

"Alan Cooper is a software god... This is a landmark book."—Stewart Alsop, Executive Vice President of InfoWorld Publishing

ABOUT FACE

THE ESSENTIALS OF USER INTERFACE DESIGN



ALAN COOPER

"Father of Visual Basic"
Microsoft Windows® Pioneer Award Honoree

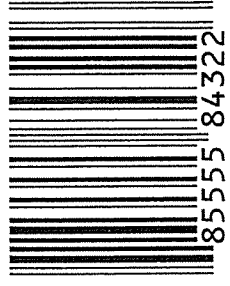
Foreword by Andrew Singer

**COVERS
WINDOWS 95**



0001

ISBN 1-56884-322-4



7

85555 84322 6

Praise for Alan Cooper's *About Face*...

“Alan Cooper is a software god. With Visual Basic, he designed one of the key tools for designing new software. Now he’s sharing his wisdom about how to make that software useable. This is a landmark book.”

—Stewart Alsop, Editor in Chief, *InfoWorld*

“*About Face* defines a new interface design vocabulary that speaks to programmers in their own terms. We have come a long way from the time when there were just modal (bad) and modeless (good) interfaces, and this book reflects that progress.”

—Charles Simonyi, Chief Architect, Microsoft Corp.

“Alan Cooper’s mind harbors a deep, compelling model of software-human interaction, which he presents clearly and applies systematically to real-world design problems in *About Face*. This book is fast-paced, irreverent, and no-nonsense. I would recommend it to any software development executive or designer.”

—John Chisholm, President, Decisive Technology Corp.,
& Columnist, *UNIX Review*

“*About Face* introduces, in common language, many new ideas and pearls of wisdom on how to design software that really is *for* the user. It will help any software designer or programmer understand how to make the user feel good about using the product and at the same time maximize his/her productivity.”

—Mike Maples, EVP, Worldwide Products, Microsoft Corp.

“Alan Cooper popularized the idea of software design as a separate and important discipline. In *About Face*, he passes along both the Big Picture strategy of good design, as well as myriad examples that bring his principles to life. For your sake and the sake of your users, don’t leave the DOS prompt without it.”

—Jesse Berst, Columnist, *PCWeek*, & Editorial Director,
Windows Watcher Newsletter

“*About Face* contains fresh ideas that are a must for the toolset of anyone who is creating an interactive product—from software to interactive Web sites.”

—Dave Carlick, Senior EVP, Poppe-Tyson Advertising

“*About Face* has no fluff: Cooper provides just the information necessary for a software designer to improve their interfaces and programs right now. The anecdotes and examples are excellent, and the axioms make it easy to remember specific issues. If this book doesn’t help people improve their interfaces, nothing will.”

—Larry Marine, Usability Engineer, Intuitive Design Engineering

“Alan Cooper is the ‘Miss Manners’ of software design, translating his deep expertise into practical information instantly useful to developers attempting to tame the Windows interface. My advice is to buy two copies—autograph the second, and send it to an engineer at Microsoft.”

—Paul Saffo, Director, Institute for the Future

“I thoroughly enjoyed Cooper’s writing style. Programmers and designers in all business domains will find this book insightful.”

—Ann Winblad, Software Venture Capitalist

“As a Visual Basic consultant, I find Cooper’s practical design principles and goal-directed approach have helped me improve the quality and usability of my user interface designs and put the best face on my software.”

—Deborah Kurata, Author & Consultant

“*About Face* is a concise and articulate explanation of user-centered design principles. This is the kind of information that takes *years* for user interface professionals to accumulate on their own. This book will surely become a classic.”

—Penny Bauersfeld, Human Interface Design Consultant,
& Author, *Software by Design*

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 5

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

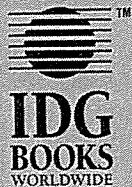
ABOUT IDG BOOKS WORLDWIDE



WINNER
Eighth Annual
Computer Press
Awards 1992



WINNER
Ninth Annual
Computer Press
Awards 1993



Welcome to the world of IDG Books Worldwide.

IDG Books Worldwide, Inc., is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. IDG was founded more than 25 years ago and now employs more than 7,700 people worldwide. IDG publishes more than 250 computer publications in 67 countries (see listing below). More than 70 million people read one or more IDG publications each month.

Launched in 1990, IDG Books Worldwide is today the #1 publisher of best-selling computer books in the United States. We are proud to have received 8 awards from the Computer Press Association in recognition of editorial excellence and three from Computer Currents' First Annual Readers' Choice Awards, and our best-selling ...*For Dummies*[®] series has more than 19 million copies in print with translations in 28 languages. IDG Books Worldwide, through a joint venture with IDG's Hi-Tech Beijing, became the first U.S. publisher to publish a computer book in the People's Republic of China. In record time, IDG Books Worldwide has become the first choice for millions of readers around the world who want to learn how to better manage their businesses.

Our mission is simple: Every one of our books is designed to bring extra value and skill-building instructions to the reader. Our books are written by experts who understand and care about our readers. The knowledge base of our editorial staff comes from years of experience in publishing, education, and journalism — experience which we use to produce books for the '90s. In short, we care about books, so we attract the best people. We devote special attention to details such as audience, interior design, use of icons, and illustrations. And because we use an efficient process of authoring, editing, and desktop publishing our books electronically, we can spend more time ensuring superior content and spend less time on the technicalities of making books.

You can count on our commitment to deliver high-quality books at competitive prices on topics you want to read about. At IDG Books Worldwide, we continue in the IDG tradition of delivering quality for more than 25 years. You'll find no better book on a subject than one from IDG Books Worldwide.

John Kilcullen
President and CEO
IDG Books Worldwide, Inc.

IDG Books Worldwide, Inc., is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. International Data Group publishes over 250 computer publications in 67 countries. Seventy million people read one or more International Data Group publications each month. International Data Group's publications include: ARGENTINA: Computerworld Argentina, GamePro, Infoworld, PC World Argentina; AUSTRALIA: Australian Macworld, Client/Server Journal, Computer Living, Computerworld, Digital News, Network World, PC World, Publishing Essentials, Reseller; AUSTRIA: Computerwelt, PC TEST; BELARUS: PC World Belarus; BELGIUM: Data News; BRAZIL: Anuário de Informática, Computerworld Brazil, Connections, Super Game Power, Macworld, PC World Brazil, Publish Brazil, SUPERGAME; BULGARIA: Computerworld Bulgaria, Networkworld/Bulgaria, PC & MacWorld Bulgaria; CANADA: CIO Canada, ComputerWorld Canada, InfoCanada, Network World Canada, Reseller World; CHILE: Computerworld Chile, GamePro, PC World Chile; COLUMBIA: Computerworld Colombia, GamePro, PC World Colombia; COSTA RICA: PC World Costa Rica/Nicaragua; THE CZECH AND SLOVAK REPUBLICS: Computerworld Czechoslovakia, Elektronika Czechoslovakia, PC World Czechoslovakia; DENMARK: Communications World, Computerworld Denmark, Macworld Denmark, PC World Denmark, PC World Denmark Supplements, TECH World; DOMINICAN REPUBLIC: PC World Republica Dominicana; ECUADOR: PC World Ecuador, GamePro; EGYPT: Computerworld Middle East, PC World Middle East; EL SALVADOR: PC World Centro America; FINLAND: MikroPC, Tietoverkko, Tietoviikko; FRANCE: Distributique, Golden, Info PC, Le Guide du Monde Informatique, Le Monde Informatique, Reseaux & Telecoms; GERMANY: Computer Business, Computerwoche, Computerwoche Extra, Computerwoche Focus, Electronic Entertainment, GamePro, I/M Information Management, Macwelt, PC Welt; GREECE: GamePro, Macworld & Publish; GUATEMALA: PC World Centro America; HONDURAS: PC World Centro America; HONG KONG: Computerworld Hong Kong, PCWorld Hong Kong, Publish in Asia; HUNGARY: ABCD CD-ROM, Computerworld Szamitastechnika, PC & Mac World Hungary, PC-X Magazine; INDIA: Computerworld India, PC World India, Publish in Asia; INDONESIA: InfoKomputer PC World, Komputek Computerworld, Publish in Asia; IRELAND: ComputerScope, PC Live!; ISRAEL: PC World 32 BIT, People & Computers; ITALY: Computerworld Italia, Computerworld Italia Special Editions, Lotus Italia, Macworld Italia, Networking Italia, PC Shopping, PC World Italia, PC World/Walt Disney; JAPAN: Macworld Japan, Nikkei Personal Computing, SunWorld Japan, Windows World Japan; KENYA: East African Computer News; KOREA: Hi-Tech Information/Computerworld, Macworld Korea, PC World Korea; MACEDONIA: PC World Macedonia; MALAYSIA: Computerworld Malaysia, PC World Malaysia, Publish in Asia; MEXICO: Computerworld Mexico, GamePro, Macworld, PC World Mexico; MYANMAR: PC World Myanmar; NETHERLANDS: Computable, Computer! Totaal, LAN Magazine, Macworld, Net Magazine; NEW ZEALAND: Computer Buyer, Computerworld New Zealand, MTB, Network World, PC World New Zealand; NICARAGUA: PC World Costa Rica/Nicaragua; NIGERIA: PC World Africa; NORWAY: Computerworld Norge, Computerworld Privat, CW Rapport Klient/Tjener, CW Rapport Nettverk & Telecom, CW Rapport Offentlig Sektor, IDG's KURSGUIDE, Macworld Norge, Multimedia World, PC World Ekspres, PC World Nettverk, PC World Norge, PC World's Produktguide, Windows Spesial; PAKISTAN: Computerworld Pakistan, PC World Pakistan; PANAMA: GamePro, PC World Panama; PARAGUAY: PC World Paraguay; P. R. OF CHINA: China Computerworld, China Infoworld, Computer & Communication, Electronic Product World, Electronics Today, Game Camp, PC World China, Popular Computer Week, Software World, Telecom Product World; PERU: Computerworld Peru, GamePro, PC World Profesional Peru, PC World Peru; POLAND: Computerworld Poland, Computerworld Special Report, Macworld, Network, PC World Komputer; PHILIPPINES: Computerworld Philippines, PC Digest, Publish in Asia; PORTUGAL: Cerebro/PC World, Correio Informático/Computerworld, Mac•In/PC•In Portugal; PUERTO RICO: PC World Puerto Rico; ROMANIA: Computerworld Romania, PC World Romania, Telecom Romania; RUSSIA: Computerworld Rossiya, Network World Russia, PC World Russia; SINGAPORE: Computerworld Singapore, PC World Singapore, Publish in Asia; SLOVENIA: MONITOR; SOUTH AFRICA: Computing S.A., Network World S.A., Software World; SPAIN: Computerworld España, COMUNICACIONES WORLD, Dealer World, Macworld España, PC World España; SWEDEN: CAP&Design, Computer Sweden, Corporate Computing, MacWorld, Maxi Data, MikroDatorm, Nätverk & Kommunikation, PC/Arkiv, PC World, Windows World; SWITZERLAND: Computerworld Schweiz, Macworld Schweiz, PCtip; TAIWAN: Computerworld Taiwan, Macworld Taiwan, PC World Taiwan, Publish Taiwan, Windows World; THAILAND: Thai Computerworld, Publish in Asia; TURKEY: Computerworld Monitor, MACWORLD Turkiye, PC WORLD Turkiye; UKRAINE: Computerworld Kiev, Computers & Software Magazine, PC World Ukraine; UNITED KINGDOM: Acorn User, Amiga Action, Amiga Computing, Amiga, Appletalk, CD Powerplay, CD-ROM Now, Computing, Connexion, GamePro, Lotus Magazine, Macation, Macworld, Open Computing, Parents and Computers, PC Home, PC Works, The WEB; UNITED STATES: Cable in the Classroom, CD Review, CIO Magazine, Computerworld, Computerworld Client/Server Journal, Digital Video Magazine, DOS World, Electronic, InfoWorld, I-Way, Macworld, Maximize, MULTIMEDIA WORLD, Network World, PC World, PUBLISH, SWATPro Magazine, Video Event, WebMaster; URUGUAY: PC World Uruguay; VENEZUELA: Computerworld Venezuela, GamePro, PC World Venezuela; and VIETNAM: PC World Vietnam. 10/17/95b

For More Information

For general information on IDG Books Worldwide's books in the U.S., please call our Consumer Customer Service department at 800-762-2974. For reseller information, including discounts and premium sales, please call our Reseller Customer Service department at 800-434-3422.

For information on where to purchase IDG Books Worldwide's books outside the U.S., contact IDG Books Worldwide at 415-655-3021 or fax 415-655-3295.

For information on translations, contact Marc Jeffrey Mikulich, Director, Foreign & Subsidiary Rights, at IDG Books Worldwide, 415-655-3018 or fax 415-655-3295.

For sales inquiries and special prices for bulk quantities, write to the address above or call IDG Books Worldwide at 415-655-3200.

For information on using IDG Books Worldwide's books in the classroom, or ordering examination copies, contact the Education Office at 800-434-2086 or fax 817-251-8174.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 508-750-4470.

About Face: The Essentials of User Interface Design is distributed in Canada by Macmillan of Canada, a Division of Canada Publishing Corporation; by Computer and Technical Books in Miami, Florida, for South America and the Caribbean; by Longman Singapore in Singapore, Malaysia, Thailand, and Korea; by Toppan Co. Ltd. in Japan; by Asia Computerworld in Hong Kong; by Woodslane Pty. Ltd. in Australia and New Zealand; and by Transword Publishers Ltd. in the U.K. and Europe.

About the Author

Alan Cooper, the “Father of Visual Basic,” is an award-winning user interface consultant and software designer. His company, Cooper Software, Inc, has worked with a broad range of clients to improve their products and help them create exciting and successful new software products. His experiences in implementing his unique approach to creating better software through goal-directed design led him to write this book.

Since 1976, Alan Cooper has designed and developed software, including *SuperProject* (Computer Associates), *MicroPhone II for Windows* (Software Ventures), and the visual programming interface for *Visual Basic* (Microsoft). In 1976, he founded Structured Systems Group, which Freiburger and Swaine, in their book *Fire in the Valley*, credited with producing “perhaps the first serious business software for a microcomputer.”

Bill Gates presented Cooper with a *Windows Pioneer* award at the Windows World conference in 1994. This rare and coveted award recognized how Cooper’s part in the invention of Visual Basic contributed to the success of Microsoft Windows.

Alan Cooper is a director of both the Association for Software Design and the Software Entrepreneur’s Forum. He founded SEF’s Windows SIG, the largest Windows developer group in the world. He is also a frequent, opinionated and engaging industry speaker and writer on the topics of user interface and conceptual software design.

Credits

**Senior Vice President
and Group Publisher**

Brenda McLaughlin

Publishing Director

John Osborn

Senior Acquisitions Manager

Amorette Pedersen

Managing Editor

Kim Field

Editorial Director

Anne Marie Walker

Editorial Assistant

Dan Hilldale

Production Director

Beth Jenkins

Production Assistant

Jacalyn L. Pennywell

**Supervisor of
Project Coordination**

Cindy L. Phipps

Supervisor of Page Layout

Kathie S. Schnorr

Supervisor of Graphics and Design

Shelley Lea

Reprint Coordination

Tony Augsburger

Theresa Sánchez-Baker

Todd Klemme

Blueline Coordinator

Patricia R. Reynolds

Project Editor

Elizabeth Rogalin

Manuscript Editor

Karen Goeller

Technical Reviewer

Neil J. Rubenking

Graphics Coordination

Gina Scott

Angela F. Hunckler

Media/Archive Coordination

Leslie Popplewell

Melissa Stauffer

Jason Marcuson

Production Page Layout

Benchmark Productions, Inc.

Elizabeth Cárdenas-Nelson

Proofreaders

Dwight Ramsey

Carl Saff

Indexer

Liz Cunningham

Book & Cover Design

Donald Maurer, Benchmark

Productions, Inc.

TonBo Design

*To Sue,
for your love and patience
while I was submerged*

Acknowledgments

Those who have tackled big writing projects know that there are few other tasks that require such a single-minded, non-stop outpouring of effort. Although this is my first book, I've written big software programs before, so I am well-acquainted with the immense demands a project of this scope makes. My friend Gary Kratkin says a big solo writing project is like having a hungry and bad-tempered monster chained up in your basement: You can go out and have fun, but eventually you must return home and feed the hungry beast. There are many people who have helped me feed this beast over the past year who deserve my sincere thanks for their patience, their contributions, or both.

Without a doubt, the people who sacrificed the most have been my family. My lovely wife (and business partner), Sue, has supported me and reassured me and read all of my drafts throughout the monster-feeding process. Thank you for lighting up my life. My two sons, Scott and Marty, missed many nights and weekends with me when I was locked in my office writing instead of playing with them. Thank you, and I love you both beyond measure.

Three of my colleagues at Cooper Software made material contributions to the quality and content of this book. Wayne Greenwood, a talented software designer, carefully read all of the chapters and made many invaluable contributions to the manuscript. In many cases, he was the first person to vet my terms and theories. He also helped with most of the illustrations. Geetha Reddy, another skilled interface designer, read many of the drafts and politely pointed out my successes and failures. Alice Blair's comments were also very useful in straightening out some dodgy prose.

Several people read an early draft of the book and provided worthwhile comments and guidance that had a major effect on the eventual shape of the manuscript. I would like to deeply thank Deborah Kurata (good luck on your own book), Mike Nelson (your moderating voice culled some too-hot flames), Diana Nelson (your insights were valuable throughout) and Frank Cohen (for your unique viewpoint).

Several other people read chapters, sent email, contributed ideas or generally helped to shovel monster food. Thank you Carl Quinn, Andrew McCarthy, Geoff Faraghan, Peter Rosberg, Janell Bandy, Liz Cunningham, Nanci Kavanagh, Andrew Singer, Mike Geary, Fran Finnegan, John Zicker, Steven List, Cynthia Lewis, Geoff Nicholls, Jeff Prorise, David Rygmyr, Paul Yao, Jim Fawcette, Gregg Irwin, Ted Young, Constance J. Petersen, Rowan Hutchinson, Harmon Rogers, Dan Barclay, J. D. Evans, Jr., Joe McGinn, Cam Marshall, Mark Pruett, Dick Grier, David K. Headley, and my best friend David Carlick for the “March of Paradigms.”

At Programmers Press, several individuals made enormous contributions to the quality of the book. Both Chris Williams and Trudy Neuhaus were the first to see the potential of this book. Anne Marie Walker stepped into this project at the eleventh hour and injected a much-needed dose of enthusiasm and energy. Amy Pedersen offered consistent support with the care and feeding of captive monsters. I owe a huge debt of thanks to my skilled editor Karen Goeller and my technical editor Neil J. Rubenking. Their comments and queries contributed materially to the final quality of the book. They both kept me from putting my foot in my mouth many times. Any mistakes that slipped by them are my responsibility. I’d also like to thank Bill Gladstone and Matt Wagner at Waterside, and John Kilcullen at IDG Books for helping to pull *About Face* out of the ordinary mass of technical books.

The publisher would like to give special thanks to Patrick McGovern, without whom this book would not have been possible.

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
1 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
1 Bringing mechanical age models into the information age	35
It's worse on a computer	39



Chapter 4: Visual Interface Design41

 Restricting the vocabulary 47

 The Canonical Vocabulary 48

 Designing for users 49

Part II: The Form51

Chapter 5: Idioms and Affordances53

 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-w73

 Unnecessary rooms 73

 Necessary rooms 75

 Windows pollution 77

Chapter 8: Lord of the Files81

 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 Systems101

 Storing versus finding 101

 It ain't document-centric 107

Chapter 10: Choosing Platforms113

 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 Behavior125

Chapter 11: Orchestration and Flow127

 Planing on the step 127

Where were you on the night of the sixteenth?144
Sensible interaction146
Chapter 12: Posture and State151
Posture151
Windows states163
MDI168
Chapter 13: Overhead and Idiocy171
Overhead171
Idiocy178
Chapter 14: The Secret Weapon of Interface Design183
Get a memory183
Task coherence186
A new way of thinking190
Part IV: The Interaction193
Chapter 15: Elephants, Mice and Minnies195
Why we use a mouse instead of a pen195
Indirect manipulation196
Mice are not here to stay197
Mousing around198
The left mouse button200
Right mouse button201
Middle mouse button202
Things you can do with a mouse202
Up and down events207
The cursor208
Focus212
Meta-keys214
Chapter 16: Selection217
Object-verb217
Concrete and discrete data219
Insertion and replacement220
Additive selection222
Group selection223
Visual indication of selection224
Chapter 17: Direct Manipulation229

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 button	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

Foreword

1876 saw the construction of many bridges, and the completion of the Brooklyn Bridge. One out of every four of those new bridges, however, failed. It is hard for us now to imagine how the outcome of so basic a construction project could be so unpredictable. It would be an extraordinary event for a bridge built today to fail. But every aspect of our understanding of the world begins with ignorance and uncertainty.

For nearly a half a century, a new field of construction, that of information technology, has been emerging. Using the most insubstantial materials, electromagnetic fields and electrons, and software — the abstract description of pure processes — we can build structures for our minds to inhabit and create fabulous tools that extend our mental reach. But this field is still very much in its infancy, and in our ignorance, many of the things we have built thus far fail.

A bridge that is too narrow for the traffic it must bear will be useless, regardless of how structurally sound it is. Likewise, our inability to clearly understand and express the purpose of a particular tool or structure, or to shape something that fits the mind that must use it, can make even the most elaborate construction efforts worthless.

Although there have already been significant efforts to understand and improve the structural integrity, the engineering, of software, thus far only modest attention has been paid to improving its *design*, the process whereby it is given form.

This book represents one of the first attempts to address this problem. As such, it constitutes an important contribution to the nascent literature on software design, especially as it is expressed in a way that is useful to the practicing designer rather than the theoretician.

You may not agree with everything presented in this book, but thoughtful software designers will undoubtedly find the issues raised to be relevant and stimulating. Unlike a number of books from the human-computer-interaction (HCI) community, it addresses issues like functionality that go beyond mere interface design.

In all likelihood, Alan Cooper will always be known principally for his role in the development of Visual Basic, but I think this book may be his greater contribution to our field. For now, it stands virtually alone on the software design bookshelf.

Andrew Singer
June, 1995

Biography of Andrew Singer

Andrew Singer is best known for his work on programming environments and work-group tools at Think Technologies, a company he co-founded in 1982, and whose product development efforts he led until its acquisition by Symantec in 1987.

He chairs the board of the Association for Software Design, a non-profit professional society he organized in 1992 with Mitchell Kapor.

Interval Research Corporation
1801 Page Mill Road
Palo Alto, CA 94304
<singer@interval.com>

Introduction

FOCUS FOR DESIGNING INTERFACES

This book is intended to provide you with effective and practical tools for designing user interfaces. These tools come in two distinct varieties: tactical and strategic. Tactical tools are hints and tips about using and creating user interface idioms, like dialog boxes and push buttons. Strategic tools are ways to think about user interface idioms—in other words, the ways in which the user and the idiom interact.

Although books are available that deal with either strategic or tactical tools, my goal has been to create a book that weaves the two together. I want to give you a cornucopia of insights about user interface design as a whole. While helping you design more attractive and effective dialog boxes, this book will simultaneously help you understand how the user comprehends and interacts with your software.

I believe that integrating the tactical and the strategic approaches is the key to designing effective software interfaces. For example, there is no such thing as an objectively good dialog box—the quality depends on the situation: who the user is and what his background and goals are.

Merely applying a set of tactical dictums will make user interface creation easier, but it won't make the end result better. Just thinking beautiful thoughts about how users "should" interact with your system won't improve the software, either. What will work is maintaining a strategic sensitivity for how users interact with specific software. This will enable you to correctly choose the appropriate tactics to apply in a particular situation.

The first three parts of this book stress strategy, but you'll find tactics interwoven throughout.

There are two steps to user interface design: the synthesis of a solution, and the testing of the validity of that solution. The latter is a discipline widely known as usability, while the former is referred to simply as user interface design. There is a significant and growing body of *usability* literature, but there is very little in print about user interface design synthesis—the invention of user interfaces from direct analysis of the tasks, the technology and the user's goals. Accordingly, I will focus exclusively on the design of user interface solutions and ignore the processes of testing those solutions. However, this is not a slur on usability: You will always achieve the best results by combining the two disciplines in a harmonious relationship.

1. USER
INTERFACE
DESIGN
2. USABILITY

Who should read this book

I wish I could say this book is for user interface designers and let it go at that. Most user interfaces are still designed by programmers, an increasing number of whom are growing uneasy as they glimpse the gulf between the skill set needed for software construction and the skill set needed for software design. Documentation writers, trainers and technical support people increasingly share this same worry. It is for this growing community of design-aware developers that this book is written.

Eighty years ago, the automobile industry came to understand that a well-engineered car is less appealing and less successful than a car that is both well-designed and well-engineered. Until the software industry comes to the same conclusion, the burden of quality design will fall largely on conscientious software engineers.

To the industry's credit, a small but growing cadre of software and user interface designers is beginning to make its presence felt. It is finally possible for software developers to hire people trained in the art of software design, both in the cauldron of industry and in forward-thinking universities. Eventually, we

will see a bifurcation in the industry: Designers will design the software and engineers will build it. This is currently considered a luxury by those development shops that haven't realized the fiscal and marketing advantages that come with professional software design.

Why I wrote this book

Since 1976 I have been creating successful software for personal computers. In the early days of the industry, I invented, designed, coded, documented, marketed, sold, supported and revised retail products including accounting, word processing, spreadsheet, project management and visual programming languages. During the 1980s, I was an independent software author—much like a freelance inventor. I identified problems and created innovative software solutions for them. Then, working alone or with a small team, I completed them for sale to a software publisher who brought them to market. For all these years, I designed my software without reflecting much on the process. More recently, I have offered my services as a software design consultant, helping other companies to design new products and improve their existing ones.

When I became a consultant, I discovered that I had to articulate to my clients the reasons *why* a certain design solution was better. I knew the answer, but I had no words with which to say it. In response to my own needs, I began to formulate the axioms, ideas and terms that are in this book. Many of my clients and people I have spoken with have requested that I record my thinking in a book; *About Face* is the result.

My twenty years of software design and development have taught me that the task of user interface design is fundamentally different from software engineering. Most of the writing available on user interface design approaches it from an engineering or a user-testing point of view. There is little on the shelves that addresses the creation of user interface design directly from the statement of the problem. The tools of the engineer are excellent ones, but not for interface design, which isn't an engineering problem and can't be well-defined in those terms.

This book is based on my personal experiences, not on studies of published works in the area of human-computer interaction, usability, cognitive psychology or ergonomics. All of the opinions, terms, axioms, tips and conclusions contained in this book derive from my own observations. Where I have knowingly adopted the thinking of others, I have said so in the text. Where my

thinking may seem to echo the work of others without credit, it is because we have independently arrived at similar views and I am ignorant of their work. It does not represent a desire to appropriate their vision or to negate their efforts.

A taxonomy of software design

Webster defines taxonomy like this:

Tax·on·o·my \tak-'sän-e-mee\ *n* 1 : the study of the general principles of scientific classification: CLASSIFICATION; *specif* : orderly classification of plants and animals according to their presumed natural relationships

Biologists, anthropologists and natural scientists of all stripes use a taxonomy as their primary tool, both for their ease in communicating concepts and as a mental model of the purposes and relationships of things in the real world. Although a taxonomy is a more formal dialect within a broader language, all language is taxonomic. Our perceptions of the way the world works are colored and influenced by the structure and usage of our language.

Physical scientists spend extravagant amounts of time learning the terms specific to their discipline. These terms not only illuminate the specific process or object at hand, but they influence how we think about them in relationship to life. Doctors must learn the names of every bone, muscle, nerve and organ in the human body as well as terms that indicate their direction, orientation and condition, in health, trauma or illness. How else could one doctor express to another a question, a concern or a discovery? A thorough taxonomy is the cornerstone of each science, from the study of spiders to the behavior of printing presses.

