PAYRANGE - EXHIBIT 2011
Alliance Laundry v. PayRange
PGR2025-00027

"Alan Cooper is a software god... This is a landmark book."—*Stewart Alsop, former Editor-in-Chief,* InfoWorld

ABOUT FACE

THE ESSENTIALS OF USER INTERFACE DESIGN



ALAN COOPER

"Father of Visual Basic"
Microsoft Windows Pioneer Award Honoree

Foreword by Andrew Singer

IDG BOOKS WORLDWIDE

RECEIVED
NOV 2 0 1998
WILSON, SONSINI, GOODRICH
& ROSATI LIBRARY

About Face

The Essentials of User Interface Design

Alan Cooper



A Division of IDG Books Worldwide, Inc.
Foster City, CA • Chicago, IL • Indianapolis, IN • Braintree, MA • Southlake, TX

About Face: The Essentials of User Interface Design
Published by
IDG Books Worldwide, Inc.
An International Data Group Company
919 East Hillsdale Boulevard, Suite 400
Foster City, CA 94404

Copyright
Copyright © 1995 by IDG Books Worldwide, Inc. All rights reserved. No part of this book (including interior design, cover design, and illustrations) may be reproduced or transmitted in any form, by any means, (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher. For authorization to photocopy items for internal corporate use, personal use, or for educational and/or classroom use, please contact: Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923 USA, Fax 508-750-4470.

Library of Congress Catalog Card No.: 95-75055
ISBN 1-56884-322-4
Printed in the United States of America
First Printing, August, 1995
10 9 8 7 6

Distributed in the United States by IDG Books Worldwide, Inc.

Limit of Liability/Disclaimer of Warranty
The author and publisher of this book have used their best efforts in preparing this book. IDG Books Worldwide, Inc., International Data Group, Inc., and the author make no representation or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose, and shall in no event be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.

Trademarks
All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. IDG Books Worldwide, Inc., is not associated with any product or vendor mentioned in this book.

Published in the United States

Table of Contents

Introduction 1
Who should read this book 2
Why I wrote this book 3
A taxonomy of software design 4
Conventions used in this book 6
Let's design 8

Part I: The Goal 9

Chapter 1: Goal-Directed Design 11
The user's goals 12
The essence of user interface design 16
A fresh look at features 18

Chapter 2: Software Design 21
Software isn't designed 21
Conflict of interest 23
The profession of software design 24
Supporting software design disciplines 24

Chapter 3: The Three Models 27
The manifest model 27
Bringing mechanical age models into the information age 35
It's worse on a computer 39

Chapter 4: Visual Interface Design 41
Restricting the vocabulary 47
The Canonical Vocabulary 48
Designing for users 49

Part II: The Form 51

Chapter 5: Idioms and Affordances 53
The Myth of Metaphor 53
Manual affordances 64
Understanding what it means 65
Chapter 6: An Irreverent History of Rectangles on the Screen . . .67
Xerox PARC 67
Chapter 7: Windows-with-a-Small-w 73
Unnecessary rooms 73
Necessary rooms 75
Windows pollution 77
Chapter 8: Lord of the Files 81
The tragedy of the file system 81
Designing software with the proper model 86
Unify the file model 91
Document management 92
How did we get here? 96
Chapter 9: Storage and Retrieval Systems 101
Storing versus finding 101
It ain't document-centric 107
Chapter 10: Choosing Platforms 113
Software is the expensive part 113
The half-life of a desktop computer 114
Choosing a development platform 116
Simultaneous Multiplatform Development 119
The Myth of Interoperability 121

Part III: The Behavior 125

Chapter 11: Orchestration and Flow 127
Planing on the step 127

Where were you on the night of the sixteenth? 144
Sensible interaction 146
Chapter 12: Posture and State 151
Posture 151
Windows states 163
MDI 168
Chapter 13: Overhead and Idiocy 171
Overhead 171
Idiocy 178
Chapter 14: The Secret Weapon of Interface Design 183
Get a memory 183
Task coherence 186
A new way of thinking 190

