The Essence of Software

Published by Princeton University Press
41 William Street, Princeton, New Jersey 08540
6 Oxford Street, Woodstock, Oxfordshire OX20 1TR
press.princeton.edu

All Rights Reserved
ISBN 978-0-691-22538-8
ISBN (e-book) 978-0-691-23054-2

British Library Cataloging-in-Publication Data is available
Editorial: Hallie Stebbins and Kristen Hop
Production Editorial: Jenny Wolkowicki
Jacket design: Emily Weigel
Production: Danielle Amatucci
Publicity: Sara Henning-Stout and Kate Farquhar-Thomson
Copyeditor: Bhisham Bherwani
Interior design: Daniel Jackson
Set in Adobe Arno Pro and Stone Magma using Adobe InDesign
Printed on acid-free paper. ∞
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Contents

How to Read This Book

A micromaniac is someone obsessed with reducing things to their smallest possible form. This word, by the way, is not in the dictionary.
—Edouard de Pomiane, French Cooking in Ten Minutes

Concept design is a simple idea that you'll be able to apply in your own work—using and designing software—without having to master any complex technicalities. Many of the concepts I'll use as examples will be recognizable old friends. So I'll take it as a compliment if your conclusion, after reading this book, is that concepts are a natural and even obvious way to think about software, and that you learned nothing more than a systematic framework for an intuitive idea.

But even if the underlying theme of this book resonates and seems familiar, I suspect that for many readers this new way of thinking about software will be disorienting, at least initially. Although software designers have talked for decades about conceptual models and their importance, concepts have never been placed at the center of software design. What would design look like if every software app or system were described in terms of concepts? What exactly would those concepts be? How would they be structured? And how would they be composed together to form the product as a whole?

To answer these questions as best I can, this book is longer than I would have liked. To mitigate that, I have organized it so that different readers can take different journeys through it. Some will want to reach a pragmatic destination as quickly and expeditiously as possible; others, wanting a deeper understanding, may prefer to follow me on some detours away from the main path. This little guide should help you plan your route.

Intended Audience

In short, this book is aimed at anyone interested in software, in design, or in usability. You might be a programmer, a software architect, or a user interaction designer; a consultant, an analyst, a program manager, or a marketing strategist; a computer science student, teacher, or researcher; or maybe just one of those people who (like me) enjoys thinking about why things are designed in certain ways—and why some designs succeed so gloriously and others fail so spectacularly.

No knowledge of computer science or programming is assumed, and although many of the principles in the book can be expressed more precisely in logic, no mathematical background is required. In order to appeal to as broad an audience as possible, I've drawn examples from a wide variety of popular apps, from word processors to social media platforms. So each reader will likely encounter examples that are easy to follow and others that require more effort. A byproduct of reading the book, I hope, will be a more solid grasp (and thus greater mastery) of apps that you use but don't yet fully understand.

Goals of This Book

This book has three related goals. The first is to present some straightforward techniques that software creators can apply immediately to improve the quality of their designs. By helping you identify and disentangle the essential concepts, articulate them, and make them clear and robust, the book will enable you to design software better—and thus design better software—whatever phase of design you work in, from the earliest phases of strategic design (in which products are imagined and shaped) to the latest phases (in which every detail of interaction with the user is settled).

The second goal is to provide a fresh take on software, so you can view a software product not just as a mass of intertwined functions, but as a systematic composition of concepts, some classic and well understood, and others novel and more idiosyncratic. With this new perspective, designers can focus their work more effectively, and users can understand software with greater clarity, empowering them to exploit their software to its fullest potential.

My third and final goal is broader and perhaps easier. It is to convince the community of researchers and practitioners who work on the development of software applications and services that the design of software is an exciting and intellectually substantive discipline.

Interest in software design—especially when focused on user-facing aspects—has waned over the last few decades, even as recognition of its importance has grown. In part this is due to the misconception that there is nothing inherent to the design of software that makes it more or less usable, and that any such judgments are subjective (or better addressed as psychological or social questions, with the focus more on the user than the software itself).

The rise of empiricism in software practice—while motivated by the apt recognition that even the best designs have flaws that only user testing will reveal—has also, in my view, dulled our enthusiasm for design, as many have come to doubt the value of design expertise. But mostly, I believe, we have suffered from a lack of respectability and intellectual confidence, since our ideas about what makes software usable have more often been expressed as tentative rules of thumb rather than principles grounded in a rich theory. I hope to show, in this book, that such principles and theory do indeed exist, and to encourage others to pursue their development and refinement.

Choosing Your Path

You can take different paths through the book, depending on your goals. To do so, it will help you to know how the book is organized and what each part contains.

Part I contains three motivational chapters. The first might have served as a preface: it explains why I came to write this book, and why the problems that I was interested in hadn't already been addressed in other fields (such as human-computer interaction, software engineering and design thinking). In the second chapter, we see our first examples of concepts, and the impact that they have on usability, and I explain concept design as the top of a hierarchy of user-experience design levels. The third chapter outlines the many roles that concepts have, from being product differentiators to being the linchpin of digital transformations.

Part II is the heart of the book. Its first chapter tells you exactly what a concept is, and how it can be structured. The second explains the fundamental idea of a concept's purpose as a motivation and a yardstick. The third shows how an app or system can be understood as a composition of concepts, combined using a simple but powerful synchronization mechanism; it explains how over- or under-synchronization can damage usability; and, more subtly, how some features that we traditionally view as complex and indivisible can instead be understood as synergistic fusions of distinct concepts. The fourth shows how mapping concepts to a user interface is not always as straightforward as you might imagine, and that sometimes the problem with a design lies not with the concepts per se but in their realization as buttons and displays. The last chapter in this part introduces a way to think about software structure at a very high level as a collection of concepts that are mutually dependent on one another—not that any concept relies on another concept for its correct working, but rather that only certain combinations of concepts make sense in the context of an app.

Part III introduces three key principles of concept design each in its own chapter: that concepts should be specific (one-to-one with purposes); that they should be familiar; and that the integrity of a concept should not be violated in composition (resulting in behaviors that, when viewed through the lens of an individual concept, do not satisfy that concept's specification).