EACH SCIENTIFIC INDUSTRY HAS A VOCAB *HAS*
 The computer industry is no exception. We have a rich and complex language to describe the nuances of the field—words like “concurrency,” “recursion,” “hexadecimal” and “raster scan.” But the completeness and effectiveness of this programming terminology is really just a sham. The language of programming is too new and evolving too fast to yet have a firm foundation. While the natural taxonomy of plants and animals was developed over hundreds of years, the computer taxonomy has grown—out of control—for less than fifty. There is certainly a small core of commonly agreed-upon terms (like RAM and ROM), but there is an ocean of words that either have no meaning—like “virtual reality,” “bug-free,” and “artificial intelligence”—or that have meanings so bowdlerized, so bastardized, as to be useful only for resumes and bull-sessions

around the water cooler—words like “standard,” “object-oriented,” “macro,” and “client/server.”

I’ve heard and read countless discussions about the relative “efficiency” or “elegance” of some software artifact. But when someone speaks of an “efficient” user interface, is he referring to the code? to the gizmo-count? to the ease of programming? ease-of-learning? ease-of-use? Certainly, these are real words with real meanings that conjure up useful imagery in the minds of intelligent, technical people. But are these terms well-enough defined to base million-dollar decisions on? What would you think if your doctor said something like “Well, it seems you’ve got a swollen thingy on the front part of your arm. We’ll have to cut it off.” Cut *what* off?! The *swollen thingy or my arm*? Get that quack away from me!

All of this brings us to user interface design. Our discipline is less than half the age of the computer science field. Little of our work has been tested in the modern crucible of personal computing in which, for the first time, the majority of computer-human interaction is with non-computer-professionals.

The terms we have to work with are so weak and ill-defined that they make the computer science taxonomy seem robust by comparison. In user interface design we are dealing with so many new concepts—concepts that have no parallel in the non-digital world—that there are no terms to borrow from. We find ourselves performing functions daily that we could never imagine before we had personal computers.

The lack of consistent, specific terminology in the world of software design frustrates interface designers enormously. Without precise terminology, we are forced to speak in vague generalities and hand-waving. Without clearly differentiated terms, we accidentally group things in the wrong places, overlook significant facts and inadvertently mistake the bad for the good.

Language defines our perceptions. The words we use influence our mental picture of the world around us. To design effective software for the information age, we must have a vocabulary that accurately describes the goals we seek, the tools we use to achieve them, and the side effects of our journey. Software design will not become a real science or art or craft until we create our own taxonomy. It will not become a *successful* practice until we develop accurate ways of thinking and talking about what we do; until we develop a taxonomy.

To **neologize** means to invent words, and many of today's computer practitioners are reluctant to neologize. They imagine that having more words complicates things and makes communications more difficult. When speaking of familiar things in the familiar world, this is true, but in the mostly new world of computer-human interaction, old, ambiguous or inaccurate words hurt us more than they help. Our mental images also color our thinking.

IF COMMON
WORD CONFUSION
READER DEVELOPS
NEW WORDS

Any discipline that wants to be practiced seriously and effectively must develop a powerful, descriptive and discriminative language. User interface design is a prime example of this imperative. Not only can we not function effectively, but our credibility to the outside world, particularly to the world of software engineering, is threatened unless we can agree on terms to describe what we do, what we care about and how to judge our relative success at achieving our goals. In this spirit, I try to continuously fill the vacuum with neologisms—words that I have created to describe common ideas, things, principles, actions or conditions that relate to our practice.

Conventions used in this book

The platform

This book is about user interface design using Microsoft Windows. The majority of today's PCs run Windows and, as a result, that is where the greatest need exists for an understanding of how to create effective, goal-directed user interfaces.

Having said that, I believe that most of the material in this book transcends platforms. It is equally applicable to all desktop platforms—Macintosh, Motif, NeXT, OS/2, and others—and the majority of it is relevant even for more divergent platforms such as kiosks, handhelds, embedded systems and more.

As I write this, Microsoft is preparing Windows 95 for release. This is the fifth major release of Windows in its decade of life, and it promises to ratchet the industry forward another much-needed notch. Because of the newness of Windows 95, some of my examples come from the older Windows 3.x. However, the principles of user interface design transcend the artifacts of any particular release or platform.

The examples

I use several programs as examples. Mostly, I've used the Microsoft Office suite of Word, Excel and PowerPoint; a little Adobe Illustrator and some

CompuServe Navigator, because these are the programs I use most. I have tried to stay with mainstream programs for most examples for two reasons. First, most readers will likely be at least slightly familiar with the examples. Second, it's important to show that the user interface design of even the most finely honed products can be significantly improved with a goal-directed approach.

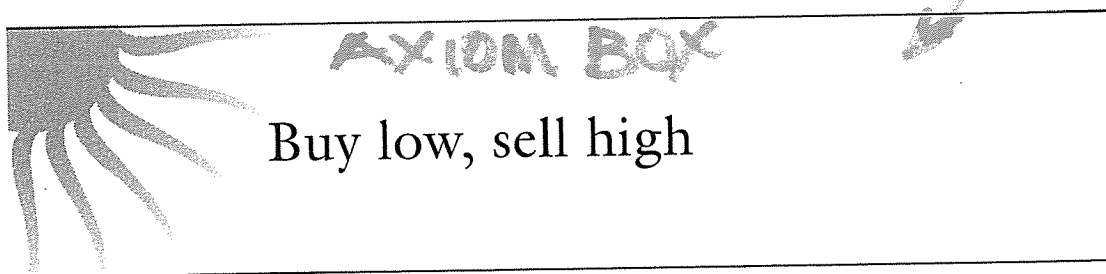
Pronouns

It is my sincere desire to be non-sexist in my writing. I have wrestled with clumsy constructions like “s/he,” “she/he” and “his or her” which seem like inkspots or thumbprints on the page to me—I have, therefore, abandoned them. I also tried the dreaded genderless plurals, “they,” “their” and “them.” In this case, the cure seemed worse than the disease to me.

The solution I finally adopted was to use feminine pronouns exclusively, and that was how the manuscript was originally drafted. Many focus groups and reviews later, my editor and publisher—both female—insisted that I return to the masculine form so as to avoid offending my readers. I want to change the world of user interface design, not rattle the world of book publishing, so I reluctantly agreed. I apologize for the male pronouns, and sincerely hope that you will read them merely as placeholders for intelligent and capable people of either sex.

Special notations

As I study software design, I find it powerful and effective to encapsulate my discoveries as axioms. These brief aphorisms encapsulate a great deal of wisdom and are easy to remember. In this book, axioms are general principles of software design and user interface design. Each one represents a guiding principle that is always true. Pose these axioms to yourself as design tests when you find yourself stuck on tough problems. All of the axioms have been highlighted in the text as shown here.



Some **aphorisms** aren't as general as axioms, but they're just as useful in their specific area. When you are working with a particular design element, the design tips from that area can help to unstick your creative mind. A complete listing of all axioms and design tips can be found in the Reference Section at the back of the book.

***Design Tip:** Keep your powder dry*

When mentioned for the first time, terms with specific meanings for the user interface design practitioner are highlighted in the text in **boldface**. Most of these terms are my own neologisms, but many of them were coined by others or are in common use. If the term is one of my own, I will introduce it by saying "I call this...." All of the design terms are mentioned in the index, with the page number where they are first mentioned indicated.

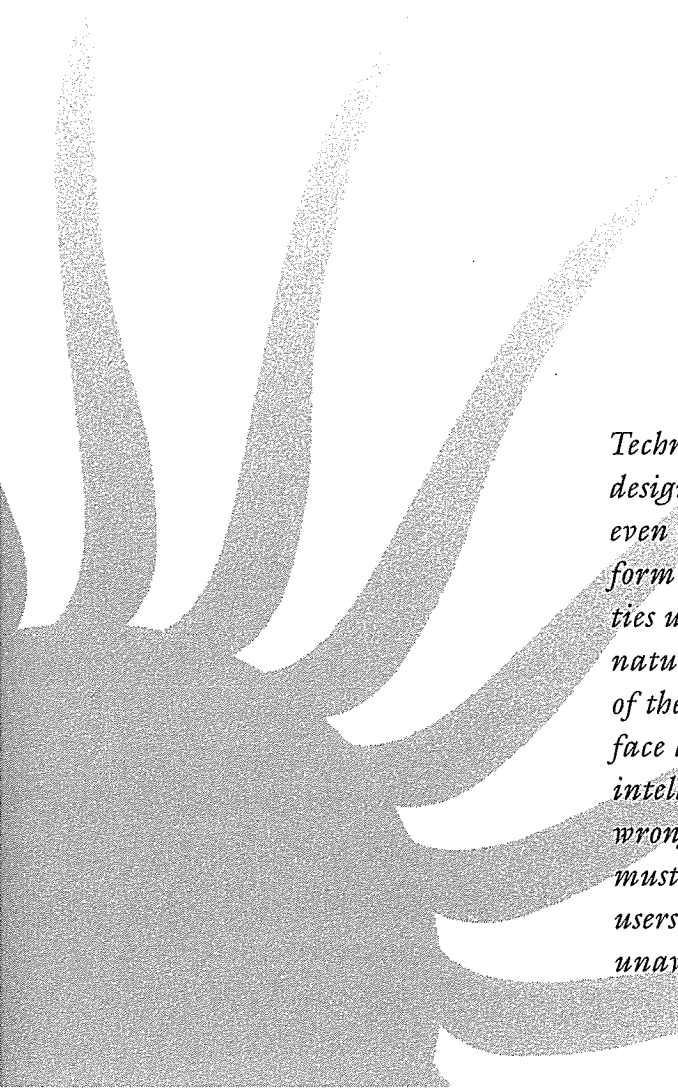
Let's design

I hope this book informs you and intrigues you, but most of all, I hope it makes you think about software design in new ways. The practice of user interface design is not only constantly changing, it is also big and varied enough to seem different to disparate observers. If you have a different opinion or just want to discuss things with me, I'd like to hear from you at alan@cooper.com.

That's enough preliminary stuff; let's design.

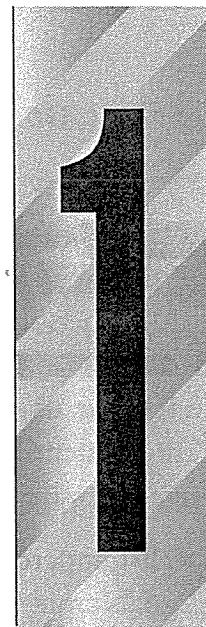
Part I: The Goal

Designing for Users



Technology is the engine that drives user interface design. This synergy is a two-edged sword, because even as the power of the technology frees us to perform great feats of invention, it simultaneously ties us to ways of thinking that are contrary to the natural direction of human behavior. Almost all of the problems with modern software user interface design originate from well-intentioned, intelligent and capable people focusing on the wrong things. Instead of technology and tasks, we must focus our gaze on the goals toward which users strive, even if they themselves are sometimes unaware of them.

Goal-Directed Design



This book has a simple premise: If achieving the user's goals is the basis of our user interface design, then the user will be satisfied and happy. If the user is happy, he will gladly pay us money, and then we will be successful.

Most software isn't designed. Rather, it *emerges* from the development team like a zombie emerging from a bubbling vat of Research and Development juice. When a discipline is hugging the ragged edge of technology, this might be expected, but much of today's software is comprised of mostly "D" and very little "R."

The little software that is consciously designed is usually designed from the point of view of the programmer, sometimes the marketing department, and occasionally from the user's point of view. None of these points of view reflect the user's *goals*. The programmer has a different set of imperatives, typically centering on technology and programming methodology. The marketing department is likely focused on what seems to create the loudest hubbub in the industry. And users tend to focus on their everyday tasks—contrary to what you might suspect, few users are consciously aware of their goals.

KEY } Software that merely enables users to perform their *tasks* will rarely be successful. If the task is to enter 5,000 names and addresses into a database, a smoothly functioning data-entry program won't satisfy the user nearly as much as an automated system that extracts the names from the invoicing system. While it is the user's job to focus on tasks, the designer's job is to look beyond the task to identify the user's goals. Therein lies the key to creating the most effective software solutions.

The well-tempered software designer must be sensitive to and aware of the users' goals amid the pressures and chaos of the software development process. This isn't as hard as you might think, as long as you know how, but it certainly isn't formulaic.

Keep in mind the old saying: "If you give a man a fish, you feed him for a day. If you teach him how to fish, you feed him for life." In this book, I'm going to teach you how to fish in these waters.

We will talk a lot about the techniques and tools of interaction, but no matter how far we stray we will always return to the user's goals. They are the bedrock upon which all good design is founded.

So, what are the user's goals? How can we identify them? How do we know that they are real? Are they the same for all users? Do they change over time?

The user's goals

The user's goals are often very different from what we might guess them to be. For example, we might think that an accounting clerk's goal is to process invoices efficiently. This is probably not true! Efficient invoice processing is more likely the goal of the company or the clerk's boss. The clerk is more likely concentrating on goals like

- Not looking stupid
- Not making any big mistakes
- Getting an adequate amount of work done
- Having fun (or at least not being too bored)

If you think about it, those are pretty common goals. Regardless of the work we do, and the tasks we must accomplish, most of us share those very basic,

simple goals. Even if you have much higher aspirations, they are still more personal than work-related:

- Be the best at what I do
- Get onto the fast track and win that big promotion
- Learn all there is to know about this field
- Be a paragon of ethics, modesty and trust

Even for the sociopath, goals don't diverge from the focus on the individual:

- Learn the boss's password
- Embezzle a million dollars
- Hide my perfidy
- Discover the original recipe for Coca-Cola

However, many of the books on user interface that I've read assume that the user's goals have something to do with the program's business purpose. Software designed to achieve purely business goals will fail, but if it is designed with the personal goals of the user in mind, it will also achieve its business goals. Of course, the program must satisfy the business problem at hand, but the people who use it cannot and will not behave like invoices, database records or modules of code.

If you examine most commercially available software today, you will find user interfaces that are particularly adept at several things:

- Making the user look stupid
- Causing the user to make big mistakes
- Slowing the user so he doesn't get an adequate amount of work done
- Preventing fun and boring the user

Most of that same software is equally bad at achieving its business purposes. Invoices don't get processed all that well. Customers don't get serviced on time. Decisions don't get properly supported. I see a connection here.

This is a sad and preventable situation, but not a surprising one, because the authors of these packages are focusing on the wrong things. Most of us pay far too much attention to the technology used to implement computer solutions, which distracts us from the user. When we do focus on the user, we pay too much attention to the tasks that users engage in and not enough attention to their goals. Software can be technologically superb and perform each business task with diligence, yet still be a critical and commercial failure. To create successful, effective software, we must see that it achieves the user's goals. We can't ignore technology or tasks, but these elements are like salt on a meal: they make it palatable, but they don't nourish all by themselves.

Let me give you some examples of the results of focusing on technology and tasks instead of on the user and his goals.

Software that is rude

F.U. Software is often rude to the user. It blames the user for making mistakes that are not the user's fault, or should not be. Error message boxes like the one in Figure 1-1 pop up like weeds announcing that the user has performed yet another dunderheaded stunt. The messages then all demand that the user acknowledge their failure by saying "OK."

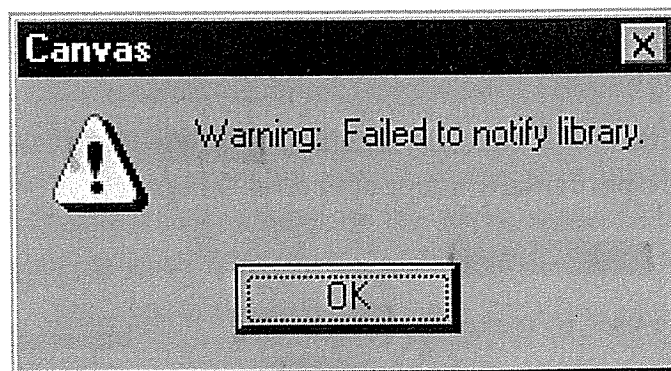


Figure 1-1

Thank you so much for sharing that pithy observation with us. Why didn't you notify the library? What did you want to notify it about? Why are you telling me? What do I care? Maybe you'd like to comment on what I'm wearing, too? And besides, what am I "OK"ing? It is *not* OK with me that this failure occurred!

Software too frequently assumes that its user is computer-literate. For example, when a user is finished editing a document, he closes it, and the program asks

if he wants to save it. The technology behind this issue is not trivial—it has to do with the ability of the CPU to directly address information stored on rotating memory—but how is the novice user to know that?

Software frequently interrogates the user, peppering him with a string of terse questions that he is neither inclined nor prepared to answer: “Where did you hide that file?” “What interrupt request line is free?”

It is difficult for the user to concentrate on the task at hand when bombarded by tangential interruptions that require acknowledgment, by managing dialog boxes that forget what he did just moments ago or by forcing him to go through unnecessary steps.

Patronizing questions like “Are you sure?” and “Did you really want to delete that file or did you have some other reason for pressing the delete key?” are equally irritating.

Software that is obscure

Software is frequently obscure, hiding meaning, intentions and actions from the user. Programs often express themselves in incomprehensible jargon that cannot be fathomed by normal users (“How many stop bits?”) and sometimes even by experts (“Please specify IRQ.”).

Features are hidden behind a veil of menus and dialogs and windows. How can the user know that the answer lies in the help system if he can’t find the help system? Even when the user finds the right dialog, he might find it populated with terse abbreviations, obscure commands and inscrutable icons.

To more frequently than you might think, software demands that its users answer tough questions before telling them the effects their answers might have. For example, how can a user possibly decide between a full installation, custom installation and laptop installation if he isn’t told what each of them means in terms of functionality as well as disk space?

Software with inappropriate behavior

If most 10-year-olds behaved like some of these software programs, they’d find themselves grounded for a week. Programs forget to shut doors behind themselves, leave shoes in the middle of the living room floor, and can’t remember what you told them only five minutes earlier. In rapid sequence I save a document, print it, then try to close it, but the program asks me if I want to save it.

first! Evidently the act of printing caused the program to think I had changed it. Sorry, Mom, I didn't hear you.

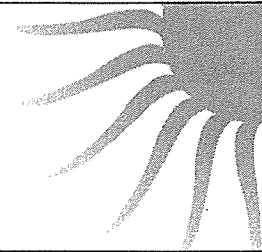
Programs often require us to step out of the main flow of tasks to perform functions that should fall immediately to hand. Dangerous instructions, though, are right up front where they can be accidentally triggered and frighten unsuspecting users.

The overall appearance of many programs is an exercise in window pollution, with popups popping up all over and making navigation difficult.

Another irritant is the "settings" that programs offer for our confirmation without allowing us to change the values we disagree with. We're forced to leave the task at hand and fight our way through thickets of dialog boxes and approvals to get to where we can enter the new values.

Let's go back to our earlier list of user goals. We can reliably say that we will make the user look stupid if we let him make big mistakes, keep him from getting an adequate amount of work done or boring him. Stating this axiomatically:

Don't make the user look stupid



This is probably the most important design guideline. In the course of this book we will examine numerous ways in which existing software makes the user look stupid and explore ways to avoid that trap.

The essence of user interface design

The practice of user interface design is not formulaic. There is no such thing as a "good user interface," just as there is no such thing as a "good furniture arrangement." A furniture arrangement can only be judged within the context of its intended use. An aesthetically pleasing arrangement of sofa, endtable and lamp is bad when placed in a bathroom. By the same token, a toilet and sink in the living room would be somewhat uncomfortable and inappropriate.

The only true test of the quality of a user interface is in context: How will the software be used? Who will use it? How frequently? For how long? How important are considerations of data integrity? Learnability? Portability? The answers to these questions vary widely and are not consistent from application to application. The first task of the software designer hasn't much to do with software: it is seeking and finding answers to these and other, user-centered questions.

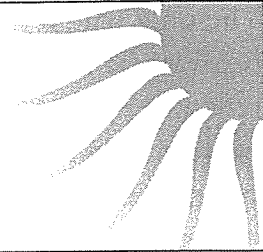
The source for determining whether or not a feature *should* be included in a product shouldn't rest solely on the technological underpinnings of that feature. The driving force behind the decision should not be that "we have the technical capability to do this." The primary factor should always be the goals of the user.

Let me illustrate: One of my clients sells an automated call-distribution system. The people who use their product are paid based on how many calls they handle, not by the hour. Their most important consideration is not ease of learning, but the efficiency with which calls can be routed to the answerer and the rapidity with which they can be completed. Ease of learning is important, as it affects the happiness and ultimately the turnover rate of employees, so both ease and throughput should be accommodated where possible. But there is no doubt that throughput is the dominant demand placed on the system and, if necessary, ease of learning takes the back seat. A program that walks the user through step-by-step will merely frustrate him once he's learned the ropes.

We assume that making things easier is the target. But if all we want is easy, we could stay in bed all day. A user interface designer who proclaims "Ease of learning is the most important consideration" is fooling himself. This statement *may* be true, but it *may* also be false depending on the goals of the user. If you test this phone-call distribution product with a dozen first-time users, you'll find ways to improve the learnability of the product. However, if you test the more-learnable version with a dozen experienced users, you'll find them impatient with the intermediate steps. You can't create good design by following rules disconnected from the goals of the user.

A more concise way to state this is to say *good design makes the user more effective*. This takes into account the universal goal of not looking stupid, along with any particular goals of business throughput or ease of use that are relevant in this situation.

The goal of all software users is to be more effective



It is up to you as a designer to determine how “effectiveness” manifests itself in the circumstances. If the software is a kiosk in a corporate lobby helping visitors find their way around, ease of use for first-time users is clearly the goal. If the software is a threat-detection and monitoring display on board an AWACS radar airplane operated by a highly trained soldier, ease of use for first-timers is a distant second consideration. The design of this is moot if the soldier cannot clearly and easily distinguish a hostile aircraft from a sky crowded with commercial and friendly aircraft. The recent incident in the Mid-East where controllers aboard an AWACS plane directed jet fighters to shoot down two friendly helicopters is evidence that their support software failed them, and that some software designer wasn’t focusing on the user’s goal. Whatever the software on board that plane was, it wasn’t “effective.”

A fresh look at features

Most software that we run on our personal computers today is feature-centric rather than goal-centric. A wildly successful program like Microsoft Word for Windows offers me hundreds of functions (I’m writing this book with it). It offers functions like paragraph formatting, field insertion, page layout view and toolbar configuration. I could easily be considered an expert user of Word, and I know how to use each of these tools in creating the many different documents needed in my writing and my business. But, none of the functions are goal-centered. If I want to write a letter, the program comes with a template for a business letter, but what if I want to write a personal letter to my Aunt Mary Lee? Rather than a canned template, I’d like to see a dialog box like the one in Figure 1-2.

With this dialog box I wouldn’t have to worry about finding the right template, nor would I have to fret over the margins, typeface, clip art and other aspects of the letter. I’d just say what I had to say to Aunt Mary Lee, and the program would take care of the rest. I would tell the program my *goals*, and it would tell each little feature how to behave to achieve them.

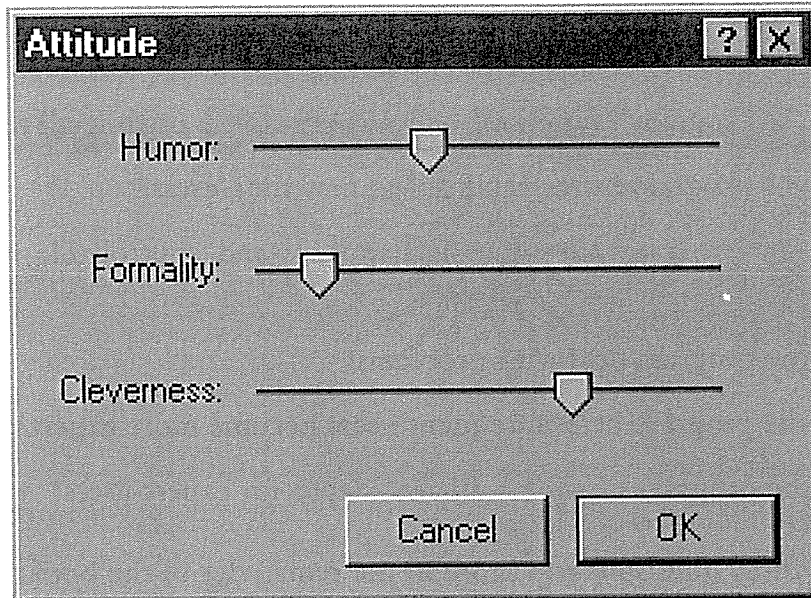


Figure 1-2

This is a “goal-directed” dialog box. It doesn’t give me tools, it gives me answers: I get to select the amount of formality, humor and cleverness in the presentation of the letter I’m typing. It would govern such things as the typeface, its regularity, its style and its arrangement on the page. It would have an effect on the margins, the spacing, the colors and additional visual elements like rules and clip art. Sure, I could control each aspect individually and get the same result, but that’s what programmers like to do, not what users like to do.

This example is purposefully overstated, but regardless of what all of us control-freak-programmer-types think about it, if someone created a letter-writing-specific word processor with dialog boxes like this one, it would be a big success. Goal-directed design is compelling to everyone, even those who aren’t intrigued by technology.

To those who *are* intrigued by the technology, which includes most of us programmer types, we share a strong tendency to think in terms of functions and features. This is only natural, as this is how we build software: function by function. The problem is that this isn’t how users want to use it. Developers are frequently frustrated by this, since it requires us to think in an unfamiliar way, but after the initial strangeness wears off, goal-directed design is a boon—it is a powerful tool for answering the most important questions that crop up during the design phase:

- What should be the form of the program?
- How will the user interact with the program?
- How can the program's functions be most effectively organized?
- How will the program introduce itself to first-time users?
- How can the program put an understandable and controllable face on technology?
- How can the program deal with problems?
- How will the program help infrequent users become more expert?
- How can the program provide sufficient depth for expert users?

We will answer these questions and more in the remainder of the book.

Software Design



When we think about complex mechanical devices, we take for granted that they have been carefully designed and engineered. Software is usually far more complex than most mechanical objects, but is rarely consciously designed.

Software isn't designed

We are just leaving what is called the **mechanical age**, an era in which the objects of value are manufactured.

We are entering an era called the **information age**, in which the objects of value are merely digital representations of things: movies, music, databases and programs.

Most manufactured objects are quite simple, and even the more complex manufactured products are immensely simpler than most digital objects, particularly programs.

A first-release software program might contain 40,000 lines of code, not including the millions of lines of supporting code in the third-party libraries and operating system needed to run it. A typical shrink-wrapped business application program, say Microsoft Excel, running on a typical desktop computer has considerably more than

1,000,000 lines of code, including many thousands of variables, conditions and comparisons—the software equivalent of moving parts. Compare this to a mechanical artifact of almost overwhelming complexity like the Navy’s swing-wing supersonic F-14 fighter jet. That jet probably has about 10,000 parts, and “only” about 1,000 of them might be *moving* parts.

Most artifacts of the mechanical age are designed by professionals. Our cars are designed by trained, professional automobile designers, not by mechanical engineers. Our houses are designed by professionally trained and certified building designers—architects—not by structural engineers. Our toys and clothes and bookcases are designed by toy designers, clothes designers and industrial designers.

The process of determining what software will do and how it will communicate with the user is closely intertwined with its construction. Most software is built like crazy Mrs. Winchester’s house, who thought that she’d die if she ever stopped building. Rooms and stairs and cupboards and walls are added in manic confusion as the need and opportunity presents itself during construction. Programmers, deep in their thoughts of algorithms and coding arcana, design user interfaces the way miners design the landscape with their cavernous pits and enormous tailing piles. The software design process alternates between the accidental and the non-existent.

As we move deeper into the information age, the overwhelming majority of “manufactured” artifacts will be software. Since our future will be dominated by vast amounts of software, the idea that it isn’t consciously and conscientiously designed by trained professionals generates some justifiable unease.

Software creators have been policing their own profession since programming began. Some of us have been warning of the inevitability of the regulation of our industry. Possibly because of the libertarian leanings of many in the programming community, these warnings are widely ignored. Unfortunately, the consumer market won’t tolerate this lack of order for very long.