Part IV: The Interaction 193
Chapter 15: Elephants, Mice and Minnies 195
Why we use a mouse instead of a pen 195
Indirect manipulation 196
Mice are not here to stay 197
Mousing around 198
The left mouse button 200
Right mouse button 201
Middle mouse button 202
Things you can do with a mouse 202
Up and down events 207
The cursor 208
Focus 212
Meta-keys 214
Chapter 16: Selection 217
Object-verb 217
Concrete and discrete data 219
Insertion and replacement 220
Additive selection 222
Group selection 223
Visual indication of selection 224
Chapter 17: Direct Manipulation 229

Manipulating gizmos 231
Repositioning 238
Resizing and reshaping 239
Arrowing 244
Direct-manipulation visual feedback 245
Chapter 18: Drag-and-Drop 247
Whither drag-and-drop? 247
Dragging where? 249
Master-and-target 249
How master-and-target works 252
Tool-manipulation drag-and-drop 256
Bomb sighting 259
Drag-and-drop problems and solutions 260

Part V: The Cast 269
Chapter 19: The Meaning of Menus 271
The command-line interface 271
The hierarchical menu interface 272
The Lotus 1-2-3 interface 274
Monocline grouping 276
The popup menu 277
The pedagogic vector 278
Chapter 20: Menus 283
Standard menus 283
The correct menus 285
Meanwhile, back on Planet Earth 288
Optional menus 289
Menu item variants 291
The system menu 297
Chapter 21: Dialog Boxes 299
Suspension of normal interaction 299
Dialog box basics 302
Modal dialog boxes 302
Modeless dialog boxes 303
The modeless dialog problem 304

Two solutions 305
A more radical, but better, solution 307
Property dialog boxes 311
Function dialog boxes 313
Bulletin dialog boxes 313
Process dialog boxes 315
Chapter 22: Dialog Box Etiquette 319
You rang? 319
The caption bar 320
Transient posture 322
Reduce excise 322
Terminating commands for modal dialog boxes 325
Keyboard shortcuts 328
Tabbed dialogs 328
Expanding dialogs 334
Cascading dialogs 335
Directed dialogs 337
Chapter 23: Toolbars 341
Visible and immediate 341
The toolbar freed the menu to teach 345
Beyond the buttcon 349
Toolbar morphing 350
Chapter 24: Roll the Credits, Please 355
Your program's name 355
Your program's icon 357
Dependencies 357
About boxes 358
Splash screens 362
Easter eggs 364

Part VI: The Gizmos 367
Chapter 25: Imperative and Selection Gizmos 369
Gizmo-laden dialog boxes 370
Gizmo liberation 370
The gizmos that Mother gives you 372

Imperative gizmos 372
Selection gizmos 375
Combobox 391
Treeview gizmo 392
Chapter 26: Entry and Display Gizmos 393
Entry gizmos 393
Bounding 394
Unbounded entry fields 397
Validation 398
Using an edit field for output 403
Display gizmos 405
Those darned scrollbars 406
Chapter 27: New Gizmos 409
Directly manipulable tools 409
Extraction gizmos 412
Visual gizmos 416
Adding visual richness 419

Part VII: The Guardian 421
Chapter 28: The End of Errors 423
Eliminating the error message box 423
Bulletin Dialog Boxes 424
Stopping the proceedings 425
Positive feedback 433
Treat error messages like GOTOs 435
Exceptions? 436
Do they work? 437
What error message dialog boxes should look like 438
The end of errors 439
Chapter 29: Managing Exceptions 441
Alerts 441
Confirmations 444
Who are we protecting, anyway? 448
Audible feedback 454
Using your powers for good 457

Failing gracefully 462
Chapter 30: Undo 465
Assisting the exploration 465
The trouble with single undo 470
Redo 472
Special undo functions 473
Deleted data buffer 475
Other manifest models 476
Undo is a global facility and should not be managed by local controls 478
Undo-proof operations 478

Part VIII: The Teacher 481
Chapter 31: Good at What You Do 483
The time users spend 483
Command vectors 486
What beginners need 490
What perpetual intermediates need 492
What experts need 492
Idiosyncratically modal behavior 493
Commensurate effort 495
The typers versus the pointers 495
Standards 499
Online help 501
The inverted meta-question 504
Chapter 32: Installation, Configuration and Personalization 507
Navigation is by reference to permanent objects 507
Pull at your own risk 510
The corporate look 513
Installation 515
Chapter 33: Shouldering the Burden 531
Let's put those idle cycles to work 533
Get our software talking to our hardware 541
Chapter 34: Where Do We Go from Here? 543
Software sucks 543

We know a lot about old technology 545
Don't ask programmers to design while they code 547
Solving the problem 548
"I'm mad as hell, and I'm not gonna take it anymore" 552

5

Idioms and Affordances

There is nothing in the world of software development that is quite as frightening as an empty screen. When we begin designing the user interface, we must first confront that awful whiteness, and ask ourselves: What does good software look like?

