medium, and large shirts? What about color—we sell them in white, black, and blue. Are there then nine rows? Do we double each of these according to gender? This is an inefficient database design, and so this fault should call out to us that we are missing an idea—missing a part of our concept.

So seeing this disconnect we must create a new object: we create the idea of the *variant*. We now have created semantic space that allows these ideas of color, size, gender, and what-have-you to be full and rich in expression and be themselves extensible (if later we add one for men's and one for women's) but each have their integrity and maintain an efficient design.

It might seem counterintuitive after all these years of false conditioning to "keep it simple." But the smart designer enlarges the problem space. You create ideas that are semantically coherent with the overall design not to add complexity, but to make the inherent complexity of the world efficiently represented in your design. You see many contexts. You attempt to blow up and undermine your design the moment you see it leap to life, knowing it will be used many different ways, only some of which you intend.

Enlarging the problem space is about identifying multiple levels of causation. You have a problem: the user needs to do *X*. First, that might or might not be the problem. Ask why do they want to do that? In many cases, the user does not want to do at all the thing they are doing. They don't want to shop for that snazzy shirt and put it in their cart and buy it. They want to wear it. The shopping is a necessary evil to the wearing. This is an area in which Amazon simply excels.

You cannot solve all of these problems by continuing to trace things back in endless deferrals. But you can perhaps arrive at a different, more general solution. This often means that you can see many benefits, more than originally hoped for.

Often, it's just as easy to do it right as it is to settle for a lesser design because that will beget workarounds and compensations.

You can reduce the set later as needed to fit the timeline, budget, and other concerns.

Copyrighted material

Sets

As you saw in the previous example, design is about thinking in sets. In this view, we see the world as a collection of collections, each containing generally three element counts: zero, one, or many.

What belongs to this object necessarily and what doesn't? What does and doesn't belong together perhaps? What is optional to add on top?

Set theory is a rich and difficult study. For our purposes, two basic ideas will get us a long way:

Extension
: What belongs in this set? What is the name that puts these things in a group? For a retailer, the group might be "All the stores of Brand X," which is rather straightforward. Now you have something to call a stake in the ground. We continue, and posit "All the stores in Kalamazoo." But where is the border precisely, or is it a gerrymandering contorted border, a zip code, or set of them? What if they want to run a campaign that allows owners to set discounts for their own store, but Oscar owns several of them?

Essence
: Essence refers to that without which, not. That is, if you don't have some part of a thing, you can no longer say you still have that thing.

Determining essence is difficult, but essential in keeping the ambiguities at the margins to a minimum, which is what will undermine your design, and make it expensive and untoward to maintain.

If you take away your hand, are you still you? I think most people would agree that they are: they don't lose their identity because they lost their hand. They can still be found guilty of crimes and identified for tax purposes. How much of you can you lose before you are not you anymore? If you suffer early onset dementia with your body healthy and well intact, are you still you? These questions are difficult to determine. Luckily, software is not as complex as people are.

Naive Set Theory

For a good introduction to set theory, I encourage you to read the mathematics textbook *Naive Set Theory* by Paul Halmos from 1960. It's short and dense. For the truly impatient, make sure you're familiar with the concepts presented on the Wikipedia page.

Copyrighted material

Relations

We already understand relations, the connections between objects. My aim is to formalize and problematize that understanding just a little bit so that you design with the edges in mind. Let's take a moment to consider these key terms:

The Axiom of Pairing
: It is the case that for any two sets there exists a set that they both belong to. When you assert a figure into the field, ask what other sets it also is a member of. Then determine validity and priority.

Domain
: We use this word regularly in software. It comes from set theory, and more formally refers to the set of input or argument values for which some function is defined.

Range
: The difference between the lowest and highest values in a set.

Intersection
: The intersection of *A* and *B* is the set of all objects that are both in *A* and in *B*.

Union
: The set consisting of all objects that are elements of *A* or of *B* or of both. For every collection of sets, there exists a set that contains all the elements that belong to at least one set of the given collection.

