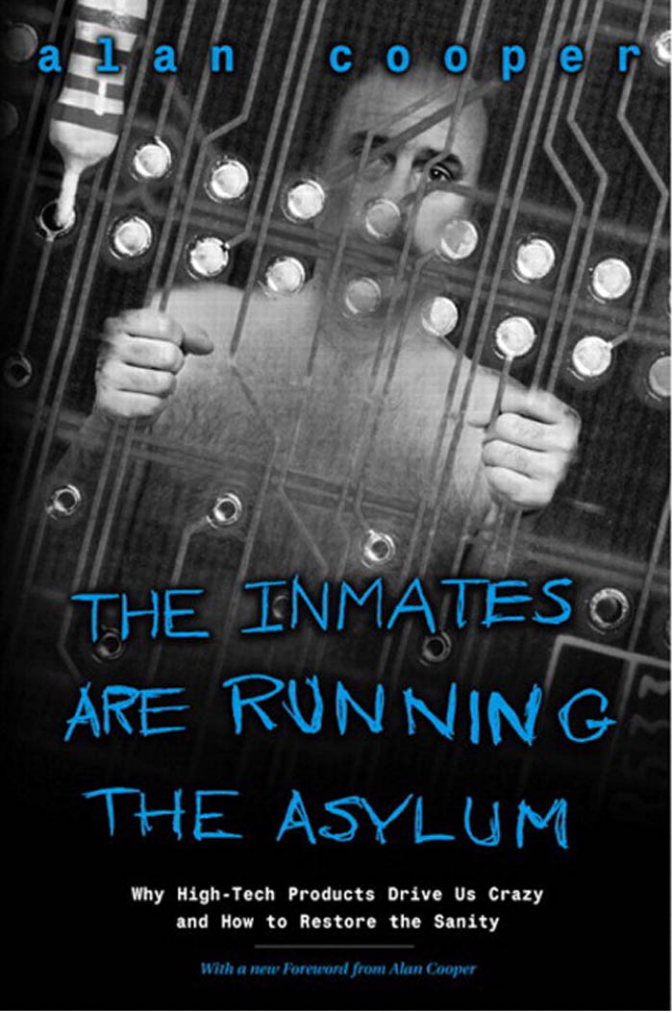



FREE SAMPLE CHAPTER

SHARE WITH OTHERS

alan cooper

# THE INMATES ARE RUNNING THE ASYLUM

SAMS
A Division of Pearson Education
800 East 96th Street, Indianapolis, Indiana 46240

## *The Inmates Are Running the Asylum*




International Standard Book Number: 0-672-32614-0

Library of Congress Catalog Card Number: 2003116997

Printed in the United States of America

First Printing: March 2004

07 06 05 4 3


### *Trademarks*

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Goal-Directed design is a trademark of Cooper Interaction Design.

### *Warning and Disclaimer*

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

### *Bulk Sales*

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**
**1-800-382-3419**
**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**
**international@pearsoned.com**


**Publisher**
Paul Boger

**Executive Editor**
Candace Hall

**Managing Editor**
Charlotte Clapp

**Project Editor**
Dan Knott

**Copy Editor**
Eileen Cohen

**Indexer**
Ken Johnson

**Proofreader**
Juli Cook

**Publishing Coordinator**
Cindy Teeters

**Interior Designer**
Karen Ruggles

**Cover Designer**
Alan Clements

**Page Layout**
Eric S. Miller

## Dedication

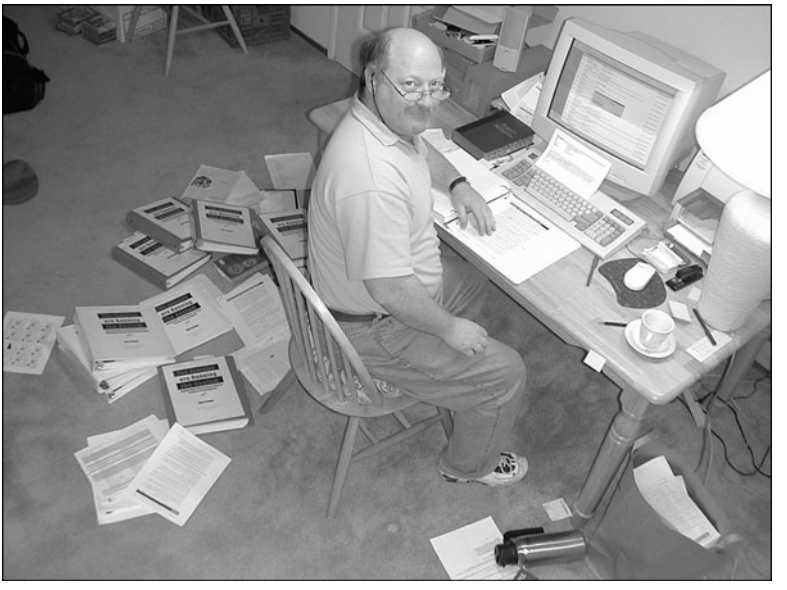
For Sue, Scott and Marty, with love.

## Acknowledgments

I could not have written this book without the care and help of many wonderful friends and colleagues. In particular, several people performed the demanding and difficult job of reading and commenting on the manuscript, sometimes more than once. Their comments made me answer tough questions, introduce my topics, sum up my points, quench my flames, and corral my wild fits of indignation. The book is far better because of the contributions of Kim Goodwin, Lane Halley, Kelly Bowman, Scott McGregor, David West, Mike Nelson, Mark Dziersk, Alan Karp, Terry Swack, Louie Weitzman, Wayne Greenwood, Ryan Olshavsky, John Meyer, Lisa Saunders, Winnie Shows, Kevin Wandryk, Glenn Halstead, Bryan O'Sullivan, Chuck Owen, Mike Swaine, and Skip Walter. I really appreciate your time, care, and wisdom. In particular, Jonathan Korman's comments and counsel were invaluable in helping me to distill my themes. I must also thank all the talented and hard-working people at Cooper Interaction Design who did my job for me while I was busy writing. Deserving of special thanks is Design Director Wayne Greenwood, who did a great job under pressure keeping our design quality and morale high.

Getting the illustrations done turned out to be one of the more interesting production challenges. Chad Kubo, the masterful creator of the images, did a remarkable job of interpreting my vague ideas into crisp and memorable images. They add a lot to the book. The illustrations could not have been done at all without the tireless art direction work of Penny Bayless and David Hale. Still others helped with the many production tasks. Thanks to Brit Katzen for fact checking and research and Mike Henry for copy editing.

Writing a book is a business, and for making it a successful one I also owe sincere thanks to my team of technology-savvy businesspersons, headed by my agent Jim Levine, and including Glenn Halstead, Lynne Bowman, Kelly Bowman, and Sue Cooper. At Pearson, Brad Jones supported this project throughout, but the most credit goes to Chris Webb, whose tenacity, focus, and hard work really made *The Inmates* happen.



I really appreciate the many people who provided moral support, anecdotes, advice, and time. Thanks very much to Daniel Appleman, Todd Basche, Chris Bauer, Jeff Bezos, Alice Blair, Michel Bourque, Po Bronson, Steve Calde, David Carlick, Jeff Carlick, Carol Christie, Clay Collier, Kendall Cosby, Dan Crane, Robert X. Cringely, Troy Daniels, Lisa Powers, Philip Englehardt, Karen Evensen, Ridgely Evers, Royal Farros, Pat Fleck, David Fore, Ed Forman, Ed Fredkin, Jean-Louis Gassee, Jim Gay, Russ Goldin, Vlad Gorelik, Marcia Gregory, Garrett Gruener, Chuck Hartledge, Ted Harwood, Will Hearst, Tamra Heathershaw-Hart, J.D. Hildebrand, Laurie Hills, Peter Hirshberg, Larry Keeley, Gary Kratkin, Deborah Kurata, Tom Lafleur, Paul Laughton, Ellen Levy, Steven List, T.C. Mangan, David Maister, Robert May, Don McKinney, Kathryn Meadows, Lisa Mitchell, Geoffrey Moore, Bruce Mowery, Nate Myers, Ed Niehaus, Constance Petersen, Keith Pleas, Robert Reimann, John Rivlin, Howard Rheingold, Heidi Roizen, Neil Rubenking, Paul Saffo, Josh Seiden, Russ Siegelman, Donna Slote, Linda Stone, Toni Walker, Kevin Weeks, Kevin Welch, Dan Willis, Heather Winkle, Stephen Wildstrom, Terry Winograd, John Zicker, and Pierluigi Zappacosta.