The Myth of Metaphor

Software designers often speak of "finding the right metaphor" upon which to base their interface design. They imagine that filling their interface with images of familiar objects from the real world will give their users a pipeline to easy learning. So they create an interface masquerading as an office filled with desks, file cabinets, telephones and address books, or as a pad of paper or a street of buildings. And if you, too, search for that magic metaphor, you will be in august company; some of the best and brightest designers in the interface world put metaphor selection as one of their first and most important tasks.

Searching for that magic metaphor is one of the biggest mistakes you can make in user interface design. Searching for an elusive guiding metaphor is like searching for the

correct steam engine to power your airplane, or searching for a good dinosaur on which to ride to work.

Basing a user interface design on a metaphor is not only unhelpful, it can often be quite harmful. The idea that good user interface design relies on metaphors is one of the most insidious of the many myths that permeate the software community.

Metaphors offer a tiny boost in learnability to first-time users, but at a tremendous cost. By representing old technologies, most metaphors firmly nail our conceptual feet to the ground, forever limiting the power of our software. They have a host of other problems as well, including the simple facts that there aren't enough metaphors to go around, they don't scale well, and the ability of users to recognize them is questionable.

The three interface paradigms

There are three dominant paradigms in the design of user interfaces. I call these three the **technology paradigm**, the **metaphor paradigm** and the **idiomatic paradigm**. The technology paradigm is based on *understanding* how things work—a difficult proposition. The metaphor paradigm is based on *intuiting* how things work—a risky method. The idiomatic paradigm, however, is based on *learning* how to accomplish things—a natural, human process.

The field of user interface design has progressed from an orientation focused on technology, into one that focuses overmuch on metaphor. We are just now becoming aware of idiomatic design. There is ample evidence of all three paradigms in contemporary software design, even though the metaphor paradigm is the only one that has been named and described. We pay metaphors lots of lip service and, all too often, hamper the creation of really good interfaces by following their false trail.

The technology paradigm

The **technology paradigm** of user interface design is simple and incredibly widespread in the computer industry. The technology paradigm merely means that the interface is expressed in terms of its construction—of how it was built. In order to successfully use it, the user must understand how the software works. Following the technology paradigm means user interface design based exclusively on the implementation model.

There was a genre of building architecture, popular in the 1960s, called Metabolist. In Metabolist architecture, the elevator shafts, air conditioning ducts, cable runs, steel beams and other construction impedimenta are left uncovered and visible. The muscles, bones and sinews of the building are exposed—even emphasized—without any hint of modesty. The idea was that the building is a machine for living and its form should follow its implementation details. The overwhelming majority of software programs today are Metabolist in that they show us, without any hint of shame, precisely how they are built. There is one button per function, one function per module of code, and the commands and processes precisely echo the internal data structures and algorithms.

We can see how a technology program ticks merely by learning how to run it; The problem is that the reverse is also true: We *must* learn how it ticks in order to run it.



Users would rather be successful than knowledgeable

Engineers want to know how things work, so the technology paradigm is very satisfying to them (which, of course, is why so much of our software follows it). Engineers prefer to see the gears and levers and valves because it helps them understand what is going on inside the machine. That those artifacts needlessly complicate the interface seems a small price to pay. Engineers may want to understand the inner workings, but most users don't have either the time or desire. They'd much rather be successful than be knowledgeable, a state that is often hard for engineers to understand.

The metaphor paradigm

In the 1970s, the modern graphical user interface was invented at Xerox Palo Alto Research Center (PARC). It has swept the industry, but what, exactly, is it? The GUI—as defined by PARC—consisted of many things: windows, buttons, mice, icons, metaphors, pull-down menus. Some of these things are good and some are not so good, but they have all achieved a kind of holy stature in the industry by association with the empirical superiority of the ensemble.