Complement
: The set of all objects that belong to *A* but *not* to *B*.

There are three ways to talk about equivalence:

Reflexive
: A relation is *reflexive* if all the members of a set have the same relation to the set. So equality is a reflexive relation. "Less than" is not reflexive.

Symmetric
: A relation is *symmetric* if, for all *A* and *B* in a set *X*, *A* is related to *B* if and only if *B* is related to *A*. Examples include:

  - Is married to
  - Is a sibling of

Copyrighted material

Transitive

A relation is *transitive* if it has the following property: if *A* is related to *B* and *B* is related to *C*, then *A* is also related to *C*. Examples include:

- Being a subset of
- Implies
- Divides

Even though we might be familiar with some of these terms from programming languages and databases, using this lens in your system analysis and design is sure to come in handy. The only point here is to encourage you to explore your concepts using this framework of how objects relate to one another.

Advantages of Semantic Design

> *On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.*
>
> —Charles Babbage

So we have thought of ourselves awkwardly as engineers and architects, and we've enjoyed none of the materials, methods, or tools, and that has meant we have misconceived of our field and misapplied a lot of square pegs into a lot of round holes. The only thing in our field that comes close, really, to the discourse of engineering is that the speed of light means we can enjoy an understanding of the limit and measurable rate of data transfer.

With the advent of user stories in Scrum and related Agile methods, we have lost a lot of our focus on communicating coherently and specifically. This leads to a culture in which there's never enough time to do it right, but somehow always enough time to do it over. This makes projects fail. The Semantic methodology offers a list of documents that together make it practicable and repeatable in your organization, capturing an incredibly rich and robust set of perspectives on the software, with various forms of communication. It focuses equally on the functional and nonfunctional requirements, which are often missing.

But if you think through your concept, you will purposely reveal more of the semantic field that is your representation of the world. As you work through the concept, the semantics evolve and are challenged and refined. Your resulting ideas and the language and logic overall will be more sound, more robust, more comprehensive, and more customer-centric, and your requirements, both functional and nonfunctional, will be far, far better than what you're used to. Your design will be fit to purpose, sturdy, harmonious, and beautiful. You will have expressed. You will have created the

Copyrighted material

context in which fantastic software is born. That software will be reliable, maintainable, extensible, scalable, available, secure, and delightful to the user.

And that's the whole point.

There are a variety of other advantages in this method:

- It focuses the team and encourages them to be personally engaged and motivated.
- It unleashes more creativity.
- It offers informal methods for testing your logic and your biases at the point in the project where it will never be cheaper, quicker, or easier to change.
- It takes a comprehensive view. It's synthetic, from many sources, more open, less narrow and rigid. The ideas are native to software more so than engineering or architecture.
- It is failure-oriented, as much as success-oriented. By foregrounding opposites and contradictions and teasing them out, we predict more problems earlier and can work to prevent them.
- It encourages you to focus on not just dividing the existing cake, but on making more cakes because the only cost that matters in an innovative landscape is opportunity cost.
- It does not use metaphors that do not apply, which misguides our thinking. In software, that matters considerably given that logic and language are the only tools we have.
- Contrary to much of what we see in Agile, you insert the concept design as an upfront phase. This does not make it waterfall. And waterfall is not inherently bad. It is bad, however, to presume to spend years of dozens of people's time and millions of dollars of other people's money making software that you haven't thought through. Thinking it through as we outline here will make better requirements and make you far more likely to do it right the first time.
- The focus on setting the context helps developers be productive while owning—and being accountable for—the software they make.
- It's prescriptive in certain documents and very loose in other areas of the method. This allows for easy incorporation into the many other processes that you must or like to use, while retaining the flexibility of an Agile process.
- It underscores the multiplicity of "customers" of the software, which makes it more robust and usable to all the actual diverse users of your software.
- It sheds several false notions that lead us astray, such as a definition of "done." Software is almost never "done" the way a building gets done. One of the systems in my charge is nearly 20 years old, and yet 200 people still work on it every day.