The Intel Pentium bug scandal of 1995 made headlines and became rich fodder for talk-show monologues. The fact that there were equally serious bugs in the 286, 386 and 486 processors lulled Intel into a false sense of security. The difference was that the Pentium was the first CPU widely advertised on prime-time TV. Intel failed to realize that when you sell directly to consumers, they apply their own standards, which are often enormously different from those of

industry insiders. The family of programmers will forgive minor bugs in a CPU chip because they know how complex it is and understand what the potential impact of the bug will be. The consumer doesn't care. He expects perfection and, in today's litigious, consumer-advocate climate, will get it.

Another story in the news lately tells of parents who purchased CD-ROM-equipped computers for their families, only to learn firsthand of the nightmarish difficulties getting them to work as advertised. Assembling bicycles on Christmas Eve was a cakewalk compared to getting *The Lion King* CD-ROM to work.

This state of affairs cannot continue for long. Either the software industry will regulate itself like doctors and architects do, or the government will regulate it like hairdressers and taxi-drivers. The choice is in the industry's hands.

Conflict of interest

There is a conflict of interest in the world of software development because the people who build it are also the people who design it. If carpenters designed houses, they would certainly be easier or more interesting to build, but not necessarily better to live in. The architect, besides being trained in the art of what works and what doesn't, is an advocate for the client, for the user. An equivalent role in the world of software has not fully developed yet, although several groups are eyeing it jealously.

Many software tools are available to describe software, but almost all of them double as programming tools. There is a real danger in using programming tools as design tools. Programming has a life of its own, and once something has been set into code, even if it's just hack code in a prototyping tool, it tends to exhibit a powerful inertia. Any code, even prototype code, tends to never be thrown away. It's as though the scaffolding is so labor-intensive that the urge to incorporate it into the finished house is irresistible. If designers give coding tools a wide berth—including prototyping tools—they will avoid the conflict of interest between the practices of design and development.

All of the designers at my company work on paper with a pencil. We also use computers, but only word processors and drawing programs. Prototyping is useful for design verification, but we are very wary of mixing the design process with the prototyping process.

The profession of software design

Thankfully, there is a growing awareness of this conflict in the software industry. More and more developers are thinking about design and viewing it as a separate discipline from programming. Many observers of digital technology sense the increasing pervasiveness of software in every aspect of our lives. They are also beginning to see the need for professional software designers, and this trend is very encouraging.

There is some confusion over the correct terminology to refer to those who design software. The term “software architect” is a good one, and it benefits from the fairly accurate analogy with building architects. However, that term has long been appropriated by the software engineers who build system internals. The term “software designer” is the one I and many others have settled on (including the Association of Software Design). Unfortunately, this term also is losing some of its value because it is widely used as a boutique term for senior programmers.

I define **software design** as that portion of the development process that is responsible for determining how the program will achieve the user’s goals. The questions answered by this phase include

1. What the software program will do
2. What it will look like
3. How it will communicate with the user

User interface design is a subset of software design that encompasses items 2 and 3, although it is often difficult to separate them from item 1. This book focuses on user interface design, so it emphasizes interactive visual communications more than application problem solving.

Supporting software design disciplines

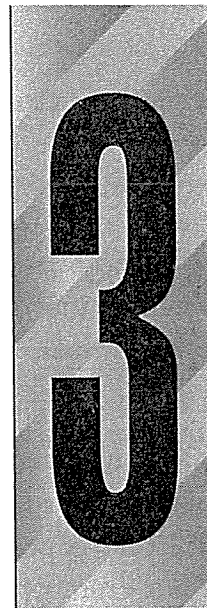
Members of another rapidly growing group call themselves “usability professionals.” These people do not necessarily come from the ranks of programmers. Rather, they specialize in the study of how people interact with software. They primarily conduct interviews and focus groups with users, observe them using software, and then evaluate the quality of user interfaces and make recommendations. Their efforts are a great help in both weeding out bad user interfaces

and in raising the awareness—inside and outside the industry—of the crisis in software design.

Another discipline, called variously “human factors engineering,” “human-computer interaction” or “ergonomics,” researches the behavior of people as they interact with computers and other technological artifacts. It provides significant insight into the nuances of how we relate to our technical devices.

Another growing academic specialty is cognitive psychology, popularized at the University of California (San Diego) by Donald Norman. This discipline looks at how people think and understand the world around them, particularly the technical artifacts they work with.

The Three Models



People in the computer industry frequently toss around the term “computer literacy.” They talk about how some people have it and some don’t; about how those who have it will succeed in the information age and those who lack it will fall between the social and economic cracks of the new age. But computer literacy is nothing more than a euphemism for making the user stretch to reach an information age appliance rather than having the appliance stretch to meet the user.

DONT MAKE A FEW EXPECTS,
MAKE ONLY (OR NO) “

The manifest model

Any given machine has a method for accomplishing its purpose. A motion picture projector, for example, uses a complicated sequence of intricately moving parts to create its illusion. It shines a very bright light through a translucent, miniature image for a fraction of a second. It then blocks out the light for a split second while it moves another miniature image into place. Then it unblocks the light again for another moment. It repeats this process with a new image twenty-four times per second. The actual method of how a device works is what I call its **implementation model**.

From the movie-goer's point of view, it is easy to forget the nuance of sprocket holes and light-interrupters while watching an absorbing drama. The viewer imagines that the projector merely throws onto the big screen a picture that moves. This is what is called the user's **mental model**, or sometimes his **conceptual model**.

People don't need to know all of the details of how some complex process actually works in order to use it, so they create a mental shorthand for explaining it, one that is powerful enough to cover all instances, but that is simple and easy. For example, many people imagine that when they plug their vacuums and blenders into outlets in the wall, electricity travels up to them through little black tubes. This mental model is perfectly adequate for using all household appliances. The fact that the implementation model of household electricity involves nothing actually travelling up the cord or that there is a reversal of electrical potential 120 times per second is irrelevant to the user, although the power company needs to know these details.

In the digital world, however, the differences between a user's mental model and an actual implementation model may be stretched far apart. We ignore the fact that a cellular telephone might swap connections between a dozen different cell antennas in the course of a two-minute phone call. Knowing this doesn't help us to understand how to work our car phones. This is particularly true for computer software, where the complexity of implementation can make it nearly impossible for the user to see the connections between his action and the program's reaction. When we use the computer to digitally edit sound or display video effects like morphing, we are bereft of analogy to the mechanical world, so our mental models are necessarily different from the implementation model. Even if the connections were visible, they would remain inscrutable.

Computer software has a behavioral face it shows to the world, one made up by the programmer or designer. This posture is not necessarily an honest representation of what is really going on inside the computer, although it frequently is. This ability to *represent* the computer's functioning independent of its true actions is far more pronounced in software than in any other medium. It allows a clever designer to hide some of the more unsavory facts of how the software is really getting the job done. This disconnection between what is real and what is offered as explanation gives rise to a *third* model in the digital world, which I call the **manifest model**. It is the way the program represents its functioning to the user.

In the world of software, a program's manifest model can be quite divergent from the actual processing structure of the program. For example, an operating system can make a network file server look as though it were a local disk. The fact that the physical disk drive may be miles away is not made manifest by the model. This concept of the manifest model has no widespread counterpart in the mechanical world. The relationship between the three models is shown in Figure 3-1.

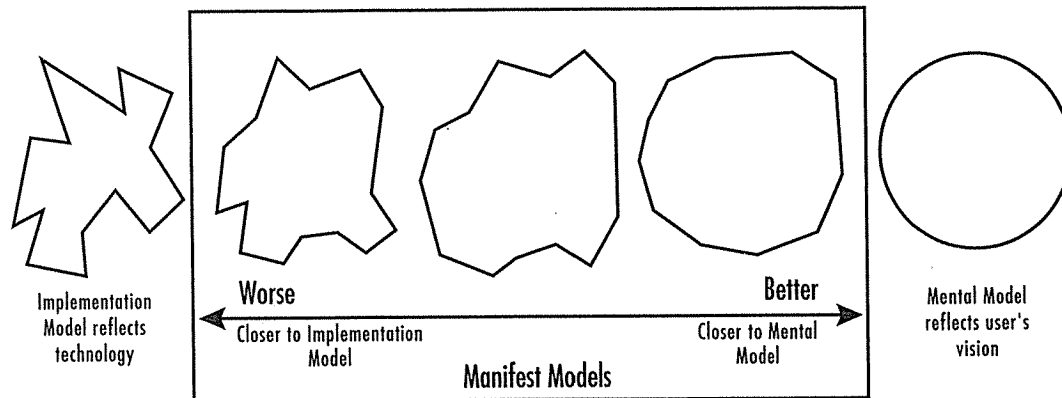


Figure 3-1

The way the engineer must build the program is usually a given. We call this the implementation model. The way the user perceives the program is usually beyond our control. He will conjure up a likely image that we call the mental model. The way the designer chooses to render the program we call the manifest model; this is the one aspect of the program that we can change significantly. If we use logic and reason to make the manifest model follow reality—the implementation model—shown on the left, we will create a bad interface. On the other hand, if we abandon logic and make the manifest model follow the user's imagination—the mental model—shown on the right, we will create a good interface.

Although software developers have absolute control over a program's manifest model, considerations of efficiency will strongly dictate their choice. Designers, on the other hand, have considerable leeway in their choice of manifest model. The closer our manifest model comes to the user's mental model, the easier he will find the program to use and to understand. Generally, offering a manifest model that closely follows the implementation model will reduce the user's ability to use and learn the program.

We tend to form mental models that are simpler than reality, so creating manifest models that are simpler than the actual implementation model can help the user achieve a better understanding. Pressing the brake pedal in your car, for example, may conjure a mental image of pushing a lever that rubs against the wheels to slow you down. The actual mechanism includes hydraulic cylinders, tubing and metal pads that squeeze on a perforated disk, but we simplify all of that in our minds, creating a more effective, albeit less accurate, mental model. In software, we imagine that a spreadsheet “scrolls” new cells into view when we click on the scrollbar. Nothing of the sort actually happens. There is no sheet of cells out there, but a tightly packed heap of cells with various pointers between them, and the program synthesizes a new image from them to display in real-time.

The ability to tailor the manifest model is a powerful lever that the software designer can use positively or negatively. If the manifest model takes the trouble to closely represent the implementation model, the user can get confused by useless facts. Conversely, if the manifest model closely follows a likely mental model, it can take much of the complexity out of a user interface.

When we interact with computer software, we tend to create anthropomorphic mental models. My program “reads” what I type in and “answers” me back with an appropriate response. It doesn’t really do anything of the sort, but this mental model is still a very effective tool to manage the complexity of a system. If the software manifests this same anthropomorphic model, it will be easier for the user to relate to.

Even hard-core propeller-heads anthropomorphize computers in order to better understand them. This mental model isn’t “real,” but it is analogically and symbolically valid, and very practical. Programmers often curse at their recalcitrant computers, even though they know they aren’t listening. We do this partly because our bodies have a mechanical structure. Our limbs, hands and fingers are levers, so we think of automobile suspension systems as arms or ankles, even though they are much more complex than that.

A mental model doesn’t necessarily have to be true or accurate, but it enables the user to work effectively with the modeled process. For example, most non-technical computer users imagine that their video screen is the heart of their computer. This is only natural because the screen is what they stare at all the time and is the place where they see what the computer is doing. If you point out that the computer is actually a little chip of silicon in that big steel box

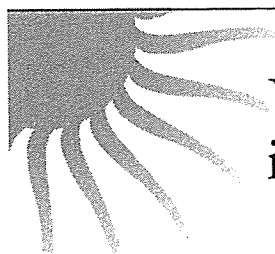
sitting under their desk, they will probably shrug and ignore this pointless factoid. The fact that the CPU isn't actually in the video display doesn't help them think about how they work with their computer, even though it is a more technically accurate concept. The industry doesn't invest a lot of effort in disabusing people of this mental model because it so clearly helps and it doesn't seem to get in anybody's way.

Most software conforms to implementation models

Because software interfaces are often designed by engineers who know exactly how the software works, the result is software with a manifest model very consistent with its implementation model. This is logical and truthful, but not very effective. The user doesn't care all that much about how a program is actually implemented. Of course, he cares about any problems that arise because of the difference between the models, but the difference itself is of no particular interest. There is a real communication gap between technical people who understand implementation models and non-technical users who think purely in terms of mental models. Any time a user telephones a software company's technical support hotline, he will probably fall into that gap.

Understanding how software actually works will always help someone to use it, but this understanding usually comes at a significant cost. The manifest model allows software creators to solve the problem by simplifying the apparent way the software works. The cost is entirely internal, and the user never has to know. User interfaces that abandon implementation models to follow mental models more closely are better.

In Adobe PhotoShop the user can adjust the color balance of an illustration. A small dialog box, instead of offering a numeric setting—the implementation model—shows a series of small, sample images, each with a different color balance. The user can click on the image that best represents the desired color setting. Because the user is thinking in terms of colors, not in terms of numbers, the dialog more closely follows his mental model.



User interfaces that conform to
implementation models are bad

A prime example of a user interface conforming to the implementation model instead of to the user's mental model can be found in Delrina's WinFax LITE product. Every step of the process is agonizingly wrought in discrete steps that the user must laboriously control, and none of which are necessary from the user's point of view. The interaction with the user is rendered in perfect conformance with the internal logic of the software. Every possible user action is duly represented by a separate dialog box. You can see some of this in Figure 3-2. The user is prompted for information when it is convenient for the program to receive it—not when it is natural for the user to provide it. The manifest model of the WinFax program closely follows the implementation model and ignores the user's mental model. Instead of imagining the steps the *user* might take to create and send a fax, the designer imagined what the *program* had to do. This is typical of a user interface designed by programmers.

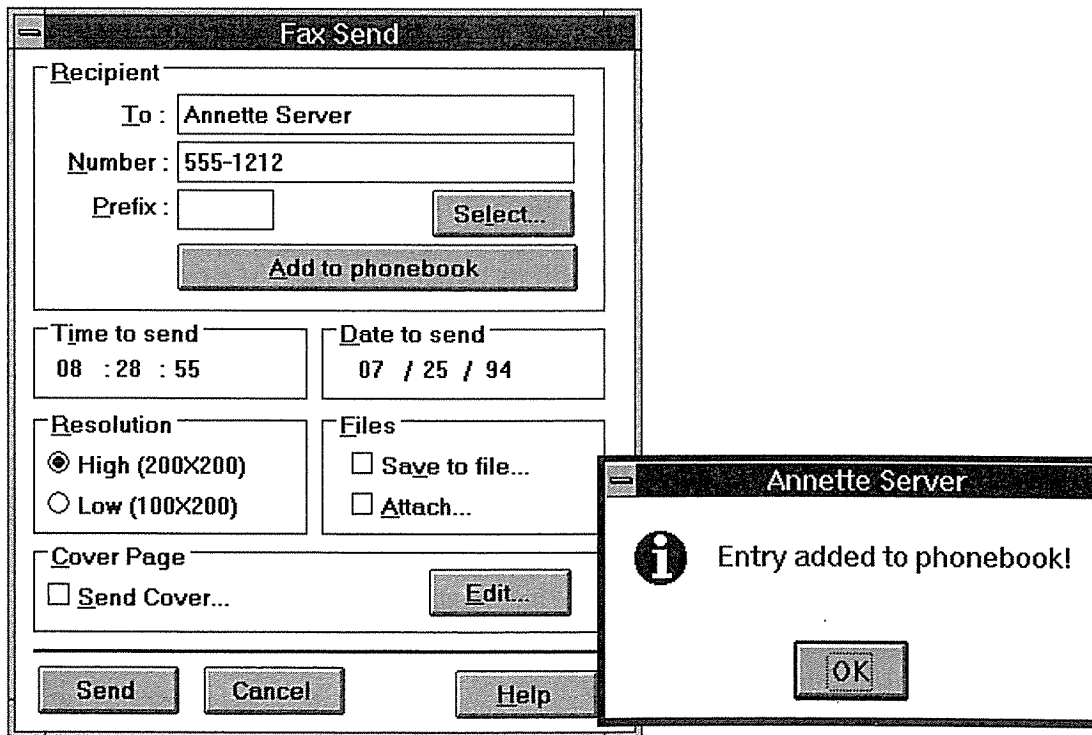


Figure 3-2

Delrina's WinFax LITE is a great study in aggravating users. Even in an application as simple as this one, they can't seem to resist adding complications. For most people on *this* planet, there are only two options: selecting an existing number or entering a new one. They have to ask for both options explicitly, and both options force a secondary dialog. Even though you can add a new number to the phonebook in place, they force a completely extraneous dialog box on you just to make sure that any ease-of-use you might be experiencing is destroyed.

When I want to send a fax, it will either be to a person whose name I haven't yet entered into the program or one I already have. The code that performs these functions is encased in separate modules inside the program, so the program encases these functions in separate dialog boxes. To either select or enter names, I have to sidestep the main program, even though selecting and entering names is the program's primary function. Similar logic in automobile design would have us manually setting the spark advance lever as we accelerated and manually flipping the brake light switch when we decelerated.

Here's a better way to manage the WinFax problem: Whenever I enter a new fax name and number, the program should automatically record it. WinFax's main window should display a list of the names of previous fax recipients, allowing me to quickly choose one from the list if I want.

Even Windows 95 misses this point. The Explorer attempts to show all of the storage devices on the computer as a unified system, but to successfully communicate that to the user, their behavior must also be unified. Instead, their behavior depends on the physical nature of the particular storage device. If you drag a file between directories on the same hard drive, the program interprets this as a MOVE, that is, the file is removed from the old directory and added to the new directory, closely following the mental model. However, if you drag a file from hard drive C to hard drive D, the action is interpreted as a COPY; that is, the file is added to the new directory but *not* removed from the old directory. This is consistent with the implementation model—the way the underlying file system actually works. When the operating system moves a file from one directory to another on the same drive, it merely relocates the file's entry in the disk's table of contents. It never actually erases and rewrites the file. But when it moves a file to another physical drive, it must physically copy the data onto the new drive. To conform to the user's mental model, it should then erase the original, even though that contradicts the implementation model. Microsoft's programmers evidently couldn't bring themselves to manifest it in any terms other than the physical ones. Actually, this behavior can be desirable, especially when copying files from a hard drive to a floppy drive, so many people aren't aware that it is just a terrifically inconsistent side effect. As computers mature and logical "volumes" represent more than just physical drives, the side effects stop being useful and become merely irritating because you have to memorize the idiosyncratic behavior.

Mathematical thinking

The interface designer must shield the user from the implementation models that the software engineer used to solve the internal problems of the software. Just because a certain tool is well-suited to attacking a problem in software construction doesn't necessarily mean that it is well-suited as a mental model for the user. In other words, just because your house is constructed of two-by-four studs and sixteen-penny nails, it doesn't mean that you should have to be skilled with a hammer to live there.

Most of the data structures and algorithms used to represent and manipulate information in software are logic tools based on mathematical models. All programmers are fluent in these models, including such things as recursion, hierarchical data structures and multi-threading. The problem arises when the user interface manifests the concepts of recursion, hierarchical data or multi-threading.

Mathematical thinking is an implementation model trap that is particularly easy for programmers to fall into. They solve programming problems by thinking mathematically, so they naturally see these mathematical models as appropriate terms for inventing user interfaces. Nothing could be further from the truth.

Design tip: Users don't understand Boolean

For example, one of the most durable and useful tools in the programmer's toolbox is Boolean algebra. It is a compact mathematical system that conveniently describes the behavior of the strictly on-or-off universe that exists inside all digital computers. There are only two main operations: AND and OR. The problem is that the English language also has an "and" and an "or," and they are usually interpreted—by non-programmers—as the exact opposite of the Boolean AND and OR. If the program expresses itself with Boolean notation, the user can be *expected* to misinterpret it.

For example, this problem crops up frequently when querying databases. If I want to extract from a file of employees those who live in Arizona along with those who live in Texas, I would say, in English, "get employees in Arizona and Texas." To say that properly in Boolean algebraic terms, I would say "get employees in Arizona OR Texas." No employee lives in two states at once, so saying "get employees in Arizona AND Texas" is nonsensical in Boolean and will always return the empty set as an answer. If you want to extract from that

database all of the employees who started work between January 1st and February 28th, it seems natural to say, in English, “get employees with start dates of January and February.” In Boolean, you would say “get employees with start dates of January OR February.”

A database query program—or any other program, for that matter—that interacts with the user in Boolean is doomed to suffer severe user interface problems. It is unreasonable to expect users to penetrate the confusion. They are well-trained in English, so why should they have to express things in an unfamiliar language that—annoyingly—redefines key words.

Bringing mechanical age models into the information age

We are experiencing an incredible transformation from a mechanical age to an information age. The change has only begun, and the pace is accelerating rapidly. The upheaval that society underwent as a result of industrialization will be dwarfed by that associated with the information age.

It is only natural for us to drag the imagery and taxonomy of the earlier era into the new one. As the history of the Industrial Revolution shows, the fruits of new technology can often only be expressed at first with the language of an earlier technology. For example, we called railroads “iron horses,” automobiles “horseless carriages,” and radio “wireless.” Unfortunately, this imagery and taxonomy colors our thinking more than we might admit.

Importing linguistic or mental images directly from the pre-digital world is an example of what I call **mechanical age modeling**.

We use old representations in the new environment. Sometimes, the usage is valid since the function is identical, even if the underpinning technology is different. For example, when we translate the process of typewriting with a typewriter into word processing on a computer, we are doing mechanical-age modeling of a common task. Typewriters used little metal tabs to slew the carriage rapidly over several spaces and come to rest on a particular column. The process, as a natural outgrowth of the technology, was called “tabbing” or “setting tabs.” Word processors also have tabs because their function is the same: whether you are working on paper rolled around a platen or on images on a video screen, you need to rapidly slew to a particular margin offset.

Sometimes, however, the mechanical-age model can't make the cut into the digital world. We don't use reins to steer our cars, or even a tiller, although both of these older models were tried in the early days of autos. It took many years to develop an idiom that was unique to and appropriate for the car.

When technology changes dramatically, we often find that the nature of the tasks we perform generates what I call **information age models**.

These are tasks, processes or concepts that arise solely because the new technology makes them possible for the first time. With no reason to exist in the non-digital version, they were not conceived of in advance. When the telephone was first invented, for example, it was touted solely as a business tool. Its use as a personal tool wasn't conceived of until it had been in use for 40 years. Today, of course, the phone is used at least as much for personal reasons as it is for business. When your teenage son spends an hour on the phone, it is a usage model that was invisible from the older world.

New conceptual models are not exclusive to the digital world; they are part of any rapidly shifting context, and technology is our current context. Digital technology is the most rapidly shifting context humankind has witnessed so far, so new and surprising information-age models are and will be plentiful.

An interesting thing about information-age models is that we have a hard time seeing them with our mechanical-age mindset. Often, the real advantages of the software products we create remain invisible until they have a sizable population of users. For example, the real advantage of e-mail isn't that it's faster mail, but rather the flattening and democratization that it promotes in the modern business organization—the information-age advantage. The real advantage of making it possible for everybody to communicate online isn't cheaper and more-efficient communications—the mechanical-age viewpoint. Instead, it is the creation of virtual communities—the information-age advantage that was revealed only after it materialized in our grasp.

The language we bring to the new environment creates a problem because it is always derived from mechanical-age models. Forty years ago, the computer was envisioned as a big collating machine, and we applied the collation model to it. We saw it as a “unit-record” device for 80-column-wide keypunch cards. Today, when computers are ubiquitous personal productivity machines, we still find vestigial indications of that 80-column, unit-record world.

The taxonomy of the mechanical-age model tends to obscure the recognition of information-age models. The mechanical taxonomy hinders invention and goal-directed design by focusing our thinking on old-paradigm goals. For example, in the non-digital world calendars are made of paper and are usually divided up into a one-month-per-page format. This is a reasonable compromise based on the size of paper, file folders, briefcases and desk drawers.

Now that we have desktop computers, we frequently see programs with graphic representations of calendars, and they almost always show one month at a time. Why? Paper calendars showed a single month because they were limited by the size of the paper, and a month was a convenient breaking point. Computer screens are not so constrained, but they copy the mechanical-age artifact faithfully. On a computer, the calendar could easily be a continuously scrolling sequence of days, weeks or months as shown in Figure 3-4, rather than a series of discrete pages, as in Figure 3-3. Scheduling something from April 28th to May 4th would be simple if weeks were contiguous instead of broken up by the arbitrary monthly division.

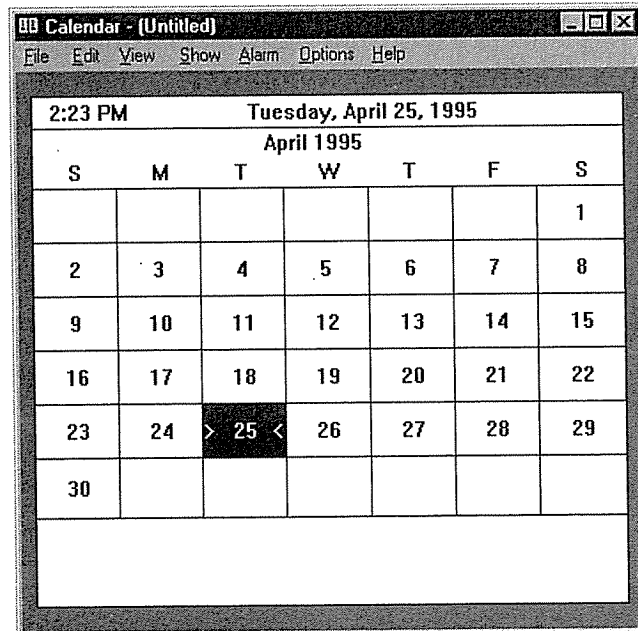


Figure 3-3

The ubiquitous calendar is so familiar that we rarely stop to apply our information-age design sensibilities to it, but that old calendar was designed for small pieces of paper, not for computer screens. The one shown here is from the Calendar in Windows 3.1. How would you redesign it? What aspects of the calendar are artifacts of its old, mechanical-age platform?

96	SUN	MON	TUE	WED	THU	FRI	SAT
APRIL	21	22	23	24	25	26	27
	28	29	30	1	2	3	4
MAY	5	6	7	8	9	10	11
	12	13	14	15	16	17	18
	19	20	21	22	23	24	25
	26	27	28	29	30	31	1

Figure 3-4

Scrolling is an idiom extremely familiar to computer users. Why not add scrolling to the calendar to create a better one? This perpetual calendar can do everything the old one can, and it also solves the mechanical-age problem of scheduling things across monthly boundaries. Why drag old limitations onto new platforms just out of habit? What other improvements can you think of?

Similarly, the grid pattern in digital calendars is almost always of a fixed size. Why couldn't the width of columns of days or the height of rows of weeks be adjustable like a spreadsheet? Certainly you'd want to adjust the sizes of your weekends to reflect their relative importance over your weekdays. Likewise, your vacation-week calendar would demand more space than a working week. The idioms are as well known as spreadsheets—that is to say, universal—but the mechanical-age models are so firmly set in our taxonomy that we rarely see software publishers deviate from the trajectory of the past. We have the tools, we just don't have the language.

The designer of the software thought of calendars as a canonical image—one that couldn't be altered from the familiar. This calendar software often exhibits interesting new information-age features, like the ability to page instantly forward and backward months or years at a time, or to add graphic representations of holidays to the little day rectangles. These same calendars rarely break the one-month-per-screen archetype, though, and it is this one thing that really holds digital calendars back. Surprisingly, most time-management software

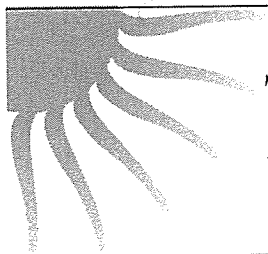
probably handles time internally—its implementation model—as a continuum, and only renders it as discrete months in its user interface—its manifest model!