In particular, the idea that metaphors are a firm foundation for user interface design is a very misleading proposition. It's like worshipping 5.25" floppy diskettes because so much good software once came on them.

The first commercially successful implementation of the PARC GUI was the Apple Macintosh, with its desktop, wastebasket, overlapping sheets of paper and file folders. The Mac didn't succeed because of these metaphors, however, but because it was the first computer that defined a tightly restricted vocabulary—a canonical vocabulary based on a very small set of mouse actions—for communicating with users. It also offered richer visual interaction. The metaphors were just nice paintings on the walls of a well-designed house.

Metaphors don't scale very well. A metaphor that works well for a simple process in a simple program will often fail to work well as that process grows in size or complexity. File icons were a good idea when computers had floppies or 10 MB hard disks with only a couple of hundred files, but in these days of multi-gigabyte hard disks and thousands of files, file icons can get pretty clumsy.

When we talk about metaphors in the user interface design context, we really mean visual metaphors: a picture of something used to represent that thing. Users recognize the imagery of the metaphor and, by extension, can understand the purpose of the thing. Metaphors range from the tiny images on toolbar buttons to the entire screen on some programs. They can be a tiny scissors on a button indicating "cut," or a full-size checkbook in Quicken. We understand metaphors intuitively. Webster's defines intuition like this:

> **in·tu·i·tion** \in-'tu-wi-shen\ *n* 1 : quick and ready insight 2 **a**: immediate apprehension or cognition **b**: knowledge or conviction gained by intuition **c**: the power or faculty of attaining direct knowledge or cognition without evident rational thought and inference

The dictionary highlights the magical quality of intuition, but it doesn't say *how* we intuit something. Intuition works by inference, where we see connections between disparate subjects and learn from these similarities while not being distracted by their differences. We grasp the meaning of the metaphoric controls in an interface because we mentally connect them with other processes or things we have already learned. This is an efficient way to take advantage of the awesome power of the human mind to make inferences, something that CPUs are incapable of. But this method also depends on the creaky, cantankerous, idiosyncratic human mind, which may not have the requisite language,

knowledge or inferential power necessary to make the connection. Metaphors are not dependable in the way that understanding is. Sometimes the magic works, sometimes it doesn't.

Metaphors rely on associations perceived in similar ways by both the designer and the user. If the user doesn't have the same cultural background as the designer, it is easy for metaphors to fail. Even in the same or similar cultures, there can be significant misunderstandings. Does a picture of an airplane mean "send via airmail" or "make airline reservations"?

The **metaphor paradigm** relies on intuitive connections in which there is no need to understand the mechanics of the software, so it is a step forward from the technology paradigm, but its power and usefulness has been inflated to unrealistic proportions.

Recall from our definition of intuition that no rational thought is evident in the process. I think it is silly to imagine that we can base good user interface design on a kind of mental magic that thumbs its nose at thinking. In the computer industry, and particularly in the user interface design community, the word *intuitive* is often used to mean easy-to-use or easy-to-understand. I'm a big fan of easy-to-use, but it doesn't promote our craft to attribute its success to metaphysics. Nor does it help us to devalue the precise meaning of the word. There are very real reasons why people understand certain interfaces and not others.

There are certain sounds, smells and images that make us respond without any previous conscious learning. When a small child encounters an angry dog, she *instinctively* knows that bared fangs are a sign of great danger, even without any previous learning. The encoding for such recognition goes deep. Instinct is a hard-wired response that involves no conscious thought. Intuition is one step above instinct because, although it also requires no conscious thought, it is based on a web of knowledge learned consciously.

Examples of instinct in human-computer interaction include the way we are startled and made apprehensive by gross changes in the image on the screen, or react to sudden noises from the computer or the smell of smoke rising from the CPU.

Intuition is a middle ground between having consciously learned something and knowing something instinctively. If we have learned that things glowing red can burn us, we tend to classify all red-glowing things as potentially dangerous until proven otherwise. We don't necessarily know that the particular

red-glowing thing is a danger, but it gives us a safe place to begin our exploration.