The body of the book closes with a list of provocative questions addressed to readers in different roles. You might use these as a summary of the lessons of the book; or as a checklist in your subsequent practice; or even read them first as a quick overview of what the book offers.

If you want to jump in at the deep end, you could start with Part II; the motivations of Part I could be read as a conclusion, summarizing the ways in which the ideas you've learned can be applied. Each chapter ends with a summary of the main lessons and a list of practices that you can apply immediately.

Explorations & Digressions

Almost half the book is a collection of endnotes. I wrote the book like this because I wanted to make the body of the book as brief as possible, while also carefully justifying my approach and explaining its connection to existing design theories. So the main part of the book contains no discussion of related work (not even a single citation), ignores many subtle points, and leaves out many ideas I have about design more generally.

The endnotes make up for these omissions. There, I not only cite related work but attempt to put it all in context and explain its significance. I explain in much more detail the distinguishing characteristics of concept design, and I present examples that require more background (or persistence) to comprehend. When I have not managed to resist the temptation to fulminate (against rampant empiricism, or a myopic focus on defect elimination, for example), I have at least relegated my diatribes to these notes.

The notes are cited at appropriate points in the main text with superscript numerals; the first appears a few sentences from here. But to spare you the annoyance of flipping back and forth, I have grouped them into free standing sections with their own titles, so they can be read independently, and you can just dive in and read them randomly at your leisure.

Multiple Indexes

Rather than just one index, this book has four separate ones: an index of the applications from which examples are drawn; an index of concepts; an index of names; and a general index of topics. The entries under *concept* in the index of topics might be particularly useful, as they highlight the key qualities of concepts and point to various mini essays in the endnotes.

Warning: Micromaniac at Work

De Pomiane, author of the inimitable *French Cooking in Ten Minutes,*[1] confesses in his introduction to being a micromaniac. I readily admit to the same pathology. I don't want to hear that a design failed or succeeded for a myriad of amorphous reasons. Even if that were sometimes true, what use would it be? I want to get to the essence, to put my finger on the one essential spot, on the crucial design decision that launched the product to dizzying success or sunk the entire enterprise.

I am not naive, and I know that being cognizant of multiple factors in design—especially when analyzing the causes of accidents—is wise and sensible. But it's just not a valuable way to draw lessons from prior experiences. To do that, I believe we all need to be micromaniacs: to focus on the tiniest of details in the search for an elusive but potent explanation whose generalization offers a lasting and widely applicable lesson. So be warned: the devil is in the details—and the angels are too.[2]

PART I

MOTIVATIONS

1
Why I Wrote This Book

As an undergraduate in physics, I'd been entranced by the idea that the world could be captured by simple equations like $F = ma$. When I became a programmer, and later a computer science researcher, I gravitated towards the field of formal methods, because it promised to do something similar for software: to express its very essence in a succinct logic.

A Passion for Design

My main research contribution in the 30 years since my PhD has been Alloy,[3] a language for describing software designs and analyzing them automatically. It's been an exciting and satisfying journey for me, but I came to realize over time that the essence of software doesn't lie in any logic or analysis. What really fascinated me wasn't the question that consumed most formal methods researchers—namely how to check that a program's behavior conforms exactly to its specification—but rather the question of *design*.[4]

I mean "design" here in the same sense that the word is used in other design disciplines: the shaping of some artifact to meet a human need. Design, as the architect Christopher Alexander put it, is about creating a *form* to fit a *context*. For software, that means determining what the behavior of the software should be: what controls it will offer, and what responses it will provide in return. These questions have no right or wrong answers, only better or worse ones.[5]

I wanted to know why some software products seem so natural and elegant, react predictably once you master the basics, and let you combine their features in powerful ways. And to pinpoint why other products just seem wrong: cluttered with needless complexity, and behaving in unexpected and inconsistent ways. Surely, I thought, there must be some essential principles, some theory of software design, that could explain all of this. It would not only explain why some software products are good and some are bad, but it would help you fix the problems and avoid them in the first place.

Design in Computer Science and Other Fields

I started to look around. Within my own subfield (formal methods, software engineering and programming languages), such a theory exists for what you might call "internal design"—namely the design of the structure of the code. Programmers have a rich language of design, and well-established criteria for what distinguishes good designs from bad ones. But no such language or criteria exist for software design in the user-facing sense, namely design that determines how software is experienced as a form in context.[6]

Internal code design is very important and influences primarily what software engineers call "maintainability," which means how easy (or hard) the code is to change over time as needs evolve. It also influences performance and reliability. But the key decisions that determine whether a software application or system is useful and fulfills its users' needs lie elsewhere, in the kind of software design in which the functionality and the patterns of interaction with the user are shaped.

These big questions were at one time more central in computer science. In the field of software engineering, they came up in workshops on software design, specification and requirements; in the field of human-computer interaction, they permeated early work on graphical user interfaces and computational models of user behavior.[7]

But as time passed, they became less fashionable, and they faded away. Research in software engineering narrowed, and eliminating defects—whether by testing or more sophisticated means such as program verification—became synonymous with software quality.[8] But you can't get there from here: if your software has the wrong design, there's no amount of defect elimination that will fix it, short of going back to the very start and fixing the design itself.[9]

Research in human-computer interaction (HCI) shifted to novel interaction technologies, to tools and frameworks, to niche domains, and to other disciplines (such as ethnography and sociology). Both software engineering and HCI embraced empiricism enthusiastically, largely in the misguided hope that this would bring respectability. Instead, the demand for concrete measures of success seems to have led researchers towards less ambitious projects that admit easier evaluation, and has stymied progress on bigger and more important questions.[10]