This “year long” project took 20 months, and my family showed great patience with me. I owe the greatest debt of love and thanks to my wife, Sue Cooper, and to my handsome young sons, Scott and Marty. I love you with all of my heart.

# Table of Contents

# Introduction

Run for your lives—the computers are invading. Awesomely powerful computers tackling ever more important tasks with awkward, old-fashioned interfaces. As these machines leak into every corner of our lives, they will annoy us, infuriate us, and even kill a few of us. In turn, we will be tempted to kill our computers, but we won't dare because we are already utterly, irreversibly dependent on these hopeful monsters that make modern life possible.

Fortunately, we have another option. We need to fundamentally rethink how humans and machines interact. And rethink the relationship in deep and novel ways, for the fault for our burgeoning problems lies not with our machines, but with us. Humans designed the interfaces we hate; humans continue to use dysfunctional machines even as the awkward interfaces strain their eyes, ache their backs, and ruin their wrist tendons. We all are, as the title of this book suggests, the inmates running the techno-asylum of our own creation.

This book is a guide to our escape. Or rather, Alan Cooper reveals that the door to the asylum lies wide open. We are free to leave any time we want, but mad as we have all become, we never noticed until now. The secret lies in redefining the way we interact with our computers in a larger context.

Alan Cooper is not merely a fellow inmate; he is also a heretic whose ideas will likely infuriate those who would want to keep us locked up. These are the engineers who built the systems we hate and who still believe the way out of this mess is to build better interfaces. But the very notion of *interface* is itself an artifact of an age when computers were scarce and puny, and barely able to interact with their human masters. *Interface* made sense when the entire interaction took place across the glass-thin no-man land of a computer screen. Now it is an utterly dangerous notion in a world where computers are slipping into every corner of our lives. Computers no longer interface with humans—they interact, and the interaction will become steadily deeper, more subtle, and more crucial to our collective sanity and ultimate survival.

Alan Cooper understands the shift from interface to interaction better than anyone I know. His ideas come from years of experience in helping design products that slip elegantly and unobtrusively into our lives. He has walked his talk for years, and now he has finally found the time to turn his practice into a lucid description of the challenge we face, and a methodology for escaping the asylum we have so lovingly built. Read on and you will find your freedom.

Paul Saffo
Director
Institute for the Future

## Foreword to the Original Edition

### *The Business-Case Book*

I intended to write a very different book from this one: a how-to book about the interaction-design process. Instead, in May 1997 on a family visit to Tuscany, my friends Don McKinney and Dave Carlick talked me into this one. They convinced me that I needed to address a business audience first.

They knew I wanted to write a how-to design book, and—although they were encouraging—they expressed their doubts about the need for interaction design, and they wanted me to write a book to convince them of its value. Their argument was intriguing, but I was unsure that I could write the book they wanted.

Late one night on the veranda of our shared ochre villa overlooking Firenze, I was having an earnest conversation with Dave and Don. Several empty bottles of Chianti stood on the table, along with the remains of some bread, cheese, and olives. The stars shone brightly, the fireflies danced over the lawn, and the lights on the ancient domes of the Tuscan capital twinkled in the distance. Once again, Dave suggested that I postpone the idea of a how-to book on design and instead "make the business case for interaction design."

I protested vigorously, "But Dave, I don't know how to write that book." I ticked off the reasons on my fingertips. "It means that I'd have to explain things like how the current development process is messed up, how companies waste money on inefficient software construction, how unsatisfied customers are fickle, and how a better design process can solve that."

Dave interrupted me to say simply, "They're called chapters, Alan."

His remark stopped me dead in my tracks. I realized that I was reciting an old script, and that Dave was right. A book that made "the business case" *was* necessary—and more timely—than a book that explained "how to." And both Dave and Don convinced me that I really could write such a book.

## *Business-Savvy Technologist/Technology-Savvy Businessperson*

The successful professional for the twenty-first century is either a *business-savvy technologist* or a *technology-savvy businessperson*, and I am writing for this person.

The technology-savvy businessperson knows that his success is dependent on the quality of the information available to him and the sophistication with which he uses it. The business-savvy technologist, on the other hand, is an entrepreneurial engineer or scientist trained for technology, but possessing a keen business sense and an awareness of the power of information. Both of these new archetypes are coming to dominate contemporary business.

You can divide all businesspeople into two categories: those who will master high technology and those who will soon be going out of business. No longer can an executive delegate information processing to specialists. Business *is* information processing. You differentiate yourself today with the quality of your information-handling systems, not your manufacturing systems. If you manufacture anything, chances are it has a microchip in it. If you offer a service, odds are that you offer it with computerized tools. Attempting to identify businesses that depend on high technology is as futile as trying to identify businesses that depend on the telephone. The high-tech revolution has invaded every business, and digital information is the beating heart of your workday.

It's been said, "To err is human; to really screw up, you need a computer." Inefficient mechanical systems can waste a couple of cents on every widget you build, but you can lose your entire company to bad information processes. The leverage that software-based products—and the engineers that build them—have on your company is enormous.

Sadly, our digital tools are extremely hard to learn, use, and understand, and they often cause us to fall short of our goals. This wastes money, time, and

opportunity. As a business-savvy technologist/technology-savvy businessperson, you produce software-based products or consume them—probably both. Having better, easier-to-learn, easier-to-use high-tech products is in your personal and professional best interest. Better products don't take longer to create, nor do they cost more to build. The irony is that they don't have to be difficult, but are so only because our process for making them is old-fashioned and needs fixing. Only long-standing traditions rooted in misconceptions keep us from having better products today. This book will show you how you can demand—and get—the better products that you deserve.

The point of this book is uncomplicated: We can create powerful and pleasurable software-based products by the simple expedient of *designing* our computer-based products *before* we build them. Contrary to the popular belief, we are not already doing so. Designing interactive, software-based products is a specialty as demanding as constructing them.

⌘

Having made my choice to write the business-case book rather than the how-to design book, I beg forgiveness from any interaction designers reading this book. In deference to the business audience, it has only the briefest treatment of the actual nuts and bolts of interaction-design methodology (found primarily in Part IV, "Interaction Design Is Good Business"). I included only enough to show that such methodology exists, that it is applicable to any subject matter, and that its benefits are readily apparent to anyone, regardless of their technical expertise.

Alan Cooper
Palo Alto, California
http://www.cooper.com
inmates@cooper.com

# Foreword

I recently met with a senior executive at one of the world's largest technology companies. His official title is Vice President for Ease of Use, and he is responsible for a great number of software products, large and small. He is a brilliant and accomplished fellow with roots in the formal Human-Computer Interaction community. He is steeped in the ways of "usability"—of testing and observing behind one-way mirrors—as is his company. But he came to talk about design, not testing, and about personas, not users. He said that his company has completely ceased all postdevelopment usability testing and has instead committed to predevelopment design efforts. He further asserted that all of his staffers trained in the art of *in vitro* user observation were being retrained to do *in situ* ethnographic research.

This executive and his company are emblematic of the sea of change that has occurred in the industry in the five short years since *The Inmates* was first published. The book has served as both a manifesto for a revolution and a handbook for a discipline. Countless midlevel product managers have sent me email describing why—after reading *The Inmates*—they purchased a copy of the book for each of their departments' senior executives. Meanwhile, software builders and universities alike have used the three chapters in Part IV, "Interaction Design Is Good Business," as a rudimentary how-to manual for implementing Goal-Directed® design using personas.

I am deeply grateful to all of the managers, programmers, executives, and usability practitioners who have used the ideas in this book to help bring usability out of the laboratory and into the field and changed its focus from testing to design. Because of their efforts, the entire landscape of the usability profession has changed. Today, most of the organizations I have contact with have one or more interaction-design professionals on their payrolls, who have an ever-increasing influence over the quality and behavior of the software products and services being created. It's gratifying to know that this book has contributed to their success.

I recall giving a keynote presentation at a programmer's conference in 1999, shortly after this book was first published. That talk had the same title as the book, and I opened by asserting that "inmates are running the asylum, and *you* are the inmates." You could hear a pin drop as the more than 2,500 engineers in the audience grappled with that accusation. In the silence that engulfed the auditorium, I went on to present the basic premise of this book, and an hour later, that crowd of *Homo logicus* was so sufficiently convinced that they honored me with a standing ovation. Surprisingly, most programmers have become enthusiastic supporters of design and designers. They know that they need help on the human side of software construction, and they are very happy to be finally receiving some useful guidance. They recognize that any practice that improves the quality and acceptance of their programs doesn't threaten them.