Sometimes people counter that the one-month-per-page calendar is better because it is familiar and unthreatening to users. I doubt it. Most people's mental models don't break time into monthly chunks, but rather see it as a continuum of days. Nor do people find it difficult to adapt to newer, simpler manifestations of familiar systems. We adapted to electric from gas stoves without a hitch. Similarly, the transition from manual transmissions to automatics, from AM radio to FM, from conventional to microwave ovens and from vinyl records to compact discs was simple and painless.

All of those paper-style calendars on various personal information managers (PIMs) and schedulers are mute testimony to how our taxonomy—our language—influences our designs. If we depend on words from the mechanical age, we will build software from the mechanical age. Better software is based on information-age thinking.

It's worse on a computer

We encounter another big problem when we bring our familiar mechanical-age models over to the computer. Simply put, mechanical-age processes are a lot worse when computerized. Procedures are easier by hand than they are with computers. Try to type someone's address on an envelope with a computer. The only time it gets easier is if you have 500 envelopes to address.



Transliterated mechanical models are always worse on computers

Another example, a name and address list on a computer—if it is faithfully rendered like a little bound book—will be much more complex, inconvenient and difficult to use than the actual book. The name and address book, for example, stores names in alphabetical order by last name, but what if you want to find someone by his first name? The mechanical-age artifact doesn't help you: you have to scan the pages manually. So, too, does the computerized version: it can't search by first name either. The difference is that, on the computer screen, you lose many subtle visual cues offered by the paper-based book. The

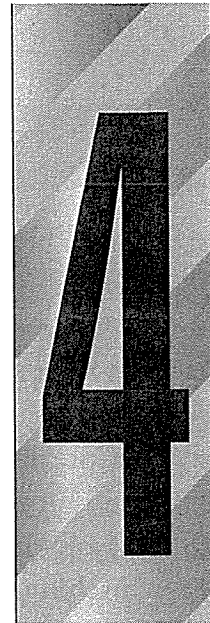
scrollbars and dialog boxes are harder to use, to visualize and to understand than flipping pages. They are rocks thrown at your feet.

Whenever you take a mechanical process and put it on a computer, the user of that process will suffer. The only situation where transliterated processes yield an advantage is if the sheer quantity of items to be processed is large enough to justify doing the task en masse. Early data-processing systems did this with applications like invoicing and billing. Most of our desktop computing jobs don't involve sufficiently large quantities of information for this to remain true.

But there is another, bigger problem with transliterated mechanical models. The old mechanical method will always have the strengths and weaknesses of its medium, like pen and paper. Software has a completely different set of strengths and weaknesses, yet when those old models are brought across without change, they combine the weaknesses of the old with the weaknesses of the new. In our address book example, the computer could easily search for an entry by first name, but, by storing the names in the same paradigm as the mechanical artifact, we deprive ourselves of new ways of searching. We limit ourselves to not much more than what we could do in the world of paper and ink, but this time we have to do it through dialog boxes and menus.

When designers rely on mechanical-age paradigms to guide them, they are blinded to the far greater potential of the computer to do information management tasks in a better, albeit different, way.

Visual Interface Design



The commonly accepted wisdom of the post-Macintosh era is that graphical user interfaces, or GUIs, are better than character-based user interfaces. This is generally a true statement but, while there are certainly GUI programs that dazzle us with their ease of use, the vast majority of GUI programs irritate and annoy us in spite of their graphic nature. Why is this?

Visual Software

It's not merely the graphic nature of an interface that makes it better. Using a bit-mapped system to render the lines and characters of a character-mode program doesn't change the essential nature of the program. It's very easy to create a program with a "graphical user interface" that has the same extreme difficulty-of-use as a CP/M, DOS or UNIX application.

The qualities that make a user interface good are user-centric and not technology-centric. "Graphicalness" is a technology-centric concept. There are two really important user-centric qualities: the "visualness" of the software and the program's vocabulary.

Most humans process information better visually than they do textually. Sure, we learn by reading, but we learn much more, much faster by seeing things whole and in context. In order to realize the advantages of the technology, the interaction with the user must become visual. The issue isn't the graphic nature of the program, it's the visualness of the interaction. Instead of GUI, it's a visual user interface—a VUI—that we are looking for. Software that recognizes this is called **visual interface design**. When done well, a VUI has a feeling of fluency, of moving along smoothly and effortlessly towards the user's goals without hitching or stopping on confusing little problems of comprehension.

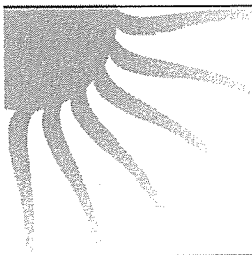
Visual processing

The human brain is a superb pattern-processing computer. It uses this strength to make sense of the dense quantities of visual information we are bombarded with from the moment we open our eyes in the morning. The acuity of the human eye is tremendous, and if our brain couldn't impose some management system on what our eyes report, we would collapse from overload. Look out the window. See the trees, the water, the waves, the clouds, the people, the windows, the people in the windows, the guy carrying the box, the name printed on the box, the letters in the name... If we had a difficult time with visual complexity, the sheer quantity of visual information we take in when we look out the window would put us in a state of shock. But we clearly aren't bothered by this visual complexity. When we look out the window, our eyes encompass a huge scene filled with constantly changing terabytes of complex information. Our brain manages the input by unconsciously discerning patterns, and by using these patterns to manage what we are looking at. Our brains establish a system of priorities for the things we see that allow us to consciously analyze the visual input.

Text, when viewed from a distance, forms a recognizable pattern and shape that our brains categorize. This is a different act from reading, where we scan the individual words and interpret them. Even then, we use pattern-matching more than we actually sound out each syllable the way we did as children. Each word has a recognizable shape, and this is why WORDS TYPED IN ALL CAPITAL LETTERS ARE HARDER TO READ than upper/lower case—our familiar pattern-matching hints are absent in all capitals, so we must pay much closer attention to decipher what is written. This same pattern-processing talent explains why body text in books is always in a relatively standard, serif typeface like the one you are looking at now. *However, if this book were printed using a sans serif font, or a font with unusual proportions, you would find it not a strain on the eyes, but a strain on the brain.*

When we look at the complex scene out the window, our brain gathers big chunks of the view into manageable pieces—building, street, ocean, sky—and lets our conscious processes grapple with higher-level issues.

If, for example, we find ourselves taking a second look at one person in the crowd on the street below, it is because our subconscious pattern-matching equipment got a hit. We next study the person's face, searching for details in order to make a positive identification. We go through the identical process when we read documents. Our unconscious mind is constantly reducing visual input to patterns, and our conscious mind is constantly ordering those patterns into hierarchies. When our eye-brain-pattern system reports an “envelope,” our brain-hierarchy system isolates it and examines it for our name. The pattern system detects the envelope pattern; then the conscious system disambiguates that pattern into either a letter for us or a letter for someone else.



Visually show what,
Textually show which

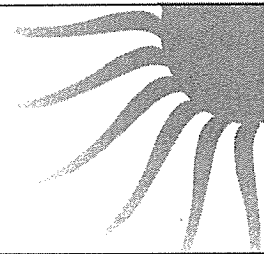
If our unconscious mind could not classify the pattern as an envelope, we would have to get our conscious mind involved in the preliminary processing. It is much faster when our unconscious mind provides the first cut because pattern-matching is so much faster and more efficient than having to think about it.

Visual patterns

If our conscious mind had to grapple with every detail of what our eyes saw, we would be overwhelmed with meaningless detail. The ability of our unconscious mind to group things into patterns based on visual cues is what allows us to process visual information so quickly and efficiently. Understanding and applying this model of how the human mind processes visual information is one of the key elements of visual interface design. The philosophy is to present the program's components on the screen as recognizable visual patterns with accompanying text as a descriptive supplement. The user can choose, on a purely pattern-matching, unconscious level, which objects to consider consciously. The accompanying text only comes into play once the user has decided it's important.

You build an effective visual interface from visual patterns. Notice that I did not say pictures or images or icons. Representational images are useful, but patterns are the engine of unconscious recognition. For the user to discern a particular icon from a screenful of similar but different icons is just as difficult as discerning a particular word from a screenful of similar but different words. Icons that must be consciously recognized or deciphered are no better—and possibly much worse—than plain text.

A visual interface is based on visual patterns



The pecking order of visual understanding always regards visual pattern-matching as superior to verbal or pictographic reading. Pattern-matching is unconscious and reading is conscious. Our visual user interface must create readily recognizable patterns. It will certainly include text, but only in a secondary role of distinguishing between objects with similar patterns.

We create patterns in very simple ways. Possibly the simplest is by creating recognizable graphic symbols and giving them value by association. As you drive down the highway, you read all of the signs you see. After a while, you begin to notice a pattern. Every time the highway you are on is identified, its number is accompanied or even enclosed by the symbol “☞.” You probably don’t pay much attention to this trivial detail, and why should you? You are usually well aware of what highway you are on. Your unconscious mind filters out the ☞ signs. Then one day you are on an unfamiliar highway and you want to know exactly which one you are on. Your conscious mind wants to know this, so your unconscious mind alerts you to the presence of each ☞ it sees. Your conscious mind then reads the numbers on the sign to separate it from all of the other ☞s you have seen. The ☞ is not representational. It is not metaphoric. It is idiomatic: you learn the shape from the context in which it is used, and from then on it represents its context.


This is exactly what you do with visual interface design. You create symbols for the objects in the interface. If the program you are creating manages a restaurant,

for example, you will find that tables, checks, orders, specials, and waitpersons are the fundamental elements—the building blocks—with which you must create the interface. In other words, these are the objects that the users will manipulate to achieve their goals. What you need to do is create a recognizable visual symbol for each of these primary types:

-  Tables
-  Checks
-  Orders
-  Specials
-  Waitpersons

The symbols don't have to be representational, but it doesn't hurt. If you do choose a representational image, don't kid yourself about its value as a teaching tool. On the other hand, don't ignore the value of mnemonics. Each user can form his own mental cues to help him remember what the symbols represent: factories and tables both produce value; ducks and waitpersons both fly from place to place.

In order to drive home the connection between symbol and object, you must use the symbol everywhere the object is represented on the screen. Whether the object is an item in a listbox, an entire dialog box, a mention in text, or a gizmo on the toolbar, it must be accompanied by the visual symbol. You don't have to spell this out to the user: you are teaching it to his unconscious mind, and its presence alone over time is sufficient to do that. I call this a **visual fugue**.

If you have a list of waitpersons, prefix each one with the  symbol as in Figure 4-1.

The power of this technique is even greater if you have a listbox filled with heterogeneous objects. Imagine a similar listbox filled with both tables and orders as shown in Figure 4-2.

Our minds differentiate each line—each object—by its visual symbol, and once we have identified the type we are interested in, we read the text to separate it from its siblings. We don't have to read about objects we are not interested in. This type of processing is very natural to humans and we can perform it rapidly and with little effort.

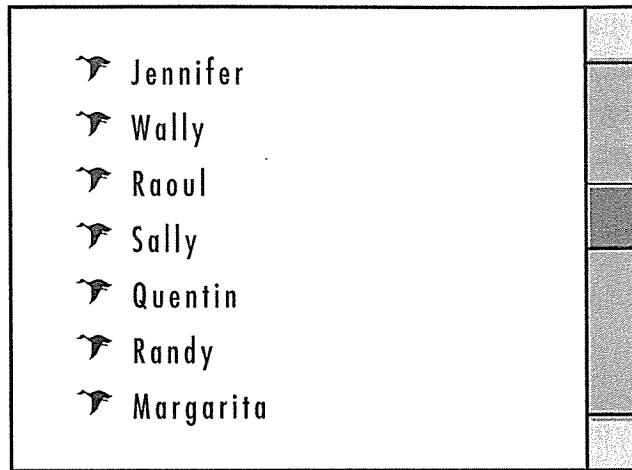


Figure 4-1

This listbox is filled with several objects of one type. You can see that unconsciously, because your mind discerns the identical symbols associated with each entry. It will probably take some additional reading to disambiguate which object is which, but without the symbols, we'd have to read them all just to know what they are and that they are all the same type. Symbols should always be associated with text in visual user interfaces.

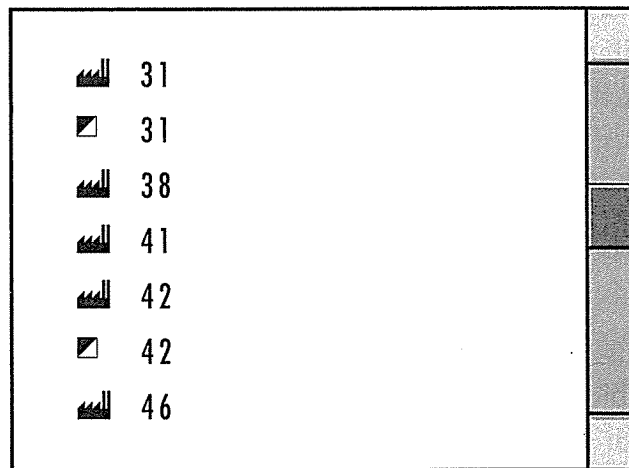


Figure 4-2

This listbox is filled with objects of two different types. Without the symbols to differentiate between tables and orders, it would be impossible to make sense of the list. We would have to label each entry with text, "Table 31," "Order 31," and so on. The symbols are much faster, letting our unconscious minds recognize the patterns before our relatively slow conscious minds even have to pay attention.

Restricting the vocabulary

When graphical user interfaces were first invented, they were so clearly superior that many observers credited their success to their graphics. This was a natural reaction, but it was only part of the story. One of the most important reasons why those first GUIs were better was that they were the first user interfaces to restrict the range of their vocabulary for communicating with the user. In particular, the input they could accept from the user went from a virtually unrestricted command line to a tightly restricted set of mouse-based actions. In a command line interface, the user can enter any combination of characters in the language—a virtually infinite number. In order for the user's entry to be correct, he needs to know exactly what the program expects. He must remember the letters and symbols with exacting precision. The sequence can be important. The capitalization can be vital.

In the GUI, the user can point to images or words on the screen with the mouse cursor. Using the buttons on the mouse, the user can click, double-click or click-and-drag. That is it. The keyboard is used for data entry, not for command entry or navigation. Instead of 26 letters, 10 digits and a couple of dozen other keys available in an infinite number of combinations in the command line interface, the user has just three basic actions to choose from. The number of atomic elements in the user's input vocabulary dropped from millions to just three, even though the range of tasks that could be performed by GUI programs wasn't restricted any more than that of command-line systems.

The more atomic elements there are in a communications vocabulary, the more time-consuming and difficult the learning process is. Vocabularies like the English language take at least ten years to learn thoroughly, and its complexity requires constant use to maintain fluency. Of course, English is a fantastically expressive language and, in the hands of an artist, can be a most compelling medium. Our users aren't artists, though, and they shouldn't have to invest that much effort in becoming effective with our software. Merely restricting the number of elements in the vocabulary reduces the expressiveness of it, so that alone is not the solution. The answer lies in the way we build our vocabularies—some parts are restricted in size, while others can be huge.

A properly formed vocabulary is shaped like an inverted pyramid. All easy-to-learn communications systems obey this pattern. It is so fundamental that I call it the **canonical vocabulary**. You can see a picture of it in Figure 4-3.

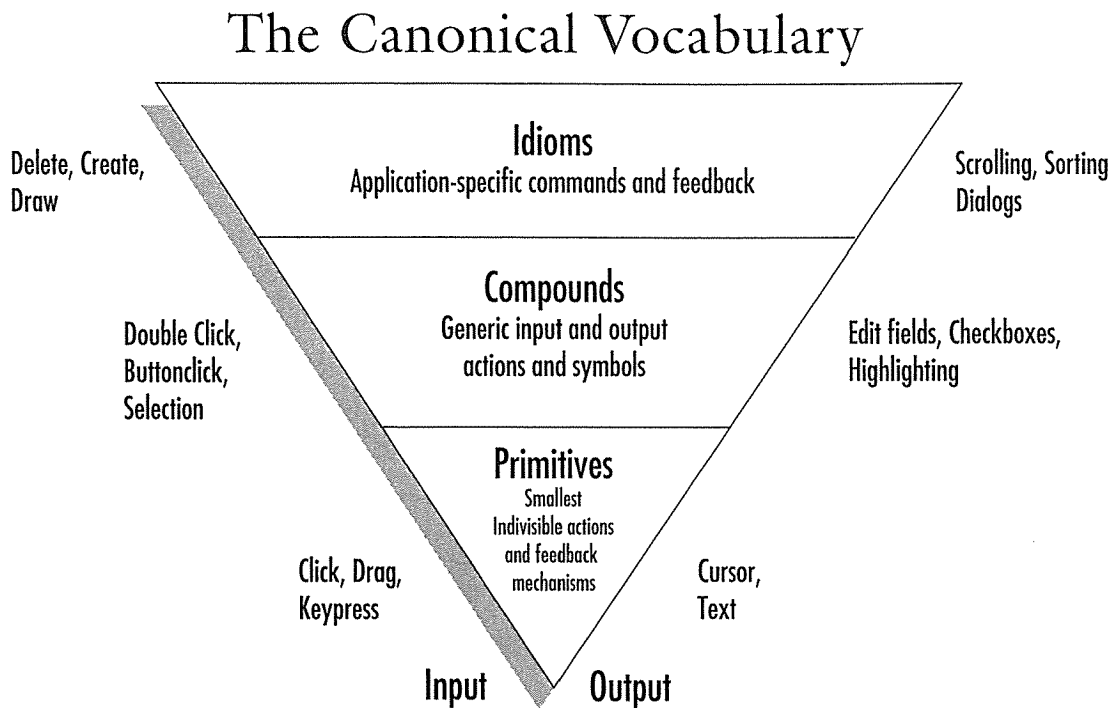


Figure 4-3

The main reason GUIs are so much easier to use is that they were the first platform to enforce a canonical vocabulary. It has very little to do with graphics. All vocabularies follow this archetypal form.

At the lowest level is a set of primitives from which all else is constructed. Generally, the set of primitives shouldn't exceed four elements. The middle layer consists of more complex constructs built from combinations of the primitives. The upper-level idioms are compounds with the addition of domain knowledge.

The bottom segment contains what I call the **primitives**, the atomic elements of which everything in the language is comprised.

Paraphrasing Albert Einstein, this set should be as small as possible, but no smaller. In a GUI, it consists of pointing, clicking and dragging. A set of primitives of two to four items is about right. More than that leads to trouble.

The middle trapezoid contains what I call the **compounds**.

These are more complex constructs created by combining one or more of the primitives. Nothing else is added; they are built exclusively from elements below them in the pyramid. In a GUI, it contains such actions as double-clicking, click-and-dragging and manipulable objects like push-buttons and checkboxes.

The uppermost layer of the pyramid contains what I call the **idioms**. Idioms combine compounds with knowledge of the problem under consideration, known as **domain knowledge**. Domain knowledge is information related to the user's application area and not specifically to the computerized solution. The set of idioms opens the vocabulary to information about the particular problem the program is trying to address. In a GUI, it would include things like OK buttons, caption bars, listboxes and file icons.

Any language that does not follow the canonical form will be very hard to learn. Many effective communications systems outside of the computer world follow canonical vocabularies. Street signs follow a simple pattern of shapes and colors: Yellow triangles are cautionary, red octagons are imperatives and green rectangles are informative.

Our telephone system has a tiny set of primitives consisting of simple audio tones. Hearing a buzz—a dial tone—means the system is available. When the buzz alternates with silence, it means the number is busy. A warble means the phone is ringing. Silence means we have failed to enter valid numbers, or there is some other problem and we should try again.

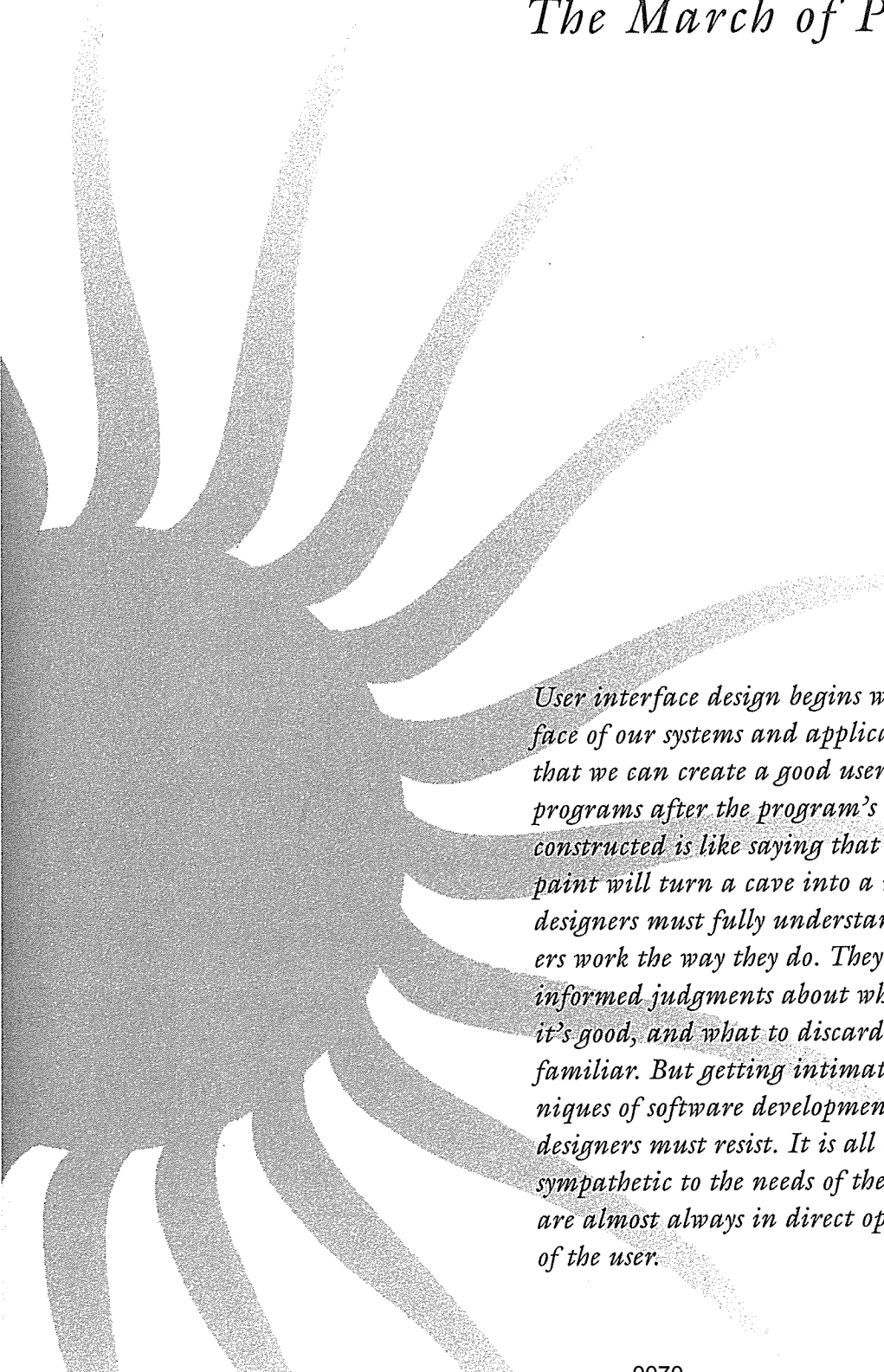
Designing for users

Successful user interfaces are those that focus on the user's goals even if they have to ignore the technology of the implementation. Professional software designers are the primary group today acting as advocates for the user.

To create effective visual interfaces, designers must create interaction from a canonically formed vocabulary that is expressed visually. This vocabulary follows the user's mental model, even if it diverges from the physically correct model. As Frederick Brooks says, "The [designer] sits at the focus of forces which he must ultimately resolve in the user's interest."

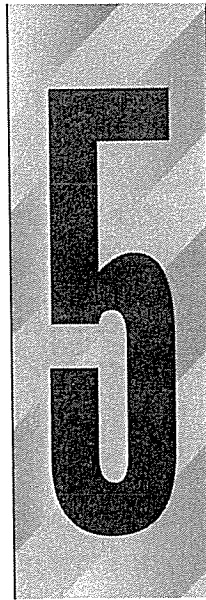
Part II: The Form

The March of Paradigms



User interface design begins well-below the surface of our systems and applications. Imagining that we can create a good user interface for our programs after the program's internals have been constructed is like saying that a good coat of paint will turn a cave into a mansion. Software designers must fully understand why our computers work the way they do. They must make informed judgments about what to keep because it's good, and what to discard even though it is familiar. But getting intimate with the techniques of software development is a seduction that designers must resist. It is all too easy to become sympathetic to the needs of the computer, which are almost always in direct opposition to the needs of the user.

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.

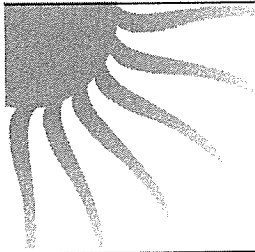
The field of user interface design has progressed from an orientation focused on technology, into one that focuses overmuch on metaphor. We are just now becoming aware of idiomatic design. There is ample evidence of all three paradigms in contemporary software design, even though the metaphor paradigm 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.

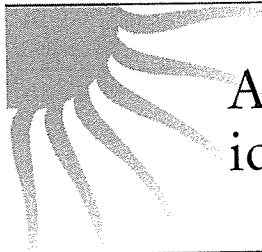
This is where the human mind is really outstanding: learning and remembering idioms very easily without relying on comparisons to known situations or an understanding of how they work. This is a necessity, because many idioms don't have any metaphoric meaning at all, and the stories behind most others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

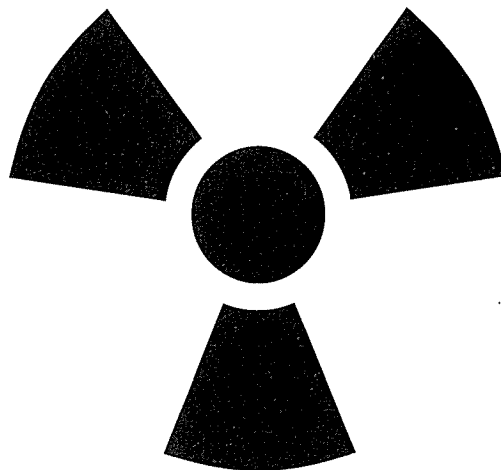


Figure 5-1

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.

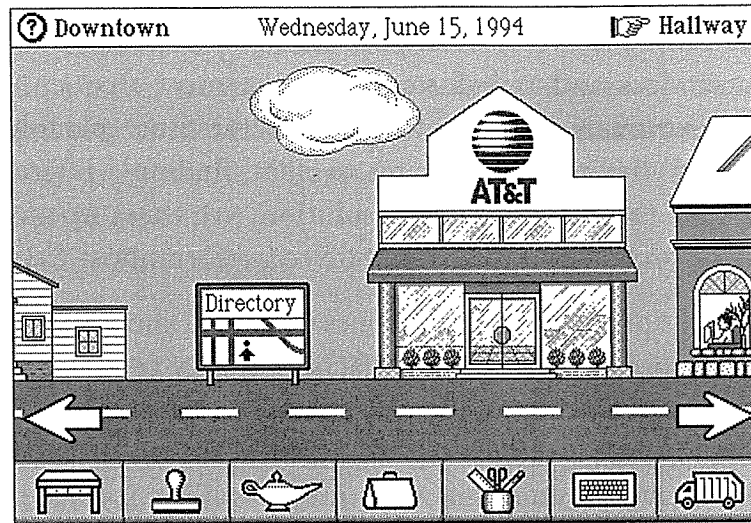
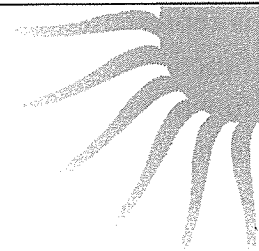


Figure 5-2

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 push-able 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 scrollbar 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.

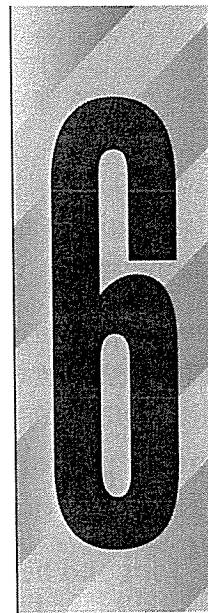
In the canonical vocabulary (described in Chapter 4), manual affordances have no meaning in the uppermost tier, in idioms. This is why gizmos must have writing on them to make sense. If the answer isn't written directly on the gizmo, we can only learn what it does by one of two methods: experimentation 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.