Puzzlingly, even as interest in design seems to have waned, talk of "design" is everywhere. This is not in fact a contradiction. The talk, almost exclusively, is about the *process* of design, whether in the context of "design thinking" (a compelling packaging of iterative design processes), or of "agile" software development. These processes are undoubtedly valuable (so long as they are applied judiciously and not as panaceas), but they are for the most part content-free. I mean that not to disparage but to describe. Design thinking, for example, might tell you to develop your solution hand in hand with your understanding of the problem, or to engage in alternating phases of brainstorming ("divergence") and reduction ("convergence"). But no design thinking book that I have read talks in depth about any particular designs and how the process sheds light on them. The very domain-independence of design thinking may be the key to its widespread appeal and applicability—but also the reason it has little to say about deeper challenges of design in a particular domain such as software.[11]

Clarity & Simplicity in Design

When I began the Alloy project, with the goal of creating a design language that was amenable to automatic analysis, I was critical of existing modeling and specification languages whose lack of tool support rendered them "write-only." This snide dismissal was not entirely unwarranted. After all, why would you go to the trouble of constructing an elaborate design model if you couldn't then do anything with it? I argued, in particular, that the designer's effort should be rewarded immediately with "push-button automation" that would instantly give you feedback in the form of surprising scenarios that would challenge you to think more deeply about your design.[12]

I don't think I was wrong, and Alloy's automation did indeed change the experience of design modeling. But I had underestimated the value of writing down a design. In fact, it was a not very well guarded secret amongst formal methods researchers (who were eager to demonstrate the efficacy of their tools by finding flaws in existing designs) that a high proportion of the flaws were detected *before* the tools were even run! Just transcribing the design into logic was enough to reveal serious problems. The software engineering researcher Michael Jackson credits not the logic per se but the very difficulty of using it, and once mischievously suggested that the quality of software systems might be improved if designers were simply required to record their designs in Latin.

Clarity is good not only for finding design flaws after the fact. It is also the key to good design in the first place. In teaching programming and software engineering over the last thirty years, I've become increasingly convinced that the determinant of success when you're developing software isn't whether you use the latest programming languages and tools, or the management process you follow (agile or otherwise), or even how you structure the code. It's simply whether you know what you are trying to do. If your goals are clear, and your design is clear—and it's clear how your design meets the goals—your code will tend to be clear too. And if something isn't working, it will be clear how to fix it.[13]

It is this clarity that distinguishes great software from the rest. When the Apple Macintosh came out in 1984, people could see immediately how to use folders to organize their files; the complexities of previous operating systems (such as Unix, which made even the command to move files between folders complicated) seemed to have evaporated.

But what exactly is this clarity, and how is it achieved? As early as the 1960s, the central role of "conceptual models" has been recognized. The challenge was not merely to *convey* the software's conceptual model to the user so that her internal version ("mental model") was aligned with the programmers', but to treat it as a subject of design in its own right. With the right conceptual model, the software would be easy to understand and thus easy to use. This was a great idea, but nobody seems to have pursued it, and so until now "concepts" have remained a vague, if inspiring, notion.[14]

How This Project Came About

Convinced that conceptual models were indeed the essence of software, I started about eight years ago trying to figure out what they might be. I wanted to give them concrete expression, so that I could point to some software's conceptual model, compare it to others (and to the mental models of users), and have an explicit focus for design discussions.

That didn't seem so hard. After all, a plausible first cut at a conceptual model might be just a description of the software's behavior, made suitably abstract to remove incidental and "non-conceptual" aspects (such as the details of the physical user interface). What proved much harder was finding appropriate structure in the model. I had an inkling that a conceptual model should be made up of concepts, but I didn't know what a concept was.

In a social media app such as Facebook, for example, it seemed to me that there should be a concept associated with liking things. This concept surely wasn't a function or action (such as the behavior bound to the button you click to like a post); there are too many of those, and they only tell part of the story. It also surely wasn't an object or entity (such as the "like" itself that your action produced), since at the very least the concept seemed to be about the *relationship* between things and their likes. It also seemed essential to me that the concept of liking was not associated with any particular kind of thing: you could like posts, comments, pages, and so on. The concept, in programming lingo, is "generic" or "polymorphic."

This Book: Opening a Conversation

This book is the result of my explorations to date. Driven by dozens of design issues in widely used applications, I've evolved a new approach to software design, refining and testing it along the way. A happy aspect of this project has been that every app failure or frustration had a silver lining: a chance to extend my repertoire of examples. It has also given me greater sympathy and respect for the designers when my analysis revealed the full complexity of the problem they faced.

Of course, the problem of software design is not solved. But as my friend Kirsten Olson wisely advised me: a book should aim to start a conversation, not to end one. In the course of giving many talks about this project, I've been thrilled to discover that it seems to resonate with audiences more than any of my previous ones. I suspect this is because software design is something we all want to talk about, but we have not known how to have that conversation.

So to you, my readers—fellow researchers, designers and users—I present this book as my opening gambit in what I hope to be a fruitful and enjoyable conversation.

2
Discovering Concepts

A software product—from the smallest app that runs on your phone to the largest enterprise system—is made of *concepts*, each a self-contained unit of functionality. Even though concepts work in tandem for a larger purpose, they can be understood independently of one another. If an app is like a chemical mixture, concepts are like molecules: although bound together, their properties and behavior are similar wherever they are found.

You're already familiar with many concepts, and know how to interact with them. You know how to place a phone *call* or make a restaurant *reservation*, how to *upvote* a comment in a social media forum and how to organize files in a *folder*. An app whose concepts are familiar and well designed is likely to be easy to use, so long as its concepts are represented faithfully in the user interface and programmed correctly. In contrast, an app whose concepts are complex or clunky is unlikely to work well, no matter how fancy the presentation or clever the algorithms.

Since concepts have no visible form, they're rather abstract, and this is perhaps why they haven't been a focus of attention until now. I hope to persuade you, in the course of this book, that by thinking in terms of concepts, and by "seeing through" user interfaces to the concepts that lie behind them, you will be able to understand software more deeply—to use it more effectively, to design it better, to diagnose flaws more precisely, and to envision new products with greater focus and confidence.

We don't generally appreciate how something works until it breaks. You may think that your water heater just magically produces a constant stream of hot water. But then at some point someone in your household takes one shower too many, and your shower is cold. That's when you might learn that your water heater has a *storage tank* with limited capacity.