In the past, executives assumed that interaction design was a programming problem and delegated it to programmers, who diligently tried to solve the problem even though their skills, training, mindset, and work schedule prevented them from succeeding. In the spirit of problem diagnosis, this book takes pains to describe this failure, which is necessarily a story of the programmer's failure. Some of them took offense at my descriptions, imagining that I was maligning or blaming programmers for bad software. They are certainly the *agents* by which bad software is created, but they are by no means culpable. I do not blame programmers for hard-to-use software, and I'm very sorry to have given any programmer a contrary impression. With few exceptions, the programmers I know are diligent and conscientious in their desire to please end users and are unceasing in their efforts to improve their programs' quality. Just like users, programmers are simply another victim of a flawed process that leaves them too little time, too many conflicting orders, and utterly insufficient guidance. I am very sorry to have given any programmers the impression that I fault them.

The intractability of the software-construction process—particularly the high cost of programming and the low quality of interaction—is simply not a technical problem. It is the result of business practices imposed on a discipline—software programming—for which they are obsolete. With pure hearts, the best of intentions, and the blessing of upper management, programmers attempt to fix this problem by engineering even harder. But more or better engineering cannot solve these problems. Programmers sense the growing futility of their efforts, and their frustration mounts.

In my recent travels I have noticed a growing malaise in the community of programmers. Sadly, it is the best and most experienced of them who are afflicted the worst. They reflect cynicism and ennui about their efforts because they know that their skills are being wasted. They may not know exactly how they are misapplied, but they cannot overlook the evidence. Many of the best programmers

have actually stopped programming because they find the work frustrating. They have retreated into training, evangelism, writing, and consulting because it doesn't feel so wasteful and counterproductive. This is a tragic and entirely avoidable loss. (The open-source movement is arguably a haven for these frustrated programmers—a place where they can write code according to their own standards and be judged solely by their peers, without the advice or intervention of marketers or managers.)

Programmers are not given sufficient time, clear enough direction, or adequate designs to enable them to succeed. These three things are the responsibility of business executives, and they fail to deliver them for preventable reasons, not because they are stupid or evil. They are simply not armed with adequate tools for solving the complex and unique problems that confront them in the information age. Now here I am sounding like I'm slamming people again, only this time businesspeople are in my sights instead of programmers. Once again, to solve the problem one must deconstruct it. I'm questing after solutions, not scapegoats.

Management sage Peter Drucker can see the problem from his unique viewpoint, having both observed and guided executives for the majority of his 92 years. In a recent interview in CIO magazine, he commented on the wide-eyed optimism of executives in the 1950s and 1960s as digital computers first nudged their way into their businesses. Those executives imagined that computers "would have an enormous impact on how the business was run," but Drucker exclaims, "This isn't what happened. Very few senior executives have asked the question, 'What information do I need to do my job?'" Although digital computers have given executives unprecedented quantities of data, few have asked whether this data is appropriate for guiding the corporation. Operations have changed dramatically, but management has not followed suit. Drucker accuses our obsolete accounting systems, born in mercantilism, come of age in an era of steam and iron, and doddering into senility in the dawning twenty-first century information age. Drucker asserts, "The information you need the most is about the outside world, and there is absolutely none."

During the last few years of the twentieth century, as the dot-com bubble inflated, truckloads of ink were used to sell the idea that there was a "new economy" on the Internet. The pundits said that selling things on the World Wide Web, where stores were made of clicks instead of bricks, was a fundamentally different way of doing business, and that the "old economy" was as good as dead. Of course, almost all of those new-economy companies are dead and gone, the venture capitalists who backed them are in shock, and the pundits who pitched the new economy have now recanted, claiming it was all a hopeless dream. The new, new thinking says we must still be in the old, old economy.

Actually, I believe that we really *are* in a new economy. What's more, I think that the dot-coms never even participated in it. Instead, the dot-coms were the last gasp of the *old* economy: the economy of manufacturing.

In the industrial age, before software, products were *manufactured* from solid material—from atoms. The money it took to mine, smelt, purchase, transport, heat, form, weld, paint, and transport dominated all other expenditures. Accountants call these "variable costs" because that expense varies directly with each product built. "Fixed costs," as you might expect, don't vary directly and include things such as corporate administration and the initial cost of the factory.

The classic rules of business management are rooted in the manufacturing traditions of the industrial age. Unfortunately, they have yet to address the new realities of the information age, in which products are no longer made from atoms but are mostly software, made only from the arrangements of bits. And bits don't follow the same economic rules that atoms do.

Some fundamental truths hold for both the old and the new economies. The goal of all business is to make a sustainable profit, and there is only one legal way to do so: Sell some goods or services for more money than it costs you to make or acquire them. It follows that there are two ways to increase your profitability: Either reduce your costs or increase your revenues. In the old economy, reducing your costs worked best. In the new economy, increasing your revenue works much, much better.

Today's most vital and expensive products are made largely or completely of software. They consume no raw materials. They have no manufacturing cost. They have no transportation cost. There is no welding, hammering, or painting. This is the real difference between the industrial-age economy and the information-age economy: In the information age, there is little or no variable cost, whereas in the late industrial age, variable cost was the dominant factor. Indeed, the absence of variable cost is what *makes* this a new economy.

Is the salary you pay the programmers on your staff a fixed cost or a variable cost? One hour of programming is definitely not directly related to one product sale; you can sell that same code over and over again. An investment in programming can be leveraged across millions of salable items, just as an investment in a factory is leveraged across all the products built within it.

Writing software is not a variable cost, but it's not really a fixed cost either. Writing software is an ongoing, revenue-generating operation of the company, and it is not the same as constructing a factory. The expensive craftsmen who build the factory leave and go to work on some other job after the building is erected. Programmers are far more expensive than carpenters or ironworkers, and they never go away because their work is apparently never completed. Some

might suggest that programming is research and development, and there are similarities. However, R & D is the thinking and experimenting done to establish the theoretical viability of a product and is not performed the same way that products are built in a production environment. Fittingly, traditional accounting separates R & D expenditures from the daily operations that generate revenue. Writing software doesn't work well in any of those old business-accounting categories.

Now, you might discount this little terminology mismatch as a minor quibble for bean-counters with green eyeshades to debate over beers, but it actually has a huge effect on how software is funded, managed, and—most significantly—regarded by senior executives.

Programmers create software, and business executives create revenue streams and profit centers. Programmers measure their success by the quality of the product, and business executives measure their success by the profitability of their investments. They measure this profitability by applying the language of business mathematics, which recognizes fixed costs, variable costs, corporate overhead, and research and development, but, unfortunately, it has no model appropriate for software or programming. Accounting is the basic language of business, and these categories are so fundamental to all business measurement and communication that contemporary executives have completely internalized them. They see programming as simply another corporate expense to be fitted into an already existing category. In practice, most executives simply treat programming as a manufacturing effort—a variable cost. (For tax purposes, most software companies account for programming as R & D, but it is regarded as a variable cost in every other respect.) This is the worst possible choice because it hopelessly prejudices their business decision making.

The key advantage of the industrial age was that products could be mass-produced, which means they could be made available to the masses at affordable prices. The advantage to customers was the availability of functions that were previously unavailable or only expensively hand built for the wealthy. Companies competed on the basis of their sales prices, which were directly related to their variable costs: the cost of manufacturing and shipping. In the information age, it is taken for granted that products are available at affordable prices to everyone. After all, software can be downloaded and distributed to any number of customers for essentially no cost and with little or no human effort.

Remember, businesses can grow profits by increasing revenue or reducing costs. That is, a business can increase its fixed-cost investment, improving its product's quality, which increases its pricing strength, or it can reduce its variable cost, which means decreasing the cost of manufacturing. In the old manufacturing economy of atoms, reducing costs was simple and effective, and it was the

preferred tactic. When today's executives regard programming the same as manufacturing, they imagine that reducing the cost of programming is similarly simple and effective. Unfortunately, those rules don't apply anymore.