An Irreverent History of Rectangles on the Screen



Any book on Windows user interface design must devote considerable space to windows-with-a-lower-case-w. I consider it necessary to place these omnipresent rectangles in some historical perspective to keep the reader from imbuing too much intrinsic value in them.

Xerox PARC

Microsoft's Windows derives its appearance either from the Apple Macintosh or from the Alto, an experimental desktop computer system developed in the late '70s at Xerox PARC. This is, however, a distinction without a difference, since the Macintosh was derived directly from the Alto.

Xerox PARC and the Alto contributed many significant innovations to the vernacular of desktop computing. These include several things we now regard as commonplace: the mouse, the rectangular window, the scrollbar, the push-button, the "desktop metaphor," object-oriented programming, pulldown menus and Ethernet.

PARC's effect on the industry and contemporary computing was profound. Both Steve Jobs and Bill Gates, chairmen of

Apple Computer and Microsoft, respectively, saw the Alto at PARC and were indelibly impressed.

Xerox tried to commercialize the Alto with a computer system called Star, but it was expensive, complex, agonizingly slow and a commercial failure. The brain trust at PARC, realizing that Xerox had blown an opportunity of legendary proportions, began an exodus that greatly enriched other software companies, particularly Apple and Microsoft.

Steve Jobs and his PARC refugees immediately tried to duplicate the Alto/Star with the Lisa. In many ways they succeeded, including copying the Star's failure to deal with reality. The Lisa was remarkable, accessible, exciting, too expensive and frustratingly slow. Even though it was a decisive commercial failure, it ignited the imagination of many people in the small but booming micro-computer industry.

Bill Gates was less impressed by the sexy "graphicalness" of the Alto/Star than he was by the more systemic advantages of an object-oriented presentation and communication model. Software produced by Microsoft in the early eighties, notably the spreadsheet Multiplan (the forerunner of Excel), reflected this thinking.

Steve Jobs wasn't deterred by the failure of the Lisa. He was convinced that its lack of success was due to compromises in its design and that PARC's vision of a truly graphical personal computer was an idea whose time had come. He added to his cadre of PARC refugees by raiding Apple's various departments for skilled and energetic individuals, then set up a "skunk works" to develop a commercially viable incarnation of the Alto. The result was the legendary Macintosh, a machine that has had enormous influence on our computing technology, design and culture. The Mac single-handedly brought an awareness of design and aesthetics to the industry. It not only raised the standards for user-friendliness, but it also enfranchised a whole population of skilled individuals from disparate fields who were previously locked out of computing because of the industry's self-absorption in techno-trivia.

The almost-religious aura surrounding the Macintosh was also associated with many aspects of the Mac's user interface. The pull-down menus, metaphors, dialog boxes, rectangular overlapping windows and other elements all became part of the mystique. Unfortunately, because its design has acquired these heroic proportions, its failings have often gone unexamined.

PARC's Principles

One of the ideas that emerged from PARC was the visual metaphor. At PARC, the global visual metaphor was considered critical to the user's ability to understand the system, and thus critical to the success of the product and its concept. In the last chapter I wrote at length about the problems of such metaphoric design.

Modes

Another principle associated with the modern GUI is the notion that modes are bad. A **mode** is a state the program can enter where the effects of a user's action changes from the norm—essentially a behavioral detour.

For example, older programs would demand that you shifted into a special state to enter records, then shift into another state to print them out. These behavioral states are modes, and they can be extremely confusing and frustrating. Former PARC staffer and current Chief Scientist at Apple, Larry Tesler, was an early advocate of eliminating modes from software and was pictured in an influential magazine wearing a T-shirt with the bold legend "Don't mode me in." His license plate reads "NOMODES." In a command-line environment, modes are indeed poison. However, in the object-verb world of a GUI, they aren't inherently bad. Unfortunately, the don't-mode-me-in principle has become an unquestioned part of our design vernacular.

Arguably, the most influential program on the Macintosh was MacPaint, a program with a thoroughly modal interface. This program enables the user to draw pixel-by-pixel on the computer screen. The user selects one tool from a palette of a dozen or so and then draws on the screen with it. Each tool is a mode, because it restricts the program to behave in one way. When a tool is selected, the behavior of the program conforms, modally, to the attributes of that tool.

Of course, the PARC researchers weren't wrong, just misunderstood. The user interface benefits of MacPaint compared with contemporary programs were great, but they didn't accrue from its imagined modelessness. Rather, they resulted from the ease with which the user could see which mode the program was in and the effortlessness of changing that mode.

Generally, modes based on the implementation model are confusing modes. "Edit" mode versus "Print" mode is convenient only for the program, not the user. But modes based on the user's mental model are often harmless. The "Spray can" mode or the "Paint brush" mode, for example.

Overlapping Windows

Another Mac fundamental that emerged from PARC (and which has metastasized in Microsoft Windows) is the idea of overlapping rectangular windows. The rectangular theme of modern GUIs is so dominating and omnipresent that it is somehow seen as vital to the success of visual interaction. Actually, it is a by-product of the technology: of our TV-screen-like video display terminals. They are excellent devices for showing rectangles, but much less efficient for manipulating non-orthogonal shapes. Rectangles are an effect rather than a cause of GUI design.

Overlapping windows demonstrated clearly that there are other, and better, ways to transfer control between concurrently running programs than typing in obscure commands.

Overlapping rectangular windows were intended to represent overlapping sheets of paper on the user's desktop. Okay, I'll buy that, but why? The stated reason is that it makes it easy to see which programs are running and to shift between them, but if this were true, Microsoft wouldn't be offering us the button-lined, program-changer tool called the Startbar in Windows 95. The overlapping window *concept* is good, but its execution is impractical in the real world. The number of pixels on today's video screens is way too small and users can't afford to waste them. Leaving an edge of one application's rectangle peeking out from behind the active window is an egregious waste of precious pixels.

The overlapping-sheets-of-paper metaphor starts to suffer when you get three or more applications on the screen—it just doesn't scale up well. The idiom has other problems, too. A user who clicks the mouse just one pixel away from where he thought he was can find his program disappearing, to be replaced by another one. User testing at Microsoft has shown that a typical user might launch the same word processor several times in the mistaken belief that he has somehow "lost" the program and must start over.

Part of the confusion regarding overlapping windows comes from several other idioms that happen to be implemented using an overlapping window. The familiar dialog box is one, as are all menus and tool palettes. Such overlapping within a single application is completely natural and a well-formed idiom. It even has a faint metaphoric trace: that of your faithful secretary handing you an important note.

We have windows largely because rectangular objects are very easy to draw and to manage on a raster scan device—a video screen. We have rectangular windows because they are the easiest to program, not because they offer cognitive superiority or information-management leverage.

In the grand tradition of focusing on a trivial aspect of the new PARC GUI, Bill Gates named his hastily cobbled together response to the Macintosh's success "Windows." Ever since then, the eponymous rectangle has dominated the development of our commercial products. It has been taken for granted in many circles that would otherwise be questioning such accidental dominance.

Tiling

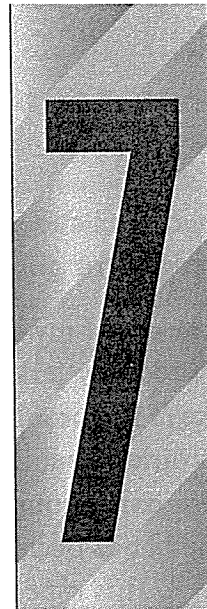
The first version of Microsoft Windows diverged somewhat from the pattern established by Xerox and Apple. Instead of using overlapping rectangular windows to represent the overlapping sheets of paper on one's desktop, Windows 1.0 relied on what was called "tiling" to allow the user to have more than one application on screen at a time. Tiling meant that applications would divide up the available pixels in a uniform, rectilinear tessellation, evenly parsing out the available space to running programs. I suspect that tiling was invented as an idealistic way to solve the orientation and navigation problems caused by overlapping windows. Navigation with tiled windows is much easier than with overlapped ones, but the cost in pixels is horrendous. Tiling died as a mainstream idiom, although it can still be found in the most interesting places: try right clicking on the Windows 95 Startbar. No doubt tiling will stage a comeback when computer screens grow to six feet square and cost \$50.

Overlapping windows fail to make it easy to navigate between multiple, running programs, so other vendors continue to search for new ways. For example, the "virtual desktop" on the UNIX-based OpenWindows platform extends the desktop to six times the size of the visible window. In the upper left corner of the screen is a small superimposed, black-and-white image of all six desktop spaces, all of which can be running different things simultaneously and each of which can have many open windows. You switch between these virtual desktops by clicking on the one you want to make active.

Microsoft braved a double-barreled breach-of-contract and patent infringement lawsuit from Apple to add overlapping to Windows 2.0. In all of this controversy, the basic problem seems to have been forgotten: How can the user easily navigate from one program to another? Multiple windows *sharing* a small

screen—whether overlapping or tiled—is not a good solution. We are moving rapidly to a world of full-screen programs. Each application occupies the entire screen when it is “at bat.” A tool like the Startbar borrows the minimum quantity of pixels from the running application to provide a visual method of changing the lineup. This solution is much more pixel-friendly, and the day of the overlapping main window is waning fast.

Much contemporary software design begins with the assumption that the user interface will consist of a series of overlapping windows, without modes, informed by a global metaphor. The PARC legacy is a strong one. Most of what we know about modern graphical user interface design came from these origins, whether right or wrong. But the well-tempered designer will push the myths aside and approach software design from a fresh viewpoint, using history as a guide, not as a dictator.



Windows-with-a-Small-w

Our programs are constructed of two kinds of windows: main windows and subordinate windows (like documents and dialog boxes). Determining which windows to use for a program is a primary step in determining its look and feel. If we expect to create an effective user interface, we cannot simply guess at which windows to use. We must choose them carefully and understand *why* we make our choices.

Unnecessary rooms

If we imagine our program as a house, we can picture each window as a separate room. The house itself is represented by the program's main window, and each room is a document window or dialog box. In real life, we don't add a room to our house unless it has a purpose that cannot be served by other rooms. Similarly, we shouldn't add windows to our program unless they have a special purpose that can't or shouldn't be served by existing windows.

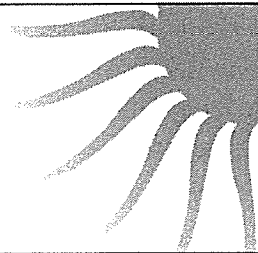
"Purpose" is a goal-directed term. It implies that using a room is associated with a goal, but not necessarily with a particular task or function. For example, you might shake

someone's hand at your front door, but it will probably have quite different connotations (or goals) than shaking someone's hand in the kitchen or bedroom.

If I held out my hand and asked you to shake, you would certainly think it odd if I suddenly jerked it away and said, "Wait! Let's go into this other room to shake." It doesn't matter what room we are in, since we both understand the motivations behind the handshake, but having to move to another room to do it is incongruous. There can be no good reason for changing rooms just to shake hands because, regardless of where we are, the task can be performed just as well. It is especially ridiculous if, after shaking in the other room, we trudge back into the first room to continue what we're doing.

If you think of dialog boxes as rooms, you can easily find examples of programs that change rooms to shake hands. The WinFax program you saw back in Figure 3-2 is one. When I use the program, it is certain that I am going to send a fax, but it sends me to another room to select a previously recorded fax number, and to yet another room to record a new fax number. WinFax LITE is a one-room program, but it divides its interface into several unnecessary rooms.

A dialog box is another room. Have a good reason to go there



In most drawing programs, for example, the depth of a drop-shadow is usually set by selecting a menu item that triggers a dialog box. A winder, text field or similar gizmo on the dialog then sets the shadow depth. After the setting is made, the program returns to the main screen that contains the drawing. This sequence is so commonplace that it is completely unremarkable, and yet it is undeniably bad design. In a drawing program, changing the image is the primary task. The image is in the main window, so that's where the tools that affect it should be also. Setting the depth of a drop-shadow isn't a tangential task but one quite integral to the drawing process. If the drawing were being done with pencil on paper, the artist might bring a new tool to bear—an eraser—but he would not shift to a different table or sheet of paper just to change the depth of the drop-shadow. The drop-shadow depth could be set with a gizmo right on the toolbar, for example, or—better yet—the user could click on the shadow with the mouse and just drag it to a new position.

Putting functions in a dialog box emphasizes their separateness from the main task. Putting the drop-shadow adjustment in a dialog box works just fine, but it creates an interaction that is stilted and rough. Going into an adjacent room to shake hands works fine, too, but it is a distracting waste of effort.

From the programmer's point of view, changing the drop-shadow is a separate function, so it seems natural to treat it like one. From the user's point of view, however, it is an integral function and should be integrated into the main window.

Design tip: Build function controls into the window where they are used

This is one of the most frequently violated tips in user interface design. Because the construction of programs is so function-centric, the user interface is often constructed in close parallel. Combine this with the incredible ease with which we can build dialog boxes, and the result is one (or more) dialog box per function. Our modern GUI-building tools tend to make dialogs easy to create, but adding gizmos to the surface of a document window or creating direct-manipulation idioms is generally not supported by these handy tools. The developer who wants to create a better user interface often must roll-his-own without much help from the tool vendors.

Necessary rooms

When it is time to go swimming, you'll think it odd if I offer you the crowded living room to change your clothes. Decorum and modesty are excellent reasons for you to want a separate room in which to change. It is entirely inappropriate for me not to provide a separate room when one is needed.

When I want to perform a function that is out of the normal sequence of events for a particular program, that program should provide a special place in which to perform it. For example, purging a database is not a normal activity. It involves setting up and using features and facilities that are not part of the normal operation of the database program. The more prosaic parts of the program will support daily tasks like entering and examining records, but erasing records en masse is not an everyday occurrence. The purge facility correctly belongs in a separate dialog box. It is entirely appropriate for the program to shunt me into a separate room—a window—to handle that function.

Using goal-directed thinking, we can examine each function to good effect. If the user is using a graphics program to develop a drawing, his goal is to create an appealing and effective image. All of the drawing tools are directly related to this goal, but the various pencils and sprayers and erasers are the most tightly connected functions. These tools should be intimately integrated into the workspace itself in the same way that the conventional artist will arrange his pencils, pens, knives, tweezers, erasers and other drawing equipment right on his drawing board, close at hand. The tools are ready for immediate use without having to reach far, let alone having to get up and walk into the next room. In the program, equivalent drawing tools should be arrayed on the edges of the drawing space, available with a single click of the mouse. The user shouldn't have to go to the menu or to dialog boxes to accomplish these tasks. The new Version 3 of Fractal Design Painter arranges artists' tools in trays, and lets you move the things that you use frequently to the front of the tray. While you can hide the various trays and palettes if you want, they appear as the default and are part of the main drawing window. They can be positioned anywhere on the window, as well. And if you create a brush that is, for example, thin charcoal in a particular shade of red that you're going to need again, you simply "tear it off" the palette and place it wherever you want on your workspace—just like laying that charcoal in the tray on your easel. This tool selection design closely mimics the way we manipulate tools while working with most software.

If the user decides to import a piece of clip art, the function is still closely related to the goal of ending up with a good drawing, but the tools to be used are different and somewhat unrelated to drawing. Clip art is usually held in a directory of pre-recorded art and may include a facility for previewing and selecting the desired piece. The clip art directory is clearly not congruent with the user's goal of drawing—it is only a means to an end. The conventional artist probably does not keep a book of clip art right on his drawing board, but you can expect that it is close by, probably on a bookshelf adjacent to the drawing board and available without even getting up. In the program, the clip art facility should be very easy to access but, because it involves a whole suite of tools that aren't normally needed, should be placed in a separate facility: a dialog box.

When the user is done creating the artwork, he has now achieved his initial goal of creating an effective image. At this point, his goals change. His new goal is to preserve the picture, protect it, and communicate with it. The need for pens and pencils is over. The need for clip art is over. Leaving these tools behind now

is no hardship. The conventional artist would now unpin the drawing from his board, take it into the hall and spray it with fixative, then roll it up and put it in a mailing tube. He purposely leaves behind his drawing tools—he doesn't want them affected by fixative overspray and doesn't want accidents with paint or charcoal to mar the finished work. Mailing tubes are used infrequently and are sufficiently unrelated to the drawing process that he stores them in a closet. In the software equivalent of this process, the user ends the drawing program, puts away his drawing tools, finds an appropriate place on the hard disk to store the image, and sends it to someone else via electronic mail. These functions are clearly separated from the drawing process by the goals of the user and are well-suited to residing in their own dialog box.

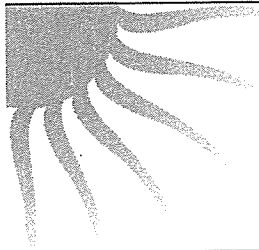
By examining the user's goals, we are naturally guided to an appropriate form for the program. Instead of merely putting every function in a dialog box, we can see that some functions shouldn't be enclosed in a dialog at all, others should be put into a dialog that is integral to the main body of the interface, and still other functions should be completely removed from the program.

Windows pollution

Some designers take the approach that each dialog box should embody a single function. It is unclear to me why they think this. What they end up with is what some call **windows pollution**.

Achieving many user goals involves executing a series of functions. If there is a single dialog box for each function, things can quickly get visually crowded and navigationally confusing. The CompuServe Navigator (Version 1.0.1) program, shown in Figure 7-1, is a case in point.

Adding a squirt of oil to my bicycle makes it pedal easier, but it doesn't mean that dumping a gallon of oil all over it will make it pedal itself. It seems to me as though the designer of Navigator was on a mission to put more windows in our lives in the mistaken belief that windows are inherently good. He certainly



A gallon of oil won't make
a bicycle pedal itself

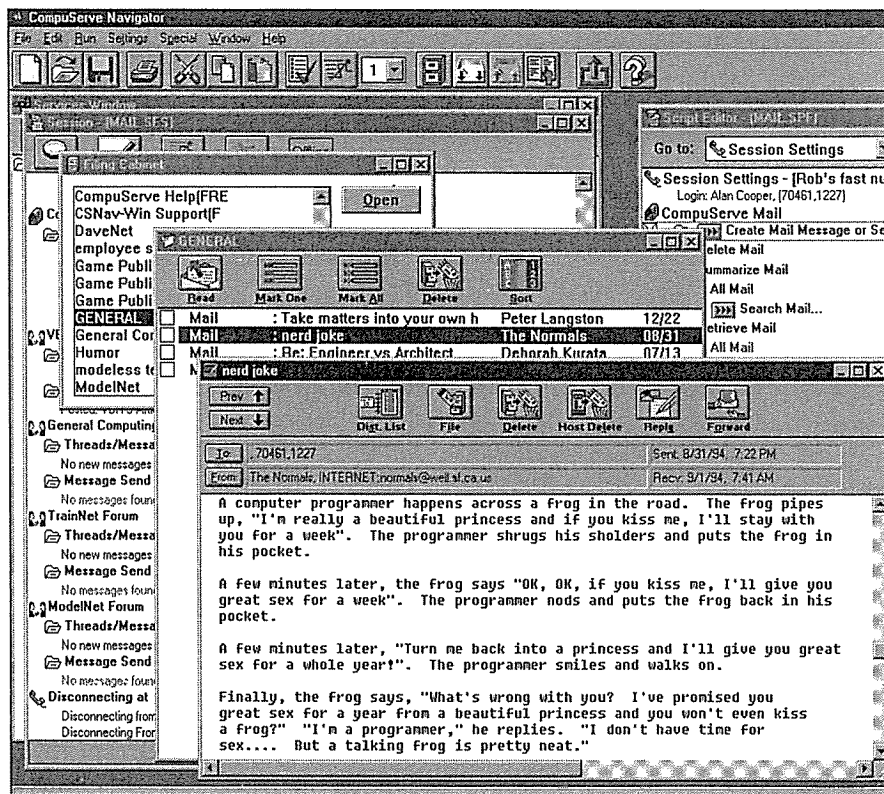


Figure 7-1

Version 1.0.1 of CompuServe Navigator suffers from tragic windows pollution. Just normal downloading of my mail requires that three windows be open. To examine a filed message demands that I open three more windows in turn. First, I get the “Filing Cabinet”; then I call up the “GENERAL” window. Finally, I can open a particular mail message in its own separate window. This is all one integral function and should occupy one integral window. But the worst is yet to come: I must put each window away separately in the reverse order of opening them.

succeeded in putting lots of windows in my life, but he didn’t make things any better.

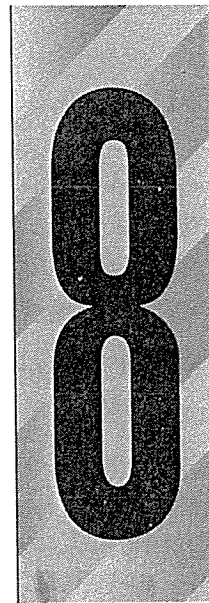
From the user’s point of view, examining a saved piece of email is not three functions, but one. One dialog would not only be perfectly sufficient to accomplish this task, it would also more closely correspond to the user’s goal of “viewing an email.” It would also correspond more closely to the user’s mental model of what is happening inside the computer. The designer has instead faithfully rendered the actual processing to the user, sort of like forcing the driver to turn two steering wheels, one for each front wheel, instead of combining the two functions into a single, conceptual whole.

A much better solution to the Navigator problem would have been to create a single “mail” box, with tools strategically positioned along the top row—a toolbar would be perfect—for managing searches. Intermediate results of the search could be shown in the window along with the final message itself. One goal—finding and reading a message—should be implemented as one dialog box.

There is no way to show the connections between lots of windows, so don’t create lots of windows. Modal dialogs, however, always get you back immediately to the point of departure, so they don’t count against you. This is a particularly annoying problem with Visual Basic (VB) where it is easy to create “forms.” Forms are independent, top-level windows. In terms of behavior, they are the same as modeless dialog boxes. Creating applications as collections of several modeless dialog boxes is a questionable strategy that was never very common until VB made it easy to do. And, as I’ve said before, just because it’s easy to do doesn’t mean it is good design.

Each added window contributes more to the user’s burden of window management excise. This overhead can grow to be really obnoxious if the program is used daily. If your program has a couple of dozen windows because you honestly feel that each of those windows moves the user towards that many different goals, then you should divide up your program into several smaller ones, each one true to its own goal. A program shouldn’t have more than two or three goals, which means it shouldn’t have more than two or three windows.

A VB programmer once explained to me proudly that his program was especially difficult to design because it had 57 forms. No program can be used effectively with 57 forms. Each form may be excellent in its own right, but collectively, it’s simply too many. It’s like saying you’re going to taste 57 vintage Chardonnays at a sitting, or test-drive 57 sedans on Saturday.



Lord of the Files

If you have ever tried to teach your Mom how to use a computer, you will know that “difficult” doesn’t really do the problem justice. Things start out all right: you fire up the word processor and key in a letter. She’s with you all the way. When you are finally done, you press the Close button and up pops that mutant ninja turtle of a dialog box asking “Do you want to save changes?” and you and Mom hit the wall together. She looks at you and asks, “What does this mean? What changes? Is everything OK?” How can you answer her?

The tragedy of the file system

The part of modern computer systems that is the most difficult to understand is the file system, the facility that stores programs and data files on disk. Telling the uninitiated about disks is very difficult. The difference between “main memory” and “disk storage” is not clear to most people. Unfortunately, the way we design our software forces users—even your Mom—to know the difference. Every program exists in two places at once: in memory and on disk. The same is true of every file, but many users never quite grasp the difference. When that “Save Changes?” dialog box,

shown in Figure 8-1, comes up, they just suppress a twinge of fear and confusion and press the YES button out of habit. A dialog box that is always answered the same way is a redundant dialog box that should be eliminated.

The Save Changes dialog box is based on a bad assumption. The very presence of the dialog assumes that saving and not saving are equally probable. The dialog gives equal weight to these two options, even though the YES button is pressed orders of magnitude more frequently than the NO button. As I discuss in Chapter 11, this is a case of putting *might* on *will*. The *user might* say no, but the *user will* almost always say yes. **VS.**

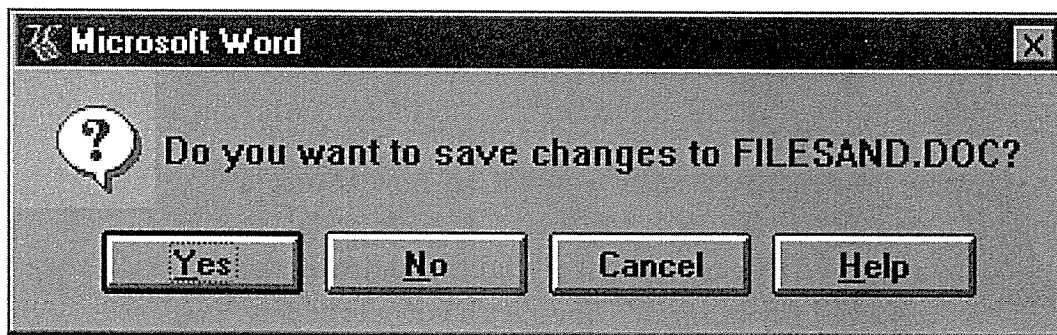


Figure 8-1

This is the question Word asks me when I close a file after I have edited it. Yes, of course I want to save it; otherwise, I wouldn't have made the changes in the first place. The origin of this dialog box is not the user's mental model, but rather the programmer's manifestation of the implementation model. In other words, the physical characteristics of the disk system are imposed on the user's work habits. This dialog is so unexpected for new users that they often say "No" inadvertently.

There is another odd thing about the dialog, and Mom will probably wonder about it. Why does it ask about saving changes when you are all done? Why didn't it ask when you actually made them? The connection between closing a document and saving changes isn't all that natural, even though we power-users have gotten quite familiar with it.

Mom is thinking "If I didn't want those changes, I would have undone them long ago." To her, the question is absurd. The program issues the dialog box when the user requests CLOSE or QUIT because that is the time when it has to reconcile the differences between the copy of the document in memory with

the copy on disk. The way the technology actually implements the facility associates changes with the CLOSE and QUIT operations, but the user doesn't naturally see the connection. When we leave a room, we don't consider discarding all of the changes we made while we were there. When we put a book back on the shelf, we don't first erase any comments we wrote in the margins.

Computer geeks are very familiar with the connection between saving changes and closing or quitting. They don't want to lose this ability because it is familiar to them, but familiarity is a really bad design rationale. We don't want to keep repairing our car just because we are familiar with the shop. We don't want to keep getting root canals just because we are familiar with the drill.

As experienced users, we have learned to use this dialog box for purposes for which it was never intended. There is no other easy way to undo massive amounts of changes, so we just use the Save Changes dialog and answer it with a NO. If you discover yourself making big changes to the wrong file, you use this dialog as a kind of escape valve to return things to the status quo.

The problems caused by disks

The computer's file system is the tool it uses to manage data and programs stored on disk. This means the big hard disks where most of your information resides, but it also includes your floppy drives and your CD-ROM if you have one. The File Manager program in Windows 3.x and the Explorer in Windows 95 graphically represent the file system. Without a doubt, the file system—and the disk storage facility it manages—is the primary cause of disaffection with computers for non-computer-professionals.



Disks and files make users crazy

Even though the file system is an internal facility that shouldn't—by all rights—even affect the user, it creates a large problem because the influence of the file system on the interface of most programs is very deep. The most intractable problems facing user interface designers usually concern the file system and its demands. It affects our menus, our dialogs, even the procedural framework of

Yes

our programs, and this influence is likely to continue indefinitely unless we make a concerted effort to stop it.

Currently, most software treats the file system in much the same way that the operating system shell does (Explorer, File Manager). This is tantamount to **LIKE** you dealing with your car the same way your mechanic does. Even though this approach is tragically bad, it is an established, de facto standard and there is considerable resistance to improving it.