What we commonly refer to as intuition is actually a mental comparison between something and the things we have already learned. You instantly intuit how to work a wastebasket icon, for example, because you once learned how a real wastebasket works, thereby preparing your mind to make the connection years later. But you didn't *intuit* how to use the original wastebasket. It was just an extremely easy thing to learn. This brings us to the third paradigm, which is based on the fact that the human mind is an incredibly powerful learning machine, and that learning isn't hard for us.

The idiomatic paradigm

The idiomatic method of user interface design solves the problems of both of the previous two. I call it the **idiomatic paradigm** because it is based on the way we learn and use idioms, or figures of speech, like "beat around the bush" or "cool."

These idiomatic expressions are easily understood but not in the same way metaphors are. There is no "bush" and nobody is beating anything. We understand the idiom simply because we have learned it and because it is distinctive, not because we understand it or because it makes subliminal connections in our minds.

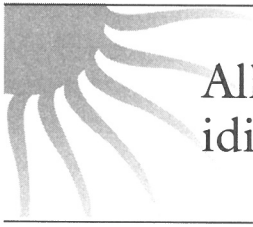
This is where the human mind is really outstanding: learning and remembering idioms very easily without relying on comparisons to known situations or an understanding of how they work. This is a necessity, because many idioms don't have any metaphoric meaning at all, and the stories behind most others were lost ages ago.

Most of the elements of a GUI interface are idioms. Windows, caption bars, close boxes, screen-splitters and drop-downs are things we learn idiomatically rather than intuit metaphorically.

We are inclined to think that learning is hard because of our conditioning from the technology paradigm. Those old interfaces were very hard to learn because you also had to understand how they worked. Most of what we know we learn *without* understanding: things like faces, social interactions, attitudes, the arrangement of rooms and furniture in our houses and offices. We don't "understand" why someone's face is composed the way it is, but we "know" his face. We recognize it because we have looked at it and automatically (and easily) memorized it.

The familiar mouse is not metaphoric of anything, but rather is learned idiomatically. There is a scene in *Star Trek IV* where Scotty returns to twentieth-century Earth and tries to speak into a mouse. It is one of the few parts of that movie that is not fiction. There is nothing about the physical appearance of the mouse that indicates its purpose or use, nor is it comparable to anything else in our experience, so learning it is not intuitive. However, learning to point at things with a mouse is incredibly easy. Someone probably spent all of three seconds showing it to you the first time, and you mastered it from that instant on. We don't know or care how mice work, and yet even small children can operate them just fine. That is idiomatic learning.

Not only can you not intuit an idiom, neither can you reason it out. Our language is filled with idioms that, if you haven't been taught them, make no sense. If I say my Uncle Joe "kicked the bucket," you know what I mean even though there is no bucket or kicking involved. You can't know this because you have thought through the various permutations of smacking pails with your feet. You can only learn this from context in something you read or by being consciously taught it. You remember this obscure connection between buckets, kicking and dying only because humans are good at remembering stuff like this.



All idioms must be learned. Good idioms only need to be learned once

The key observation about idioms is that although they must be learned, good ones only need to be learned once. It is quite easy to learn idioms like "neat" or "politically correct" or "the lights are on but nobody's home" or "in a pickle" or "inside the Beltway" or "take the red-eye" or "grunge." The human mind is capable of picking up idioms like these from a single hearing. It is similarly easy to learn idioms like check-boxes, radio buttons, push-buttons, close-boxes, pulldown menus, icons, tabs, comboboxes, keyboards, mice and pens.

Branding

Marketing professionals know this idea of taking a simple action or symbol and imbuing it with meaning. After all, synthesizing idioms is the essence of product branding, whereby a company takes a product or company name and

imbues it with a desired meaning. Tylenol is, by itself, a meaningless word, an idiom, but the McNeil company has spent millions to make you associate that word with safe, simple, trustworthy pain relief. Of course, idioms are visual, too. The golden arches of McDonalds, the three diamonds of Mitsubishi, the five interlocking rings of the Olympics, even Microsoft's flying window are non-metaphoric idioms that are instantly recognizable and imbued with common meaning. The example of idiomatic branding shown in Figure 5-1 illustrates its power.

Ironically, many of the familiar GUI elements that are often thought of as metaphoric are actually idiomatic. Artifacts like window close boxes, resizable windows, infinitely nested file folders and clicking and dragging are non-metaphoric operations—they have no parallel in the real world. They derive their strength only from their easy idiomatic learnability.

