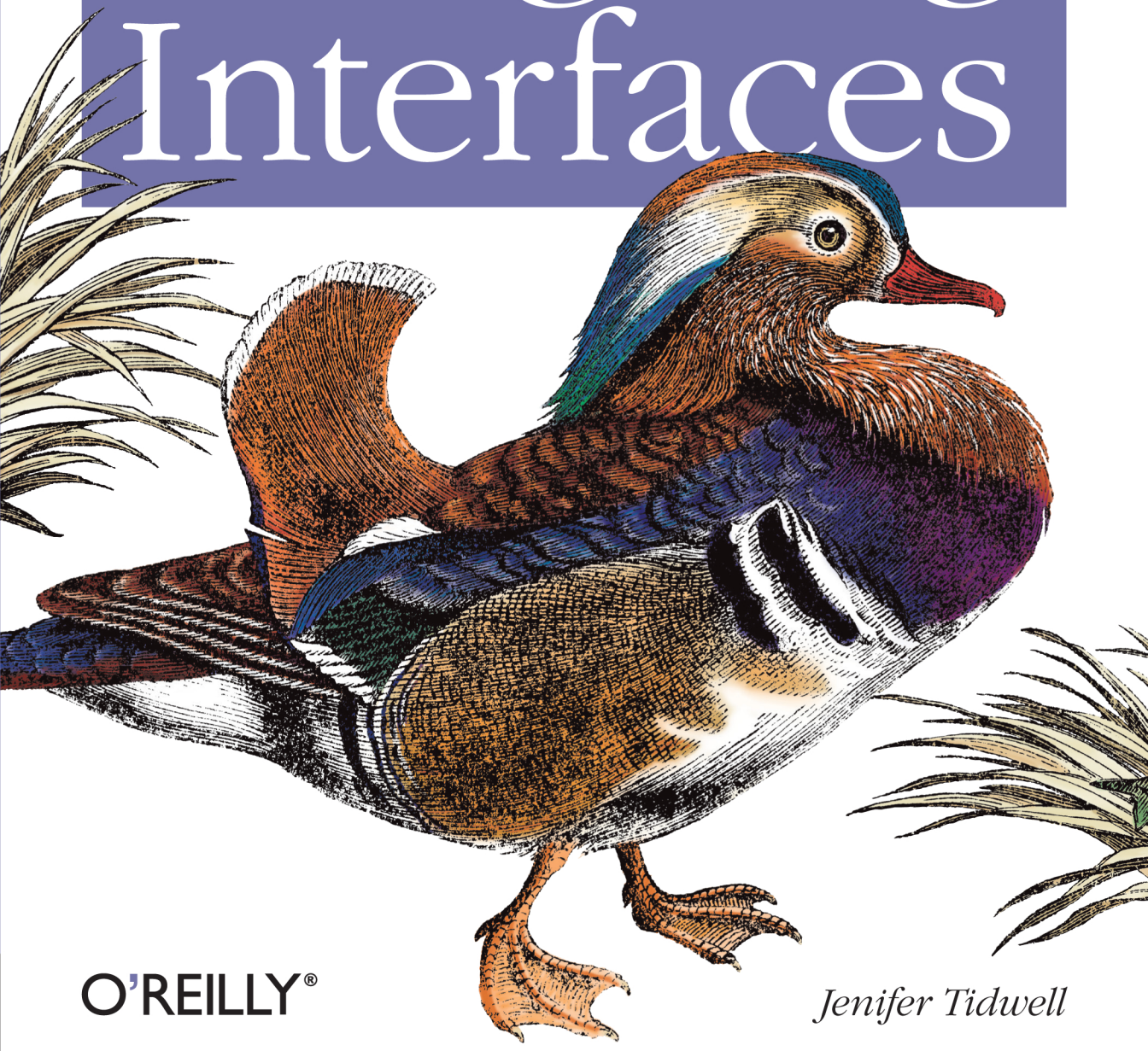
Patterns for Effective Interaction Design

2nd Edition

Designing Interfaces



O'REILLY®

Jenifer Tidwell

Designing Interfaces

Despite all of the UI toolkits available today, it's still not easy to design good application interfaces. This bestselling book is one of the few reliable sources to help you navigate through the maze of design options. By capturing UI best practices and reusable ideas as design patterns, *Designing Interfaces* provides solutions to common design problems that you can tailor to the situation at hand.

This updated edition includes patterns for mobile apps and social media, as well as web applications and desktop software. Each pattern contains full-color examples and practical design advice that you can use immediately. Experienced designers can use this guide as a sourcebook of ideas; novices will find a roadmap to the world of interface and interaction design.

- Design engaging and usable interfaces with more confidence and less guesswork
- Learn design concepts that are often misunderstood, such as affordances, visual hierarchy, navigational distance, and the use of color
- Get recommendations for specific UI patterns, including alternatives and warnings on when *not* to use them
- Mix and recombine UI ideas as you see fit
- Polish the look and feel of your interfaces with graphic design principles and patterns

"Anyone who's serious about designing interfaces should have this book on their shelf for reference. It's the most comprehensive cross-platform examination of common interface patterns anywhere."

–Dan Saffer
author of Designing Gestural Interfaces *(O'Reilly) and* Designing for Interaction *(New Riders)*

Jenifer Tidwell is a writer and consultant in interaction design, information architecture, and pre-design analysis. She has designed and built user interfaces for companies such as Google and The MathWorks.

O'REILLY®
oreilly.com

US $49.99 **CAN $57.99**
ISBN: 978-1-449-37970-4

Designing Interfaces

Designing Interfaces

Second Edition

Jenifer Tidwell

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Designing Interfaces, Second Edition
by Jenifer Tidwell

Printed in Canada.

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

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

Editor: Mary Treseler
Production Editor: Rachel Monaghan
Copyeditor: Audrey Doyle
Proofreader: Emily Quill
Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: Ron Bilodeau
Illustrator: Robert Romano

Printing History:

November 2005: First Edition.

December 2010: Second Edition.

Revision History:

2010-12-06	First release
2011-07-08	Second release
2012-02-24	Third release
2013-03-15	Fourth release

ISBN: 978-1-449-37970-4
[TI]

Contents

Introduction to the Second Edition

In the five years since the first edition of *Designing Interfaces* was published, many things have changed.

Most user interface designers—who might now play the roles of user experience (UX) designers, or interaction designers, or information architects, or any of several other titles—now do their work on the Web. Countless websites, web services, web-delivered software, blogs, and online stores need good design, and it's becoming easier and easier to deliver these finished products in ridiculously short turnaround times. Many of these are highly interactive, but even traditional websites—static and straightforward in the past—now contain components that are dynamic and interactive, such as video players and social network content. There's a lot of designing going on!

Compared to a few years ago, not as much of that designing is being done for desktop applications. Of course, all of us technology users depend upon the complex software installed on our laptops and desktops. Our email clients, browsers, document editors, domain-specific software, and operating systems are still important parts of our online lives. But many aspects of their interface designs have stabilized. As a result, since the early 2000s, the audience for design books has shifted away from desktop design toward web-based design.

Here's another change: mobile design, which was still immature in 2005, has flourished. With iPhones and other complex mobile devices now spreading everywhere, putting the whole Web in our pockets, many designers have been forced to face the special problems inherent to mobile design. How should mobile concerns change interface design, especially for websites? That's a question we're still collectively trying to answer, but the design community has learned some approaches and techniques that work.

Also, designers cannot ignore the influence of online social networks. When I'm in the early phase of a design project, I need to think about its connections to blogs, Twitter, Facebook, comment areas, forums, and all the other ways that people talk to one another online. I would be remiss not to do so. Users spend a lot of their online time "doing" social

interaction, and sophisticated users expect social-network support as a matter of course. It's unusual now to find any website that doesn't somehow connect to or from a social service (and usually several).

But wait, there's more! Since this book was first published, the UX design world has discovered the value of patterns, and other UX-related pattern collections have appeared on the scene. Many of them are quite good. Some took patterns originally set forth here and elaborated upon them, changed them, renamed them according to emergent conventions, or presented new information about them. Others created new patterns in areas that this book didn't cover well—especially social, mobile, gestural, search, and RIA-style interfaces. (I list the best of these other pattern collections in the preface, in the References section, and in the patterns themselves.)

So is the material written in 2005 still relevant?

To a large extent, yes. The human mind hasn't changed—visual hierarchies still work, progressive disclosure still works, and moving things still attract the attention of our reptilian brains. Good patterns based on fundamental design principles are just as valid now as they were 5, 10, or 20 years ago. But other patterns weren't as well grounded or have fallen out of favor. This second edition gave me the privilege of hindsight: I was granted the time to figure out how well these patterns have endured, and then report on them. And, indeed, a few have been removed from this book.

But most of them remain, because they still work. They've been updated with fresh examples, and in some cases with fresh research into their effectiveness. In addition, I've written (or borrowed) new patterns to reflect the changes of the last five years. The next section describes these changes in some detail.

Changes in the Second Edition

Here's what you're getting in this book:

A chapter about social media
: Chapter 9, *Using Social Media*, lays out some tactics and patterns for integrating social media into a site or application. The chapter does not cover all aspects of social interfaces; it's meant to be complementary to existing works on the subject, especially *Designing Social Interfaces* (O'Reilly, *http://oreilly.com/catalog/9780596154936/*).

A chapter about mobile design
: Chapter 10, *Going Mobile*, contains some patterns that are specific to mobile devices. In particular, the patterns are aimed at the platforms most designers are likely to target: touch-screen devices with full connectivity, such as iPhones. Both apps and websites are covered. Again, this is not intended to cover all aspects of mobile design—simply the patterns and ideas that can help you create a graceful mobile interface even if you're not a mobile UI specialist.

The existence of this chapter brings up an interesting point. A "good" pattern should be invariant across different platforms, perhaps including mobile ones. However, mobile design introduces so many new constraints on screen size, interactive gestures, social expectations, and latency that some patterns simply don't work well for it. Conversely, most of the patterns written specifically for mobile contexts don't work well (or aren't particularly salient design solutions) for larger screens; those patterns have a home in Chapter 10.

Reorganized chapters and rewritten introductions

Because there were so many old and new patterns about how to present lists of items, I chose to "refactor" three chapters to account for that. Chapter 5 is now simply about lists. It pulled patterns from the first edition's Chapter 2 (Two-Panel Selector, One-Window Drilldown) and Chapter 7 (Row Striping and Cascading Lists). I also added several new ones, such as List Inlay and Alphabetic Scroller.

Furthermore, the introductions to the chapters on information architecture (Chapter 2), navigation (Chapter 3), and page layout (Chapter 4) have been rewritten to reflect recent design thinking and a new emphasis on web-based or web-like designs.