Likewise, to learn about concepts, we need to see what happens when they go wrong. Much of this book, therefore, will involve examples of concepts that fail in seemingly unlikely scenarios, or that turn out to be much harder to understand than you'd expect them to be. In this chapter, we'll see our first examples of concepts, and how they can explain some unexpected (and surprisingly complicated) behaviors.

But don't be put off, or draw the conclusion that the *idea* of concepts is itself obscure and complicated. On the contrary, the idea is straightforward, and adopting it will help you to design software that is simpler and more powerful than much of the software we use today.

A First Example: Baffling Backups

To protect my work from corrupted disks and accidental deletion, I use a terrific backup utility called Backblaze, which copies my files to the cloud, and lets me restore old versions if I need to. It runs invisibly and continuously in the background, keeping an eye on every file in my computer, copying it to the cloud if it changes.

Recently, I edited a video and wanted to make sure the new version had been backed up before I deleted the old one to save space. I checked the backup status, and it said "You are backed up as of: Today, 1:05 PM." Since I had created the new video *before* 1:05 PM, I assumed it had been backed up. Just to be sure, I tried to restore it from the cloud. But it wasn't there.

I contacted tech support, and they explained to me that files aren't exactly backed up continuously. There's a periodic scan that compiles a list of new or modified files; when the next backup runs, only files on that list are uploaded. So any changes made between the scan and the backup fall between the cracks until they're discovered in the next scan.

I could force a rescan, they told me, by clicking the "Backup Now" button while holding down the option key. I followed this advice, and waited for the scan and subsequent backup to complete. Now, surely, my new video would show up on the restore list! But no such luck. At this point, I was totally confused, and asked for more help. It turned out that my video *had* been uploaded, but only to a special "staging" area, from which files are moved to the restore area every few hours.

My problem was that I misunderstood the key *backup* concept of Backblaze. I had imagined that files were uploaded continuously, and moved directly to the restore area (Figure 2.1, left). In fact, only the files on the list produced by the last

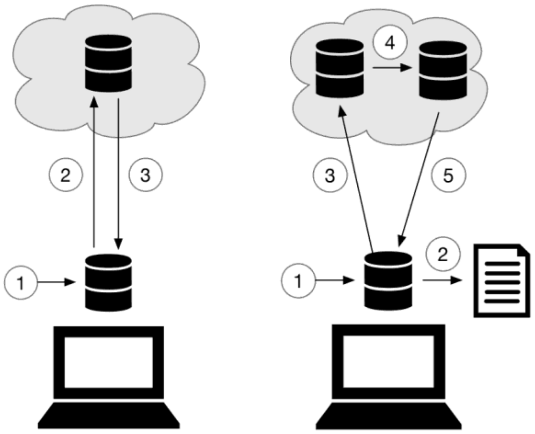


FIG. 2.1 *Backblaze's backup concept. On the left, what I assumed: (1) I make a change to a file; (2) when the backup runs, the file is copied to the cloud; (3) I can then restore it. On the right, what actually happens: (1) I make a change to a file; (2) a scan runs and adds the file to a list of files for backup; (3) the backup runs, copying to the cloud only those files that were added in the last scan; (4) periodically, backed-up files are moved to a cloud location (5) from where they can be restored.*

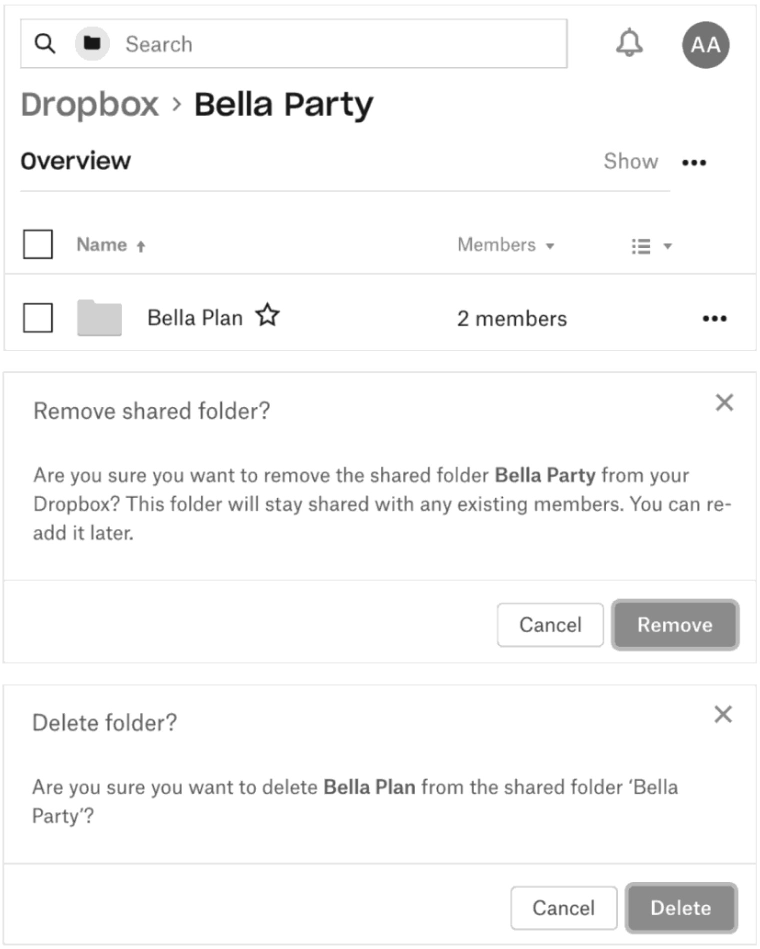
scan are uploaded, and even then remain unavailable until they have been transferred sometime later from the upload destination to the restore area (Figure 2.1, right).

This is a small example but it illustrates my key point. I'm not taking a stand on whether the design of Backblaze is flawed or not; I suspect it could be improved though (see Chapter 8 for a suggestion). Certainly, had I taken the backup message at face value and not known about the scan, I might have lost some crucial files.