Because software has relatively insignificant variable costs, there is little business advantage to be had in reducing them. Programmers' salaries appear to be a variable cost from an accountant's point of view, but they are much more like a long-term investment—a fixed cost. Reducing the cost of programming is not the same as reducing the cost of manufacturing. It's more like giving cheap tools to your workers than it is like giving the workers smaller paychecks. The companies that are shipping programming jobs overseas in order to pay reduced salaries are missing the point entirely.

What's more, the only available economic upside comes from making your product or service more desirable by improving its quality, and you can't do that by reducing the money you spend designing or programming it. In fact, you need to invest *more* time and money on the research, thinking, planning, and designing phase to make your results better suited to your customers' needs.

Of course, this requires a mode of thinking that is quite unfamiliar to twenty-first century businesspeople. Instead of *reducing* what they spend to build *each* object, they need to *increase* what they spend to build *all* objects. This is the essence of the real new economy and precisely what Peter Drucker was talking about.

Modern pharmaceutical companies inventing high-tech drugs share some similarities to the new software economy. The actual manufacturing cost of a single pill is miniscule, but the development costs can run to billions of dollars over a decade or more. The upside of shipping a new miracle drug can be boundless, but there is only a catastrophic downside in shipping that drug before it has been developed completely. Pharmaceuticals know that reducing development costs is not a viable business strategy.

Like inventing medicine, building software isn't the same as building a factory. The factory is a physical asset that a company owns, and the factory workers are largely interchangeable. The intangible but extremely complicated patterns of thought that is software has value only when accompanied by the programmers who wrote it. No company can treat programmers the same as a factory. Programmers demand continuous attention and support well above that of any factory.

Architecture—the human design part of programming, in which users are studied, use scenarios are defined, interaction is designed, form is determined, and behavior is described—is the part of the software-construction process that is most frequently dispensed with as a cost-saving measure. It is certainly possible to do too much design, but there is no advantage in reducing it. Every dollar or hour spent on architecture will yield tenfold savings during programming. Additionally, when

you invest a sufficient amount of competent design, your product becomes very desirable, which means that it will make more money for you. Its desirability will establish your brand, increase your ability to raise prices, generate customer loyalty, and give your product a longer, stronger lifespan. Although there's no advantage in cost reduction, there is big advantage in quality enhancement. Ironically, the best way to increase profitability in the information age is to spend more.

Unfortunately, most executives have an almost irresistible desire to reduce the time and money invested in programming. They see, incorrectly, the obsolete advantage in reducing costs. What they don't see is that reduction in investment in programming has strong negative effects on a product's long-term quality, desirability, and therefore profitability. Of course, simply spending more money doesn't guarantee improvement, and it can often make things worse when additional money is unaccompanied by wisdom, analysis, and guidance. My first mentor, Dan Joaquin, used to say that the old maxim "You get what you pay for" should properly be inverted to "You *don't* get what you *don't* pay for." Proceeding without proper planning risks spending *way* too much. The trick is to spend the correct amount, and that demands significant expertise in software-construction management. It also demands process tools that provide managers with the insight and information they need to make the correct decisions. Providing those tools is this book's goal.

The dot-com boom was populated with companies whose entire business model consisted of the reduction of variable costs. Although many dot-coms claimed various online advantages, their Web sites were sufficiently ponderous and unhelpful to be far less satisfying than simply driving to the mall. Dot-com founders swooned with ecstasy (as did the press) because they could establish a retail enterprise for a remarkably lower variable cost. Their complete and spectacular failure demonstrated beyond doubt that the economic rules of the information age are different from those of the industrial age.

In the old economy, lower variable costs meant wider distribution and lower retail costs. Those twin advantages directly benefited the consumer, and they are the foundation for the economic success of the industrial revolution. In the new economy, business success depends on adding something new and better for the consumer. The actual quality of every part of the transaction, from browsing to comparison shopping to comprehensiveness, must be noticeably better for the end user. Wading through 11 screens only to have to telephone the company anyway is far less satisfying than making the purchase conventionally. Entering your name, address, and credit card information three or four times, only to find that the site can't sell you everything you need and a trip to the atom-based store is necessary anyway, has the unfortunate effect of making the entire online sale completely unnecessary and undesirable. Today, simply lowering costs for the vendor doesn't guarantee success.

When Pets.com sold dog food over the Internet, it didn't offer better dog food, and it didn't offer a customer experience better than you could get at the local brick-and-mortar pet store; it didn't offer any better information, intelligence, or confidence. All it offered was cheaper shipping, stocking, and selling—variable costs all—for Pets.com. It was a classic industrial-age-economy tactic of cost reduction that ignored the fundamental principles of the new economy. Far from being the first breath of a new economy, it was the last gasp of the old.

I am absolutely convinced that you can sell *anything* on the Internet profitably and successfully. The trick is that your online store must offer a measurably greater degree of shopper satisfaction than any competing retail medium, and price is only one small component of satisfaction. There is only one way to accomplish this: You must architect your system to deliver the highest possible end-user satisfaction. Treating any aspect of software design and construction as if it were a manufacturing process courts failure. The design and programming of software is simply not a viable target for conventional cost-reduction methods. It's certainly possible to spend too much time and money on building software, but the danger of spending too little is far greater.

Such danger is probably not shocking or unfamiliar to you, but it is nearly inconceivable to most senior business executives who are responsible for running big companies. Those execs are still using accounting models popular in the age of steam, yet every aspect of their companies is fully dependent on software for operations, decision making, communications, and finance. The terms and concepts those executives use are simply not cognizant of the unique nature of doing business in an era when the tools and products of commerce are intangible arrangements of bits instead of railroad carloads of iron. The sock puppets were cool, though.

Even though corporations are hiring interaction designers and applying goal-directed methods, the quality of our software products hasn't actually improved that much. What's more, the high cost of programming and the basic intractability of the software-construction process remain ever-present. Why?

*Change is impossible until senior business executives realize that software problems are not technical issues, but are significant business issues.* Our problems will remain unsolved until we change our *process* and our *organization*.

Not only do companies follow obsolete financial models, but they also follow an inappropriate organizational model. This model is copied directly from academia, where the act of creating software is entangled with the planning and engineering of that software. Such is the nature of research. Tragically, and apparently without notice, this paradigm has been carried over intact into the world of business, where it does not belong.

All modern manufacturing disciplines have roots in preindustry except software, whose unique medium appeared well after industrialization was a fait accompli. Only programming comes directly from academia, where there are no time limits on research, student power is dirt cheap, profit is against the rules, and a failing program can be considered a very successful experiment. It's not a coincidence that Microsoft, IBM, Oracle, and other leading software companies reside in "campuses." Universities never have to make money, hit deadlines, or build desirable, useful products.

All nonsoftware businesses begin with research and end with mass production and distribution of their products or services. They plan carefully in between, cognizant of the dangers to both bank account and reputation if they attempt premature production of an ill-conceived product. They know that time, thought, and money invested in planning will pay big dividends in the smoothness and speed of manufacturing and the popularity and profitability of their end products.

In all other construction disciplines, engineers plan a construction strategy that craftsmen execute. Engineers don't build bridges; ironworkers do. Only in software is the engineer tasked with actually building the product. Only in software is the "ironworker" tasked with determining how the product will be constructed. Only in software are these two tasks performed concurrently instead of sequentially. But companies that build software seem totally unaware of the anomaly. Engineering and construction are so crossbred as to be inseparable and apparently indistinguishable by practitioners or executives. Planning of all sorts is either omitted or delayed until far too late. Profoundly complex technical engineering problems are habitually left unsolved until construction of code intended for public release is well underway, when it is too economically embarrassing to back up.

Architecture must be integrated into early-stage engineering planning. In fact, it should drive early-stage engineering, but because such engineering is typically deferred until construction has begun and is corrupted by intermingling with production coding, the architectural design lacks an entry point into the construction process. Despite the fact that companies are hiring interaction designers and retraining their usability testers to create personas, their work has little effect on either the cost of construction or the quality of the finished product.

The solution lies in the hands of corporate presidents and chief executive officers. When these execs delegate the solution to their chief technology officers or vice presidents of engineering they miss the point. Those worthy officers are technicians, and the problem is not a technical one. As Drucker pointed out, the accounting tools CEOs depend on simply do not represent the true state of their organizations. It's like saying that because the speedometer is accurate the car is headed in the right direction. In a business world dominated by digital technology, that is simply no longer true.