New patterns that capture popular new interactions

Some techniques have really caught on in the last five years, and the ones that seem to be "pattern-like"—they are abstractable and cross-genre, they're common enough to be easy to find, and they can noticeably improve the user experience—are represented here. Examples include Fat Menus, Sitemap Footer, Hover Tools, Password Strength Meter, Data Spotlight, and Radial Table.

New patterns that aren't really "new," but that were not included in the first edition

These ideas have been kicking around for a while, but either I didn't recognize them as being important back in 2005, or they weren't especially salient back then. They are now. This list of patterns includes Dashboard, News Stream, Carousel, Grid of Equals, Microbreaks, Picture Manager, and Feature, Search, and Browse.

Renamed patterns, and patterns whose scope has changed

For instance, Card Stack was renamed to Module Tabs, and Closable Panels to Collapsible Panels; I made these changes to conform to current terminology and other pattern libraries. Similarly, Accordion was factored out from Collapsible Panels and made into its own pattern, since other designers, design writers, and pattern collections have converged on the term "accordion" for this particular technique. Meanwhile, One-Window Drilldown and Two-Panel Selector—both from the original book's chapter on information architecture—have been narrowed down to deal specifically with lists of items.

New examples, new research, and new connections to other pattern libraries

Almost every pattern has at least one new pictorial example, and many of them have an "In other libraries" section that directs the reader to the same pattern (or patterns that closely resemble it) in other collections. These might provide you with new insights or examples. Also, some patterns in this book have been slightly rewritten to

account for new thinking or research on the issue. Row Striping is one of these; some experiments were run to find out the value of the technique, and the pattern refers you to those results.

Some individual patterns have been removed
: Many of these have passed into the realm of "blindingly obvious to everyone," and while they're still useful as design tools, their value as part of this book is diminished. This list includes Extras on Demand, Intriguing Branches, Global Navigation, and Illustrated Choices. Others are no longer used much in contemporary designs, such as Color-Coded Sections.

The "Builders and Editors" chapter is gone
: Designers still work on these types of applications, of course, but I honestly couldn't find much to change in that set of patterns in terms of new work and updated examples. I also discovered in a survey that readers found this to be one of the least valuable chapters. Because I wanted to keep the book size down to something reasonable, I chose to remove that chapter to make room for the new material.

Finally, I want to talk briefly about what you won't find in this new edition. The following areas are so well covered by other published (or forthcoming) pattern collections that I saw little need to put them into this edition:

- Search
- General social interfaces
- Gestural interfaces
- More depth in mobile design
- Types of animated transitions
- Help techniques

I hope that in the next few years, we'll see new sets of patterns for other areas of design: online games, geographic systems, online communities, and more. I see a rich and rewarding area of inquiry here, and that's terrific. I encourage other design thinkers to jump in and write other patterns—or challenge us pattern writers to make the existing collections better!

Preface

Once upon a time, interface designers worked with a woefully small toolbox.

We had a handful of simple controls: text fields, buttons, menus, tiny icons, and modal dialogs. We carefully put them together according to the Windows Style Guide or the Macintosh Human Interface Guidelines, and we hoped that users would understand the resulting interface—and too often, they didn't. We designed for small screens, few colors, slow CPUs, and slow networks (if the user was connected at all). We made them gray.

Things have changed. If you design interfaces today, you work with a much bigger palette of components and ideas. You have a choice of many more user interface toolkits than before, such as the Java toolkits, HTML/CSS, JavaScript, Flash, and numerous open source options. Apple's and Microsoft's native UI toolkits are richer and nicer-looking than they used to be. Display technology is better. Web applications often look as professionally designed as the websites they're embedded in, and some of those web sensibilities have migrated back into desktop applications in the form of blue underlined links, Back/Next buttons, beautiful fonts and background images, and non-gray color schemes.

But it's still not easy to design *good* interfaces. Let's say you're not a trained or self-taught interface designer. If you just use the UI toolkits the way they should be used, and if you follow the various style guides or imitate existing applications, you can probably create a mediocre but passable interface.

Alas, that may not be enough anymore. Users' expectations are higher than they used to be—if your interface isn't easy to use "out of the box," users will not think well of it. Even if the interface obeys all the standards, you may have misunderstood users' preferred workflow, used the wrong vocabulary, or made it too hard to figure out what the software even does. Impatient users often won't give you the benefit of the doubt. Worse, if you've built an unusable website or web application, frustrated users can give up and switch to your competitor with just the click of a button. So the cost of building a mediocre interface is higher than it used to be, too.

Devices like phones, TVs, and car dashboards once were the exclusive domain of industrial designers. But now those devices have become smart. Increasingly powerful computers drive them, and software-based features and applications are multiplying in response to market demands. They're here to stay, whether or not they are easy to use. At this rate, good interface and interaction design may be the only hope for our collective sanity in 10 years.

Small Interface Pieces, Loosely Joined

As an interface designer trying to make sense of all the technology changes in the last few years, I see two big effects on the craft of interface design. One is the proliferation of *interface idioms:* recognizable types or styles of interfaces, each with its own vocabulary of objects, actions, and visuals. You probably recognize all the ones shown in Figure P-1, and more are being invented all the time.

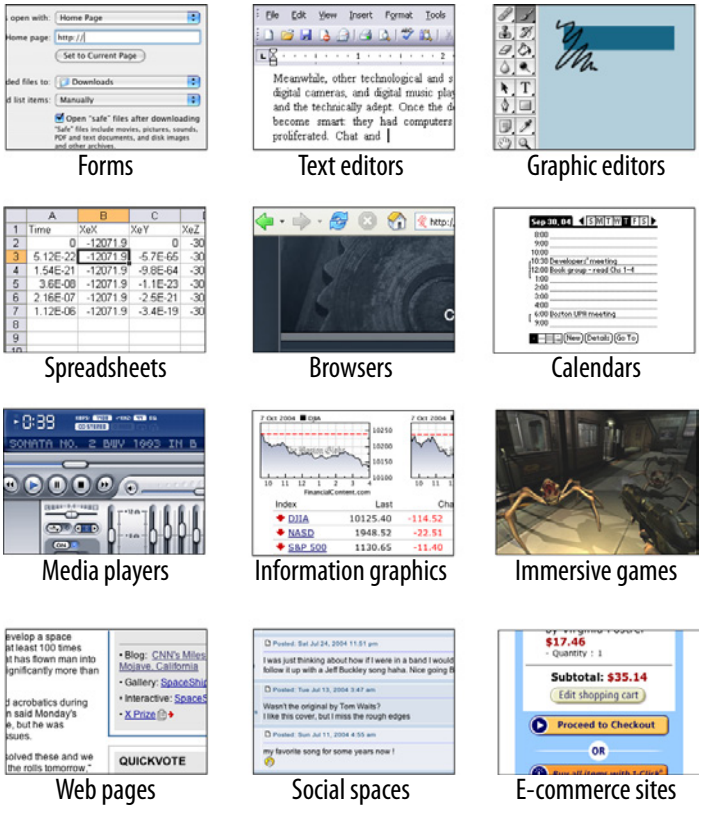


Figure P-1. *A sampler of interface idioms*

The second effect is a loosening of the rules for putting together interfaces from these idioms. It no longer surprises anyone to see several of these idioms mixed up in one interface, for instance, or to see parts of some controls mixed up with parts of other controls. Online help pages, which long have been formatted in hypertext anyway, might now include interactive applets, animations, or links to a web-based bulletin board. Interfaces themselves might have help texts on them, interleaved with forms or editors; this situation used to be rare. Combo boxes' drop-down menus might have funky layouts, like color grids or sliders, instead of the standard column of text items. You might see web applications that look like document-centered paint programs, but have no menu bars, and save the finished work only to a database somewhere.

The freeform-ness of web pages seems to have taught users to relax their expectations with respect to graphics and interactivity. It's OK now to break the old Windows style-guide strictures, as long as users can figure out what you're doing.

And that's the hard part. Some applications, devices, and web applications are easy to use. Many aren't. Following style guides never guaranteed usability anyhow, but now designers have even more choices than before (which, paradoxically, can make design a *lot* harder). What characterizes interfaces that are easy to use?

One could say, "The applications that are easy to use are designed to be intuitive." Well, yes. That's almost a tautology.

Except that the word "intuitive" is a little bit deceptive. Jef Raskin once pointed out that when we say "intuitive" in the context of software, we really mean "familiar." Computer mice aren't intuitive to someone who's never seen one (though a growling grizzly bear would be). There's nothing innate or instinctive in the human brain to account for it. But once you've taken ten seconds to learn to use a mouse, it's familiar, and you'll never forget it. Same for blue underlined text, play/pause buttons, and so on.

Rephrased: "The applications that are easy to use are designed to be *familiar*."

Now we're getting somewhere. "Familiar" doesn't necessarily mean that everything about a given application is identical to some genre-defining product (e.g., Word, Photoshop, Mac OS, or a Walkman). People are smarter than that. As long as the parts are recognizable enough and the relationships among the parts are clear, then people can apply their previous knowledge to a novel interface and figure it out.

That's where patterns come in. This book catalogs many of those familiar parts, in ways you can reuse in many different contexts. Patterns capture a common structure—often a very local one, such as a list layout—without being too concrete on the details, which gives you the flexibility to be creative.

If you know what users expect of your application, and if you choose carefully from your toolbox of idioms and frameworks (large-scale), individual elements (small-scale), and patterns (covering the range), then you can put together something that "feels familiar" while remaining original.

And that gets you the best of both worlds.

About Patterns in General

In essence, patterns are structural and behavioral features that improve the "habitability" of something—a user interface, a website, an object-oriented program, or a building. They make things easier to understand or more beautiful; they make tools more useful and usable.

As such, patterns can be a description of best practices within a given design domain. They capture common solutions to design tensions (usually called "forces" in pattern literature) and thus, by definition, are not novel. They aren't off-the-shelf components; each implementation of a pattern differs a little from every other. They aren't simple rules or heuristics either. And they won't walk you through an entire set of design decisions—if you're looking for a complete step-by-step description of how to design an interface, a pattern catalog isn't the place to find it!

Patterns are:

Concrete, not general
: All designers depend upon good design principles, like "Prevent errors," "Create a strong visual hierarchy," and "Don't make the user think." It's rather hard, however, to design an actual working interface starting from fundamental principles! Patterns are concrete enough to help fill the space between high-level general principles and the low-level "grammar" of user interface design (widgets, text, graphic elements, alignment grids, and so on).

Valid across different platforms and systems
: Patterns may be more concrete than principles or heuristics, but they do define abstractions—the best patterns aren't specific to a single platform or idiom. Some even work in both print and interactive systems. Ideally, each pattern captures some minor truth about how people work best with a created artifact, and it remains true even while the underlying technologies and media change.

Products, not processes
: Unlike heuristics or user-centered design techniques, which usually advise on how to go about *finding* a solution to an engineering or design problem, patterns *are* possible solutions.