What I *am* claiming is that any discussion of this design must revolve around the fundamental concepts, in this case the *backup* concept, and an assessment of whether the behavioral pattern that it embodies is fit for purpose. The user interface matters too, but only to the extent that it serves the app's concepts by representing them to the user. If we want to make software more usable, concepts are where we must start.

Dropbox Delusions

A friend of mine was running out of space on her laptop. So she cleverly sorted the files by size, and looked down the list to see if there were any large and



Remove shared folder?

Are you sure you want to remove the shared folder **Bella Party** from your Dropbox? This folder will stay shared with any existing members. You can re-add it later.

Delete folder?

Are you sure you want to delete **Bella Plan** from the shared folder 'Bella Party'?

FIG. 2.3 *Dropbox's folder deletion messages. The folder Bella Party has been shared (top). If that folder is deleted, the message (middle) informs you that the deletion will not be propagated to other users. If the folder Bella Plan contained within it is deleted, a different message (bottom) appears, surprisingly not warning that other users will lose the folder too.*

Explaining Dropbox

To see what's going on in these sharing scenarios, it helps first to articulate what our expectations might have been. A simple and familiar design for names would treat them as if they were sticky labels attached to physical objects—like a cat collar, or a license plate—with at most one label per object (Figure 2.4, left). We might call this approach "name as metadata," and it would be an in-

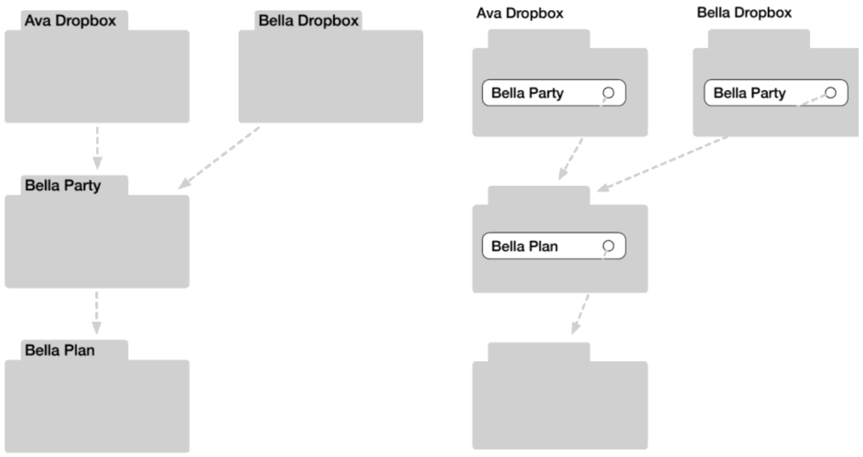


FIG. 2.4 *Two possible concepts for folders in Dropbox: in the metadata concept (left), names are labels attached to folders; in the unix folder concept (right), names belong to entries within the parent folder.*

stance of a more general *metadata* concept in which data that describes an object—such as the title or caption of a photo—can be attached to it.

With respect to deletion, the simplest design would be to make a file or folder disappear when it's deleted. We might call this (using a technical term) the "deletion as poof" approach to deletion: you click delete, and "poof!"—it's gone. The underlying concept here—that a pool of items can be stored, with actions to add and remove items from the pool—is so basic and familiar it has no name. In this design, we'd expect a separate *sharing* concept, with an *unshare* action so that you can remove a file or folder that someone else shared with you and free up the space in your own account without deleting their copy.

Both of these understandings—that names are metadata and deletion simply removes items from a pool—are wrong (at least for Dropbox). The concepts behind these understandings are themselves fine; they're just not the concepts that Dropbox uses. If you hold the wrong conceptual model of a software app, you might get away with it for a while. We've seen that in some scenarios these explanations would work successfully. But in other scenarios, they'll fail, perhaps with disastrous consequence.

The actual concepts that Dropbox uses are very different (Figure 2.4, right). When an item sits in a folder, the name of that item belongs *not* to the item itself but rather to the folder containing it. Think of a folder as being a collection of tags, each containing the name of an item (a file or folder) and a link to it. This concept, which I'll call *unix folder*, was not invented by Dropbox, but, as its name suggests, was borrowed from Unix.[15]

Look at the diagram in Figure 2.4 (right). Each of Ava and Bella has her own, top-level Dropbox folder, and these two folders have *separate* entries for the single, shared folder called *Bella Party*. When Bella renames *Bella Party*, this alters the entry in her own Dropbox folder, and the entry in Ava's folder is unchanged.

In contrast, there is only a single entry holding the name of the second-level shared folder, *Bella Plan*, belonging to the single, shared parent folder called *Bella Party*. Since there is only *one* entry for the folder—the same entry seen by both Ava and Bella—when Bella renames the folder, she is changing that one entry in their shared folder, so Ava sees the change too.

Using this same *unix folder* concept, we can now explain the deletion behaviors. Deletion doesn't remove the folder per se; it removes its entry. So if Bella deletes the folder *Bella Party*, she removes the entry from her own folder, and Ava's view is unchanged. But if Bella deletes *Bella Plan*, she removes the entry from the shared folder, and the deleted folder is now inaccessible to Ava too.

What Kind of Flaw is This?

At this point, you might be saying to yourself: Well, this is all obvious. I knew Dropbox behaved like this and I'm not in the least bit surprised. There's nothing wrong with Dropbox, and someone who doesn't understand it shouldn't be using it. But if you think this, I'm pretty sure you'd be in the minority of readers. We presented this scenario to MIT computer science students and found that many of them, even those who used Dropbox regularly, were confused.[16]

Even if you understood all these subtleties, I'd argue that there's still a problem. The distinction between the two cases—whether the folder that is the subject of the action is shared at the top level, or belongs to another folder that is itself shared—isn't readily discernible in the user interface, so it's a constant annoyance having to figure out which situation you're in.

Moreover, it doesn't seem reasonable that this rather arbitrary distinction should determine the behavior. Why should I be able to give my own name only to the top-level folder? Why can't I give private names to all the folders shared with me? Or conversely, if renaming folders for both of us is part of our shared work, why can I only do it for some folders and not others?