One of the biggest problems of applying incorrect accounting and organizational methods to software construction is that executives don't realize how much of their programming dollar is wasted. An accurate system would show that at least one half of every dollar is misspent and that it takes another two or three dollars to fix the problems caused by the initial bad investment. In any other business, such statistics would be cause for revolution, but in software we remain in a state of blissful ignorance.

Over the past 13 years my company, Cooper, has consulted with hundreds of companies. My talented designers have provided most of them with blueprints for products that would help them enormously, yet only a handful have been able to take full advantage of them. Most of them treat interaction design and software architecture as advice, and their programmers and engineers *always* have the last word. None of those companies' CEOs has any clue as to what is really going on in the engineers' cubicles, so they squeeze the schedule without reason. The programmers are always working in an environment of scarcity, primarily lacking time to program well, but also lacking the time to determine what should be programmed at all. They are forced to protect themselves by rejecting advice and prevaricating to their managers.

I believe that there are two kinds of executives: those who are engineers, and those who are terrified of engineers. The former propagate the familiar problems because their viewpoint is hopelessly blinkered by a conflict of interest. The latter propagate them because they cannot speak the language of programmers. I don't mean Java or C#. I mean that business people and programmers lack common tools and common goals. *Homo sapiens* delegate human problems to *Homo logicus* and are unaware that the solution could be so much better if they applied—at the executive level—appropriate financial and organizational models instead.

There is a colossal opportunity for companies to break this logjam and organize around customer satisfaction instead of around software, around personas instead of around technology, around profit instead of around programmers. I eagerly await the enlightened executive who seizes this chance and forever alters the way software is built by providing the industry with a bold and successful example.

Alan Cooper
Menlo Park, California
October 2003
`http://www.cooper.com`
`inmates@cooper.com`

# 3

## Wasting Money

It's harder than you might think to squander millions of dollars, but a flawed software-development process is a tool well suited to the job. That's because software development lacks one key element: an understanding of what it means to be "done." Lacking this vital knowledge, we blindly bet on an arbitrary deadline. We waste millions to cross the finish line soonest, only to discover that the finish line was a mirage. In this chapter I'll try to untangle the expensive confusion of deadline management.

### *Deadline Management*

There is a lot of obsessive behavior in Silicon Valley about time to market. It is frequently asserted that shipping a product *right now* is far better than shipping it later. This imperative is used as a justification for setting impossibly ambitious ship dates and for burning out employees, but this is a smoke screen that hides bigger, deeper fears—a red herring. Shipping a product that angers and frustrates users in three months is *not* better than shipping a product that pleases users in six months, as any businessperson knows full well.

Managers are haunted by two closely related fears. They worry about when their programmers will be done building, and they doubt whether the product will be good enough to ultimately succeed in the marketplace. Both of these fears stem from the typical manager's lack of a clear vision of what the finished product actually will consist of, aside from mother-and-apple-pie statements such as "runs on the target computer" and "doesn't crash." And lacking this vision, they cannot assess a product's progress towards completion.

The implication of these two fears is that as long as it "doesn't crash," there isn't much difference between a program that takes three months to code and one that takes six months to code, except for the prodigious cost of three months of unnecessary programming. After the programmers have begun work, money drains swiftly. Therefore, logic tells the development manager that the most important thing to do is to get the coding started as soon as possible and to end it as soon as possible.

The conscientious development manager quickly hires programmers and sets them coding immediately. She boldly establishes a completion date just a few months off, and the team careens madly toward the finish line. But without product design, our manager's two fears remain unquelled. She has not established whether the users will like the product, which indeed leaves its success a mystery. Nor has she established what a "complete" product looks like, which leaves its completion a mystery. Later in the book, I'll show how interaction design can ease these problems. Right now, I'll show how thoroughly the deadline subverts the development process, turning all the manager's insecurities into self-fulfilling prophecies.

## *What Does "Done" Look Like?*

After we have a specific description of what the finished software will be, we can compare our creation with it and really *know* when the product is done.

There are two types of descriptions. We can create a very complete and detailed physical description of the actual product, or we can describe the reaction we'd like the end user to have. In building architecture, for example, blueprints fill the first requirement. When planning a movie or creating a new restaurant, however, we focus our description on the feelings we'd like our clients to experience. For software-based products, we must necessarily use a blend of the two.

Unfortunately, most software products never *have* a description. Instead, all they have is a shopping list of features. A shopping bag filled with flour, sugar, milk, and eggs is not the same thing as a cake. It's only a cake when all the steps of the recipe have been followed, and the result looks, smells, and tastes substantially like the known characteristics of a cake.

Having the proper ingredients but lacking any knowledge of cakes or how to bake, the ersatz cook will putter endlessly in the kitchen with only indeterminate results. If we demand that the cake be ready by 6 o'clock, the conscientious cook will certainly bring us a platter at the appointed hour. But will the concoction be a cake? All we know is that it is on time, but its success will be a mystery.


In most conventional construction jobs, we know we're done because we have a clear understanding of what a "done" job looks like. We know that the building is completed because it looks and works just like the blueprints say it should look and work. If the deadline for construction is June 1, the arrival of June doesn't necessarily mean that the building is done. The relative completeness of the building can only be measured by examining the actual building in comparison to the plans.

Without blueprints, software builders don't really have a firm grasp on what makes the product "done," so they pick a likely date for completion, and when that day arrives they declare it done. It is June 1; therefore, the product is completed. "Ship it!" they say, and the deadline becomes the sole definition of project completion.

The programmers and businesspeople are neither stupid nor foolish, so the product won't be in complete denial of reality. It will have a robust set of features, it will run well, and it won't crash. The product will work reasonably well when operated by people *who care deeply* that it works well. It might even have been subjected to usability testing, in which strangers are asked to operate it under the scrutiny of usability professionals[1]. But, although these precautions are only reasonable, they are insufficient to answer the fundamental question: Will it succeed?

## *Parkinson's Law*

Managers know that software development follows Parkinson's Law: Work will expand to fill the time allotted to it. If you are in the software business, perhaps you are familiar with a corollary to Parkinson called the Ninety-Ninety Rule, attributed to Tom Cargill of Bell Labs: "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time." This self-deprecating rule says that when

[1] *Usability professionals are not interaction designers. I discuss this difference in detail in Chapter 12, "Desperately Seeking Usability."*

the engineers have written 90% of the code, they *still* don't know where they are! Management knows full well that the programmers won't hit their stated ship dates, regardless of what dates it specifies. The developers work best under pressure, and management uses the delivery date as the pressure-delivery vehicle.

In the 1980s and 1990s, Royal Farros was the vice president of development for T/Maker, a small but influential software company. He says, "A lot of us set deadlines that we *knew* were impossible, enough so to qualify for one of those Parkinson's Law corollaries. 'The time it will take to finish a programming project is twice as long as the time you've allotted for it.' I had a *strong* belief that if you set a deadline for, say, six months, it would take a year. So, if you had to have something in two years, set the deadline for one year. Bonehead sandbagging, but it always worked."

When software entrepreneur Ridgely Evers was with Intuit, working on the creation of QuickBooks, he experienced the same problem. "The first release of QuickBooks was supposed to be a nine-month project. We were correct in estimating that the development period would be the same as a gestation period, but we picked the wrong species: It took almost two-and-a-half years, the gestation period for the elephant."

Software architect Scott McGregor points out that Gresham's Law—that bad currency drives out good—is also relevant here. If there are two currencies, people will hoard the good one and try to spend the bad one. Eventually, only the bad currency circulates. Similarly, bad schedule estimates drive out good ones. If everybody makes bogus but rosy predictions, the one manager giving realistic but longer estimates will appear to be a heel-dragger and will be pressured to revise his estimates downward.

Some development projects have deadlines that are unreasonable by virtue of their arbitrariness. Most rational managers still choose deadlines that, while reachable, are only reachable by virtue of extreme sacrifice. Sort of like the pilot saying, "We're gonna make Chicago on time, but only if we jettison all our baggage!" I've seen product managers sacrifice not only design, but testing, function, features, integration, documentation, and reality. *Most product managers that I have worked with would rather ship a failure on time than risk going late.*

## *The Product That Never Ships*

This preference is often due to every software development manager's deepest fear: that after having become late, the product will never ship at all. Stories of products never shipping are not apocryphal. The project goes late, first by one year, then two years, then is euthanized in its third year by a vengeful upper management or board of directors. This explains the rabid adherence to deadlines, even at the expense of a viable product.