Following the implementation model

Before I go any further, let me make clear that the file systems on modern personal computer operating systems, like Windows 95, are technically excellent. I have no gripe with the way they are implemented. The problem stems from the simple mistake of rendering that implementation model to the user.

The implementation model of the file system runs contrary to the mental model almost all users bring to it. In other words, they picture files or documents as typical documents in the real world, and they imbue them with the behavioral characteristics of those real objects. In the simplest terms, users visualize two salient facts about all documents: First, there is only one document, and second, it belongs to the user. The file system's implementation model violates both of these rules: There are always at least two copies of the document, and both belong to the program.

Saying that someone is "computer literate" is really a euphemism meaning that he has been indoctrinated and trained in the irrational and counter-intuitive way that file systems work, and once you have been properly subverted into thinking like a computer nerd, the obvious ridiculousness of the way the file system presents itself to the user doesn't seem so foolish.

Every data file, every document and every program, while in use by the computer, exists in a minimum of two places at once: on disk and in main memory. The user, though, imagines his document as a book on a shelf. Let's say it is a journal he is keeping. Occasionally, it comes down off the shelf to have some words added to it. There is only one journal, and it either resides on the shelf or it resides in the user's hands. On the computer, the disk drive is the shelf, and main memory is the place where editing takes place, equivalent to the user's hands. But in the computer world, the journal doesn't come "off the shelf." Instead a copy is made, and that *copy* is what resides inside the computer. As the user makes changes to the document, he is actually making changes

to the in-memory copy, while the original remains untouched on disk. When the user is done and closes the document, the program is faced with a decision. It must decide whether to replace the original on disk with the changed copy from memory. From the programmer's point of view, equally concerned with all possibilities, this choice could go either way. From the software's implementation model point of view, the choice is the same either way. However, from the user's point of view, there is rarely a decision to be made at all. He made his changes already; now he is just putting the document away. If this were happening with a paper journal in the physical world, the user would have pulled it off the shelf, pencilled in some additions, and is now replacing it on the shelf. It's as if the shelf suddenly spoke up, asking if he really wants to keep those changes!

Right now, the seriously computer-holic readers are beginning to squirm in their seats. They are thinking that I'm treading on holy ground, and a pristine copy on disk is a wonderful thing and that I had better not be advocating getting rid of it. Relax! As I said before, there is nothing wrong with our file systems. I am only advocating that we hide its existence from the user. We can still offer to him all of the advantages of that extra copy on disk without exploding his mental model. I'll show you how.

Dispensing with the disk model

If we begin to render the file system according to the user's mental model, we achieve a significant advantage. The primary one is that we can all teach our Moms how to use computers. We won't have to answer her challenging questions about the inexplicable behavior of the interface. We can show her the program and explain how it allows her to work on the document and, upon completion, she can store the document on the disk as though it ~~were~~ ^{WAS} a journal on a shelf. Our sensible explanation won't be interrupted by that "Save changes?" dialog. Not to put too fine a point on this, but I'm just using Mom as a surrogate representing the mass market of computer buyers.

The other big advantage is that software user interface designers won't have to incorporate clumsy file-system awareness into their products. We can structure the commands in our programs according to the goals of the user instead of according to the needs of the operating system.

We no longer need to call the left-most menu the "File" menu. This older nomenclature is a bold reminder of how the technology pokes through the facade of our programs. We can label this menu after the type of document we

are processing—for example, we can call it “Spreadsheet,” “Invoice,” or “Picture.” Alternatively, we can give it a more generic name like “Document,” which is a reasonable choice for horizontal programs like word processors or spreadsheets.

Changing the name and contents of the “File” menu violates an established standard. I recognize the impact of this proposal and don’t make it lightly. I have tremendous respect for standards, unless they are wrong. This one is wrong, and it’s existence makes life more difficult than it has to be for every user of computers, particularly newcomers and casual users. The benefits will far outweigh any dislocation the change might cause. There will certainly be an initial cost as experienced users get used to the new presentation, but it will be far less than you might suppose. This is because these power-users have already shown their ability and tolerance by learning the implementation model. For them, learning the better model will be a slam-dunk, and there will be no loss of functionality.

The advantage for new users will be immediate and big. We computer professionals forget how tall the mountain is once we’ve climbed it, but everyday newcomers approach the base of this Everest of knowledge we sit atop and are severely discouraged. Anything we can do to lower the height can make a big difference, and this step removes a considerable obstacle.

Designing software with the proper model

Properly designed software will always treat documents as single instances, never as a copy on disk and a copy in memory. I call this the **unified file model**.

Saving

One of the most important functions every computer user must learn is how to “save.” Invoking this function means taking whatever changes the user has made to the in-memory copy and writing them onto the on-disk copy of the document. In the unified model, we abolish all user-interface recognition of the two copies, so the “save” function disappears completely from the mainstream interface. Of course, that doesn’t mean that it disappears from the program. It is still a very necessary operation.

The program will automatically save the document. At the very least, when the user is done with the document and requests the close function, the program will merely go ahead and write the changes out to disk without stopping to ask for confirmation with the “Save Changes” dialog box.

In a perfect world, that would be enough, but computers and software can crash, power can fail, and other unpredictable, catastrophic events can conspire

to erase your work. If the power fails before you have pressed CLOSE, all of your changes will be lost as the memory containing them scrambles. The original copy on disk will be all right, but hours of work can still be lost. To keep this from happening, the program must also save the document at intervals during the user session. Ideally, the program will save every single little change as soon as the user makes it. In other words, after each keystroke. For most programs on modern computers, this is quite feasible. Only certain programs—word processors leap to mind—would find difficulty with this level of saving (but the solution would still not be impossible). Most documents can be saved to the hard disk in just a fraction of a second, so generally this is not a problem. Still, this is a sensitive area, because the program will hesitate noticeably in a very disturbing way. Word has a facility for automatically saving files to disk, and I never use it for that reason. The problem is caused by the save facility's logic, not because the principle of automatic saving is bad. Word automatically saves the file according to a countdown clock, and you can set the delay to any number of minutes. If you ask for a save every two minutes, for example, after precisely two minutes the program stops accepting your input to write your changes out to disk, regardless of what you are doing at the time. If you are typing when the save begins, it just clamps shut in a very realistic and disconcerting imitation of a broken program. It is a very unpleasant experience. If the algorithm would pay attention to the user instead of the clock, the problem would disappear. Nobody types continuously. Everybody stops to gather his thoughts, or flip a page, or take a sip of coffee. All the program needs to do is wait until the user stops typing for a couple of seconds and *then* save.

This automatic saving every few minutes and at CLOSE time will be adequate for everybody except the really twisted computer-freaks who have been using computers since Bill Gates was just a thousandaire. I include myself in this group. I'm so paranoid about my computer crashing and losing data that I habitually press the CTRL-S key after every paragraph I type, and sometimes after every sentence. (Pressing CTRL-S is the keyboard accelerator for the SAVE function.) I'll typically save a document—like a chapter in this book—more than 1,000* times before it's done! There is no way in the world I would even use a program that didn't provide such manual save capabilities, and all programs should have them. I just don't think that my compulsive behavior should be forced on new or occasional users who are writing the occasional letter or spreadsheet and haven't begun writing a book yet.

* Using the revision number feature of Microsoft Word, I print the exact number of saves at the bottom of all of my drafts. I'm not exaggerating.

Right now in Word, the SAVE function is prominently placed in-your-face on the primary program menu. The SAVE dialog is forced on all users when they ask to close the document or to QUIT or EXIT the program. These artifacts must go away, but the SAVE functionality can remain in place exactly as it is now.

Closing

There is no inherent connection between closing and saving in my unified model because there is no concept of saving.

We computer geeks are conditioned to think that CLOSE is the time and place for abandoning unwanted changes if we made some error or were just what-if-ing. This is not correct, though, because the proper time to reject changes is when the changes are made. We even have a well-established idiom to support this. The UNDO function is the proper facility for eradicating changes. We have bent and contorted our thinking so much to accommodate the implementation model that I often hear people bleat in protest over losing the ability to refuse a request to “save changes.”

In Chapter 30, “Undo,” I’ll talk about some more sophisticated variants of undo that allow us to create multiple versions of a document. Currently, savvy computer users who understand the technology can accomplish this by working cleverly with the file system. A better interface could offer these desirable features directly and explicitly.

When you answer YES to the Save changes dialog, virtually every program then presents you with the “Save As” dialog box. A typical example is shown in Figure 8-2.

Neither the typical user nor the unified file model recognizes the concept of manual saving, so, from their point of view, the name of this dialog box doesn’t make much sense. Functionally, this dialog offers the user two things. It lets you name your file, and it lets you choose which directory you wish to place it in. Both of these functions demand intimate knowledge of the file system. The user must know how to formulate a file name and how to navigate through the directory tree. I know of many users who have mastered the name portion but who have completely given up on understanding the directory tree. They put all their documents in whatever directory the program chooses for a default. All of their files associated with a particular program are stored in a single directory. Occasionally, some action will cause the program to forget its default directory, and these users must call in an expert to find their files for them. My next door neighbor, Bill, calls me about every six months to help him find his Lotus 1-2-3

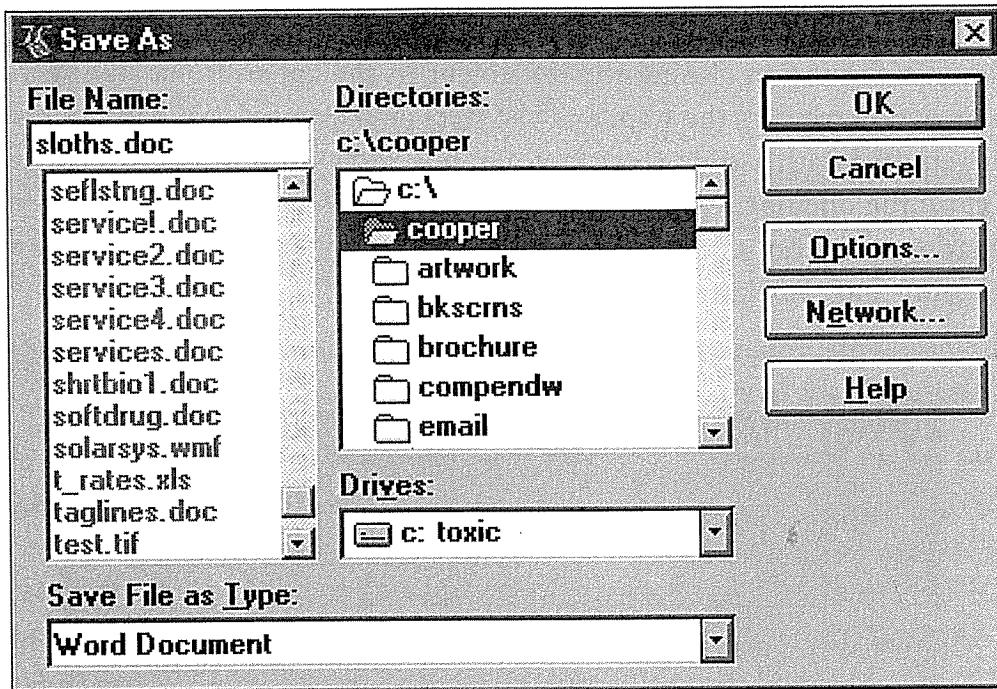


Figure 8-2

The Save As dialog box offers two functions: It lets you name your file, and it lets you place it in the directory of your choice. From the user's perspective, remember, he has no concept of "saving," so the name of this dialog is incorrect. Also, if a dialog enables me to name and place a document, shouldn't it also allow me to rename and replace the document? I certainly think so.

files. The first time he called, I asked him where he normally keeps his spreadsheets. He answered innocently "In 1-2-3." Bill's mental model is very different from the software's implementation model and, ultimately, Bill is right.

The Save As dialog needs to decide what its purpose is. If it exists to name and place files, then it does a very bad job of it. Once the user has named and placed a file, he cannot then change its name or directory. At least he can't with this dialog that purports to offer naming and placing privileges. Nor with any other tool in the application itself.

Beginners are out of luck, but experienced users learn the hard way that they can close the document, change to the Explorer, rename the file, return to the application, summon the Open File dialog, and reopen the document. In case you were wondering, the Open File dialog doesn't allow renaming or repositioning either.

Forcing the user to go to the Explorer to rename the document is a minor hardship, but therein lies a hidden trap; its teeth sharp and its spring strong. The

bait is the fact that Windows easily supports several applications running simultaneously. Attracted by this feature, the user tries to rename the file in the Explorer without first closing the document in the application. This very reasonable action triggers the trap, and the steel jaws clamp down hard on his leg. He is rebuffed with a rude error message box shown in Figure 8-3. He didn't first close the document—how would he know? Trying to rename an open file is a sharing violation, and the operating system summarily rejects it with a truly frightening and unhelpful error message box.

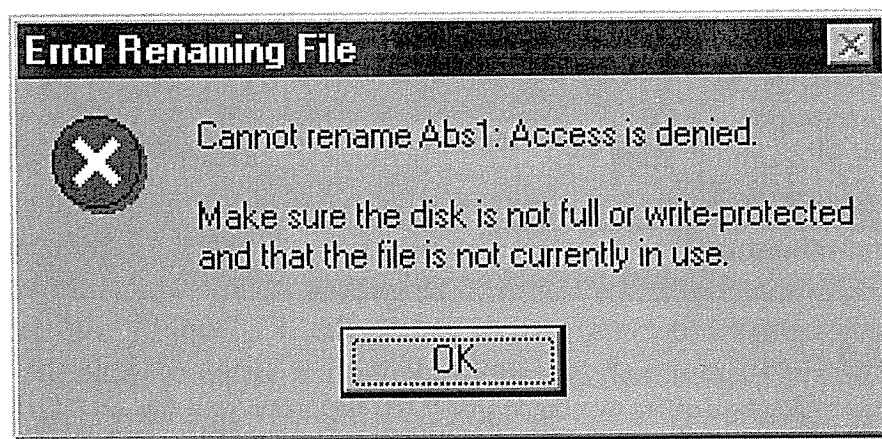


Figure 8-3

If the user attempts to rename a file using the Explorer while it is still being edited by Word, the Explorer is too stupid to get around the problem and make it work. It is also too stupid to figure out what happened so it can report it correctly. It is also too rude to be nice about it, and it puts up this frightening error message box. Rebuffed by both the program and the Explorer, it is easy for a new user to imagine that a document cannot be renamed at all.

The innocent user is merely trying to name his document, and he finds himself lost in an archipelago of operating-system arcana. Ironically, the one program that has both the authority and the responsibility to change the document's name while it is still open is the application itself, yet it refuses to even try.

Archiving

There is no explicit function for making a copy of, or archiving, a document. The user must accomplish this with the Save As dialog, and doing so is as clear as mud. Even if there were a "Copy" command, users visualize this function in different ways. If we are working, for example, on a document called "Alpha,"

some people imagine that we would create a file called “Copy of Alpha” and store it away. Others imagine that we put Alpha away and continue work on Copy of Alpha.

I suspect that the latter option will only occur to those who are already experienced with the implementation model of file systems. That is, of course, how we would do it today with the Save As dialog: you have already saved the file as Alpha; then you explicitly call up the Save As dialog and change the name. Alpha will be closed and put away on disk, and the new copy will be the version being edited. This action makes very little sense from the single-document viewpoint of the world, and it also offers a really nasty trap for the user.

Here is the completely reasonable scenario that leads to trouble: Let’s say that I have been editing Alpha for the last twenty minutes and now wish to make an archival copy of it on disk so I can make some big, but experimental, changes to the original. I call up the Save As dialog box and change the file name to “New Alpha.” The program puts Alpha away on disk leaving me to edit New Alpha. Ahhh, but Alpha was never “Saved,” so it gets written to disk without the changes I made in the last twenty minutes! Those changes only exist in the “New Alpha” copy that is currently in memory—in the program. As I begin cutting and pasting in New Alpha—trusting that my handiwork is backed up by “Alpha”—I am actually trashing the sole copy of this information.

Everybody knows that you can use a hammer to drive a screw or a pliers to bash in a nail, but any skilled craftsperson knows that using the wrong tool for the job will eventually catch up with you. The tool will break or the work will be hopelessly ruined. The Save As dialog is the wrong tool for making and managing copies, and it is the user who will eventually have to pick up the pieces caused by the developer’s laziness.

Unify the file model

The application program refuses to rename and reposition the file out of respect for the file system. The file system is the facility whose job it is to manage information that is not in main memory, and it does so by maintaining a second copy on disk. This method is correct, but it is an implementation detail that only confuses the user. Application software should conspire with the file system to hide this unsettling detail from the user.

If the file system is going to show the user a file that cannot be changed because it is in use by another program, the file system should indicate this to the user.

Showing the file name in red, or with a special icon next to it would be sufficient. A new user might still get that awful message in Figure 8-3, but at least some visual clues would be present to show him that there is a *reason* why that error cropped up.

Not only are there two copies of all data files but, when they are running, there are two copies of all programs. When I go to the Startbar's Start menu and launch my word processor, a button corresponding to Word appears on the Startbar. But if I return to the Start menu, Word is still there! From the user's point of view, I have pulled my hammer out of my toolbox, only to find that my hammer is still in my toolbox!

I'm not saying that this should not be the case. After all, one of the great strengths of the computer is its ability to have multiple copies of software running simultaneously. I do think that the software should help the user to understand this very non-intuitive action, however. Maybe the Start menu could make some reference to the already-running program.

Document management

The established standard suite of file management for most applications consists of the Save As dialog, the Save Changes dialog, and the Open File dialog. Collectively, these dialogs are, as I've shown, confusing for some tasks, and completely incapable for others. Here is how I would design an application that really managed documents according to the user's mental model.

Besides rendering the document as a single entity, there are several goal-directed functions that the user may need, and each one should have its own corresponding function.

- Creating a copy of the document
- Creating a milestone copy of the document
- Naming and renaming the document
- Placing and repositioning the document
- Specifying the stored format of the document
- Reversing some changes
- Abandoning all changes

Creating a copy of the document

This should be an explicit function called “Make Snapshot Copy.” The word “snapshot” makes it clear that the copy is identical to the original, while also making it clear that the copy is not tied to the original in any way. That is, subsequent changes to the original will have no effect on the copy. The new copy should be given a name with a standard form like “Copy of Alpha,” where “Alpha” is the name of the original document. If there is already a document with that name, the new copy should be named “Second Copy of Alpha.” The copy should be placed in the same directory as the original.

It is very tempting to envision the dialog box that accompanies this command, but there should be no such interruption. The program should take its action quietly and efficiently and sensibly, without badgering the user with silly questions. Make a copy. In the user’s mind, it is a simple command. If there are any anomalies, the program should make a constructive decision on its own authority.

Naming and renaming the document

The name of the document should be shown right on the application’s toolbar. If the user decides to rename the document, he can just click on it and edit it in place. What could be simpler and more direct than that?

Placing and repositioning the document

Most documents that are edited already exist. They are opened, rather than created from scratch. This means that their position in the file system is already established. Although we think of establishing the home directory for a document at either the moment of creation or the moment of first saving, neither of these events is particularly meaningful outside the implementation model. The new file should be put somewhere reasonable where the user can find it again.

If the user wants to explicitly place the document somewhere in the file-system hierarchy, he can request this function from the menu. A relative of the Save As dialog appears with the current document highlighted. The user can then move the file to any desired location. Essentially, all files are placed automatically by the program, and this dialog is used only to reposition them.

Specifying the stored format of the document

There is an additional function implemented on the Save As dialog in Figure 8-2. The combobox at the bottom of the dialog allows the user to specify the physical format of the file. This function should not be located here. By tying the physical format to the act of saving, the user is confronted with additional, unnecessary complexity. Saving should be a very simple act. In Word, if the user innocently changes the format, both the save function and any subsequent close action are accompanied by a frightening and unexpected confirmation box.

From the user's point of view, the physical format of the document—whether it is rich text, ASCII, or Word format, for example—is a characteristic of the document rather than of the disk file, so specifying the format shouldn't be associated with the act of saving the file to disk. It belongs more properly in a "Document Properties" dialog.

Overriding the physical format of a file is a rare occurrence. Saving a file is a very common occurrence. These two functions should not be combined.

The physical format of the document should be specified by way of a small dialog box callable from the main menu. This dialog box should have cautions built into its interface to make it clear to the user that the function can involve significant data loss.

Reversing some changes

If the user inadvertently makes changes to the document that must be reversed, the tool already exists for correcting these actions: undo. The file system should not be called in as a surrogate for undo. The file system may be the mechanism for supporting the function, but that doesn't mean it should be rendered to the user in those terms. The concept of going directly to the file system to undo changes merely undermines the undo function.

The milestone function (description follows) tells how a file-centric vision of undo can be implemented so that it works well with the unified file model.

Abandoning all changes

It is not uncommon for the user to decide that he wants to discard all of the changes he has made since opening or creating a document, so this action should be explicitly supported. Rather than forcing the user to understand the file system to achieve his goal, a simple "Abandon" function on the main menu

would suffice. Because this function involves significant data loss, it should be protected by clear warning signs. Additionally, making this function undoable for a week or two would be relatively easy to implement and appreciated more than you might imagine.

Creating a milestone copy of the document

Making a milestone is very similar to using the copy command.

The difference between them is that the milestone copy is managed by the application after it is made. The user can call up a “Milestone” dialog box that lists each milestone copy along with various statistics about it, like the time it was recorded and its length. With a click, the user can select a milestone copy and, by doing so, immediately return to it as the active document. The version that was current at the time of the reversion will be milestone itself, for example, under the name “Displaced by Milestone of Alpha 12/17/97, 13:53.”

The new menu

Our new File menu now looks like the one shown in Figure 8-4.

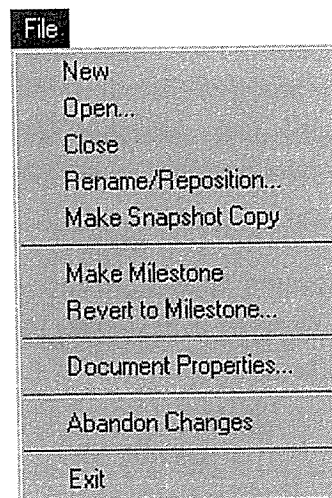


Figure 8-4

The revised File menu now reflects the user’s mental model instead of the programmer’s implementation model. There is only one file and the user owns it. If he wants, he can clone it, discard any changes he has made to it or change its format. He doesn’t have to worry about the copy in RAM and the copy on disk.

“New” and “Open” function as before, but “Close” will just quietly close the document without a dialog box or any fuss, after assuring that it is completely saved. “Rename/Reposition...” brings up a small dialog box that lets the user rename the current file and/or move it to another directory. “Make Snapshot Copy” quietly creates a new file that is a copy of the current document. “Make Milestone” does the same thing, except that the program manages these copies by way of the dialog box summonable with the “Revert to Milestone” item. “Document Properties” also brings up a dialog box that lets the user change the physical format of the document. The final item is “Abandon Changes” and it discards all changes made to the document since it was opened or created.

File menu?

Of course, now that we are manifesting a monolithic model of storage instead of the bifurcated implementation model of disk and RAM, we no longer need to call the left-most menu the “File” menu. This older nomenclature is a bold reminder of how the technology has been inflicted on the user instead of the user’s model being reflected in the technology. There are two pretty good alternatives to solving this problem.

As I said earlier, we can label the menu after the type of document we are processing. For example, a spreadsheet might label its left-most menu “Sheet.” An invoicing program might label it “Invoice.” I designed a patent management program for a client, and in that program we called it “Patent.”

Alternatively, we can give the left-most menu a more generic label like “Document.” This is certainly a reasonable choice for broad programs like word processors and spreadsheets, but is less appropriate for narrower programs like the patent manager.

Conversely, those few programs that do represent the contents of disks as files—generally operating system shells and utilities—*should* have a “File” menu, because they are addressing files with a studied ignorance of their contents.

How did we get here?

If you are still not convinced that disks and their file system are the cause of great user interface confusion, I’d like to show how our disks came to have such a profound effect on our software.

From the user's point of view, there is no reason for disks to exist. From the computer engineer's point of view, there are three:

1. Disks are cheaper than solid-state memory. ~~\$\$\$-CASP~~
2. Once written to, disks don't forget when the power is off. ~~NVSRAM~~
3. Disks provide a physical means of moving information from one computer to another. ~~XPORTABLE~~

Reasons number two and three are certainly useful but are not the exclusive domain of disks. Other technologies work as well or better. There are varieties of RAM that don't forget their data when the power is turned off. CMOS memory is solid state, yet it retains its setting without external power.

Networks and phone lines can be used to physically transport data to other sites, often more easily than with removable disks.

Reason number one—~~cost~~—is the *real* reason why disks exist. CMOS is a lot more expensive than disk drives. Reliable, high-bandwidth networks haven't been around as long as removable disks, and they are still more expensive.

Disk drives have many drawbacks when compared to RAM. Disk drives have always been much slower than solid-state memory. They are much less reliable, too, since they depend on moving parts. They consume more power and can take up more space, as well. The real whammy when it comes to disks, though, is that computers, the actual CPU, can't really read or write to them! Data must first be brought into main, solid-state memory by the CPU's helpers before the CPU can work with it. When the processor is done, the helpers must once again step in to move the data back out to the disk! This means that all processing involving disks is necessarily orders of magnitude slower and more complex than working in plain RAM.



Disks are a hack,
not a design feature

The time and complexity penalty for using disks is so severe that nothing short of enormous cost-differential could compel us to rely on them. Disk drives are

a cost-saving hack. Mind you, there is nothing wrong with using this sophisticated technology to save money, but keep in mind that the technology isn't there to provide us with services we couldn't get in other ways. This means that any changes we make to our interfaces to adjust to the disk technology are likely to be inappropriate from a goal-directed point of view.

So we can see that disks are not a law of nature; they are not architectural features that make computers better, more powerful, faster or easier to use. Instead, they make computers weaker, slower and more complex. They are a compromise, a dilution, an adulteration, a corruption of the pure architecture of digital computers. If early computer designers could have economically used RAM instead of disks, they would have done so without hesitation. Whatever other problems RAM exhibited could have been overcome with technologies simpler than the complexity of disk drives.

The difference between RAM and disk is merely a matter of economics, much like the way you go to a lending library instead of personally owning copies of every book. This means that wherever disk technology has left its mark on the design of our software, it has done so purely for implementation purposes and not for any goal-directed design rationale. While this difference should be of interest only to programmers, in reality, it is imposed on nearly every program and users are forced to master it. Any construction that supports disks is for the convenience of the programmer and the computer, and not to help the user.

The pervasiveness of the file system in our thinking and our design of software is as though refrigeration technology dominated the design of every room in our houses. Certainly, the invention of cheap mechanical refrigeration affected our domestic lives, but we don't turn our houses into shrines to Freon. Yet this is largely what we have done on our desktop computers.

It is one thing to weave a technology invisibly into our lives. It is another thing altogether to allow our lives to be dominated by that technology. Refrigeration plays a big part in our lives in many ways, including food preparation, the production and storage of some medicines and air conditioning, yet we don't usually find ourselves expressing our desires in terms of it. We don't go into a restaurant and say, "I'll have the salmon. It's been refrigerated, hasn't it?" We don't say, "You'll love working here, it's air conditioned." Omnipresent technologies don't have to intrude on our conscious thoughts to work well for us. Unfortunately, this realization hasn't yet dawned on the computer industry, and we remain sadly dependent on the file-system model.

The last gasp

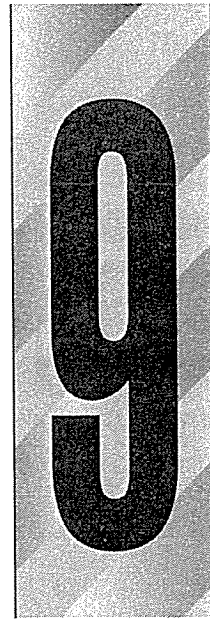
There are only two arguments that can be mounted in favor of application software implemented in the file-system model: Our software is already designed and built that way, and users are used to it.