Suggestions, not requirements
: You should almost always follow good design principles and heuristics, of course. And organizations need designers to follow style guides so that their products stay self-consistent. But patterns are intended to be only suggestions; you can follow them or reject them, depending on your design context and user needs.

Relationships among elements, not single elements
: A text field is not a pattern. The spatial relationships between a text field and a piece of help text near it, however, might be a pattern. Likewise, changes in a set of elements over time—as a user interacts with the software—may constitute a pattern, though some patterns capture only static relationships.

Customized to each design context
: When a pattern is instantiated in a design, the designer should adjust the pattern as needed to fit the situation. You could use some of the pattern examples verbatim, but as long as you understand why the pattern works, why not be creative? Fit the pattern to your particular users and requirements.

Some very complete sets of patterns make up a "pattern language." These patterns resemble visual languages in that they cover the entire vocabulary of elements used in a design (though pattern languages are more abstract and behavioral; visual languages talk about shapes, icons, colors, fonts, etc.). The set in this book isn't nearly as complete, and it contains techniques that don't qualify as traditional patterns. But at least it's concise enough to be manageable and useful.

Other Pattern Collections

The text that started it all dealt with physical buildings, not software. Christopher Alexander's *A Pattern Language* and its companion book *The Timeless Way of Building* established the concept of patterns and described a 250-pattern multilayered pattern language. It is often considered the gold standard for a pattern language because of its completeness, its rich interconnectedness, and its grounding in the human response to our built world.

In the mid-1990s, the publication of *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides profoundly changed the practice of commercial software architecture. This book is a collection of patterns describing object-oriented "micro-architectures." If you have a background in software engineering, this is the book that probably introduced you to the idea of patterns. Many other authors have written books about software patterns since this book. Software patterns such as these do make software more habitable—for those who write the software, not those who use it!

The first substantial set of user-interface patterns was "Common Ground," the predecessor to the book you're reading now. Many other collections and languages followed, notably Martijn van Welie's *Interaction Design Patterns;* van Duyne, Landay, and Hong's *The Design of Sites*; the Little Springs mobile patterns, now known as Design4Mobile; the Yahoo! Design Pattern Library, which morphed into *Designing Web Interfaces;* and the rest of the O'Reilly design pattern library, including *Designing Social Interfaces, Designing Gestural Interfaces,* and the first edition of this book.

About the Patterns in This Book

So there's nothing really new in here. If you've done any web or UI design, or even thought much about it, you should say, "Oh, right, I know what that is" to most of these patterns. But a few of them might be new to you, and some of the familiar ones may not be part of your usual design repertoire.

These patterns work for both desktop applications and highly interactive websites. Many patterns also apply to mobile devices or TV-based interfaces (like digital recorders).

Though this book won't exhaustively describe all the interface idioms mentioned earlier, these idioms help to organize the book. Some chapters focus on the more common idioms: forms, information graphics, mobile interfaces, and interactions with social networks. Other chapters address subjects that are useful across many idioms, such as organization, navigation, actions, and visual style. (The book does not address idioms such as online games or communities, simply due to lack of space.)

This book is intended to be read by people who have some knowledge of such interface design concepts and terminology as dialog boxes, selection, combo boxes, navigation bars, and whitespace. It does not identify many widely accepted techniques, such as copy-and-paste, since you already know what they are. But, at the risk of belaboring the obvious, this book describes some common techniques to encourage their use in other contexts or to discuss them alongside alternative solutions.

This book does *not* present a complete process for constructing an interface design. When doing design, a sound process is critical. You need to have certain elements in a design process:

- Field research, to find out what the intended users are like and what they already do
- Goal and task analysis, to describe and clarify what users will do with what you're building
- Design models, such as personas (models of users), scenarios (models of common tasks and situations), and prototypes (models of the interface itself)
- Empirical testing of the design at various points during development, like usability testing and *in situ* observations of the design used by real users
- Enough time to iterate over several versions of the design, because you won't get it right the first time

These topics transcend the scope of this book, but there are plenty of other excellent resources and workshops out there that cover them in depth.

But there's a deeper reason why this book won't give you a recipe for designing an interface. Good design can't be reduced to a recipe. It's a creative process, and one that changes under you as you work—in any given project, for instance, you won't understand some design issues until you've designed your way into a dead end. I've personally done that many times.

And design isn't linear. Most chapters in this book are arranged more or less by scale, and therefore by their approximate order in the design progression: large decisions about

content and scope are made first, followed by navigation, page design, and eventually the details of interactions with forms and toolbars and such. But you'll often find yourself moving back and forth through this progression. Maybe you'll know very early in a project how a certain screen should look, and that's a "fixed point;" you may have to work backward from there to figure out the right navigational structure. (It's not ideal, but things like this do happen in real life.)

Here are some ways you can use these patterns:

Learning
: If you don't have much design experience, a set of patterns can serve as a learning tool. You may want to read over it to get ideas, or refer back to specific patterns as the need arises. Just as expanding your vocabulary helps you express ideas in language, expanding your interface design "vocabulary" helps you create more expressive designs.

Examples
: Each pattern in this book has at least one example. Some have many; they might be useful to you as a sourcebook. You may find wisdom in the examples that is missing in the text of the pattern. If you're a designer who knows the patterns already, the examples may be the most useful aspect of the book for you.

Terminology
: If you talk to users, engineers, or managers about interface design, or if you write specifications, then you could use the pattern names as a way of communicating and discussing ideas. This is another well-known benefit of pattern languages. (The terms "singleton" and "factory," for instance, were originally pattern names, but they're now in common use among software engineers.)

Comparison of design alternatives
: If you initially decided to use Module Tabs to organize material on a page and it's not working quite as well as you hoped, you might use these patterns to come up with alternatives, such as Titled Sections or an Accordion. Other sets of "either/or" patterns are presented in this book, often with reasons to choose one pattern or another. Skilled designers know that presenting alternative designs to clients frequently leads to a better choice in the end.

Inspiration
: Each pattern description tries to capture the reasons why the pattern works to make an interface easier or more fun. If you get it, but want to do something a little different from the examples, you can be creative with your "eyes open." You could also use the book to jumpstart your creative process by flipping through it for ideas.

One more word of caution: a catalog of patterns is not a checklist. You cannot measure the quality of a thing by counting the patterns in it. Each design project has a unique context, and even if you need to solve a common design problem (such as how to fit too much content onto a page), a given pattern might be a poor solution within that context. No reference can substitute for good design judgment. Nor can it substitute for a good design process, which helps you find and recover from design mistakes.

Ultimately, you should be able to leave a reference like this behind. As you become an experienced designer, you will internalize these ideas to the point that you don't even notice you're using them anymore; the patterns become second nature and a permanent part of your toolbox.

Audience

If you design user interfaces in any capacity, you might find this book useful. It's intended for people who work on:

- Desktop applications
- Websites
- Web applications or "rich internet applications" (RIAs)
- Software for mobile devices or other consumer electronics
- Turnkey systems like kiosks
- Operating systems

Of course, profound differences exist among these different design platforms. However, I believe they have more in common than we generally think. You'll see examples from many different platforms in these patterns, and that's deliberate—they often use the same patterns to achieve the same ends.

From what readers said about the previous edition, this book has been more valuable to less experienced designers than to those who have been designing sites or interfaces for a while—they know this material already. However, even if you're just starting out with design, you should already know the basic "grammar" of UI design, such as available toolkits and control sets, concepts like drag-and-drop and focus, and the importance of usability testing and user feedback. If you don't, some excellent books listed in the References section can get you started with the essentials.

Specifically, this book targets the following audiences:

- Software developers who need to design the UIs that they build.
- Web page designers who are now asked to design web apps or sites with more interactivity.
- New interface designers and usability specialists.
- More experienced designers who want to see how other designs solve certain problems; the examples can serve as a sourcebook for ideas.
- Professionals in adjacent fields, such as technical writing, product design, and information architecture.

- Managers who want to understand what's involved in good interface design.
- Open source developers and enthusiasts. This isn't quite "open source design," but the idea is to open up interface design best practices for everyone's benefit.

How This Book Is Organized

The patterns in this book are grouped into thematic chapters, and each chapter has an introduction that briefly covers the concepts those patterns are built upon. I want to emphasize *briefly.* Some of these concepts could have entire books written about them. But the introductions will give you some context; if you already know this stuff, they'll be review material, and if not, they'll tell you what topics you might want to learn more about. The first set of chapters is applicable to almost any interface you might design, whether it's a desktop application, web application, website, hardware device, or whatever you can think of:

- Chapter 1, *What Users Do,* talks about common behavior and usage patterns supported by good interfaces.
- Chapter 2, *Organizing the Content,* discusses information architecture as it applies to highly interactive interfaces. It deals with different organizational patterns, recognizable interface types, and "guilds" of patterns (groups of smaller-scale patterns that work well together to support a certain type of interface).
- Chapter 3, *Getting Around,* discusses navigation. It describes patterns for moving around an interface—between pages, among windows, and within large virtual spaces.
- Chapter 4, *Organizing the Page,* describes patterns for the layout and placement of page elements. It talks about how to communicate meaning simply by putting things in the right places.
- Chapter 5, *Lists,* enumerates a set of patterns for displaying lists of items, along with criteria for choosing among them.
- Chapter 6, *Doing Things,* talks about how to present actions and commands; use these patterns to handle the "verbs" of an interface.

Next comes a set of chapters that deal with specific idioms. It's fine to read them all, but real-life projects probably won't use all of them. Chapters 7 and 8 are the most broadly applicable, since most modern interfaces use trees, tables, or forms in some fashion.

- Chapter 7, *Showing Complex Data,* contains patterns for trees, tables, charts, and information graphics in general. It discusses the cognitive aspects of data presentation and how to use them to communicate knowledge and meaning.
- Chapter 8, *Getting Input from Users,* deals with forms and controls. Along with the patterns, this chapter has a table that maps data types to various controls that can represent them.

- Chapter 9, *Using Social Media,* discusses the ways that one might integrate contemporary social media into a website or application design. Although designers don't always make these choices for a site, they sometimes do, and social media may influence your design in any case.
- Chapter 10, *Going Mobile,* presents techniques and concepts that designers ought to know in order to help their designs translate well to a mobile device. Patterns throughout the book may contain examples from mobile devices, but the patterns in this chapter are mobile-specific.

Finally, the last chapter comes at the end of the design progression, but it too applies to almost anything you design.

- Chapter 11, *Making It Look Good,* deals with aesthetics and fit-and-finish. It uses graphic design principles and patterns to show how (and why) to polish the look-and-feel of an interface once its behavior is established.

I chose this book's examples based on many factors. The most important is how well an example demonstrates a given pattern or concept, of course, but other considerations include general design fitness, printability, variety—desktop applications, websites, devices, etc.—and how well known and accessible these applications might be to readers. As such, the examples are weighted heavily toward Microsoft and Apple software, certain big-name websites such as Google and Yahoo! properties, and easy-to-find consumer software and devices. This is not to say that they always are paragons of good design—they're not, and I do not mean to slight the excellent work done by countless designers on lesser-known applications. If you know of examples that might meet most of these criteria, please suggest them to me.