For example, in the late 1990s, at the much-publicized start-up company Worlds, Inc., many intelligent, capable people worked on the creation of a virtual, online world where people's avatars could wander about and engage other avatars in real-time conversation. The product was never fully defined or described, and after tens of millions of investment capital was spent, the directors mercifully pulled the plug.

In the early 1990s, another start-up company, Nomadic Computing, spent about $15 million creating a new product for mobile businesspeople. Unfortunately, no one at the company was quite sure what its product was. They knew their market, and most of the program's functions, but weren't clear on their users' goals. Like mad sculptors chipping away at a huge block of marble hoping to discover a statue inside, the developers wrote immense quantities of useless code that was all eventually thrown away, along with money, time, reputations, and careers. The saddest waste, though, was the lost opportunity for creating software that really was wanted.

Even Microsoft isn't immune from such wild goose chases. Its first attempt at creating a database product in the late 1980s consumed many person-years of effort before Bill Gates mercifully shut it down. Its premature death sent a shock wave through the development community. Its successor, Access, was a completely new effort, staffed and managed by all new people.

## *Shipping Late Doesn't Hurt*

Ironically, shipping late generally isn't fatal to a product. A third-rate product that ships late often fails, but if your product delivers value to its users, arriving behind schedule won't necessarily have lasting bad effects. If a product is a hit, it's not a big deal that it ships a month—or even a year—late. Microsoft Access shipped several years late, yet it has enjoyed formidable success in the market. Conversely, if a product stinks, who cares that it shipped on time?

Certainly, some consumer products that depend on the Christmas season for the bulk of their sales have frighteningly important due dates. But most software-based products, even consumer products, aren't that sensitive to any particular date.

For example, in 1990 the PenPoint computer from GO was supposed to be the progenitor of a handheld-computer revolution. In 1992, when the PenPoint crashed and burned, the Apple Newton inherited the promise of the handheld revolution. When the Newton failed to excite people, General Magic's Magic Link computer became the new hope for handhelds. That was in 1994. When the Magic Link failed to sell, the handheld market appeared dead. Venture capitalists declared it a dry hole. Then, out of nowhere, in 1996, the PalmPilot arrived to universal acclaim. It seized the handheld no-man's-land *six years late*. Markets are always ready for good products that deliver value and satisfy users.

Of course, companies with a long history of making hardware-only products now make hybrid versions containing chips and software. They tend to underestimate the influence of software and subordinate it to the already-established completion cycles of hardware. This is wrong because as Chapter 1, "Riddles for the Information Age," showed, these companies are now in the software business, whether or not they know it.

## *Feature-List Bargaining*

One consequence of deadline management is a phenomenon that I call "feature-list bargaining."

Years ago programmers got burned by the vague product-definition process consisting of cocktail-napkin sketches, because they were blamed for the unsuccessful software that so often resulted. In self-defense, programmers demanded that managers and marketers be more precise. Computer programs are procedural, and procedures map closely to features, so it was only natural that programmers would define "precision" as a list of features. These feature lists allowed programmers to shift the blame to management when the product failed to live up to expectations. They could say, "It wasn't my fault. I put in all the features management wanted."

Thus, most products begin life with a document variably called a "marketing specification," "technical specification," or "marketing requirements document." It is really just a list of desired features, like the list of ingredients in the recipe for cake. It is usually the result of several long brainstorming sessions in which managers, marketers, and developers imagine what features would be cool and jot them down. Spreadsheet programs are a favorite tool for creating these lists, and a typical one can be dozens of pages long. (Invariably, at least one of the line items will specify a "good user interface.") Feature suggestions can also come from focus groups, market research, and competitive analysis.

The managers then hand the feature list to the programmers and say, "The product must ship by June 1." The programmers—of course—agree, but they have some stipulations. There are far too many features to create in the time allotted, they claim, and many of them will have to be cut to meet the deadline. Thus begins the time-honored bargaining.

The programmers draw a dividing line midway through the list. Items above it will be implemented, they declare, while those below the "line of death" are postponed or eliminated. Management then has two choices: to allow more time or to cut features. Although the project will inevitably take more time, management is loath to play that trump so early in the round, so it negotiates over features. Considerable arguing and histrionics occur. Features are traded for time; time is traded for features. This primitive capitalist negotiation is so human and

natural that both parties are instantly comfortable with it. Sophisticated parallel strategies develop. As T/Maker's Royal Farros points out, when one "critical-path feature was blamed for delaying a deadline, it would let a dozen other tardy features sneak onto the list without repercussion." Lost in the battle is the perspective needed for success.

Farros described T/Maker's flagship product, a word processor named WriteNow, as "a perfect product for the university marketplace. In 1987, we actually shipped more copies of WriteNow to the university market than Microsoft shipped Word. However, we couldn't hold our lead because we angered our very loyal, core fans in this market by not delivering the one word-processor feature needed in a university setting: *endnotes*. Because of trying to make the deadline, we could never slip this feature into the specification. We met our deadline but lost an entire market segment."

## *Programmers Are in Control*

Despite appearances, programmers are completely in control of this bottom-up decision-making process. They are the ones who establish how long it will take to implement each item, so they can force things to the bottom of the list merely by estimating them long. The programmers will—in self-defense—assign longer duration to the more nebulously defined items, typically those concerned with substantive user-interface issues. This inevitably causes them to migrate to the bottom of the list. More familiar idioms and easy-to-code items, such as menus, wizards, and dialog boxes, bubble to the top of the list. All of the analysis and careful thinking done by high-powered and high-priced executives is made moot by the unilateral cherry picking of a programmer following his own muse or defending his turf.

Like someone only able to set the volume of a speaker that isn't within hearing distance, managers find themselves in the unenviable position of only having tools that control ineffective parameters of the development process. It is certainly true that management needs to control the process of creating and shipping successful software, but, unfortunately, our cult of deadline ignores the "successful" part to concentrate only on the "creating" part. We give the creators of the product the reins to the process, thus relegating management to the role of passenger and observer.

## ***Features Are Not Necessarily Good***

Appearances to the contrary, users aren't really compelled by features. Product successes and failures have shown repeatedly that users don't care that much about features. Users only care about achieving their goals. Sometimes features are needed to reach goals, but more often than not, they merely confuse users

and get in the way of allowing them to get their work done. Ineffective features make users feel stupid. Borrowing from a previous example, the successful PalmPilot has far fewer features than did General Magic's failed Magic Link computer, Apple's failed Newton, or the failed PenPoint computer. The PalmPilot owes its success to its designers' single-minded focus on its target user and the objectives that user wanted to achieve.

About the only good thing I can say about features is that they are quantifiable. And that quality of being countable imbues them with an aura of value that they simply don't have. Features have negative qualities every bit as strong as their positive ones. The biggest design problem they cause is that every well-meant feature that *might possibly* be useful obfuscates the few features that *will probably* be useful. Of course, features cost money to implement. They add complexity to the product. They require an increase in the size and complexity of the documentation and online help system. Above all, cost-wise, they require additional trained telephone tech-support personnel to answer users' questions about them.

It might be counterintuitive in our feature-conscious world, but you simply cannot achieve your goals by using feature lists as a problem-solving tool. It's quite possible to satisfy every feature item on the list and still hatch a catastrophe. Interaction designer Scott McGregor uses a delightful test in his classes to prove this point. He describes a product with a list of its features, asking his class to write down what the product is as soon as they can guess. He begins with 1) internal combustion engine; 2) four wheels with rubber tires; 3) a transmission connecting the engine to the drive wheels; 4) engine and transmission mounted on metal chassis; 5) a steering wheel. By this time, every student will have written down his or her positive identification of the product as an automobile, whereupon Scott ceases using features to describe the product and instead mentions a couple of user goals: 6) cuts grass quickly and easily; 7) comfortable to sit on. From the five feature clues, not one student will have written down "riding lawnmower." You can see how much more descriptive goals are than features.

## *Iteration and the Myth of the Unpredictable Market*

In an industry that is so filled with money and opportunities to earn it, it is often just easier to move right along to another venture and chalk up a previous failure to happenstance, rather than to any *real* reason.