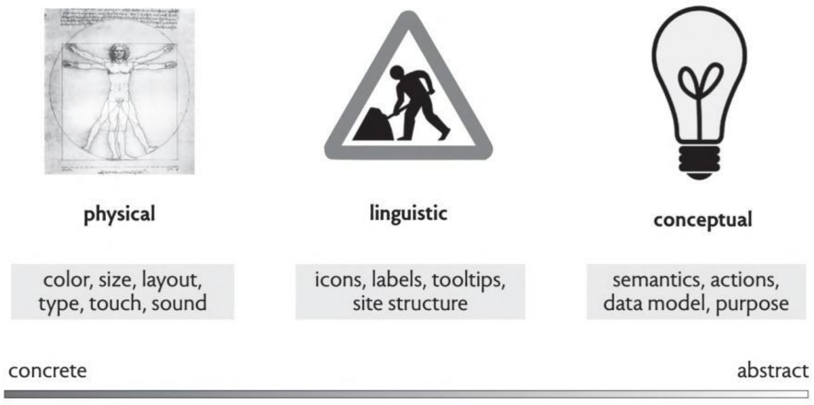

physical
linguistic
conceptual
color, size, layout, type, touch, sound
icons, labels, tooltips, site structure
semantics, actions, data model, purpose
concrete
abstract

FIG. 2.5 *Levels of interaction design.*

Assuming then, that these scenarios are indeed evidence of a flaw in Dropbox, we can ask: what kind of flaw is it? It's certainly not a bug; Dropbox has behaved like this for years. We might wonder if it's a flaw in the user interface. That seems implausible too. It would be possible, of course, for Dropbox to give more informative messages when a change you make affects other users. But this might just be perceived as additional complexity, and experience suggests that users ignore warning messages if they come up too often.[17]

The real problem runs deeper. It's in the very essence of how files and folders are named, and how those names are related to the containment relationship between folders and their contents. This is what I call a *conceptual* design issue. The flaw is that the Dropbox developer has certain concepts in mind that have been faithfully implemented. But those concepts, at the very least, are not consistent with the concepts in most users' minds. And, at worst, these concepts are not a good match for the users' purposes.[18]

Levels of Design

To put conceptual design in perspective, it helps to break software design into levels, as shown in Figure 2.5. This classification is my own, but it is similar to schemes previously proposed.[19]

The first level of design, the *physical level*, is about the physical qualities of the artifact. Even software whose interface is no more than a touch-sensitive piece of glass has such qualities, limited though they might be.[20] At this level, the designer must take into account physical capabilities of human beings. It's

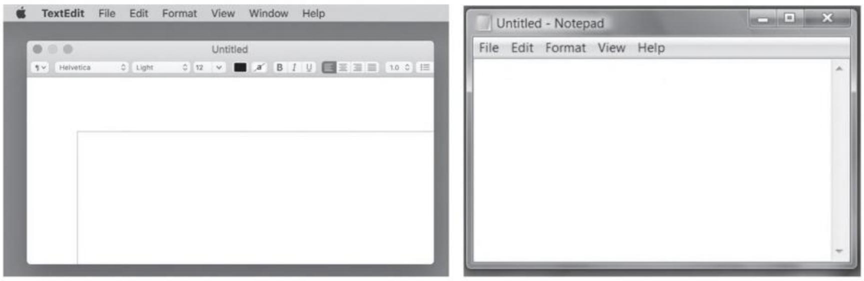

TextEdit File Edit Format View Window Help
Untitled
Helvetica
Light
12
B I U
1.0
Untitled - Notepad
File Edit Format View Help

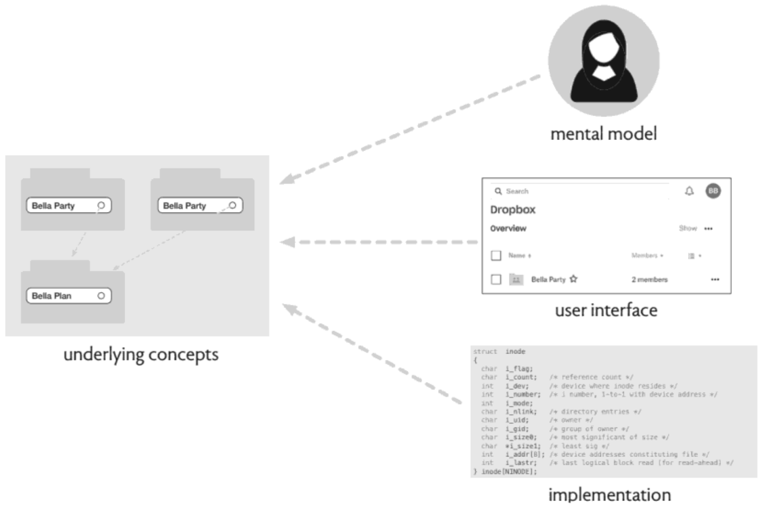
FIG. 2.6 *A design issue at the physical level, and a classic example of applying Fitts's Law. Which menu placement allows for more convenient access: the macOS placement (on the left) in which an application's menu bar always appears at the top of the desktop, or the Windows placement (on the right) in which the menu bar is part of the application window?*

where accessibility concerns arise, as the designer considers how a visually impaired, or color-blind, or deaf user might interact.

Common human characteristics dictate certain design principles. For example, the fact that our limited visual sampling rate results in *perceptual fusion,* making it hard to distinguish events that occur within about 30 ms of each other, suggests that 30 frames/second is enough for a movie to look smooth. It also tells us that system reactions that take much longer than 30 ms will be perceived as delays by the user, and should be avoided, or given progress bars, and if very much longer, an opportunity to abort. Likewise, Fitts's Law predicts the time it takes for a user to move a pointing device to a target, and explains why the menu bar should be positioned at the top of the screen, as in the Macintosh desktop, and not in the application window, as in Windows (Figure 2.6).[21]

The second level of design is the *linguistic level.* This level concerns the use of language for conveying the behavior offered by the software, to help the user navigate the software, understand what actions are available and what impact they will have, what has happened already, and so on. While design at the physical level must respect diversity amongst the physical characteristics of its users, design at this level must respect differences of culture and language.