Comments and Questions

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:

http://oreilly.com/catalog/9781449379704/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

First of all, I am indebted to my editor, Mary Treseler, who got this project rolling at just the right time. You knew a second edition was needed, and with the patience of a saint, you made sure I followed through with it. Thanks also to the rest of the O'Reilly production team: Rachel Monaghan, Audrey Doyle, Robert Romano, Ron Bilodeau, and anyone else I may have inadvertently missed. You all rocked.

The technical reviewers for this edition gave me fantastic feedback. Barbara Ballard, Erin Malone, Dan Saffer—thanks to you all!

The ideas in this second edition have been cooking for a long time. Both direct and indirect conversations with other UI designers and pattern writers have helped shape my thinking: Bill Scott, Luke Wroblewski, Martijn van Welie, Erin Malone, Christian Crumlish, Dan Saffer, James Reffell, Scott Jenson, and my UX colleagues at Google. I learned a ridiculous amount from all of you. I'm also grateful to the people who gave me feedback at the various and sundry presentations I've done for conferences and mini-conferences over the last few years.

To all who bought or read the first edition: thanks to you too! Without you, there would have been no second edition.

Finally, I am enormously grateful to Rich, who supported me wholeheartedly throughout this second-edition project; and to Matthew, who right now is too young to understand how helpful his sweet hugs actually were. I love you both!

Chapter 1
What Users Do

This book is almost entirely about the look and behavior of applications, web apps, and interactive devices. But this first chapter is the exception to the rule. No screenshots here; no layouts, no navigation, no diagrams, no visuals at all.

Why not? After all, that's probably why you picked up the book in the first place.

It's because good interface design doesn't start with pictures. It starts with an understanding of people: what they're like, why they use a given piece of software, and how they might interact with it. The more you know about them, and the more you empathize with them, the more effectively you can design for them. Software, after all, is merely a means to an end for the people who use it. The better you satisfy those ends, the happier those users will be.

Each time someone uses an application, or any digital product, he carries on a conversation with the machine. It may be literal, as with a command line or phone menu, or tacit, like the "conversation" an artist has with her paints and canvas—the give and take between the craftsperson and the thing being built. With social software, it may even be a conversation by proxy. Whatever the case, the user interface mediates that conversation, helping users achieve whatever ends they had in mind.

As the user interface designer, then, you get to script that conversation, or at least define its terms. And if you're going to script a conversation, you should understand the human's side as well as possible. What are the user's motives and intentions? What "vocabulary" of words, icons, and gestures does the user expect to employ? How can the application set expectations appropriately for the user? How do the user and the machine finally end up communicating meaning to each other?

There's a maxim in the field of interface design: "Know thy users, for they are not you!"

So, this chapter will talk about people. It covers a few fundamental ideas briefly in this introduction, and then discusses some patterns that differ from those in the rest of the book. They describe human behaviors—as opposed to system behaviors—that the software you design may need to support. Software that supports these human behaviors better helps users achieve their goals.

A Means to an End

Everyone who uses a tool—software or otherwise—has a reason for using it. For instance:

- Finding some fact or object
- Learning something
- Performing a transaction
- Controlling or monitoring something
- Creating something
- Conversing with other people
- Being entertained

Well-known idioms, user behaviors, and design patterns can support each of these abstract goals. User experience designers have learned, for example, how to help people search through vast amounts of online information for specific facts. They've learned how to present tasks so that it's easy to walk through them. They're learning ways to support the building of documents, illustrations, and code.

The first step in designing an interface is to figure out what its users are really trying to accomplish. Filling out a form, for example, is almost never a goal in and of itself—people only do it because they're trying to buy something online, renew their driver's license, or install software. They're performing some kind of transaction.

Asking the right questions can help you connect user goals to the design process. Users and clients typically speak to you in terms of desired features and solutions, not of needs and problems. When a user or client tells you he wants a certain feature, ask why he wants it—determine his immediate goal. Then to the answer of this question, ask "why" again. And again. Keep asking until you move well beyond the boundaries of the immediate design problem.*

Why should you ask these questions if you have clear requirements? Because if you love designing things, it's easy to get caught up in an interesting interface design problem. Maybe you're good at building forms that ask for just the right information, with the right controls, all laid out nicely. But the real art of interface design lies in *solving the right problem*.

So, don't get too fond of designing that form. If there's any way to finish the transaction without making the user go through that form at all, get rid of it altogether. That gets the user closer to his goal, with less time and effort spent on his part (and maybe yours, too).

Let's use the "why" approach to dig a little deeper into some typical design scenarios.

* This is the same principle that underlies a well-known technique called *root-cause analysis*. But root-cause analysis is a tool for fixing organizational failures; here, we use its "five whys" (more or less) to understand everyday user behaviors and feature requests.

- Why does a mid-level manager use an email client? Yes, of course—"to read email." Why does she read and send email in the first place? To converse with other people. Of course, other means might achieve the same ends: the phone, a hallway conversation, a formal document. But apparently, email fills some needs that the other methods don't. What are they, and why are they important to her? Privacy? The ability to archive a conversation? Social convention? What else?
- A father goes to an online travel agent, types in the city where his family will be taking a summer vacation, and tries to find plane ticket prices on various dates. He's learning from what he finds, but his goal isn't just to browse and explore different options. Ask why. His goal is actually a transaction: to buy plane tickets. Again, he could have done that at many different websites, or over the phone with a live travel agent. How is this site better than those other options? Is it faster? Friendlier? More likely to find a better deal?
- A mobile phone user wants a way to search through his contacts list more quickly. You, as the designer, can come up with some clever ideas to save keystrokes while searching. But why does he want it? It turns out that he makes a lot of calls while driving, and he doesn't want to take his eyes off the road more than he has to—he wants to make calls while staying safe (to the extent that that's possible). The ideal case is that he doesn't have to look at the phone at all! A better solution is voice dialing: all he has to do is speak the name, and the phone makes the call for him.
- Sometimes goal analysis really isn't straightforward at all. A snowboarding site might provide information (for learning), an online store (for transactions), and a set of Flash movies (for entertainment). Let's say someone visits the site for a purchase, but she gets sidetracked into the information on snowboarding tricks—she has switched goals from accomplishing a transaction to browsing and learning. Maybe she'll go back to purchasing something, maybe not. And does the entertainment part of the site successfully entertain both the 12-year-old and the 35-year-old? Will the 35-year-old go elsewhere to buy his new board if he doesn't feel at home there, or does he not care?

It's deceptively easy to model users as a single faceless entity—"The User"—walking through a set of simple use cases, with one task-oriented goal in mind. But that won't necessarily reflect your users' reality.

To do design well, you need to take many "softer" factors into account: gut reactions, preferences, social context, beliefs, and values. All of these factors could affect the design of an application or site. Among these softer factors, you may find the critical feature or design factor that makes your application more appealing and successful.

So, be curious. Specialize in finding out what your users are really like, and what they really think and feel.

The Basics of User Research

Empirical discovery is the only really good way to obtain this information. To get a design started, you'll need to characterize the kinds of people who will be using your design (including the softer factors just mentioned), and the best way to do that is to go out and meet them.

Each user group is unique, of course. The target audience for, say, a new mobile phone app will differ dramatically from the target audience for a piece of scientific software. Even if the same person uses both, his expectations for each are different—a researcher using scientific software might tolerate a less-polished interface in exchange for high functionality, whereas that same person may stop using the mobile app if he finds its UI to be too hard to use after a few days.

Each user is unique, too. What one person finds difficult, the next one won't. The trick is to figure out what's *generally* true about your users, which means learning about enough individual users to separate the quirks from the common behavior patterns.

Specifically, you'll want to learn:

- Their goals in using the software or site
- The specific tasks they undertake in pursuit of those goals
- The language and words they use to describe what they're doing
- Their skill at using software similar to what you're designing
- Their attitudes toward the kind of thing you're designing, and how different designs might affect those attitudes

I can't tell you what your particular target audience is like. You need to find out what they might do with the software or site, and how it fits into the broader context of their lives. Difficult though it may be, try to describe your potential audience in terms of how and why they might use your software. You might get several distinct answers, representing distinct user groups; that's OK. You might be tempted to throw up your hands and say, "I don't know who the users are" or "Everyone is a potential user." But that doesn't help you focus your design at all—without a concrete and honest description of those people, your design will proceed with no grounding in reality.

Unfortunately, this user-discovery phase will consume serious time early in the design cycle. It's expensive. But it's worth it, because you stand a better chance at solving the right problem—you'll build the right thing in the first place.

Fortunately, lots of books, courses, and methodologies now exist to help you. Although this book does not address user research, here are some methods and topics to consider:

Direct observation

Interviews and onsite user visits put you directly into the user's world. You can ask users about what their goals are and what tasks they typically do. Usually done "on location," where users would actually use the software (e.g., in a workplace or at home), interviews can be structured—with a predefined set of questions—or unstructured, where you probe whatever subject comes up. Interviews give you a lot of flexibility; you can do many or a few, long or short, formal or informal, on the phone or in person. These are great opportunities to learn what you don't know. Ask why. Ask it again.

Case studies

Case studies give you deep, detailed views into a few representative users or groups of users. You can sometimes use them to explore "extreme" users that push the boundaries of what the software can do, especially when the goal is a redesign of existing software. You can also use them as longitudinal studies—exploring the context of use over months or even years. Finally, if you're designing custom software for a single user or site, you'll want to learn as much as possible about the actual context of use.

Surveys

Written surveys can collect information from many users. You can actually get statistically significant numbers of respondents with these. Since there's no direct human contact, you will miss a lot of extra information—whatever you don't ask about, you won't learn about—but you can get a very clear picture of certain aspects of your target audience. Careful survey design is essential. If you want reliable numbers instead of a qualitative "feel" for the target audience, you absolutely must write the questions correctly, pick the survey recipients correctly, and analyze the answers correctly—and that's a science.

Personas

Personas aren't a data-gathering method, but they do help you figure out what to do with your data once you've got it. This is a design technique that "models" the target audiences. For each major user group, you create a fictional person that captures the most important aspects of the users in that group: what tasks they're trying to accomplish, their ultimate goals, and their experience levels in the subject domain and with computers in general. Personas can help you stay focused. As your design proceeds, you can ask yourself questions such as "Would this fictional person really do X? What would she do instead?"

You might notice that some of these methods and topics, such as interviews and surveys, sound suspiciously like marketing activities. That's exactly what they are. Focus groups can be useful, too (though not so much as the others), and the concept of market segmentation resembles the definition of target audiences used here. In both cases, the whole point is to understand the audience as best you can.