I was a party to one of these failures in the early 1990s. I helped to start a venture-funded company whose stated goal was to make it absurdly simple to network PCs together.[2] The product worked well and was easy to use, but a tragic series of

[2] *Actually, we said that we wanted to make it "as easy to network Intel/Windows computers as it was to network Macintosh computers." At the time, it was ridiculously simple to network Macs together with AppleTalk. Then, as now, it was quite difficult to network Wintel PCs together.*

self-inflicted marketing blunders caused it to fail dismally. I recently attended a conference where I ran into one of the investors who sat on the doomed company's board of directors. We hadn't talked since the failure of the company, and—like veterans of a battlefield defeat meeting years later—we consoled each other as sadder but wiser men. To my unbridled surprise, however, this otherwise extremely successful and intelligent man claimed that in hindsight he had learned a fundamental lesson: Although the marketing, management, and technical efforts had been flawless, the buying public "just wasn't interested in easy-to-install local area networks." I was flabbergasted that he would make such an obviously ridiculous claim and countered that surely it wasn't lack of desire, but rather our failure to satisfy the desire properly. He restated his position, arguing forcefully that we had demonstrated that easy networking just wasn't something that people wanted.

Later that evening, as I related this story to my wife, I realized that his rationalization of the failure was certainly convenient for all the parties involved in the effort. By blaming the failure on the random fickleness of the market, my colleague had exonerated the investors, the managers, the marketers, and the developers of any blame. And, in fact, each of the members of that start-up has gone on to other successful endeavors in Silicon Valley. The venture capitalist has a robust portfolio of other successful companies.

During development, the company had all the features itemized on the feature list. It stayed within budget. It shipped on schedule. (Well, actually, we kept extending the schedule, but it shipped on *a* schedule.) All the quantitatively measurable aspects of the product-development effort were within acceptable parameters. The only conclusion this management-savvy investor could make was the existence of an unexpected discontinuity in the marketplace. How could *we* have failed when all the meters were in the green?

The fact that these measures are objective is reassuring to everyone. Objective and quantitative measure is highly respected by both programmers and businesspeople. The fact that these measures are usually ineffective in producing successful products tends to get lost in the shuffle. If the product succeeds, its progenitors will take the credit, attributing the victory to their savvy understanding of technology and marketing.

On the other hand, if the product fails, nobody will have the slightest motivation to exhume the carcass and analyze the failure. Almost any excuse will do, as long as the players—both management and technical—can move along to the next high-tech opportunity, of which there is an embarrassment of riches. Thus, there is no reason to weep over the occasional failure. The unfortunate side effect of not understanding failure is the silent admission that success is not predictable—that luck and happenstance rule the high-tech world. In turn, this gives rise to what

the venture capitalists call the "spray and pray" method of funding: Put a little bit of money into a lot of investments and then hope that one of them gets lucky.

⌘

Rapid-development environments such as the World Wide Web—and Visual Basic before it—have also promoted this idea of simply iterating until something works. Because the Web is a new advertising medium, it has attracted a multitude of marketing experts who are particularly receptive to the myth of the unpredictable market and its imperative to iterate. Marketers are familiar with the harsh and arbitrary world of advertising and media. After all, much of advertising *really is* random guesswork. For example, in advertising, "new" is the single most effective marketing concept, yet when Coca-Cola introduced "New Coke" in the mid-1980s, it failed utterly. Nobody could have predicted this result. People's tastes and styles change randomly, and the effectiveness of marketing can appear to be random.

On the Web, the problem arises when a Web site matures from the online-catalog stage into the online-store stage. It changes from a one-way presentation of data to an interactive software application. The advertising and media people who had such great success with the first-generation site now try their same iteration methods on the interactive site and run into trouble, often without realizing it. Marketing results may be random, but interaction is not. The cognitive friction generated by the software's interactivity is what gives the impression of randomness to those untrained in interaction design.

The remarkably easy-to-change nature of the World Wide Web plays into this because an advertisement or marketing campaign can be aired for a tiny fraction of the cost (and time) of print or TV advertising. The savvy Web marketer can get almost instantaneous feedback on the effectiveness of an ad, so the speed of the iteration increases dramatically, and things are hacked together overnight. In practice, it boils down to "throw it against the wall and see what sticks." Many managers of Web start-ups use this embarrassingly simple doctrine of design by guesswork. They write any old program that can be built in the least time and then put it before their users. They then listen to the complaints and feedback, measure the patterns of the user's navigation clicks, change the weak parts, and then ship it again.

Generally, programmers aren't thrilled about the iterative method because it means extra work for them. Typically, it's managers new to technology who like the iterative process because it relieves them of having to perform rigorous planning, thinking, and product due diligence (in other words, interaction design). Of course, it's the users who pay the dearest price. They have to suffer through one halfhearted attempt after another before they get a program that isn't too painful.

Just because customer feedback improves your understanding of your product or service, you cannot then deduce that it is efficient, cheap, or even effective to toss random features at your customers and see which ones are liked and which are disliked. In a world of dancing bears, this can be a marginally viable strategy, but in any market in which there is the least hint of competition, it is suicidal. Even when you are all alone in a market, it is a very wasteful method.

Many otherwise sensitive and skilled managers are unashamedly proud of this method. One mature, experienced executive (a former marketing man) asked me, in self-effacing rhetoric, "How could anyone presume to know what the users want?" This is a staggering question. Every businessperson presumes. The value that most businesspeople bring to their market is precisely their "presumption" of what the customer wants. Yes, that presumption will miss the mark with *some* users, but not to presume at all means that *every* user won't like it. This foolish man believed that his customers didn't mind plowing through his guesses to do his design work for him. Today, in Silicon Valley, there might be lots of enthusiastic Web-surfing apologists who are willing to help this lazy executive figure out his business, but how many struggling survivors did he alienate with that haughty attitude? As he posted sketchy version after sketchy version of his site, reacting only to those people with the stamina to return to it, how many customers did he lose permanently? What did *they* want? It has been said that the way Stalin cleared a minefield was to march a regiment through it. Effective? Yes. Efficient, humanitarian, viable, desirable? No.

The biggest drawback, of course, is that you immediately scare away all survivors, and your only remaining users will be apologists. This seriously skews the nature and quality of your feedback, condemning you to a clientele of technoid apologists, which is a relatively small segment. This is one reason why so few personal-computer software-product makers have successfully crossed over into mass markets.

I am not saying that you cannot learn from trial and error, but those trials should be informed by something more than random chance and should begin from a well-thought-out solution, not an overnight hack. Otherwise, it's just giving lazy or ignorant businesspeople license to abuse consumers.

## *The Hidden Costs of Bad Software*

When software is frustrating and difficult to use, people will avoid using it. That is unremarkable until you realize that many people's jobs are dependent on using software. The corporate cost of software avoided is impossible to quantify, but it is real. Generally, the costs are not monetary ones, anyway, but are exacted in far more expensive currencies, such as time, order, reputation, and customer loyalty.

People who use business software might despise it, but they are paid to tolerate it. This changes the way people think about software. Getting paid for using software makes users far more tolerant of its shortcomings because they have no choice, but it doesn't make it any less expensive. Instead—while the costs remain high—they become very difficult to see and account for.

Badly designed business software makes people dislike their jobs. Their productivity suffers, errors creep into their work, they try to cheat the software, and they don't stay in the job very long. Losing employees is very expensive, not just in money but in disruption to the business, and the time lost can never be made up. Most people who are paid to use a tool feel constrained not to complain about that tool, but it doesn't stop them from feeling frustrated and unhappy about it.

One of the most expensive items associated with hard-to-use software is technical support. Microsoft spends $800 million annually on technical support. And this is a company that spends many hundreds of millions of dollars on usability testing and research, too. Microsoft is apparently convinced that support of this magnitude is just an unavoidable cost of doing business. I am not. Imagine the advantage it would give your company if you didn't make the same assumption that Microsoft did. Imagine how much more effective your development efforts would be if you could avoid spending over five percent of your net revenue on technical support.

Ask any person who has ever worked at any desktop-software company in technical support, and he will tell you that the one thing he spends most of his time and effort on is the file system. Just like Jane in Chapter 1, users don't understand the recursive hierarchy of the file system—the Finder or Explorer—on Windows, the Mac, or Unix. Surprisingly, very few companies will spend the money to design and implement a more human-friendly alternative to the file system. Instead, they accept the far more expensive option of answering phone calls about it in perpetuity.

You can blame the "stupid user" all you want, but you still have to staff those phones with expensive tech-support people if you want to sell or distribute within your company software that hasn't been designed.