Obviously, the button labels and tooltips on an app will vary depending on whether it's intended for English or Italian speakers. (I remember as a small child on holiday in Italy learning the hard way that the faucet marked *calda* is not the cold one.) The designer must be aware of cultural differences too. In Europe, a road sign comprising a red circle with a white interior means that no vehicular traffic is permitted at all; most American drivers would not be able to



```
struct  inode
{
  char  i_flag;
  char  i_count;   /* reference count */
  int   i_dev;     /* device where inode resides */
  int   i_number;  /* i number, 1-to-1 with device address */
  int   i_mode;
  char  i_nlink;   /* directory entries */
  char  i_uid;     /* owner */
  char  i_gid;     /* group of owner */
  char  i_size0;   /* most significant of size */
  char  *i_size1;  /* least sig */
  int   i_addr[8]; /* device addresses constituting file */
  int   i_lastr;   /* last logical block read (for read-ahead) */
} inode[NINODE];
```

FIG. 2.8 *The central role of concepts (left) in aligning the user's mental model (top right) and the developer's design model embodied in the code (bottom right). By mapping the concepts carefully to the user interface (middle right), the concepts are not only fully supported but also conveyed implicitly to the user.*

Figure 2.8 depicts this. At the top, there is the user; at the bottom, the code written by the programmer; and between the two, the user interface. For the software to be successful, we need to understand the user (by investigating her needs, working environment and psychological qualities); ensure that the code meets its specification (by testing, review and verification); and craft a usable interface. But most important of all is aligning the model in the head of the user with the model in the head of the programmer, and that is achieved by explicitly designing concepts that are shared by user and programmer alike, and conveyed clearly in the user interface.

Lessons & Practices

Some lessons from this chapter:

· Major usability problems in software applications can often be traced to their underlying concepts. In Dropbox, for example, confusions over whether deletions will affect other users is explained by Dropbox's adoption of a concept that originated in Unix.

· Software design is conducted at three levels: the physical level, which involves designing buttons, layouts, gestures and so on that are matched to the physical and cognitive capabilities of human users; the linguistic level, which involves designing icons, messages and terminology to communicate with users; and the conceptual level, which involves designing the underlying behavior as a collection of concepts. The two lower levels are concerned with representing the concepts in the user interface.

- For users, having the right mental model is essential for usability. To ensure this, we need to design concepts that are simple and straightforward, and to map the concepts to the user interface so that the concepts are intelligible and easy to use.

And some practices you can apply now:

- Take an app you have trouble using. Ask yourself what concepts are involved, and check that your hypotheses about how they work match the actual behavior. If not, can you find different concepts that explain the behavior more accurately?
- As the designer of an app, consider the functions that users find hardest to use (or easiest to misuse). Can put your finger on one or more concepts that are responsible?
- When designing, know what level you're working at. Start at the conceptual level, and move down. Sketching concepts out at the lower levels can help you grasp them more intuitively, but resist the temptation to polish the physical interface (for example, worrying about typefaces, colors and layout details) before you have a clear sense of your concepts.
- When you hear complaints about an app that focus on physical or linguistic aspects, ask if the underlying issue may lie at the conceptual level instead.

3
How Concepts Help

In traditional design disciplines, design evolves from a conceptual core. This core differs from field to field. Architects call it the *parti pris*: an organizing principle for the work that follows, represented by a diagram, a short statement or an impressionistic sketch. Graphic designers call it *identity*, and it typically comprises a few elements that capture the spirit of the project or organization. Composers build music around *motifs*—sequences of notes—that can be altered, repeated, layered, and sequenced together to form larger structures. Book designers start from a *layout* that specifies the dimensions of the text block and margins, and the typefaces and sizes in which the text will be set.

When the core is well chosen, the subsequent design decisions can seem almost inevitable. The design as a whole emerges with a coherence that makes it look like the product of a single mind even if it was the work of large team. Users perceive a sense of integrity and uniformity, and the underlying complexity gives way to an impression of simplicity.

For a software application, the conceptual core consists of—no surprise here—a collection of key concepts. In this chapter, we'll explore the roles that such concepts play, such as characterizing individual applications, application families, and even entire businesses; exposing complexity and usability snags; ensuring safety and security; and enabling division of labor and reuse.

Concepts Characterize Apps

If you're trying to explain an app, outlining the key concepts goes a long way. Imagine encountering someone who time-traveled from the 1960s and wanted to know what Facebook (Figure 3.1) was and how to use it. You might start with the concept of *post*, explaining that people author short pieces that can be read by others; that these are called "status updates" in Facebook (and "tweets" in Twitter) is a small detail. Then there's the concept of *comment*, in which one person can write something in response; the concept of *like* in which people