Neither of these arguments holds water, though. The first one is irrelevant because new programs written with a unified file model can freely coexist with the older implementation model applications. The underlying file system doesn't change at all. In much the same way that toolbars have invaded the interfaces of most Windows applications in the last few years accompanied only by cheers and encouragement, the unified file model could also be implemented.

The second argument is more insidious, because its proponents are placing the user community in front of them like a shield. What's more, if you ask the users themselves, they will reject the new solution because they abhor change, particularly when that change affects something they have already worked hard to master—like the file system. In the '80s, the Chrysler company showed car-buyers early sketches of a dramatically new automobile design: the minivan. The buyers were asked if they would be interested in this new vehicle, and the public uniformly gave a thumbs-down to the new design. Chrysler went ahead and produced the Caravan anyway, convinced that the design was superior. They were right, and those same people who rejected the design have not only made the Caravan the best-selling minivan, but have made the minivan the most popular new automotive archetype since the convertible.

People will gladly give up painful, poorly designed software for easier, better software even if they don't understand the explanations. After all, users aren't software designers and they cannot be expected to visualize the larger effect of the change. Saying that users want to keep their familiar file-system model is like saying you want to break your leg again so you can return to the hospital because the food was so good the last time you were in there.

Storage and Retrieval Systems



The document is a well-established concept in the mechanical world. It is an object that can be read by those who care to, and it often can be manipulated with writing or drawing instruments. Beyond that, a document can be transported, owned and stored. These latter qualities exist even if the former do not. In other words, I can hold or own a book on calculus even though I have never learned calculus. Our disk file systems are not so forgiving. A program can do little with a PowerPoint file, for example, unless it is intimately familiar with how to process PowerPoint slides. Our document-centric systems are really just file-centric systems and are harder to understand and use—and in some ways less powerful—than our manual systems.

Storing versus finding

In the physical world, storing and retrieving an item are inextricably linked; putting an item on a shelf (storing it) also gives us the means to find it later (retrieving it). In the digital world, the only thing linking these two concepts is our faulty thinking. Computers will enable remarkably sophisticated retrieval techniques if only we break our thinking out of its traditional box.

A **storage system** is a tool for placing goods into a repository for safekeeping. It is composed of a physical container and the tools necessary to put objects in and take them back out again.

A **retrieval system** is a method for finding goods in a repository. It is a logical system that allows the goods to be located according to some abstract value, like its name, position or some aspect of its contents.

As we saw in the last chapter, disks and files are usually rendered in implementation terms, rather than in accord with the user's mental model of how information is stored. This is even more true in the methods we use for *finding* information after it has been stored. This is extremely unfortunate, because the computer is the one tool capable of providing us with significantly better methods of finding information than is physically possible from mechanical systems.

In the real world of books and paper on library shelves, we have at least three indices: author, subject and title. Although our desktop computers can handle hundreds of different indices, we ignore this capability and have no indices at all pointing into the files stored on our disks. Instead, we have to remember where we put our files and what we called them before we can find them again. This omission is one of the most destructive backward steps in modern software design. This failure can be attributed to the interdependence of files and the organizational systems in which they exist, an interdependence that doesn't exist in the mechanical world.

We can own a book or a hammer without giving it a name or a permanent place of residence in our houses. A book can be identified by characteristics other than a name—a color or a shape, for example. Even if we do assign a “proper place” for a physical tool, it often resides away from that place for stretches of time. A volume may properly reside on our bookshelf, but when it is being read, it may be left on night stands and coffee tables, or stuffed into briefcases or purses, and it still serves us well. Of course, these places merely act as temporary locations for the book.

For the book or the hammer, it is important that there be a proper place for them, because that is how we find them when we need them. We can't just whistle and expect them to find us; we must know where they are, then go there and fetch them. In the physical world, the actual location of a thing is the means to finding it. In the real world, where the systems of storage and retrieval are the same, remembering where we put something—its address—is vital both

to putting it away and to finding it again. When we want to find a spoon, for example, we go to the place where we keep our spoons. We don't find the spoon by referring to any inherent characteristic of the spoon itself. Similarly, when we look for a book, we either go to where we left the book, or we guess that it is stored with other books. We don't find the book by association. That is, we don't find the book by referring to its contents.

Retrieval methods

There are three fundamental ways to find a document. You can find it by remembering where you left it, which I call **positional retrieval**. You can find it by remembering its identifying name, which I call **identity retrieval**. The third method, which I call **associative retrieval**, is based on the ability to search for a document based on some inherent quality of the document itself.

For example, if I wanted to find a book with a red cover, or one that discusses light rail transit systems, or one that contains photographs of steam locomotives, or one that mentions Theodore Judah, the method I must use is associative.

Both positional and identity retrieval are methods that also function as storage systems. Associative retrieval is the one method that is not also a storage system. If our retrieval system is based solely on storage methods, we deny ourselves any associative searching and we must depend on the user's memory. He must know what information he wants and where it is stored in order to find it. To find the spreadsheet in which he calculated the amortization of his home loan, he has to know that he stored it in the directory called "home" and that it was called "amort1." If he doesn't remember either of these factoids, finding the document can become quite difficult.

The document and the system it lives in

In the physical world, a complex case like a library might have many thousands or millions of objects to store. To handle this, we assign books proper places somewhere on the shelves and then concoct other schemes for finding them based on some associative value: a characteristic of the book itself.

A book doesn't have to have a place on a shelf in order to exist. Books and the physical systems we store them in, shelves, are not physically dependent on each other. The book can just as easily exist without participating in any storage system.

A file on a disk, on the other hand, is not separate from the organizational structure of its filing system. What defines that file is not its contents but its presence in the filing system. *A disk file cannot exist outside of the filing system in which it lives.*

We can own, read and pass a book between us without ever entering it into a book filing system such as the Dewey Decimal system or a specific library. In order to own, read or pass on a computer “document,” it must first be entered into the computer’s file system.

There is no such concept as a collection of data—a document—other than as a participant in the host file system. The file systems in Windows, DOS, Macintosh and UNIX are the same in this respect: None support the existence of independent documents, only the existence of files tied intimately to their storage systems.

An independent book or document in the physical world doesn’t need to have any identifying information; its physical presence is sufficient. Usually each book or document is given a title, but this is not a requirement for its existence. In order to be stored in a manual or electronic filing system, however, it must have a unique identifier (usually its name, though bigger collections require more specific identifiers).

Indexing

In libraries, where names can be too disparate or insufficiently unique or otherwise confusing, each book is also assigned a unique serial number, called a Dewey Decimal number. The book is then stored in sequence according to this number. This numbering scheme is very convenient for storing the books but, by itself, doesn’t help in their retrieval. For that, we need a separate index: the traditional card catalog.

Libraries usually provide three indices: author, subject and title. Each index is associative, allowing the user to find the book according to an inherent property of the book other than its identifying number or its location on the shelf. When the book is entered into the library system and assigned a number, three index cards are created for the book, including all particulars and the serial number. Each card is headed by either the author’s name, the subject or the title. These cards are then placed in their respective indices in alphabetical order. When you want to find a book, you look it up in one of the indices and find its number. You then find the row of shelves that contain books with

numbers in the same range as your target, by examining signs. You then search those particular shelves, narrowing your view down by the lexical order of the numbers until you find the one you want.

You *actually, physically* retrieve the book by participating in the system of storage, but you *conceptually, logically* find the book you want by participating in a system of retrieval. The shelves and numbers are the storage system. The card indices are the retrieval system. You identify the desired book with one and fetch it with the other. In a typical university or professional library, customers are not allowed into the stacks. As a customer, you identify the book you want by using only the retrieval system. The librarian then fetches the book for you by participating only in the storage system. The unique serial number is the bridge between these two interdependent systems. In the physical world, both the retrieval system and the storage system may be very labor intensive. Particularly in older, non-computerized libraries, they are both inflexible. Adding a fourth index based on acquisition date, for example, would be prohibitively difficult in the library.

Conversely, it's not all that hard to add an index in the computer. Ironically, in a system where easily implementing dynamic, associative retrieval mechanisms is at last possible, we often don't implement *any* retrieval system. Astonishingly, we don't use indices at all.

In most of today's computer systems, there is no retrieval system other than the storage system. If you want to find a file on disk, you need to know its name and its place. It's as if we went into the library, burned the card catalog, and told the patrons that they could easily find what they want by just remembering the little numbers painted on the spines of the books. We have put 100% of the burden of file retrieval on the user's memory while the CPU just sits there idling, executing billions of NOP instructions.

An associative retrieval system

We have rendered the retrieval system in strict adherence to the implementation model of the storage system, ignoring the power and ease-of-use of a system for *finding* files that is distinct from the system for *keeping* files.

An associative retrieval system would enable us to find documents by their contents. For example, we could find all documents that contain the text string "superelevation." For such a search system to really be effective, it should know where all documents can be found, so the user doesn't have to say "Go look in

such-and-such a directory and find all documents that mention superelevation.” This system would, of course, know a little bit about the domain of its search, so it wouldn’t try to search the entire Internet, for example, for “superelevation” unless we insisted.

An associative retrieval system would also help the user create temporary or permanent groups of documents and use them as the basis for searches. For example, I frequently like to search for passages in the manuscript for this book, which is stored as dozens of small text files. I would like to first search for all documents containing the phrase “About Face” and have the program remember that set of files as the *book* set. Then, when I wanted to find the discussion of associative file retrieval systems, I could search the *book* set for occurrences of the phrase “associative” and gain the performance advantage of a restricted search without knowing anything about where my chapters were physically stored.

A well-crafted associative retrieval system would also enable the user to browse by synonym or related topics or by assigning attributes to individual documents. The user can then dynamically define sets of documents having these overlapping attributes. For example, imagine a consulting business where each potential client is sent a proposal letter. Each of these letters is different and is naturally grouped with the files pertinent to that client. However, there is a definite relationship between each of these letters because they all serve the same function: proposing a business relationship. It would be very convenient if a user could find and gather up all such proposal letters while each one can still retain its uniqueness and association with its particular client. A file system based on place—on its single storage location—must, of necessity, store each document by a single attribute rather than multiple characteristics.

The system can learn a lot about each document just by keeping its eyes and ears open. If the associative retrieval system remembered some of this information, much of the setup burden on the user would be made unnecessary. The program could, for example, easily remember such things as

- The program that created the document
- The type of document: words, numbers, tables, graphics
- The program that last opened the document
- If the document is exceptionally large or small

- If the document has been untouched for a long time
- The length of time the document was last open
- The amount of information that was added or deleted during the last edit
- Whether the document has been edited by more than one type of program
- Whether the document contains embedded objects from other programs
- Whether the document was created from scratch or cloned from another
- If the document is frequently edited
- If the document is frequently viewed but rarely edited
- Whether the document has been printed and where
- How often the document has been printed, and whether changes were made to it each time immediately before printing
- Whether the document has been faxed and to whom
- Whether the document has been emailed and to whom

The retrieval system could find documents for the user based on these facts without the user ever having to explicitly record anything in advance. Can you think of other useful attributes the system could remember?

There is nothing wrong with the disk file storage systems we have created for ourselves. The only problem is that we have failed to create disk file *retrieval* systems. Instead we hand the user the storage system and call it a retrieval system. This is like handing him a bag of groceries and calling it a gourmet dinner. There is no reason to change our file storage systems. The UNIX model is fine. Our programs can easily remember the names and locations of the files they have worked on, so they aren't the ones who need a retrieval system: That's just for us human users.

It ain't document-centric

The purveyors of GUIs, Microsoft Windows included, often allow themselves the conceit that we have a “document-centric” view of the world. It would be more accurate to say that we have a “file-centric” view of the world. Our so-called documents behave exactly like files and not much like documents.

When software vendors claim to have a “document-based” product, I interpret it to mean that their software supports documents independent of the supporting file system. None of the software I have seen does this.

Some programs, like those in Microsoft’s Office suite, implement an associative searching system that operates outside of, and in parallel to, the normal file system, but it doesn’t replace the need to work within the file system. Microsoft’s solution is weak because it still demands so much advance effort by the user.

In a document-centric world, documents are naturally at the center of things, and are independent of any particular program. Instead of *Word* documents or *WordPerfect* documents or 1-2-3 documents, we would have generic documents that could be worked on by any spreadsheet or word processor program.

Of course, vendors have developed a myriad of proprietary file formats that make exchanging data problematic. But the divergence of file formats is an effect, not a cause, of the failure of document-centricity. The file systems of our popular operating systems have so punted on the issue of retrieval (and management) of documents, that vendors felt unconstrained to use any kind of common form or format ... even on UNIX which actually did have a common format: ASCII. The only elements that remain common from file to file are those two lowest common denominator retrieval tools that are part of the storage system, too: name and position.

It isn’t even necessary for a company to abandon its own custom file formats. In just the same way that I can hold and own a book written in German—even though I can’t read German—WordPerfect should be able to own and hold a 1-2-3 file without necessarily having the ability to read it.

In a document-centric world, applications would be less monolithic. Instead of a giant word processor with hundreds of built-in functions, we’d have programs with more tightly targeted feature sets: chartwriters and graphwriters and tablewriters and CADwriters and animationwriters. In fact, we would find that programs could get even smaller and more specialized, yet still work well together. Imagine a heterogeneity of inventive tools like pencils, inks, erasers, animators, sound recorders, fonts, undo-ers, margin controllers, spraypainters and rubber stamps that could be freely applied to any of our documents. We wouldn’t have to wait for Microsoft or WordPerfect to think of it and decide to include it in the next release of their program. Nor would we be constrained to work on words in one program and images in another. We could combine these tools in one program based on our work habits rather than on one vendor’s

specialties. We would buy each tool from a different vendor, choosing the one whose product was best for the desired function. The result would be a program containing all of our favorite tools, all working together the way we want them to. We wouldn't be forced to use the tools from someone else's toolbox.

A utopian vision?

For this happy situation to occur, we'd have to have a standard document format independent of any one particular program. This would mean that the industry would have to reach a general agreement on the characteristics of a document—not an easy task in our competitive business world, where each player thinks the world should rally around its particular flag. SGML is an emerging standard that many vendors have adopted. It is gaining momentum as a common format, and this is a significant contribution to the industry. It may even grow into the utopian vision someday. Actually, we have an excellent model of an independent document standard in the UNIX world where streaming ASCII files are considered a generic, common file format that hundreds of programs know how to read, process and write.

In UNIX, any program can read or write an ASCII file regardless of which program created the file. The format of the file is common, rather than proprietary. UNIX is justifiably famous for the benefits of this standard. Programs are smaller and more powerful because they can concentrate on the function they do best. The system is egalitarian and open, and the suite of available tools comes from a wide variety of sources, both commercial and non-profit. Streaming ASCII files on UNIX are a model of what a true document-centric environment can produce.

Unfortunately, streaming ASCII is a pretty weak file format. It is a lowest-common-denominator format, lacking an internal structure of any kind. Vendors, in their endless quest to achieve a market edge with their product, abandon standards and create files in a proprietary format, but this has the effect of removing them from the ranks of open systems. From that point on, if they want to add functionality to their system, they must do it themselves and they will not be able to count on competitors adapting to their format. They have pretty-much closed off the avenue for third party add-ons.

The bottom line is whether a vendor owns the file format or if it is a common format owned by no one in particular. If the format is common, a document-centric architecture exists. If the format is proprietary, it is not document-centric. The issue hinges on the ownership of files. If a program

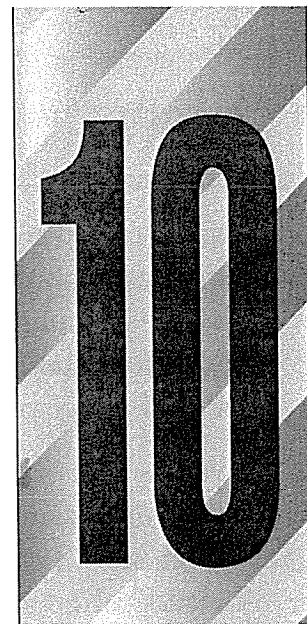
“owns” a file because of its format, the system is closed. According to this definition, only SGML ranks as a document-centric design. Almost every application currently running on Windows uses proprietary file formats, including all of those from Microsoft. We have seen over the years how open systems thrive. The only closed systems that avoid a swift and painful death in the open marketplace are those which can offer significantly better value than the competition. This is why Microsoft is working towards a common document architecture with its OLE standard.

Unfortunately, OLE is just a baby-step in this direction, and it comes with some significant flaws. In particular, OLE doesn't address the file-ownership problem. With OLE, other objects can be embedded in a document, or it can be embedded in others, but it remains strictly cast according to its type—its owner. OLE attempts to create an interchange standard by defining complex methods for programs to talk and work with each other, instead of defining a common document architecture and letting the programs do as they please. Instead of creating a network of roads, OLE tries to connect everybody's houses with one long hallway.

There are other problems with the file-centric model besides file ownership. There are countless cases where a user wants to organize his information in groups other than documents. For example, this book is a “document,” but it is composed of dozens of smaller documents, each represented by a file. The word processor that I used to create each document in the book understands how to deal with each one, but it is quite weak when it comes to handling the bigger “document,” the book itself. There are no global commands, so I can't change the phrase “abysmally bad design” to “dunderheaded design” throughout the book by using a single command. Neither is there a way to tell what number this page will have when it is part of the full book. Microsoft is aware of the problem but can't seem to solve it decently. The “Master Document” feature in Word is a game attempt, but anyone who has used it on a large document will immediately see how inadequate it is. Of course, my point isn't that word processor manufacturers can't solve this problem. Programs like Interleaf or FrameMaker can handle it. It is just that our file-centric vision tends to blind us to cross-file or multi-file problems, and they usually don't receive the attention they deserve.

System designers don't seem to be aware of these tradeoffs and, consequently, many of our most cherished notions about system design are based on tradition

rather than on sensible design. We have been doing files the same way for so long that nobody questions our methods. And these methods shape our thinking and, ultimately, shape our user interfaces.



Choosing Platforms

The very first strategic choice you will wrestle with as a designer will probably concern platforms. You must decide whether to write for UNIX, the Macintosh or the Intel/Microsoft platform—or for all of them. You also must decide whether to support the older, weaker hardware out there. These decisions are very difficult because they combine the messy uncertainties of real-world considerations with the clean, pure world of software construction. If you apply physical-world logic to these decisions, you will usually get left behind by the technology. If you use pure software logic for them, you will upset and alienate your customers. The correct answer is to blend the two, remaining aware that the proper proportions of each change daily. Here are my thoughts on resolving the platform issues. I expect you will temper them with the demands of your particular situation.

Software is the expensive part

Modern desktop computers are consumables, like paper clips and stationery, rather than fixed assets or durable goods, like buildings or desks. The problem isn't that computers can't perform or aren't valuable after a year or two, but that the technology moves ahead so rapidly that the resulting

interaction problems detract significantly from productivity. Keeping older desktop computers in critical roles in your mainstream business environment any longer than appropriate is like making your employees take the bus on their cross-country business trips instead of flying: it is penny-wise and pound-foolish.

Every aspect of software is more expensive than hardware. You might think this isn't true because you have 1,000 computers but only have to develop an application once. Let's say it takes \$350,000 to develop a program and those computers cost \$3,000 each for a total of \$3,000,000; it seems like your point is proven. But the comparison is not really between \$350,000 and \$3,000,000. Yes, the cost of the hardware is \$3,000,000, but the cost of the software also includes the cost of installing, training and supporting 1,000 users of it. It may take a week to get each person up to speed on the program. If we assume that each employee makes \$200 per day, their combined salary for the week is \$1,000,000. Then you add about \$500 per user for the teaching costs. Now don't forget the opportunity cost! While each person is learning about the program, he is *not* generating income for the company. If each employee normally generates \$5,000 worth of business for the company each week, that revenue is lost. So far, the cost of implementing the software is hovering around \$6,500 per user. You can get a pretty classy computer for that much money. The software cost of installing our 1,000 computers is now \$6,500,000!

The half-life of a desktop computer

Much of today's business wisdom regarding computers was learned in the '60s and '70s in the data-processing centers with their giant mainframes. Those machines were large, long-term, corporate assets tended to by dozens of technicians. The technicians came and went, but the mainframe was permanent. The modern desktop computer is architecturally very similar to the mainframe, but in every other respect is quite a different animal.

The desktop PC is to the mainframe as a wild lion is to a house cat. The capability and flexibility of the PC make it the king of the jungle, while the mainframe was weak as a pussycat unless it had hordes of technicians working to keep it purring. They share many physical characteristics, but one is a domesticated animal and the other is a savage beast. To treat them the same would be dangerous. The desktop PC came from a different branch of the evolutionary tree than mainframes did, and it has dramatically different purposes, goals, usage and responsibilities. Those who treat PCs as durable goods are persisting

in thinking of them as little mainframes; as permanent investments that support operations or generate revenue. But desktop PCs are, as I've said before, consumables, not investments. To be economically efficient, they must be treated as such. I'm not suggesting that you wrangle with the IRS over it (although someone should), but this is the way you should consider computers in your planning.

Think of your desktop PCs the way that Hertz thinks of their cars: certainly cars are a fundamental part of their business, but Hertz doesn't get sentimental about them. Instead, they do the math. The half-life of the price of a fleet car is about two years. That is, a car that cost \$20,000 new can be resold for \$10,000 in 24 months. I would guess that a modern desktop PC that can be purchased for \$3,000 today can be sold for \$1,500 within 12 months, because the pace of computer technology is faster than automotive technology. Hertz sells off the bulk of its fleet before they have reached their price half-life, yet most businesses won't sell off their personal computers for as long as four years; 400% of their price half-life. Are desktop computers less important for conducting your business than Hertz's cars are to them? I doubt it.

That Hertz or Avis sells its fleet cars after a year isn't an accident. These companies have performed detailed financial studies to determine the optimum amount of time to keep their cars so that their yield from resale is best with respect to the amount of rental revenues they can generate from each one. Just because Hertz sells off their fleet cars after a year doesn't mean that you can't get ten or more good years out of your family car, but it takes a considerable amount of care and attention to do so. Care and attention is expensive in business, and in today's service economy they are more effectively lavished on customers than on inanimate objects. Similarly, you can keep your family's old 386/16 with 640×480×16 VGA monitor going for several years past its prime, and it will still serve you well. You can devote the time and attention to it that it needs. In the business environment, however, you can't afford to lavish that time and attention on your office equipment. Opportunity cost is extremely expensive in modern business, and while you are baby-sitting cantankerous hardware, your competition is out stealing your market share.

Personal computers are not cars, and the dislocation involved in upgrading from one model to another is much greater than just buying a new car—it's more akin to buying a new office—so the analogy isn't precise. The point, though, is that we must begin to regard our desktop computers more like fleet cars and less like mainframes.

PCs are not little mainframes; they are unique business tools that don't age gracefully. There are enormous costs associated with keeping computers beyond their useful and most productive times. The main costs arise from interaction problems. A typical PC will have dozens of major hardware and software components, and the probability for incompatibilities between them grows exponentially as the system ages and new components are added. When you buy a brand-new computer, you start the clock ticking again at zero, and the probability for interaction problems is reduced again to a manageable level.

The potential for error inside a given modem, for example, is really small. Most hardware vendors are reputable and test their product well. However, the odds that the particular brand of modem you own is fully tested with a particular serial communications chip and a particular serial communications driver software decreases as these three products diverge in time.

Almost any mouse sold in 1995 will work with almost any computer sold in 1995. But the chances of strange, unpredictable interaction problems between that mouse and an otherwise perfectly functional computer sold in 1992 are quite high. Even the standard plugs for mice have changed between 1992 and 1995, from a seven pin DIN connector to a five-pin mini-DIN. How much will it cost your company in lost productivity to have an executive stopped from doing her job while a technician hunts down the proper connector? Is it more or less than the cost to replace her computer with a more modern one?

If the cost of keeping older desktop PCs in service is higher than their replacement cost, it makes good business sense to upgrade them. If, based on resale value, the optimal sell-off date for a computer is 10 months, you can expect that the residual value will reach zero sometime before four years have elapsed. I contend that the optimal interval to keep a computer before replacing it is roughly 24 to 30 months from the initial purchase. Before that, you pay too much in disruption. After that, you pay too much in obsolescence.

Choosing a development platform

The computer industry often makes a further miscalculation that makes keeping old computers around past their prime seem harmless by comparison. I'm referring, of course, to the decisions regarding target platforms for software development. Many development teams create software that will accommodate all existing hardware. Their management usually colludes in this error by encouraging them to support the five-, six- or seven-year-old computers that

are still ticking away in corporate offices, arguing that it would be too expensive to replace all of those computers. This ignores the fact that the cost of developing software to support both old and new hardware is generally significantly greater than the cost of purchasing and supporting the more powerful new hardware. This means that, if the software is written to accommodate those old computers, it will save money on the hardware just to spend it on the software, resulting in much stupider software at greater cost. It should be the responsibility of management to assure that the computers on desktops throughout the company are as modern as can be when the new software is ready.

To develop software for modern platforms, you must design for hardware that will be readily available six to twelve months after the product first ships. Don't forget that it might take a year to develop the software, and another six months for it to penetrate your organization, and the state-of-the-market computers will be even more powerful than today.

Design tip: The program should perform optimally on hardware that doesn't exist yet.

If you develop software for a target hardware platform that is any older than next year's standard, you are firmly anchoring your business in the past. If any of your competitors make the more intelligent choice, you will be quickly overtaken. The cost of programming is extremely high compared to the cost of hardware, but you will have to accept this development cost regardless of the hardware platform you write for, so this isn't the real problem. Instead, the problem lies in the desire to fully amortize the investment in the software by assuring that it covers all platforms. The trap is that by covering all existing platforms, you reach backwards, shutting yourself out of future platforms. And only the future platforms have, well, a future.

Just like the desire to fully amortize your investment in the hardware caused the problem, it also compounds the problem by forcing you into building weaker, less-effective software and then insisting on getting your money's worth from it, too. The unsuspecting businessperson can be trapped by his own parsimony into weakening the company's ability to perform its fundamental business.

The insight here is to never let software decisions be swayed significantly by the limitations of existing hardware. The software should, of course, be able to run on state-of-the-art computers when it is released, but it should have to stoop a

bit to do so. The product should be designed to behave optimally with the hardware that will be state-of-the-art 6 to 12 months after the software is first released.

This is a lot less important for operating system software (OSs) or language compilers, where the performance leverage is enormous and always works against you. But in the world of applications, where user interaction is intense and performance is usually measured by how productive users feel rather than by more objective measures, don't compromise software for hardware.

Controlling the hardware

If you are creating specialized software or vertical-market programs that will be sold to customers for several thousands of dollars or more, you can certainly dictate the hardware it should run on. A larger proportion of the user's budget will go for software than for hardware.

Users will inevitably argue this point. Since the beginning of the microcomputer revolution, no axiom has been truer yet more frequently violated than this one:

**Purchase the right software; then
buy the computer that runs it**



Most users will buy a computer and then look for software that solves their problem and—by the way—also runs on their computer hardware. This attitude is a carryover from the mainframe days, and often informs the thinking of software developers as much as it influences software buyers. To make a sale, developers are quick to adapt to a specific hardware platform. Yes, the realities of business sometimes dictate such choices and an adulteration of our practices, but this doesn't for a minute mean that these decisions make for good design. Software is the key, not the hardware. In a few more years, when the cost of computing machinery drops by another couple of orders of magnitude, this natural order will be apparent to all. Good designers will anticipate it.

Simultaneous Multiplatform Development

As tantalizing as it is to want to kill two birds with one stone, don't do simultaneous multiplatform development. It isn't worth it. Instead, develop only for your primary market. Then use the revenue from this product to port to your secondary platforms.

There are two ways to do simultaneous multiplatform development and both of them are bad. You can make the code more complicated, or you can homogenize the interface.

Design tip: Build the program to run on only one platform at a time.

Anything that increases the complexity of source code should be avoided at all costs. It will magnify the time it takes both to write and to debug. The main job of the software development manager is to avoid uncertainty and delay. Simultaneous multiplatform development generates more uncertainty and delay than any other tactic you might use. The compromises and confusion will ultimately result in the quality of your product suffering.