The difference is that as a designer, you're trying to understand the people who *use* the software. A marketing professional tries to understand those who buy it.

It's not easy to understand the real issues that underlie users' interactions with a system. Users don't always have the language or introspective skill to explain what they really need to accomplish their goals, and it takes a lot of work on your part to ferret out useful design concepts from what they *can* tell you—self-reported observations are usually biased in subtle ways.

Some of these techniques are very formal, and some aren't. Formal and quantitative methods are valuable because they're good science. When applied correctly, they help you see the world as it actually is, not how you think it is. If you do user research haphazardly, without accounting for biases such as the self-selection of users, you may end up with data that doesn't reflect your actual target audience—and that can only hurt your design in the long run.

But even if you don't have time for formal methods, it's better to just meet a few users informally than to not do any discovery at all. Talking with users is good for the soul. If you're able to empathize with users and imagine those individuals actually using your design, you'll produce something much better.

Users' Motivation to Learn

Before you start the design process, consider your overall approach. Think about how you might design the interface's overall interaction style—its personality, if you will.

When you carry on a conversation with someone about a given subject, you adjust what you say according to your understanding of the other person. You might consider how much he cares about the subject, how much he already knows about it, how receptive he is to learning from you, and whether he's even interested in the conversation in the first place. If you get any of that wrong, bad things happen—he might feel patronized, uninterested, impatient, or utterly baffled.

This analogy leads to some obvious design advice. The subject-specific vocabulary you use in your interface, for instance, should match your users' level of knowledge; if some users won't know that vocabulary, give them a way to learn the unfamiliar terms. If they don't know computers very well, don't make them use sophisticated widgetry or uncommon interface-design conventions. If their level of interest might be low, respect that, and don't ask for too much effort for too little reward.

Some of these concerns permeate the whole interface design in subtle ways. For example, do your users expect a short, tightly focused exchange about something very specific, or do they prefer a conversation that's more of a free-ranging exploration? In other words, how much openness is there in the interface? Too little, and your users feel trapped and unsatisfied; too much, and they stand there paralyzed, not knowing what to do next, unprepared for that level of interaction.

Therefore, you need to choose how much freedom your users have to act arbitrarily. At one end of the scale might be a software installation wizard: the user is carried through it with no opportunity to use anything other than Next, Previous, or Cancel. It's tightly focused and specific, but quite efficient—and satisfying, to the extent that it works and is quick. At the other end might be an application such as Excel, an "open floorplan" interface that exposes a huge number of features in one place. At any given time, the user has about 872 things that he can do next, but that's considered good, because self-directed, skilled users can do a lot with that interface. Again, it's satisfying, but for entirely different reasons.

Here's an even more fundamental question: how much effort are your users willing to spend to learn your interface?

It's easy to overestimate. Maybe they'll use it every day on the job—clearly they'd be motivated to learn it well in that case, but that's rare. Maybe they'll use it sometimes, and learn it only well enough to get by (Satisficing). Maybe they'll only see it once, for 30 seconds. Be honest: can you expect most users to become intermediates or experts, or will most users remain perpetual beginners?

Software designed for intermediate-to-expert users includes:

- Photoshop
- Dreamweaver
- Excel
- Code development environments
- System-administration tools for web servers

In contrast, here are some things designed for occasional users:

- Kiosks in tourist centers or museums
- Windows or Mac OS controls for setting desktop backgrounds
- Purchase pages for online stores
- Installation wizards
- Automated teller machines

The differences between the two groups are dramatic. Assumptions about users' tool knowledge permeate these interfaces, showing up in their screen-space usage, labeling, and widget sophistication, and in the places where help is (or isn't) offered.

The applications in the first group have lots of complex functionality, but they don't generally walk the user through tasks step by step. They assume users already know what to do, and they optimize for efficient operation, not learnability; they tend to be document-centered or list-driven (with a few being command-line applications). They often have entire books and courses written about them. Their learning curves are steep.

The applications in the second group are the opposite: restrained in functionality but helpful about explaining it along the way. They present simplified interfaces, assuming no prior knowledge of document- or list-centered application styles (e.g., menu bars, multiple selection, etc.). Wizards frequently show up, removing attention-focusing responsibility from the user. The key is that users aren't motivated to work hard at learning these applications—it's usually just not worth it!

Now that you've seen the extremes, look at the applications in the middle of the continuum:

- Microsoft PowerPoint
- Email clients
- Facebook
- Blog-writing tools

The truth is that most applications fall into this middle ground. They need to serve people on both ends adequately—to help new users learn the tool (and satisfy their need for instant gratification), while enabling frequent-user intermediates to get things done smoothly. Their designers probably knew that people wouldn't take a three-day course to learn an email client. Yet the interfaces hold up under repeated usage. People quickly learn the basics, reach a proficiency level that satisfies them, and don't bother learning more until they are motivated to do so for specific purposes.

You may someday find yourself in tension between the two ends of this spectrum. Naturally you want people to be able to use your design "out of the box," but you might also want to support frequent or expert users as much as possible. Find a balance that works for your situation. Organizational patterns in Chapter 2, such as Multi-Level Help, can help you serve both constituencies.

The Patterns

Even though individuals are unique, people behave predictably. Designers have been doing site visits and user observations for years; cognitive scientists and other researchers have spent many hundreds of hours watching how people do things and how they think about what they do.

So, when you observe people using your software, or doing whatever activity you want to support with new software, you can expect them to do certain things. The behavioral patterns that follow are often seen in user observations. Odds are good that you'll see them too, especially if you look for them.

(A note for pattern enthusiasts: these patterns aren't like the others in this book. They describe human behaviors—not interface design elements—and they're not prescriptive, like the patterns in other chapters. Instead of being structured like the other patterns, these are presented as small essays.)

Again, an interface that supports these patterns well will help users achieve their goals far more effectively than interfaces that don't support them. And the patterns are not just about the interface, either. Sometimes the entire package—interface, underlying architecture, feature choice, documentation, everything—needs to be considered in light of these behaviors. But as the interface designer or interaction designer, you should think about these as much as anyone on your team. You might be in a better place than anyone to advocate for the users.

1. Safe Exploration
2. Instant Gratification
3. Satisficing
4. Changes in Midstream
5. Deferred Choices
6. Incremental Construction
7. Habituation
8. Microbreaks
9. Spatial Memory
10. Prospective Memory
11. Streamlined Repetition
12. Keyboard Only
13. Other People's Advice
14. Personal Recommendations

Safe Exploration

"Let me explore without getting lost or getting into trouble."

When someone feels like she can explore an interface and not suffer dire consequences, she's likely to learn more—and feel more positive about it—than someone who doesn't explore. Good software allows people to try something unfamiliar, back out, and try something else, all without stress.

Those "dire consequences" don't even have to be very bad. Mere annoyance can be enough to deter someone from trying things out voluntarily. Clicking away pop-up windows, reentering data that was mistakenly erased, suddenly muting the volume on one's laptop when a website unexpectedly plays loud music—all can be discouraging. When you design almost any kind of software interface, make many avenues of exploration available for users to experiment with, without costing the user anything.

Here are some examples:

- A photographer tries out a few image filters in an image-processing application. He then decides he doesn't like the results, and clicks Undo a few times to get back to where he was. Then he tries another filter, and another, each time being able to back out of what he did. (The pattern named Multi-Level Undo, in Chapter 6, describes how this works.)
- A new visitor to a company's home page clicks various links just to see what's there, trusting that the Back button will always get her back to the main page. No extra windows or pop ups open, and the Back button keeps working predictably. You can imagine that if a web app does something different in response to the Back button—or if an application offers a button that seems like a Back button, but doesn't behave quite like it—confusion might ensue. The user can get disoriented while navigating, and may abandon the app altogether.

Instant Gratification

"I want to accomplish something now, not later."

People like to see immediate results from the actions they take—it's human nature. If someone starts using an application and gets a "success experience" within the first few seconds, that's gratifying! He'll be more likely to keep using it, even if it gets harder later. He will feel more confident in the application, and more confident in himself, than if it had taken a while to figure things out.

The need to support instant gratification has many design ramifications. For instance, if you can predict the first thing a new user is likely to do, you should design the UI to make that first thing stunningly easy. If the user's goal is to create something, for instance, then create a new canvas, put a call to action on it, and place a palette next to it. If the user's goal is to accomplish some task, point the way toward a typical starting point.

This also means you shouldn't hide introductory functionality behind anything that needs to be read or waited for, such as registrations, long sets of instructions, slow-to-load screens, advertisements, and so on. These are discouraging because they block users from finishing that first task quickly.

Satisficing

"This is good enough.
I don't want to spend more time learning to do it better."

When people look at a new interface, they don't read every piece of it methodically and then decide, "Hmmm, I think this button has the best chance of getting me what I want." Instead, a user will rapidly scan the interface, pick whatever he sees first that might get him what he wants, and try it—even if it might be wrong.

The term *satisficing* is a combination of *satisfying* and *sufficing*. It was coined in 1957 by the social scientist Herbert Simon, who used it to describe the behavior of people in all kinds of economic and social situations. People are willing to accept "good enough" instead of "best" if learning all the alternatives might cost time or effort.

Satisficing is actually a very rational behavior, once you appreciate the mental work necessary to "parse" a complicated interface. As Steve Krug points out in his book *Don't Make Me Think* (New Riders), people don't like to think any more than they have to—it's work! But if the interface presents an obvious option or two that the user sees immediately, he'll try it. Chances are good that it will be the right choice, and if not, there's little cost in backing out and trying something else (assuming that the interface supports Safe Exploration).

This means several things for designers:

- Use "calls to action" in the interface. Give directions on what to do first: type here, drag an image here, tap here to begin, and so forth.
- Make labels short, plainly worded, and quick to read. (This includes menu items, buttons, links, and anything else identified by text.) They'll be scanned and guessed about; write them so that a user's first guess about meaning is correct. If he guesses wrong several times, he'll be frustrated, and you'll both be off to a bad start.
- Use the layout of the interface to communicate meaning. Chapter 4 explains how to do so in detail. Users "parse" color and form on sight, and they follow these cues more efficiently than labels that must be read.
- Make it easy to move around the interface, especially for going back to where a wrong choice might have been made hastily. Provide "escape hatches" (see Chapter 3). On typical websites, using the Back button is easy, so designing easy forward/backward navigation is especially important for web apps, installed applications, and mobile devices.
- Keep in mind that a complicated interface imposes a large cognitive cost on new users. Visual complexity will often tempt nonexperts to satisfice: they look for the first thing that may work.

Satisficing is why many users end up with odd habits after they've been using a system for a while. Long ago, a user may have learned Path A to do something, and even though a later version of the system offers Path B as a better alternative (or maybe it was there all along), he sees no benefit in learning it—that takes effort, after all—and keeps using the less-efficient Path A. It's not necessarily an irrational choice. Breaking old habits and learning something new takes energy, and a small improvement may not be worth the cost to the user.