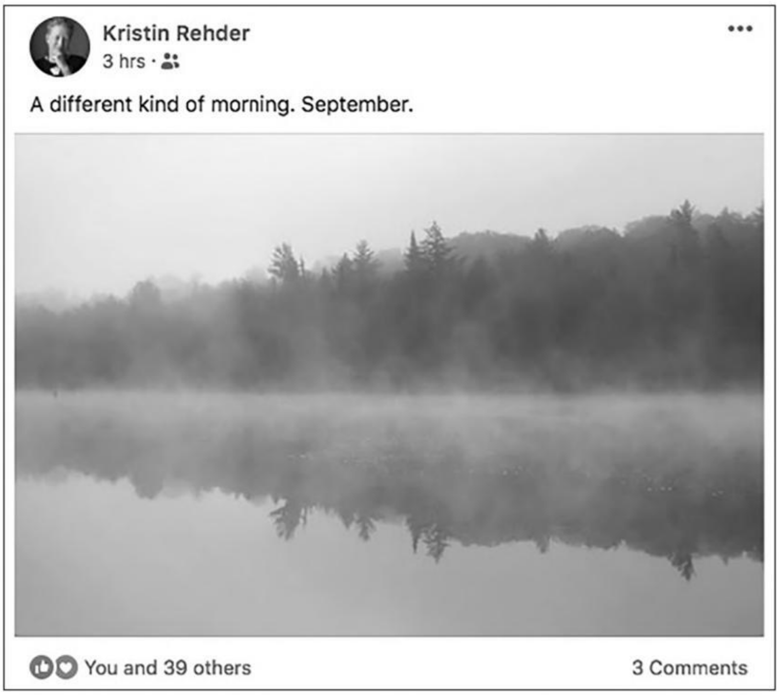
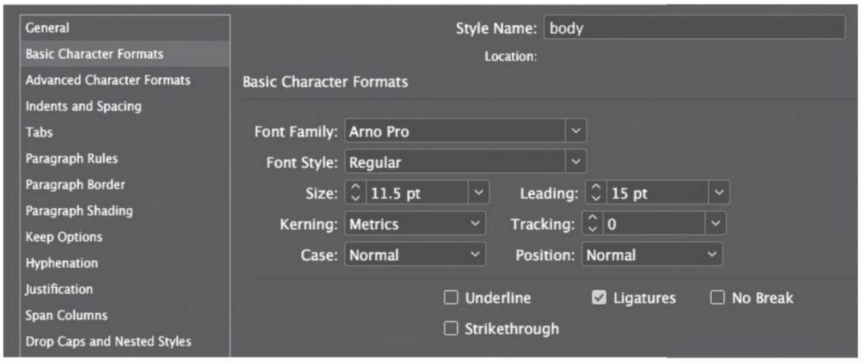


FIG. 3.1 *A screenshot of Facebook in which three concepts are evident: post (represented by the message and the associated image), like (represented by the emoticons at the bottom left), and comment (represented by the link on the bottom right).*

can register approval of a post, purportedly to boost its display ranking; and of course the concept of *friend* that is used both to filter what's shown to you and to provide access control so you can limit who sees your posts.

The difference between apps that offer similar functionality can often be explained by comparing their concepts. For example, a key difference between text messaging and email is that text messages are organized using a *conversation* concept in which all messages sent to a particular recipient appear; email messages, in contrast, are typically organized using concepts such as *mailbox, folder* or *label*. This is partly because the senders and recipients of text messages are uniquely identified by their phone numbers, whereas email users tend to have multiple addresses, which makes grouping into conversations unreliable. It also reflects different modes of interaction, with text messages relying on the context of the conversation and email messages more often interpreted in isolation (and thus often quoting previous messages explicitly).



Style Name: body
Location:
Basic Character Formats
Font Family: Arno Pro
Font Style: Regular
Size: 11.5 pt Leading: 15 pt
Kerning: Metrics Tracking: 0
Case: Normal Position: Normal
☐ Underline ✓ Ligatures ☐ No Break
☐ Strikethrough

FIG. 3.2 *The style concept, in Adobe InDesign, showing one tab of the formatting settings for a style called "body," which is the style associated with regular paragraphs in this book.*

Sometimes it takes experience and expertise to identify the key concepts in an app. Novice users of Microsoft Word, for example, might be surprised to learn that its central concept is *paragraph*. Every document is structured as a sequence of paragraphs, and all line-based formatting properties (such as leading and justification) are associated with paragraphs rather than lines. If you want to write a book in Word, you won't find any concepts that correspond to its hierarchical structure—no *chapter* or *section*, for example—and headings are treated as paragraphs like any other. Word achieves its flexibility and power through the *paragraph* concept and by the powerful way in which it is combined with other concepts.[26]

Concepts Characterize Families

Concepts not only distinguish individual apps, but also unify families of apps. Programmers, for example, commonly use *text editors* (such as Atom, Sublime, BBEdit and Emacs) to edit program code; people use *word processors* (such as Word, OpenOffice and WordPerfect) to create documents of all sorts; and professional designers use *desktop publishing* apps (such as Adobe InDesign, QuarkXPress, Scribus and Microsoft Publisher) to organize documents into finalized layouts in books and magazines.

The key concepts of text editors are *line* and *character*. The *line* concept embodies both powerful functionality (such as the ability to perform "diffs" and "merges" which are essential to programmers for managing code) as well as limitations (notably that there is no distinction between a line break and a though, I realized that I needed to understand the core concepts more deeply, so I found a book that explained layers and masks (and channels, curves, color spaces, histograms, etc.) from a conceptual point of view, and I was then able to do whatever I wanted.

Some of the most complex concepts appear in apps that are widely used by non-experts. Browser apps include the *certificate* concept for checking that the server you're talking to belongs to the company you expect—your bank, for example, rather than an interloper trying to steal your credentials—and offer the *private browsing* concept to prevent your browsing information from being available to others after you've logged out. Despite the critical importance of these concepts for security, they are poorly understood. Most users have no idea how certificates work and what they're for, and they often think that private browsing allows them to visit sites without being tracked.

Worse, some of the most basic behaviors of browsers rely on complex concepts that are invisible to most users. The *page cache* concept, for example, is used by website developers to make pages load more quickly, by using previously downloaded content. But the rules for when old content is replaced (and how these rules are modified) are obscure even to some developers, so users and developers alike may be uncertain about whether content that appears in the browser is fresh or not.

Highlighting tricky concepts is helpful because of the focus it brings. It tells us, as users, what we need to learn: if you want to be a power user, just ignore all the details of the interface—those will come easily later—and master the handful of key concepts. It helps us, as teachers, to focus on the essence: so when we teach web development, for example, we can explain the important concepts—sessions, certificates, caching, asynchronous services, etc.—without getting caught up in the idiosyncrasies of particular frameworks. And it suggests opportunities to us as designers for innovation. A better concept for server authentication, for example, might prevent a lot of phishing attacks.

Concepts Define Businesses

"Digital transformation" is a grandiose term for a simple idea: taking the core of a business and putting it online, so customers can access services through their devices. In my experience as a consultant, I've sometimes found that executives, seeking to refresh and expand their business, instead of trying to un-