Copyrighted material

They're not just doing operating system updates. An evolutionary approach works more naturally with how successful applications actually live in the real world. The semantic method establishes a framework for its further evolution by an array of teams and stakeholders.

- Because we foreground the concept and maximize context and extensibility, it is easier to adjust for changes, problems, or new ideas as they inevitably arise, minimizing churn. The abstractions will be at the optimal level across your design. Nonoptimal abstraction is often the way that lots of hacks and tacked-on additions begin to rise up like weeds or poorly executed additions to a house across your code, making it more difficult to maintain in the long run.
- Because a lack of timely, good decisions by the proper parties leads to failures, we include communication plans and clear semantic paths for working across teams in a complex environment. Decision making is an important part of the efficient flow.
- We foreground assumptions and list them along with requirements such that if they change, we can quickly plan for them.
- We thoughtfully align with the strategy and pave communication and decision routes between development teams and leadership. We do not assume, as other methods do, that software development teams exist in a vacuum, or only in some dark room decorated with *Star Trek* paraphernalia where executives never go except to slide pizza under the door. That isolation of the development teams is not one to maintain. When we foreground software design as a software problem instead of a semantic problem, we help build a wall that shouldn't exist. That wall creates divergence between the strategy and the local project and teams, which threatens the project. You can be "in the zone" when your alignment is clear.

Software projects fail because people don't know what they want, what they are making, why they're doing it, who makes what decisions about it, and what the abstractions and routes are to make those things clearer.

Our methods heretofore have improperly addressed these aspects, and they are the precise aspects of a software project that the semantic design method addresses. Let's dive deeper into what it is and how it works.

Copyrighted material

Copyrighted material

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.

Copyrighted material

```
                    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.

Copyrighted material

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.

Copyrighted material

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.

Copyrighted material

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

Copyrighted material

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.

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

Copyrighted material

(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).

Copyrighted material

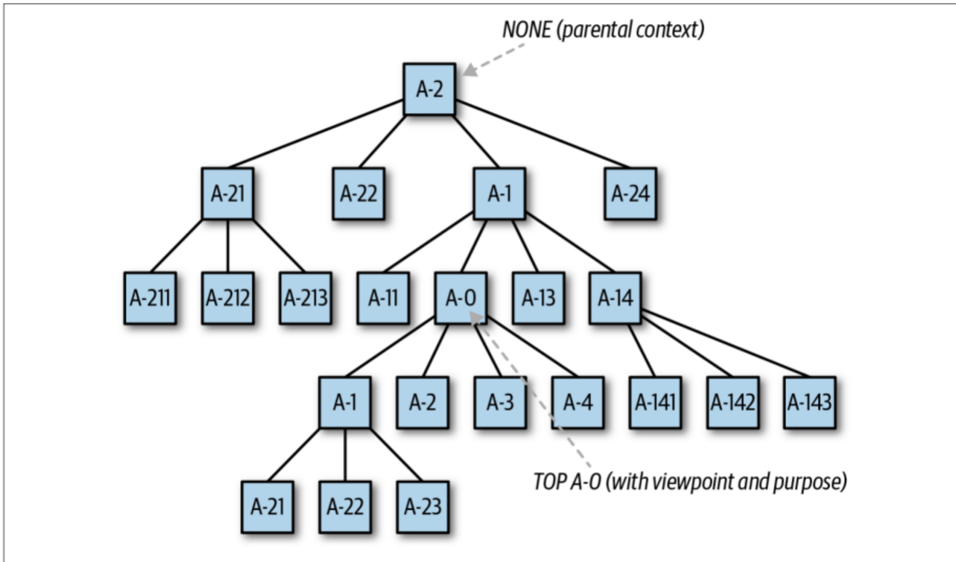


Figure 3-1. Decomposition (source: Wikipedia)

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.

Copyrighted material