Changes in Midstream

"I changed my mind about what I was doing."

Occasionally, people change what they're doing while in the middle of doing it. Someone may walk into a room with the intent of finding a key she had left there, but while she's there, she finds a newspaper and starts reading it. Or she may visit Amazon.com to read product reviews, but ends up buying a book instead. Maybe she's just sidetracked; maybe the change is deliberate. Either way, the user's goal changes while she's using the interface you designed.

This means designers should provide opportunities for people to do that. Make choices available. Don't lock users into a choice-poor environment with no connections to other pages or functionality unless there's a good reason to do so. Those reasons do exist. See the patterns called Wizard (Chapter 2) and Modal Panel (Chapter 3) for examples.

You can also make it easy for someone to start a process, stop in the middle, and come back to it later to pick up where he left off—a property often called *reentrance*. For instance, a lawyer may start entering information into a form on an iPad. Then, when a client comes into the room, the lawyer turns off the device, with the intent of coming back to finish the form later. The entered information shouldn't be lost.

To support reentrance, you can make dialogs and web forms remember values typed previously, and they don't usually need to be modal; if they're not modal, they can be dragged aside on the screen for later use. Builder-style applications—text editors, code development environments, and paint programs—can let a user work on multiple projects at one time, thus letting her put any number of projects aside while she works on another one. See the Many Workspaces pattern in Chapter 2 for more information.

Deferred Choices

"I don't want to answer that now; just let me finish!"

This follows from people's desire for instant gratification. If you ask a task-focused user unnecessary questions in the process, he may prefer to skip the questions and come back to them later.

For example, some web-based bulletin boards have long and complicated procedures for registering users. Screen names, email addresses, privacy preferences, avatars, self-descriptions…the list goes on and on. "But I just wanted to post one little thing," says the user plaintively. Why not allow him to skip most of the questions, answer the bare minimum, and come back later (if ever) to fill in the rest? Otherwise, he might be there for half an hour answering essay questions and finding the perfect avatar image.

Another example is creating a new project in a website editor. There are some things you do have to decide up front, such as the name of the project, but other choices—where on the server are you going to put this when you're done? I don't know yet!—can easily be deferred.

Sometimes it's just a matter of not wanting to answer the questions. At other times, the user may not have enough information to answer yet. What if a music-writing software package asked you up front for the title, key, and tempo of a new song, before you've even started writing it? (See Apple's GarageBand for this bit of "good" design.)

The implications for interface design are simple to understand, though not always easy to implement:

- Don't accost the user with too many upfront choices in the first place.
- On the forms that he does have to use, clearly mark the required fields, and don't make too many of them required. Let him move on without answering the optional ones.
- Sometimes you can separate the few important questions or options from others that are less important. Present the short list; hide the long list.
- Use Good Defaults (Chapter 8) wherever possible, to give users some reasonable default answers to start with. But keep in mind that prefilled answers still require the user to look at them, just in case they need to be changed. They have a small cost, too.
- Make it possible for users to return to the deferred fields later, and make them accessible in obvious places. Some dialog boxes show the user a short statement, such as "You can always change this later by clicking the Edit Project button." Some websites store a user's half-finished form entries or other persistent data, such as shopping carts with unpurchased items.
- If registration is required at a website that provides useful services, users may be far more likely to register if they're first allowed to experience the website—drawn in and engaged—and then asked later about who they are. Some sites let you complete an entire purchase without registering, then ask you at the end if you want to create a no-hassle login with the personal information provided in the purchase step.

Incremental Construction

"Let me change this. That doesn't look right; let me change it again. That's better."

When people create things, they don't usually do it all in a precise order. Even an expert doesn't start at the beginning, work through the creation process methodically, and come out with something perfect and finished at the end.

Quite the opposite. Instead, she starts with some small piece of it, works on it, steps back and looks at it, tests it (if it's code or some other "runnable" thing), fixes what's wrong, and starts to build other parts of it. Or maybe she starts over, if she really doesn't like it. The creative process goes in fits and starts. It moves backward as much as forward sometimes, and it's often incremental, done in a series of small changes instead of a few big ones. Sometimes it's top-down; sometimes it's bottom-up.

Builder-style interfaces need to support that style of work. Make it easy for users to build small pieces. Keep the interface responsive to quick changes and saves. Feedback is critical: constantly show the user what the whole thing looks and behaves like, while the user works. If the user builds code, simulations, or other executable things, make the "compile" part of the cycle as short as possible, so the operational feedback feels immediate—leave little or no delay between the user making changes and seeing the results.

When creative activities are well supported by good tools, they can induce a state of *flow* in the user. This is a state of full absorption in the activity, during which time distorts, other distractions fall away, and the person can remain engaged for hours—the enjoyment of the activity is its own reward. Artists, athletes, and programmers all know this state.

But bad tools will keep users distracted, guaranteed. If the user has to wait even half a minute to see the results of the incremental change she just made, her concentration is broken; flow is disrupted.

If you want to read more about flow, read the books by Mihaly Csikszentmihalyi, who studied it for years.

Habituation

"That gesture works everywhere else; why doesn't it work here, too?"

When one uses an interface repeatedly, some frequent physical actions become reflexive: pressing Ctrl-S to save a document, clicking the Back button to leave a web page, pressing Return to close a modal dialog box, using gestures to show and hide windows—even pressing a car's brake pedal. The user no longer needs to think consciously about these actions. They've become habitual.

This tendency helps people become expert users of a tool (and helps create a sense of flow, too). Habituation also measurably improves efficiency, as you can imagine. But it can also lay traps for the user. If a gesture becomes a habit, and the user tries to use it in a situation when it doesn't work—or, worse, does something destructive—the user is caught short. He suddenly has to think about the tool again (What did I just do? How do I do what I intended?), and he might have to undo any damage done by the gesture.

For instance, Ctrl-X→Ctrl-S is the "save this file" key sequence used by the Emacs text editor. Ctrl-A moves the text-entry cursor to the beginning of a line. These keystrokes become habitual for Emacs users. When a user presses Ctrl-A→Ctrl-X→Ctrl-S in Emacs, it performs a fairly innocuous pair of operations: move the cursor, save the file.

Now what happens when he types that same habituated sequence in Microsoft Word?

1. Ctrl-A: Select all
2. Ctrl-X: Cut the selection (the whole document, in this case)
3. Ctrl-S: Save the document (whoops)

This is why consistency across applications is important! (And also why a robust "undo" is useful.)

Just as important, though, is consistency within an application. Some applications are evil because they establish an expectation that some gesture will do Action X, except in one special mode where it suddenly does Action Y. Don't do that. It's a sure bet that users will make mistakes, and the more experienced they are—that is, the more habituated they are—the more likely they are to make that mistake.

Consider this carefully if you're developing gesture-based interfaces for mobile devices. Once someone learns how to use his device and gets used to it, he will depend on the standard gestures working consistently on all applications. Check that gestures in your design all do the expected things.

This is also why confirmation dialog boxes often don't work to protect a user against accidental changes. When modal dialog boxes pop up, the user can easily get rid of them just by clicking OK or pressing Return (if the OK button is the default button). If the dialogs pop up all the time when the user makes intended changes, such as deleting files, clicking OK becomes a habituated response. Then, when it actually matters, the dialog box doesn't have any effect, because it slips right under the user's consciousness.

(I've seen at least one application that sets up the confirmation dialog box's buttons randomly from one invocation to another. One actually has to *read* the buttons to figure out what to click! This isn't necessarily the best way to do a confirmation dialog box—in fact, it's better to not have them at all under most circumstances—but at least this design sidesteps habituation creatively.)

Microbreaks

"I'm waiting for the train. Let me do something useful for two minutes."

People often find themselves with a few minutes of down time. They might need a mental break while working; they might be in line at a store or sitting in a traffic jam. They might be bored or impatient. They want to do something constructive or entertaining to pass the time, knowing they won't have enough time to get deep into an online activity.

This pattern is especially applicable to mobile devices, because people can easily pull them out at times such as these.

Here are some typical activities during microbreaks:

- Checking email
- Reading a News Stream (in Chapter 2) such as Facebook or Twitter
- Visiting a news site to find out what's going on in the world
- Watching a short video
- Doing a quick web search
- Reading an online book
- Playing a short game

The key to supporting microbreaks is to make an activity easy and fast to reach—as easy as turning on the device and selecting an application (or website). Don't require complicated setup. Don't take forever to load. And if the user needs to sign in to a service, try to retain the previous authentication so that she doesn't have to sign in every time.

For News Stream services, load the freshest content as quickly as possible and show it in the first screen the user sees. Other activities, such as games, videos, or online books, should remember where the user left them last time and restore the app or site to its previous state, without asking (thus supporting reentrance).

If you're designing an email application, or anything else for which the user needs to do "housekeeping" to maintain order, give her a way to triage items efficiently. This means showing enough data per item so that she can identify, for instance, a message's contents and sender. You can also give her a chance to "star" or otherwise annotate items of interest, delete items easily, and write short responses and updates.

Long load times deserve another mention. Taking too long to load content is a sure way to make users give up on your app—especially during microbreaks! Make sure the page is engineered so that readable, useful content loads first, and with very little delay.

Spatial Memory

"I swear that button was here a minute ago. Where did it go?"

When people manipulate objects and documents, they often find them again later by remembering where they are, not what they're named.

Take the Windows, Mac, or Linux desktop. Many people use the desktop background as a place to put documents, frequently used applications, and other such things. It turns out that people tend to use spatial memory to find things on the desktop, and it's very effective. People devise their own groupings, for instance, or recall that "this document was at the top right over by such-and-such." (Naturally, there are real-world equivalents, too. Many people's desks are "organized chaos," an apparent mess in which the office owner can find anything instantly. But heaven forbid that someone should clean it up for him.)

Many applications put their dialog buttons—OK, Cancel, and so on—in predictable places, partly because spatial memory for them is so strong. In complex applications, people may also find things by remembering where they are relative to other things: tools on toolbars, objects in hierarchies, and so forth. Therefore, you should use patterns such as Responsive Disclosure (Chapter 4) carefully. Adding items to blank spaces in an interface doesn't cause problems, but rearranging existing controls can disrupt spatial memory and make things harder to find. It depends. Try it out on your users if you're not sure.

Along with habituation, which is closely related, spatial memory is another reason why consistency across and within a platform's applications is good. People may expect to find similar functionality in similar places. See the Sign-in Tools pattern (Chapter 3) for an example.

Spatial memory explains why it's good to provide user-arranged areas for storing documents and objects, such as the aforementioned desktop. Such things aren't always practical, especially with large numbers of objects, but it works quite well with small numbers. When people arrange things themselves, they're likely to remember where they put them. (Just don't rearrange it for them unless they ask!) The Movable Panels pattern in Chapter 4 describes one particular way to do this.