The showstoppers

If we depend on metaphors to create user interfaces, we encounter not only the minor problems already mentioned, but also two more major problems: metaphors are hard to find and they constrict our thinking.

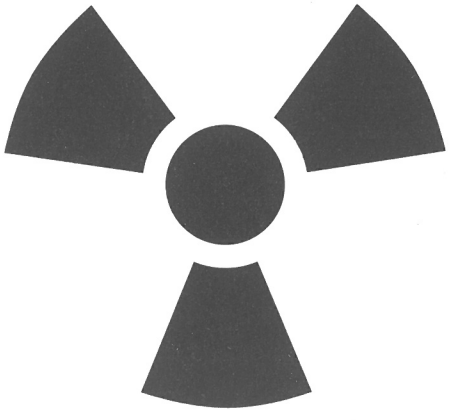


Figure 5-1

Here is a randomly chosen idiomatic symbol that has been imbued with meaning from use rather than from any inherent metaphoric value. For anyone who grew up in the '50s or '60s, this otherwise meaningless symbol has the power to cause a small shiver of fear to touch our backs. Idioms are just as powerful as metaphors. The power comes from how we use them and how we associate them, rather than from any innate imagery.

It may be easy to discover visual metaphors for physical objects like printers and documents. It can be difficult or impossible to find metaphors for processes, relationships, services and transformations—the most frequent uses of software. It can be extremely daunting to find a useful visual metaphor for buying a ticket, changing channels, purchasing an item, finding a reference, setting a format, rotating a tool or changing resolution, yet these operations are precisely the type of processes we use software to perform most frequently.

The most insidious problem with metaphors, the real showstopper, is that they tie our interfaces to mechanical age artifacts. It is easy to intuit how to use the clipboard, for example, because it is a metaphor. But if we adhere strictly to the clipboard metaphor, the facility is incredibly weak. It won't hold more than one thing, it doesn't have a memory of what it held before, it can't identify where the images came from, it can't show you thumbnails of what it holds and it doesn't save its contents from run to run. All of these actions are non-metaphoric and have to be learned. Following the metaphor gives users a momentary boost the first time they use the clipboard, but it costs them greatly after that in the arbitrary weakness of the facility.

Another really outrageous example is MagiCap, a communications interface from General Magic. It relies on metaphor for every aspect of its interface. As you can see in Figure 5-2, you metaphorically walk down a street lined with buildings representing services. You enter a building to begin a service, which is represented as a walk down a hallway that is lined with doors representing functions. This heavy reliance on metaphor means that you can intuit the basic functioning of the software, but its downside is that the metaphor restricts all navigation to a very rudimentary, linear path. You *must* go back out onto the street to go to another service. This may be normal in the physical world, but there is no reason for it in the world of software. Why not abandon this slavish devotion to metaphor and give the user services they *can't* get out on the street?

For all the limitations of metaphors, there is nothing bad about using one if it fits the situation. If I see a twenty-dollar bill lying on the sidewalk, of course I'll pick it up: I'd be a fool not to! But I'd be a bigger fool if I decided to make my living finding misplaced twenty-dollar bills. Metaphors are like that: use 'em if you find 'em, but don't bend your interface to fit some arbitrary metaphoric standard.