In the quiet of the office it seems so harmless, so easy to add a few “if-else” statements to the source code and magically reap the benefits of supporting an extra hardware platform. Don't be fooled. Everything in the already-problematic discipline of software development becomes harder and more complex. Each design decision must now be made for two platforms. Compromises slip into the product to account for the disparity between the two. If writing for dual platforms increases the amount of code by only 5%, it can increase the time to market by a *third*. This is an incredibly costly bad decision that is easily avoided.

There are several commercially available libraries of code that will let you develop on multiple platforms simultaneously. In order to do so, they demand that you design for a “generic” GUI, which the library then runs on each platform. This may be good for the development team, but the users will dislike it intensely. They will immediately detect the homogenization of the interface and will not appreciate it. Macintosh users prefer programs with a Mac sensibility. Windows users won't settle for anything but a Windows application. For example, Windows users are very comfortable with multiple, complex toolbars running horizontally across the top of the program just beneath the menubar.

Many Mac aficionados consider this idiom to be about as desirable as a shark in a swimming pool.

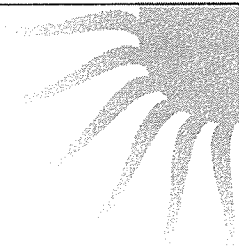
The programming staff will probably be game to do multiplatform development. They may even be the ones pushing for it. They see it as an intellectual challenge; multiplatform development is a tournament in which to compete and win. Just remember that programmers frequently don't give a hoot about deadlines—they're in it for the brain exercise.

Finessing the problem

A much simpler, safer and more effective way to solve the problem is by developing for a single platform first: your main market. This will typically be Windows, the market leader by a wide margin. You completely avoid the complexities of multiplatform development, finish the first version with the greatest possible speed and ship it to the largest possible market.

Once you've finished the Windows version, you are generating revenue while you begin development for other platforms. Development managers take note: This is your most compelling argument for convincing others that single-platform development is the proper course to take. The needs of secondary markets shouldn't delay the needs of primary markets.

Don't hamper primary markets by serving secondary markets



This doesn't mean that you need to abandon the secondary market. On the contrary, at this point you will have a fully articulated, working model of the product—running on Windows—to use as a prototype for the versions to run on other platforms. You can hire a team of programmers with proven skills on the new platform and tell them “go forth and clone.” When programmers are working from a clearly visible model, the development time can be compressed significantly because there is little time wasted going down blind design alleys. You can also hire programmers who are less experienced—and therefore less expensive—to do clone-programming because there is less design work involved. Much of the code will likely be reusable, but you now treat this as a bonus rather than as an expectation.

When I say clone, however, I mean clone functionality but not dialect. The Windows prototype will demonstrate how the program should interact with the user, but the Macintosh version must behave like a Mac program at the detail level. For that, you need local expertise. The problem is conceptually similar to localization.

The Myth of Interoperability

Windows developers often face programs with legacies as successful DOS programs. Many applications are brought to Windows after they have had a long and lucrative run in a DOS, character-based, command-line environment. Common wisdom holds that the Windows program should emulate the DOS program as closely as possible. “Thousands of satisfied customers want to move to Windows,” goes the logic, “and they will be sorely disappointed if the program is different from what they already know and love.” Besides, “Many of our corporate users work in heterogeneous environments and they want the Windows version to work the same way as their DOS-only systems.”

This concept is called **interoperability**. Believers in interoperability will tell you that your DOS customers are faithful to your product because of the way your program behaves, because they have already learned your DOS product and because they can’t afford the retraining costs of moving their people to a new Windows version. They will draw the irresistible word picture of the happy user entering data at a DOS machine, then cheerfully switching to the Windows computer and performing an identical task.

As compelling as this logic is, it is dead wrong. If you are going to create a Windows version of a program, go ahead and create a Windows version—don’t implement a DOS version on the Windows platform. If you try for interoperability, you will only hurt your product. You will find that no one is happy, least of all you and the development staff. Your job will become increasingly difficult as you try to reconcile fundamentally irreconcilable differences.

Simply stated, Windows users use Windows because they like it and because they don’t like DOS. On the other hand, DOS users use DOS because they like it and because they don’t like Windows. If your program acts like DOS on Windows, DOS users will be unhappy because they’d rather be using the genuine article on DOS, and nothing you do to simulate DOS on Windows will make them happy. Conversely, all of the Windows aficionados will turn up their noses at the pathetic DOS-ness of your program and its lack of understanding of how to behave appropriately in a Windows world.

Design tip: The program should be designed expressly for the target platform.

Your DOS customers are faithful because your DOS version is sensitive to the particular needs of DOS users. They like it because it has adapted to the local customs of the DOS environment. Given the limitations of that environment, it is a satisfactory solution. It exhibits familiar DOS-like behavior that makes users experienced in a DOS environment feel warm and fuzzy. Extrapolating this to mean that the behavior of the program itself is warm and fuzzy can be fatal when you move to another platform, particularly when exercising a paradigm shift as dramatic as moving from a character-based to a graphical platform.

Most Windows users like Windows because they were dissatisfied with the level of usability available in DOS. They are here because they want something different and better; not something similar and status quo. They came here because they wanted to leave the limitations of DOS behind, and they want you to have done the same.

Windows users expect your program to conform to the local standards in exactly the same way that DOS users expected your DOS program to conform to DOS standards. Windows users will want your Windows version to look and act like other Windows applications, not like DOS applications. They will expect your program to take advantage of the tools provided by the new platform. They will expect your program to deliver something better to justify the dislocation they had to invest in order to move from DOS to Windows.

Those people who clamor for interoperability are often motivated by fear. They are afraid of the new system; of their ability to learn it and to adapt to it. They are afraid mostly of the learning curve. They worked so hard and absorbed so much pain to learn the DOS version that they fear going through the process again on Windows. By demanding interoperability, they hope that they will be able to take their hard-earned expertise straight across to the new system.

The answer for these people, of course, is that it won't be anywhere near as difficult to learn in Windows than it was in DOS. They won't believe this, so you will just have to do the right thing despite their pleas. It's like telling a child that a tetanus shot is less painful than tetanus—all the child can see is the needle. Extending the metaphor, you must be the adult even if the frightened users hold the purse strings. It is not a good career move to make your

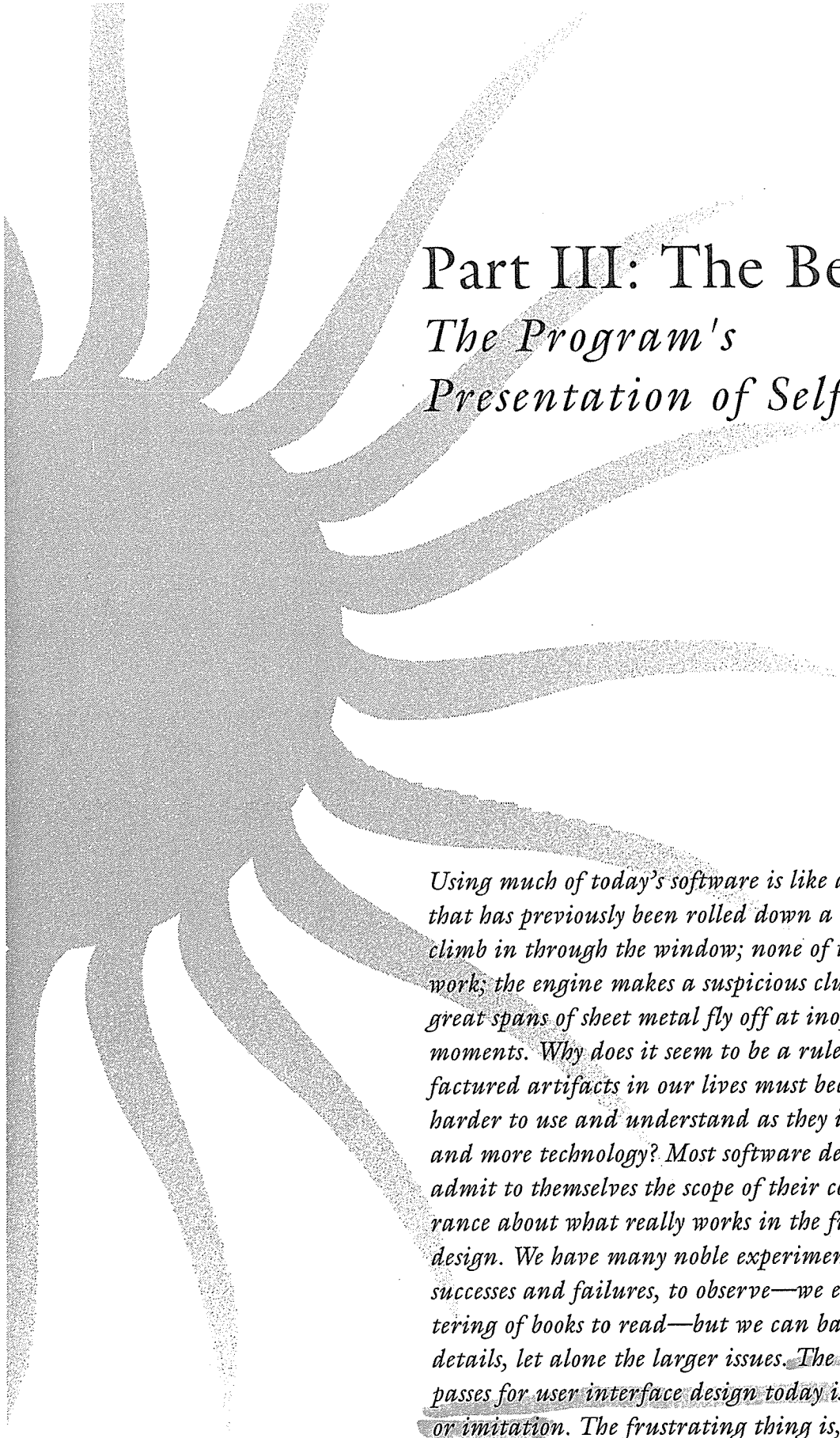
customer happy all through the development process only to eventually deliver a dud.

Of course, you have to deliver on your promises and make a graphical user interface that really is significantly easier to learn and use than the DOS version. I have no doubt that you can do it, as long as you abandon interoperable thinking and become a Windows native.

Often, the people who clamor the loudest for interoperability are the product managers, marketing managers and programmers who worked long and hard on the DOS version. They will insist that the Windows version be designed in the image of the DOS version, the company's cash cow. I have seen this situation several times. The DOS-centric forces-of-evil have the upper hand because their product makes money for the company and, for a while at least, your product merely costs the company money. They make their compelling arguments to upper management, who can't really be expected to know better, and the dictum is handed down: "Make it like the DOS version." At this point, all of the really smart people quit, the program is written, haltingly, to worship twin operating-system gods and, when it finally ships, the market emits a loud yawn. The DOS faithful remain faithful to DOS, snickering all the while about how they told you the Windows version would flop. The Windows hopefuls are very disappointed with the product because it retained its clunkiness in spite of Windows. Your competitor will release a native Windows product that was designed and written with a "when in Rome, do as the Romans do" attitude, it will begin robbing you of sales, and your company will begin its long, agonizing slide into Chapter 11. Don't let this happen to you!

The picture that I have just painted of crossing the gulf from DOS to Windows is also very true when going from the Macintosh platform to Windows. In spite of their numerous visual similarities, Mac and Windows are different cultures, and moving from one to another is not the bed of roses you might expect. If you want to sell something to Mac users and have them appreciate it, sell it to them on the Mac. Attempting to do it on a PC will just irritate Windows people and generate a yawn from the Mac folks. Macintosh users believe deeply that Macs are better than Windows. There is not much that you can do on a PC that will impress the Mac crowd, even if you adhere slavishly to Mac doctrine. The president of a prominent Mac software company once told me that "the pixels on a Mac are better than the pixels on a PC." He actually believed this, even though you can take a typical Sony or NEC video screen and plug it into either computer.

Management will make the same arguments about interoperability they always do, but the fact remains that although companies may have thousands of Macs interspersed with thousands of PCs, *very few individuals actually spend time working on both.* You can examine the market and see for yourself that there are few companies who make interoperable applications successful on both platforms. They generally have a loyal customer base on only one platform—their native one, while their customers on the *other* platform are just marking time waiting for an easier-to-use product native to *their* platform.

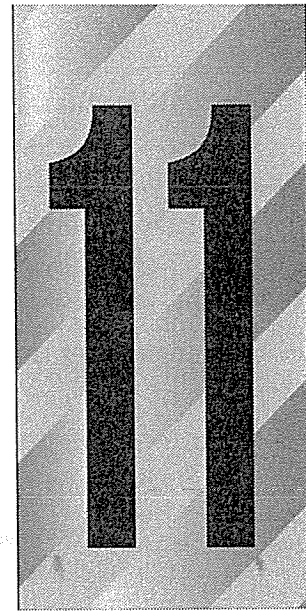


Part III: The Behavior

The Program's Presentation of Self

Using much of today's software is like driving a car that has previously been rolled down a cliff: You have to climb in through the window; none of the lights seem to work; the engine makes a suspicious clunking noise; great spans of sheet metal fly off at inopportune moments. Why does it seem to be a rule that the manufactured artifacts in our lives must become increasingly harder to use and understand as they incorporate more and more technology? Most software designers won't admit to themselves the scope of their collective ignorance about what really works in the field of interface design. We have many noble experiments, and many successes and failures, to observe—we even have a smattering of books to read—but we can barely agree on the details, let alone the larger issues. The bulk of what passes for user interface design today is either guesswork or imitation. The frustrating thing is, it doesn't have to be that way.

Orchestration and Flow



To make software more productive, we must make its users more productive. To make users more productive, we have to get them into a harmonious frame of mind. After all, it is the users' mental state that ultimately dictates how effectively they are using our program.

} PHILOSOPHY

Planing on the step

Racers of lightweight sailboats seek out a condition they call “planing.” A racing dingy planes by accelerating to the point that it actually rides on top of its own bow wake. A planing hull displaces only a fraction of the water it does normally, so the drag it generates is drastically reduced. This drag reduction can result in speeds increasing by as much as 50%. The transition between displacement sailing and planing is sharp; the planing boat will almost instantly surge ahead—sailors call it “getting on the step” and the experience is exhilarating. A minute or two longer on the step can spell the difference between winning and losing a race.

Humans can plane on the step, too, when they really concentrate on an activity. The state is generally called flow.

Tom DeMarco and Timothy Lister in their book *Peopleware, Productive Projects and Teams* (Dorset House, 1987) define flow as a “condition of deep, nearly meditative involvement.” Flow often induces a “gentle sense of euphoria” and can make you unaware of the passage of time. Most significantly, a person in a state of flow can be extremely productive, especially when engaged in process-oriented tasks such as “engineering, design, development and writing.” All of these tasks are typically performed on computers while interacting with software. Therefore, it behooves us to create a software interaction that promotes and enhances flow, rather than one that includes potentially flow-breaking or flow-disturbing behavior.

When a sailor makes a lubberly tack—changes the position of the sail clumsily—the dingy falls off the plane and slows like it hit a wall. The sailor now has to carefully accelerate until the boat can once again get on the step. Good sailors tack so smoothly that the boat is undisturbed, and the hull stays on the step. In the same way, we want our program’s interaction to be so smooth that the user is undisturbed and can remain in the state of flow. If the program rattles the user out of flow, it may take several minutes to regain that productive state.

Techniques for inducing and maintaining flow

To create flow, our interaction with software must become transparent. There are several excellent ways to make our interfaces recede into invisibility. They are

1. Follow mental models
2. Direct, don’t discuss
3. Keep tools close at hand
4. Give modeless feedback

There are other important tools for designing transparent interfaces that we will discuss in the next couple of chapters. These include “not stopping the proceedings with idiocy” (Chapter 13), and “questions aren’t the same as choices” (Chapter 14). We’ll tackle the others right here.

Follow mental models

I introduced the concept of mental models in Chapter 3. Different users will have different mental models of a process, but they will rarely visualize them in

terms of the detailed innards of the computer process. Each user naturally forms a mental image of how the software performs its task. The mind looks for some pattern of cause and effect to gain insight into the machine's behavior.

Creators of race cars place gauges on their dashboards so they follow the driver's mental model, which goes like this: "straight up is good. Anything else is bad." The engineer twists the gauges in their mounts so that every needle points straight up when everything is normal. The gauges won't look right to tyros, but the racer understands: her peripheral vision monitors the gauges easily while staying in flow to drive. If any needle deviates from the vertical, it demands the driver's conscious attention to the problem; otherwise, up means OK, just like that thumb's-up from her pit crew.

Direct, don't discuss

Many developers imagine the ideal interface to be a two-way conversation with the user. However, most users don't see it that way. Most users would rather interact with the software in the same way they interact with, say, their car. They open the door and get in when they want to go somewhere. They press on the accelerator when they want the car to move forward and the brake when it is time to stop; they turn the wheel when they want the car to turn.

This ideal interaction is not a dialog—it's more like using a tool. When a carpenter hits nails, she doesn't discuss the nail with the hammer; she directs the hammer onto the nail. In a car, the driver—the user—gives the car direction when he wants to change the car's behavior. The driver expects direct feedback from the car and its environment in terms appropriate to the device: the view out the windshield, the readings on the various gauges on the dashboard, the sound of rushing air and tires on pavement, the feel of lateral g-forces and vibration from the road. The carpenter expects similar feedback: the feel of the nail sinking, the sound of steel striking steel, the heft of the hammer's weight.

The driver certainly doesn't expect the car to interrogate him with a dialog box, nor would the carpenter appreciate one appearing on her hammer like the one in Figure 11-1.

One of the main reasons software often aggravates and upsets users is that it doesn't act like a car or a hammer. Instead, it has the temerity to try to engage us in a dialog—to inform us of our shortcomings and to demand answers from us. From the user's point of view, the roles are reversed: it should be the user doing the demanding and the software doing the answering.

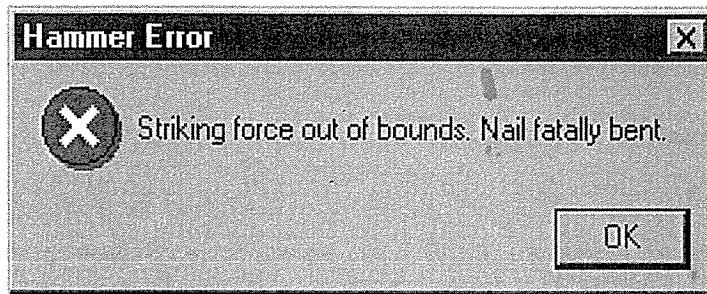


Figure 11-1

Just because programmers are accustomed to seeing messages like this doesn't mean that people from other walks of life are. Nobody wants their machines to scold them. If we guide our machines in a dunderheaded way, we expect to get a dunderheaded response. Sure, they can protect us from fatal errors, but scolding isn't the same thing as protecting.

With direct manipulation, we can point to what we want. If we want to move an object from A to B, we click on it and drag it there. As a general rule, the better, more flow-inducing interfaces are those with plentiful and sophisticated direct-manipulation idioms.

Keep tools close at hand

Most programs are too complex for one mode of direct manipulation to cover all of their features. Consequently, most programs offer a set of different tools to the user. These tools are really different modes of behavior that the program enters. Offering tools is a compromise with complexity, but we can still do a lot to make tool manipulation easy and to prevent it from disturbing the flow. Mainly, we must ensure that tool information is plentiful and easy to see and attempt to make transitions between tools quick and simple.

Tools should be close at hand, preferably on palettes or toolbars. This way, the user can see them easily and can select them with a single click. If the user must divert his attention from the application to search out a tool, his concentration will be broken. It's as if he had to get up from his desk and wander down the hall to find a pencil. And he should never have to put tools away manually.

As we manipulate tools, it's usually desirable for the program to report on their status, and on the status of the data we are manipulating with the tool. This

information needs to be clearly posted and easy to see without obscuring or stopping the action.

Modern jet fighter designers go the race car designers one better in cockpit design—this is critically important when the job involves yanking and banking 40 tons of titanium at 600 miles per hour. Jet fighters have a heads-up display, or HUD, that superimposes the readings of critical instrumentation onto the forward view of the cockpit's windscreen. The pilot doesn't even have to use peripheral vision but can read vital gauges while keeping her eyes glued on the opposing fighter.

Our software should display information like a jet fighter's HUD. The program could use the edges of the display screen to show the user information about the action in the center that is being directly manipulated.

Modeless feedback

When the program has information or feedback for the user, it has several ways to present it. The most common method is to pop up a dialog box on the screen. This technique is modal: it puts the program into a mode that must be dealt with before it can return to its normal state, and before the user can continue with his task. A better way to inform the user is with **modeless feedback**. —ROLLING PAGE COUNT
—AIX CLOCK

Feedback is modeless whenever information for the user is built into the normal interface and doesn't stop the normal flow of system activities and interaction. In Word, you can see what page you are on, what section you are in, how many pages are in the current document, what position the cursor is in and what time it is modelessly just by looking at the status bar at the bottom of the screen.

If you want to know how many words are in your document, however, you have to call up the Summary Info dialog from the File menu; then you have to press a button to summon the Statistics dialog to see a word count (see Figure 11-2). I refer to the word count figure frequently when I write magazine articles (It's hard to get them short enough!). I sure wish the word count were offered modelessly.

Orchestration

If the user could achieve his goals without the program, he would. By the same token, if the user needed the program but could achieve his goals without

going through its user interface, he would. Interacting with software is not an aesthetic experience. It is a pragmatic exercise that is best kept to a minimum. Don't kid yourself about your sexy new multimedia, interactive, online, social, point-and-click program. The user would rather just snap his fingers or say "abracadabra." No matter how cool your interface is, less of it would be better.

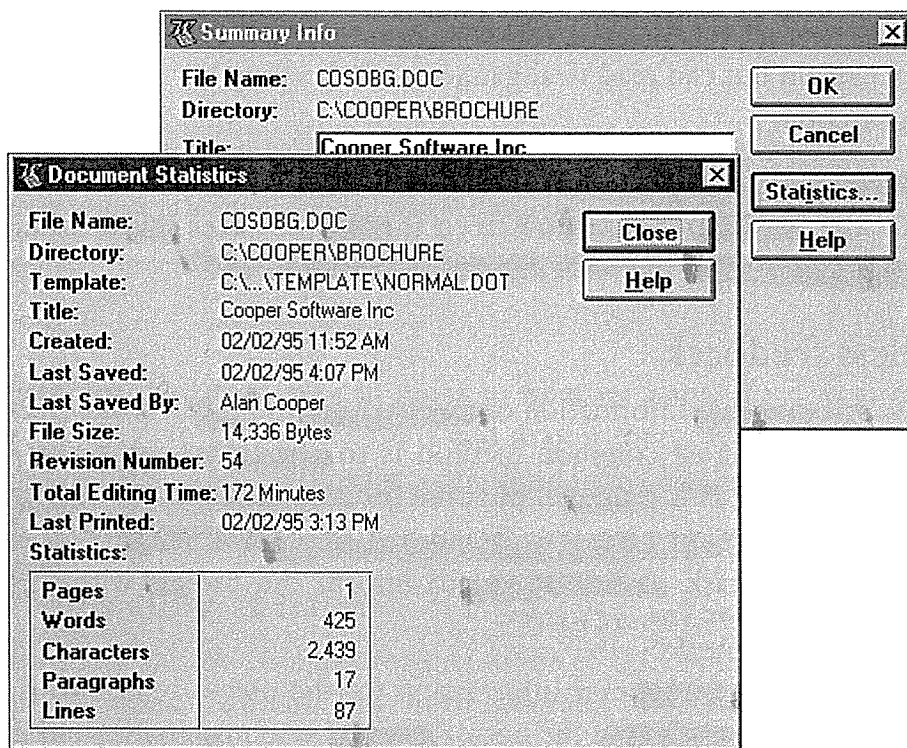
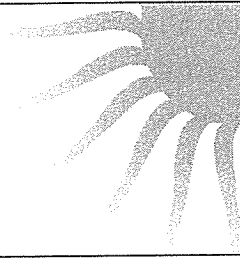


Figure 11-2

In Word, if you want to know the number of words in your document, you must first request the Summary Info dialog from the File menu. Then, by pressing the Statistics button, you call up the Document Statistics dialog box. Down in the corner, buried among other useless (for me) numbers is the one I want. After I've read it, I must press the Close key and then the Cancel key...or was it the other way around? This is the opposite of model feedback, and it brings whatever flow I might have had going to a screeching halt.

Directing your attention to the interaction itself puts the emphasis on the side effects of the tools rather than on the user's goals. A user interface is an artifact, not something directly related to the goals of the user. Next time you find yourself crowing about what cool interaction you've designed, just remember that the ultimate user interface is no interface at all.

No matter how cool your interface is, less of it would be better



It looks to me like the dialog boxes in Figure 11-2 were written by two different programmers. Maybe they didn't talk much with each other, but I can guarantee you that they never spoke with a designer—someone whose job it was to coordinate all of the user interface elements. The results look like what you'd get if the orchestra lacked a conductor. Each musician might know his part well, but when all seventy of them get together, they won't sound in accord.

It is vital that all of the elements work together towards a single goal. I call this process of achieving a coherent interface **orchestration**.

Webster's defines orchestration as "harmonious organization," a very reasonable phrase for what we should expect from interacting with software. Harmonious organization doesn't yield to fixed rules. You can't create guidelines like "five buttons on a dialog box are good" and "seven buttons on a dialog box are too many." Yet it is easy to see that a dialog box with 35 buttons is usually bad. The major difficulty with such analysis is that it treats the problem *in vitro*. It doesn't take into account the problem being solved; it doesn't take into account what the user is doing at the time or what he is trying to accomplish.

Finesse

In many things, the more there are, the better things are. In the world of interface design, the contrary is true, and we should constantly strive to reduce the number of elements in the interface without reducing the power of the program. In order to do this, we must do more with less; this is where careful orchestration becomes important. We must coordinate and control all of the power of our program without letting the interface become a gaggle of windows and dialogs, covered with a scattering of unrelated gizmos.

I often see dialog boxes that are complex but not very powerful. They typically allow the user to perform a single task without providing access to related tasks. For example, most programs allow the user to name and save a data file, but they never let him delete, rename or make a copy of that file while he is at it. The dialog leaves that task to the operating system. It may not be

trivial to add these functions to the program, but isn't it better that the programmer go through the non-trivial activities than for the user to be forced to? Today, if the user wants to do something simple like edit a new copy of file "foo," he must go through a non-trivial sequence of actions: going to the shell, selecting foo, requesting a copy from the menu, changing its name, returning to the program and then opening the new file. I'd much rather see the programmer work harder and give the user a break.

It's not as hard as it looks, actually. Orchestration doesn't mean bulldozing your way through problems. It means finessing the problems, wherever possible. Instead of adding the copy and rename functions to the File Open dialog box of every application, why not just discard that same slightly retarded File Open dialog box from every application and replace it with the shell program itself. When the user wants to open a file, the program calls the shell—which conveniently has all of those collateral file-munging functions built in—and the user can double-click on the desired document. That's pretty much what the File Open dialog does, except it doesn't do it so well.

Yes, the application's File Open dialog does show the user a filtered view of files (like only .DOC files in Word), but there are certainly ways to do that in the shell. I can think of several ways to do it better and easier in the shell than that old dialog does with its clunky combobox.

Following on this logic, we can also dispense with the Save As dialog, which is really the logical inverse of the File Open dialog. If every time we requested the Save As... function from our application, it wrote our file out to a temporary directory under some reasonable temporary name and then transferred control to the shell, we'd have all of those nice shell tools at our disposal to move things around or rename them.

Yes, there are access problems, but nothing that a little inter-process communicating wouldn't solve. Yes, there would be a chunk of coding that programmers would have to do, but look at the upside: Countless dialog boxes could be completely discarded. The user interfaces of thousands of programs would become more visually and functionally consistent, and all with a single design stroke. That is orchestration!

Invisibility

So much of today's software has stilted, jerky and inappropriate interactions. There seems to have been little attempt at orchestration anywhere.