Also, this is why changing menus dynamically can sometimes backfire. People get used to seeing certain items on the tops and bottoms of menus. Rearranging or compacting menu items "helpfully" can work against habituation and lead to user errors. So can changing navigation menus on web pages. Try to keep menu items in the same place, and in the same order, on all subpages in a site.

Incidentally, the tops and bottoms of lists and menus are special locations, cognitively speaking. People notice and remember them more than items in the middle of a list. The first and last items are perhaps the worst ones to change out from under the user.

Prospective Memory

"I'm putting this here to remind myself to deal with it later."

Prospective memory is a well-known phenomenon in psychology that doesn't seem to have gained much traction yet in interface design. But I think it should.

We engage in prospective memory when we plan to do something in the future, and we arrange some way of reminding ourselves to do it. For example, if you need to bring a book to work the next day, you might put it on a table beside the front door the night before. If you need to respond to someone's email later (just not right now!), you might leave that email on your screen as a physical reminder. Or if you tend to miss meetings, you might arrange for Outlook or your mobile device to ring an alarm tone five minutes before each meeting.

Basically, this is something almost everyone does. It's a part of how we cope with our complicated, highly scheduled, multitasked lives: we use knowledge "in the world" to aid our own imperfect memories. We need to be able to do it well.

Some software does support prospective remembering. Outlook and most mobile platforms, as mentioned earlier, implement it directly and actively; they have calendars, and they sound alarms. But what else can you use for prospective memory?

- Notes to oneself, like virtual "sticky notes"
- Windows left on-screen
- Annotations put directly into documents (such as "Finish me!")
- Browser bookmarks, for websites to be viewed later
- Documents stored on the desktop, rather than in the usual places in the filesystem
- Email kept in an inbox (and maybe flagged) instead of filed away

People use all kinds of artifacts to support passive prospective remembering. But notice that almost none of the techniques in the preceding list were designed with that in mind! What they *were* designed for is flexibility—and a laissez-faire attitude toward how users organize their stuff. A good email client lets you create folders with any names you want, and it doesn't care what you do with messages in your inbox. Text editors don't care what you type, or what giant bold magenta text means to you; code editors don't care that you have a "Finish this" comment in a method header. Browsers don't care why you keep certain bookmarks around.

In many cases, that kind of hands-off flexibility is all you really need. Give people the tools to create their own reminder systems. Just don't try to design a system that's too smart for its own good. For instance, don't assume that just because a window's been idle for a while, that no one's using it and it should be closed. In general, don't "helpfully" clean up files or

objects that the system may think are useless; someone may be leaving them around for a reason. Also, don't organize or sort things automatically unless the user asks the system to do so.

As a designer, is there anything positive you can do for prospective memory? If someone leaves a form half-finished and closes it temporarily, you could retain the data in it for the next time—it will help remind the user where she left off. (See the Deferred Choices pattern.) Similarly, many applications recall the last few objects or documents they edited. You could offer bookmark-like lists of "objects of interest"—both past and future—and make those lists easily available for reading and editing. You can implement Many Workspaces, which lets users leave unfinished pages open while they work on something else.

Here's a bigger challenge: if the user starts tasks and leaves them without finishing them, think about how to leave some artifacts around, other than open windows, that identify the unfinished tasks. Another idea: how might a user gather reminders from different sources (email, documents, calendars, etc.) into one place? Be creative!

Streamlined Repetition

"I have to repeat this how many times?"

In many kinds of applications, users sometimes find themselves having to perform the same operation over and over again. The easier it is for them, the better. If you can help reduce that operation down to one keystroke or click per repetition—or, better, just a few keystrokes or clicks for all repetitions—you will spare users much tedium.

Find and Replace dialog boxes, often found in text editors (Word, email composers, etc.), are one good adaptation to this behavior. In these dialog boxes, the user types the old phrase and the new phrase. Then it takes only one Replace button click per occurrence in the whole document. And that's only if the user wants to see or veto each replacement—if she's confident that she really should replace all occurrences, she can click the Replace All button; one gesture does the whole job.

Here's a more general example. Photoshop lets you record "actions" when you want to perform some arbitrary sequence of actions with a single click. If you want to resize, crop, brighten, and save 20 images, you can record those four steps as they're done to the first image, and then click that action's Play button for each of the remaining 19. See the Macros pattern in Chapter 6 for more information.

Scripting environments are even more general. Unix and its variants allow you to script anything you can type into a shell. You can recall and execute single commands, even long ones, with a Ctrl-P and Return. You can take any set of commands you issue to the command line, put them in a `for` loop, and execute them by pressing the Return key once.

Or you can put them in a shell script (or in a `for` loop in a shell script) and execute them as a single command. Scripting is very powerful, and when complex, it becomes full-fledged programming.

Other variants include copy-and-paste capability (preventing the need to retype the same thing in a million places), user-defined "shortcuts" to applications on operating-system desktops (preventing the need to find those applications' directories in the filesystem), browser bookmarks (so users don't have to type URLs), and even keyboard shortcuts.

Direct observation of users can help you figure out just what kinds of repetitive tasks you need to support. Users won't always tell you outright. They may not even be aware that they're doing repetitive things that could be streamlined with the right tools—they may have been doing it so long that they don't even notice anymore. By watching them work, you may see what they don't see.

In any case, the idea is to offer users ways to streamline the repetitive tasks that could otherwise be time-consuming, tedious, and error-prone.

Keyboard Only

"Please don't make me use the mouse."

Some people have real physical trouble using a mouse. Others prefer not to keep switching between the mouse and keyboard because that takes time and effort—they'd rather keep their hands on the keyboard at all times. Still others can't see the screen, and their assistive technologies often interact with the software using just the keyboard API.

For the sakes of these users, some applications are designed to be "driven" entirely via the keyboard. They're usually mouse-driven too, but there is no operation that must be done with *only* the mouse—keyboard-only users aren't shut out of any functionality.

Several standard techniques exist for keyboard-only usage:

- You can define keyboard shortcuts, accelerators, and mnemonics for operations reachable via application menu bars, such as Ctrl-S for Save. See your platform style guide for the standard ones.
- Selection from lists, even multiple selection, is usually possible using arrow keys in combination with modifiers (such as the Shift key), though this depends on which component set you use.
- The Tab key typically moves the keyboard focus—the control that gets keyboard entries at the moment—from one control to the next, and Shift-Tab moves backward. This is sometimes called *tab traversal*. Many users expect it to work on form-style interfaces.

- Most standard controls, even radio buttons and combo boxes, let users change their values from the keyboard by using arrow keys, the Return key, or the space bar.
- Dialog boxes and web pages often have a "default button"—a button representing an action that says "I'm done with this task now." On web pages, it's often Submit or Done; on dialog boxes, OK or Cancel. When users press the Return key on this page or dialog box, that's the operation that occurs. Then it moves the user to the next page or returns him to the previous window.

There are more techniques. Forms, control panels, and standard web pages are fairly easy to drive from the keyboard. Graphic editors, and anything else that's mostly spatial, are much harder, though not impossible.

Keyboard-only usage is particularly important for data-entry applications. In these, speed of data entry is critical, and users can't afford to move their hands off the keyboard to the mouse every time they want to move from one field to another or even one page to another. (In fact, many of these forms don't even require users to press the Tab key to traverse between controls; it's done automatically.)

Other People's Advice

"What did everyone else say about this?"

People are social. As strong as our opinions may sometimes be, we tend to be influenced by what our peers think.

Witness the spectacular growth of online "user comments": Amazon for books (and everything else), IMDb for movies, Flickr for photographs, and countless retailers who offer space for user-submitted product reviews. Auction sites such as eBay formalize user opinions into actual prices. Blogs offer unlimited soapbox space for people to opine about and discuss anything they want, from products to programming to politics.

The advice of peers, whether direct or indirect, influences people's choices when they decide any number of things. Finding things online, performing transactions (Should I buy this product?), playing games (What have other players done here?), and even building things—people can be more effective when aided by others. If not, they might at least be happier with the outcome.

Here's a subtler example. Programmers use the MATLAB application to do scientific and mathematical tasks. Every few months, the company that makes MATLAB holds a public programming contest; for a few days, every contestant writes the best MATLAB code he can to solve a difficult science problem. The fastest, most accurate code wins. The catch is that every player can see everyone else's code—and copying is encouraged! The "advice" in this case is indirect, taking the form of shared code, but it's quite influential. In the end,

the winning program is never truly original, but it's undoubtedly better code than any solo effort would have been. (In many respects, this is a microcosm of open source software development, which is driven by a powerful set of social dynamics.)

Not all applications and software systems can accommodate a social component, and not all should try. But consider whether it might enhance the user experience to do so. And you could get more creative than just tacking a web-based bulletin board onto an ordinary site—how can you persuade users to take part constructively? How can you integrate it into the typical user's workflow?

If the task is creative, maybe you can encourage people to post their creations for the public to view. If the goal is to find some fact or object, perhaps you can make it easy for users to see what other people found in similar searches.

Of the patterns in this book, Multi-Level Help (Chapter 2) most directly addresses this idea; an online support community is a valuable part of a complete help system for some applications.

Personal Recommendations

"My friend told me to read this, so it must be pretty good."

This pattern operates on the same principle as the previous one—we are strongly influenced by our peers. So much so, in fact, that we are much more likely to view the articles and videos that someone refers us to than those we find in some other way. The personal touch makes a big difference when we decide what to read online.

Therefore, support person-to-person sharing of content. Let people send a URL (or the content itself) to friends and family, either via email or via a social network such as Facebook or Buzz.

This implies a host of mechanisms that need to be used or designed in. First, what exactly are users sharing? If the content doesn't already have a URL, see if one can be constructed for it. (The Deep-linked State pattern in Chapter 3 talks about this.) This URL should direct the recipient to a page with the same content that the sender was seeing, to avoid confusion.

Second, whom will they share it with? Let users connect to a social network, or give them a way to send email.

Third, what implications does this reference have? If a user sends email to a few "close ties," along with a personal message—one the user typed, not an automatic "personal message!—that can potentially carry a very high recommendation. After all, someone cared enough to think about you and take time to write a note. The specialness declines as the sender *CC*s more and more email addresses, though.

When a user posts a link to her Facebook or Twitter stream, that carries other implications: "I thought this was cool, and it represents something about who I am." Followers are still likely to read these links, especially if they trust that the poster has good taste. Furthermore, followers may repost or retweet it themselves, as will their followers, ad infinitum. This is how memes start, content goes viral, and the social web rolls on.

Chapter 2

Organizing the Content: Information Architecture and Application Structure

At this point, you know what your users want out of your application or site. You're targeting a chosen platform: the Web, the desktop, a mobile device, or some combination. You know which idiom or interface type to use—a form, an e-commerce site, an image viewer, or something else—or you may realize that you need to combine several of them. If you're really on the ball, you've written down some typical scenarios that describe how people might use high-level elements of the application to accomplish their goals. You have a clear idea of what value this application adds to people's lives.