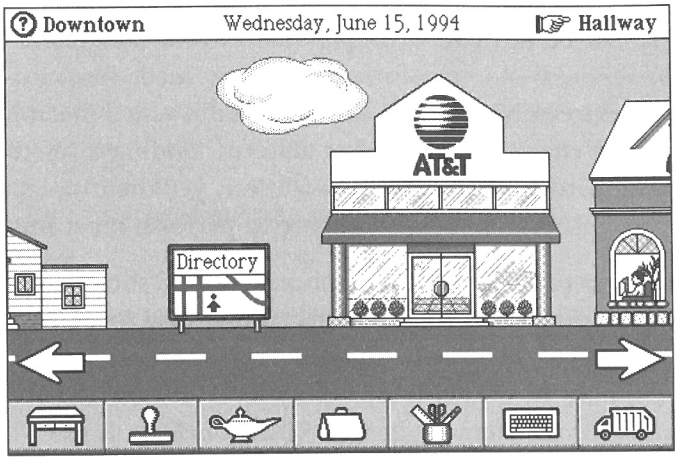


Figure 5-2

This is the MagiCap interface from General Magic. It is the acme of the expression of the metaphoric paradigm. Nothing in the program is done without a thorough metaphoric rationalization. I am in awe of its designers: the program is a tour de force of metaphor-finding above and beyond the call of duty. All of the interaction has been subordinated to the maintenance of these metaphors. I'm sure it was a lot of fun to design. I'll bet it is a real pain to use. Once you have learned that the substantial-looking building with the big "AT&T" on its facade is the phone company, you must forever live with going in and out of that building to call people. This most-modern, information-age software drags all of the limitations of the mechanical age into the future and forces us to live with them yet again. Is this progress?

Never bend your interface to fit a metaphor

On a design project for a library management system, we had to present a screen with multiple parts. Some gizmos were common to all of the parts, while others came and went depending on the active part. We made part of the screen look like a wire-bound notebook. The pages could flip like a notebook while the rest of the screen remained stationary. The gizmos on the flipping pages came and went while the gizmos outside of the notebook stayed still and worked globally. The notebook metaphor drew the user's attention to the difference and offered some visual help in understanding the scope of the controls. The metaphor fit naturally into the design of the overall product, so we used it.

General Magic's interface relies on what is called a **global metaphor**. This is a single, overarching metaphor that provides a framework for all of the other metaphors in the system. The desktop of the original Macintosh is also considered a global metaphor.

A hidden problem of global metaphors is the mistaken belief that other little daughter metaphors consistent with them enjoy cognitive benefits by association. The temptation is irresistible to stretch the metaphor beyond simple function recognition: That little software telephone also lets you "dial" with buttons just like those on our desktop telephones. We see software that has "address books" of phone numbers just like those in our pockets and purses. Wouldn't it be better to go beyond these confining technologies and deliver some of the real power of the computer? Why can't our communications software allow multiple connections or make connections by organization or affiliation, or just hide the use of phone numbers altogether?

It may seem clever to represent your dial-up service with a picture of a telephone sitting on a desk, but it actually imprisons you in a bad design. The original makers of the telephone would have been ecstatic if they could have created one that let you call your friends just by pointing to pictures of them. They couldn't because they were restricted by the dreary realities of electrical circuits and Bakelite moldings. On the other hand, today we have the luxury of rendering our communications interfaces in any way we please—showing pictures of our friends is completely reasonable—yet we insist on holding these concepts back with little pictures of obsolete technology.

There are two snares here, one for the user and one for the designer. Once the user depends on the metaphor for recognition, he expects consistency. This causes the snare for the designer, who will now be tempted to render the software in terms of the mechanical-age metaphor. As we saw in Part I, transliterating mechanical processes onto the computer just makes them worse than they were before.

Brenda Laurel says in her book *Computers as Theatre* (Addison-Wesley, 1991), "Interface metaphors rumble along like Rube Goldberg machines, patched and wired together every time they break, until they are so encrusted with the artifacts of repair that we can no longer interpret them or recognize their referents." It amazes me that software designers, who can finally create that dream-phone interface, give us the same old telephone simply because they were taught that a strong, global metaphor is a prerequisite to good user interface

design. Of all the misconceptions to emerge from Xerox PARC, the global metaphor myth is the most debilitating and unfortunate.

Idiomatic design is the future of user interface design. Using this paradigm, we depend on the natural ability of humans to learn easily and quickly as long as we don't force them to understand how and why. There is an infinity of idioms waiting to be invented, but only a limited set of metaphors waiting to be discovered. Metaphors give first-timers a penny's worth of value but cost them many dollars' worth of problems as they continue to use the software. It is always better to design idiomatically, only using metaphors when one falls in our lap.

Manual affordances

Donald Norman, in *The Psychology of Everyday Things* (Basic Books, 1988), has given us the fine term **affordance**, which he defines as "the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used."

This definition is fine as far as it goes, but it omits the key connection: *how* do we know what those properties offer us? If you look at something and understand how to use it—you comprehend its affordances—you must be using some method for making the mental connection.

I would alter Norman's definition by omitting the phrase "and actual." By doing this, affordance becomes a purely cognitive term, referring to what we *think* the object can do rather than what it can actually do. If a push-button is placed on the wall next to the front door of a residence, its affordances are 100% doorbell. If, when we push it, it causes a trapdoor to open beneath us and we fall into it, it turns out that it wasn't a doorbell, but that doesn't change its affordance as one.

So how do we know it's a doorbell? Simply because we have learned about doorbells and door etiquette and push-buttons from our complex and lengthy socialization and maturation process. We have learned about this class of pushable things by exposure to electrical and electronic devices in our environs and because—years ago—we stood on doorsteps with our parents, learning how to approach another person's abode.

But there is another force at work here, too. If we see a push-button in an unlikely place like the hood of a car, we cannot imagine what its purpose is, but

we can recognize that it is a finger-pushable object. How do we know this? I don't think we know it instinctively, because a small child wouldn't necessarily recognize it as such; certainly not the way she would recognize claws or fangs. I believe that we recognize it as a pushable object because of our tool-manipulating nature. We, as a species (genus, actually), see things that are finger-sized, placed at finger-height, and we automatically push them. We see things that are long and round, and we wrap our fingers around them and grasp them like handles. I think this is what Norman was getting at with his "affordance." For clarity, though, I call this instinctive understanding of how things are manipulated with our hands **manual affordance**. When things are clearly shaped to fit our hands or feet, we recognize that they are directly manipulable and we need no written instructions.

In fact, Norman makes much of how [manual] affordances are much more compelling than written instructions. A typical example he uses is a door that must be pushed open with a metal bar for a handle. The bar is just the right shape, height and position to be grasped by the human hand. The manual affordances of the door scream "pull me." No matter how often someone uses this diabolical door, he will always attempt to pull it open, because the affordances are strong enough to drown out any number of signs affixed to the door saying "PUSH."

There are only a few manual affordances. We pull handle-shaped things with our hands and, if they are small, we pull them with our fingers. We push flat plates with our hands or fingers. If they are on the floor we push them with our feet. We rotate round things, using our fingers for small ones—like dials—and both hands on larger ones, like steering wheels. Such manual affordances are the basis for much of our visual user interface design.

The popular three-dimensional design of systems like Windows 95, NeXT and Motif rely on shading and highlighting to make screen images appear to "pop out." These images offer manual affordances of button-like images that say "push me" to our tool-manipulating natures.

Understanding what it means

What is missing from a manual affordance is any idea of what the thing really does. We can see that it looks like a button, but how do we know what it will accomplish when we press it? For that we begin to rely on text and pictures, but most of all we rely on previous learning. The manual affordance of the scroll-bar clearly shows that it is manipulable, but the only things about it that tell us

what it does are the arrows, which hint at its directionality. In order to know that a scrollbar controls our position in a document, we have to either be taught or learn by ourselves through experimentation.

In the canonical vocabulary (described in Chapter 4), manual affordances have no meaning in the uppermost tier, in idioms. This is why gizmos must have writing on them to make sense. If the answer isn't written directly on the gizmo, we can only learn what it does by one of two methods: experimentation or training. Either we try it to see what happens, or someone who has already tried it tells us. We get no help from our instinct or intuition. We can only rely on the empirical.

Fulfilling the contract

In the real world, a thing does what it can do. A saw can cut wood because it is sharp and flat and has a handle. However, in the digital world, a thing does what it can do because some programmer imbued it with the power to do something. Our tool-using nature can tell us a lot about how a saw works merely by inspection, and it can't easily be fooled by what it sees. On a computer screen, though, we can see a raised, three-dimensional rectangle that clearly wants to be pushed like a button, but this doesn't necessarily mean that it *should* be pushed. We can be fooled because there is no natural connection—as there is in the real world—between what we see on the screen and what lies behind it. In other words, we may not know how to work a saw, and we may even be frustrated by our inability to manipulate it effectively, but we will never be fooled by it. It makes no representations that it doesn't manifestly live up to. On computer screens, canards and false impressions are very easy to create.

When we render a button on the screen, we are making a contract with the user that that button will visually change when we push it: that it will appear to depress when the mouse button is clicked over it. Further, the contract states that the button will perform some reasonable work that is accurately described by its legend. This may sound obvious, but I am constantly astonished by the number of programs I see that offer bait-and-switch visual affordances. This is relatively rare for push-buttons, but extremely common for text fields.