Now what?

You could start making sketches of the interface. Many visual thinkers do that at this stage. If you're the kind of person who likes to think visually and needs to play with sketches while working out the broad strokes of the design, go for it.

But if you're not a visual thinker by nature (and sometimes even if you are), hold off on the interface sketches. They might lock your thinking into the first visual designs you put on paper. You need to stay flexible and creative for a little while, until you work out the overall organization of the application.

It can be helpful to think about an application in terms of its underlying data and tasks. What objects are being shown to the users? How are they categorized and ordered? What do users need to do with them? And now that you're thinking abstractly about them, how many ways can you design a presentation of those things and tasks?

These lines of inquiry may help you think more creatively about the interface you're designing.

Information architecture (IA) is the art of organizing an information space. It encompasses many things: presenting, searching, browsing, labeling, categorizing, sorting, manipulating, and strategically hiding information. Especially if you're working with a new product, this is where you should start.

The Big Picture

Let's look at the very highest level of your application first. From the designer's perspective, your site or application probably serves several functions: a software service—maybe several services—sharing information, selling a product, branding, social communication, or any number of other goals. Your home page or opening screen may need to convey all of these. Via text and imagery, users should be directed to the part of your site or app that accomplishes *their* purposes.

At this level, you'll make decisions about the whole package. What interaction model will it use? The desktop metaphor? The simpler model of a traditional website? Or a richly interactive site that splits the difference? Is it a self-contained device such as a mobile phone or digital video recorder, for which you must design the interactions from scratch? The interaction model establishes consistency throughout the artifact, and it determines how users move through and among the different pieces of functionality. I won't go into more detail at this level, because almost all of the patterns in this book apply at smaller scales.

Now let's look at a smaller unit within an application or site: pages that serve single important functions. In an application, this might be a main screen or a major interactive tool; in a richly interactive website, it might be a single page, such as Gmail's main screen; in a more static website, it might be a group of pages devoted to one process or function.

Any such page will primarily do one of these things:

1. Show one single thing, such as a map, book, video, or game
2. Show a list or set of things
3. Provide tools to create a thing
4. Facilitate a task

Most apps and sites do some combination of these things, of course. A website might show a feature article (1), a list of additional articles (2), with a wiki area for members to create pages (3), and a registration form for new members (4). That's fine. Each of these parts of the site should be designed using patterns and tools to fit that particular organizing principle.

This list mirrors some of the work done by Theresa Neil with application structures in the context of rich Internet applications (RIAs). She defines three types of structures based on the user's primary goal: information, process, and creation.*

This list gives us a framework within which to fit the idioms and patterns we'll talk about in this and other chapters.

* "Rich Internet Screen Design," in *UX Magazine*: *http://www.uxmag.com/design/rich-internet-application-screen-design.*

Show One Single Thing

Is this really what your page does? The whole point of the page's design is to show or play a single piece of content, with no list of other pieces that users could also see, no comments, and no table of contents or anything like that?

Lucky you!

All you really need, then, is to manage the user's interaction with this one thing. The IA is probably straightforward. There might be small-scale tools clustered around the content—scrollers and sliders, sign-in box, global navigation, headers and footers, and so forth—but they are minor and easily designed. Your design might take one of these shapes:

- A long, vertically scrolled page of flowed text (articles, books, and similar long-form content).
- A zoomable interface for very large, fine-grained artifacts, such as maps, images, or information graphics. Map sites such as Google Maps provide some well-known examples.
- The "media player" idiom, including video and audio players.

As you design this interface, consider the following patterns and techniques to support the design:

- Alternative Views, to show the content in more than one way.
- Many Workspaces, in case people want to see more than one place, state, or document at one time.
- Deep-linked State, in Chapter 3. With this, a user can save a certain place or state within the content so that he can come back to it later or send someone else a URL.
- Sharing Widget and other social patterns, in Chapter 9.
- Some of the mobile patterns described in Chapter 10, if one of your design goals is to deliver the content on mobile devices.

Show a List of Things

This is what most of the world's digital artifacts seem to do. Lists are everywhere! The digital world has converged on many common idioms for showing lists, most of which are familiar to you—simple text lists, menus, grids of images, search results, lists of email messages or other communications, tables, trees. There are more, of course.

Lists present rich challenges in information architecture. How long is the list? Is it flat or hierarchical, and if it is a hierarchy, what kind? How is it ordered, and can the user change that ordering dynamically? Should it be filtered or searched? What information or operations are associated with each list item, and when and how should they be shown?

Because lists are so common, a solid grasp of the different ways to present them can benefit any designer. It's the same theme again—by learning and formalizing these techniques, you can expand your own thinking about how to present content in different and interesting ways.

A few patterns for designing an interface around a list are described in this chapter (others are in Chapter 5). You can build either an entire app or site, or a small piece of a larger artifact, around one of these patterns. They set up a structure that other display techniques—text lists, thumbnail lists, and so on—can fit into. Other top-level organizations not listed here might include calendars, full-page menus, and search results.

- Feature, Search, and Browse is the pattern followed by countless websites that show products and written content. Searching and browsing provide two ways for users to find items of interest, while the front page features one item to attract interest.
- Blogs, news sites, email readers, and social sites such as Twitter all use the News Stream pattern to list their content, with the most recent updates at the top.
- Picture Manager is a well-defined interface type for handling photos and other pictorial documents. It can accommodate hierarchies and flat lists, tools to arrange and reorder documents, tools to operate directly on pictures, and so on.

Once you've chosen an overall design for the interface, you might look at other patterns and techniques for displaying lists. These fit into the patterns mentioned earlier; for instance, a Picture Manager might use a Thumbnail Grid, a Pagination, or both to show a list of photos—all within a Two-Panel Selector framework. See Chapter 5 for a thorough discussion.

Provide Tools to Create a Thing

Builders and editors are the great dynastic families of the software world. Microsoft Word, Excel, PowerPoint, and other Office applications, in addition to Adobe Photoshop, Illustrator, In Design, Dreamweaver, and other tools that support designers are all in this category. So are the tools that support software engineers, such as the various code editors and integrated development environments. These have long histories, large user bases, and very well established interaction styles, honed over many years.

Most people are familiar with the idioms used by these tools: text editors, code editors, image editors, editors that create vector graphics, and spreadsheets.

Chapter 8 of the previous edition of this book discusses how to design different aspects of these tools. But at the level of application structure or IA, the following patterns are often found:

- Canvas Plus Palette describes most of these applications. This highly recognizable, well-established pattern for visual editors sets user expectations very strongly.
- Almost all applications of this type provide Many Workspaces—usually windows containing different documents, which enable users to work on them in parallel.

- Alternative Views let users see one document or workspace through different lenses, to view various aspects of the thing they're creating.
- "Blank Slate Invitation" is named and written about in *Designing Web Interfaces* (*http://oreilly.com/catalog/9780596516253/*) by Bill Scott and Theresa Neil (O'Reilly), and is a profoundly useful pattern for builders and editors. It is closely related to the Input Hints pattern in Chapter 8.

Facilitate a Single Task

Maybe your interface's job isn't to show a list of anything or create anything, but simply to get a job done. Signing in, registering, posting, printing, uploading, purchasing, changing a setting—all such tasks fall into this category.

Forms do a lot of work here. Chapter 8 talks about forms at length and lists many controls and patterns to support effective forms. Chapter 6 defines another useful set of patterns that concentrate more on "verbs" than "nouns."

Not much IA needs to be done if the user can do the necessary work in a small, contained area, such as a sign-in box. But when the task gets more complicated than that—if it's long, or branched, or has too many possibilities—part of your job is to work out how the task is structured.

- Much of the time, you'll want to break the task down into smaller steps or groups of steps. For these, a Wizard might work well for users who need to be walked through the task.
- A Settings Editor is a very common type of interface that gives users a way to change the settings or preferences of something—an application, a document, a product, and so on. This isn't a step-by-step task at all. Here, your job is to give users open access to a wide variety of choices and switches and let them change only what they need, when they need it, knowing that they will skip around.

The Patterns

Several of the patterns in this chapter are large-scale, defining the interactions for large sections of applications or sites (or sometimes the entire thing). Some of these, including Picture Manager, Canvas Plus Palette, and Feature, Search, and Browse, are really clusters of other patterns that support each other in well-defined ways—they are "guilds" of smaller-scale patterns.

1. Feature, Search, and Browse
2. News Stream
3. Picture Manager
4. Dashboard

5. Canvas Plus Palette
6. Wizard
7. Settings Editor

The last three patterns are more "meta," in the sense that they can apply to the other patterns in the preceding list. For instance, almost any content, document, or list can be shown in more than one way, and the ability to switch among those Alternative Views can empower users.

8. Alternative Views

Likewise, a user may want to instantiate the interface more than once, to maintain several trains of thought simultaneously—consider the tabs in a browser window, all showing different and unrelated websites. Offer the Many Workspaces pattern to these users.

9. Many Workspaces

Many patterns, here and elsewhere in the book, contribute in varying degrees to the learnability of an interface. Multi-Level Help sets out ways to integrate help into the application, thus supporting learnability for a broad number of users and situations.

10. Multi-Level Help

Feature, Search, and Browse

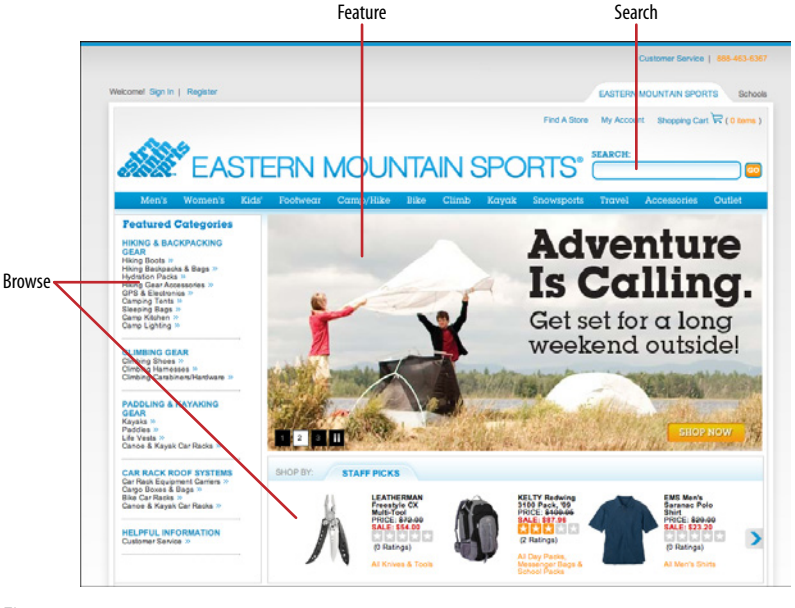


Figure 2-1. *EMS*