## *The Only Thing More Expensive Than Writing Software Is Writing Bad Software*

Programmers cost a lot, and programmers sitting on their hands waiting for design to be completed gall managers in the extreme. It seems foolish to have programmers sit and wait, when they could be programming, thinks the manager. It is false economy, though, to put programmers to work before the design is completed. After the coding process begins, the momentum of programming becomes unstoppable, and the design process must now respond to the needs of programmers, instead of vice versa. Indeed, it is foolish to have programmers wait, and by the simple expedient of having interaction designers plan your next product or release concurrently with the construction of this product or release, your programmers will never have to idly wait.

It is more costly in the long run to have programmers write the wrong thing than to write nothing at all. This truth is so counterintuitive that most managers balk at the very idea. After code is written, it is very difficult to throw it out. Like writers in love with their prose, programmers tend to have emotional attachments to their algorithms. Altering programs in midstride upsets the development process and wounds the code, too. It's hard on the manager to discard code because she is the one who paid dearly for it, and she knows she will have to spend even more to replace it.

If design isn't done before programming starts, it will never have much effect. One manager told me, "We've already got people writing code and I'm not gonna stop." The attitude of these cowboys is, "By the time you are ready to hit the ground, I'll have stitched together a parachute." It's a bold sentiment, but I've never seen it work.

Lacking a solid design, programmers continually experiment with their programs to find the best solutions. Like a carpenter cutting boards by eye until he gets one that fits the gap in the wall, this method causes abundant waste.

The immeasurability and intangibility of software conspires to make it nearly impossible to estimate its size and assess its state of completion. Add in the programmer's joy in her craft, and you can see that software development always grows in scope and time and never shrinks. We will always be surprised during its construction, unless we can accurately establish milestones and reliably measure our progress against them.

### *Opportunity Cost*

In the information age, the most expensive commodity is not the cost of building something, but the lost opportunity of what you are *not* building. Building a failure means that you didn't build a success. Taking three annual releases to get a good product means that you didn't create three good products in one release each.

Novell's core business is networking, but it attempted to fight Microsoft toe-to-toe in the office-applications arena. Although its failed efforts in the new market were expensive, the true cost was its loss of leadership in the networking market. The money is nothing compared to the singular potential of the moment. Netscape lost its leadership in the browser market in the same way when it decided to compete against Microsoft in the operating-system business.

Any developer of silicon-based products has to evaluate what the most important goals of its users are and steadfastly focus on achieving them. It is far too easy to be beguiled by the myriad of opportunities in high tech and to gamble away the main chance. Programmers—regardless of their intelligence, business acumen, loyalty, and good intentions—march to a slightly different drummer and can easily drag a business away from its proper area of focus.

## *The Cost of Prototyping*

Prototyping is programming, and it has the momentum and cost of programming, but the result lacks the resiliency of real code. Software prototypes are scaffolds and have little relation to permanent, maintainable, expandable code—the equivalent of stone walls. Managers, in particular, are loath to discard code that works, even if it is just a prototype. They can't tell the difference between scaffolding and stone walls.

You can write a prototype much faster than a real program. This makes it attractive because it seems so inexpensive, but real programming gives you a reliable program, and prototyping gives you a shaky foundation. Prototypes are experiments made to be thrown out, but few of them ever are. Managers look at the running prototype and ask, "Why can't we just use this?" The answer is too technically complex and too fraught with uncertainty to have sufficient force to dissuade the manager who sees what looks like a way to avoid months of expensive effort.

The essence of good programming is deferred gratification. You put in all of the work up front, and then you reap the rewards later. There are very few tasks that aren't cheaper to do manually. Once written, however, programs can be run a million times with no extra cost. The most expensive program is one that runs once. The cheapest program is the one that runs ten billion times. However, any inappropriate behavior will also be magnified ten billion times. Once out of the realm of little programs, such as the ones you wrote in school, the economics of

software take on a strange reversal in which the cheapest programs to own are the ones that are most expensive to write, and the most expensive programs to own are the cheapest to write.

Writing a big program is like making a pile of bricks. The pile is one brick wide and 1,000 bricks tall, with each brick laid right on top of the one preceding it. The tower can reach its full height only if the bricks are placed with great precision on top of one another. Any deviation will cause the bricks above to wobble, topple, and fall. If the 998th brick deviates by a quarter of an inch, the tower can still probably achieve 1,000 bricks, but if the deviation is in the fifth brick, the tower will never get above a couple dozen.

This is very characteristic of software, whose foundations are more sensitive to hacking than the upper levels of code. As any program is constructed, the programmer makes false starts and changes as she goes. Consequently, the program is filled with the scar tissue of changed code. Every program has vestigial functions and stubbed-out facilities. Every program has features and tools whose need was discovered sometime after construction began grafted onto it as afterthoughts. Each one of these scars is like a small deviation in the stack of bricks. Moving a button from one side of a dialog box to the other is like joggling the 998th brick, but changing the code that draws all button-like objects is like joggling the 5th brick. Object-oriented programming and the principles of encapsulation are defensive techniques whose sole purpose is to immunize the program from the effects of scar tissue. In essence, object orientation divides the 1,000-brick tower into 10 100-brick towers.

Good programmers spend enormous amounts of time and energy setting up to write a big program. It can take days just to set up the programming environment, before a line of product code is written. The proper libraries must be selected. The data must be defined. The storage and retrieval subsystems must be analyzed, defined, coded, and tested.

As the programmers proceed into the meat of the construction, they invariably discover mistakes in their planning and flaws in their assumptions. They are then faced with Hobson's choice of whether to spend the time and effort to back up and fix things from the start, or to patch over the problem wherever they are and accept the burden of the new scar tissue—the deviation. Backing up is always very expensive, but that scar tissue ultimately limits the size of the program—the height of the bricks.

Each time a program is modified for a new revision to fix bugs or to add features, scar tissue is added. This is why software must be thrown out and completely rewritten every couple of decades. After a while, the scar tissue becomes too thick to work well anymore.

Prototypes—by their very nature—are programs that are slapped together in a hurry so that the results can be assayed. What the programmer exchanges in order to build the prototype so speedily is the perfect squaring of the bricks. Instead of using the "right" data structures, information is thrown in helter-skelter. Instead of using the "right" algorithms, whatever code fragments happen to be lying around are drafted for service. Prototypes *begin* life as masses of scar tissue. They can never grow very large.

Some software developers have arrived at the unfortunate conclusion that modern rapid-prototyping tools—such as Visual Basic—are effective design tools. Rather than designing the product, they just whip out an extremely anemic version of it with a visual programming tool. This prototype typically becomes the foundation for the product. This trades away the robustness and life span of the product for an illusory benefit. You can get a better design with pencil and paper and a good methodology than you can with any amount of prototyping.

For those who are not designers, visualizing the form and behavior of software that doesn't yet exist is difficult, if not impossible. Prototypes have been drafted into the role of a visualization tool for these businesspeople. Because a prototype is a rough model created with whatever prebuilt facilities are most readily available, prototypes are by nature filled with expedient compromises. But software that actually works—regardless of how badly—exerts a powerful pull on those who must pay for its development. A running—limping—prototype has an uncanny momentum out of proportion to its real value.

It is all too compelling for the manager to say, "Don't throw out the prototype. Let's use it as the foundation for the *real* product." This decision can often lead to a situation in which the product never ships. The programmers are condemned to a role of perpetually resuscitating the program from life-threatening failures as it grows. Like the stack in which the first 25 bricks were placed haphazardly, no matter how precisely the bricks above them are placed, no matter how diligently the mason works, no matter how sticky and smooth the mortar, the force of gravity inevitably pulls it down somewhere around the 50th level of bricks.

The value of a prototype is in the education it gives you, not in the code itself. Developer sage Frederick Brooks says, "Plan to throw one away." You will anyway, so you might as well do it under controlled circumstances.

In 1988, I sold a program called Ruby to Bill Gates. Ruby was a visual programming language that, when combined with Bill's QuickBasic product, became Visual Basic. What Gates saw was just a prototype, but it demonstrated some significant advances both in design and technology. (When he first saw it, he asked, "How did you *do* that?") The Microsoft executive in charge of then-under-construction Windows 3.0, Russ Werner, was also assigned to Ruby. The subsequent deal we struck included having me write the actual program to completion. The first thing I did was to throw Ruby-the-prototype away and start over from scratch with nothing but the wisdom and experience. When Russ found out, he was astonished, angry, and indignant. He had never heard of such an outrageous thing, and was convinced that discarding the prototype would delay the product's release. It was a fait accompli, though, and despite Russ's fears we delivered the completed program on schedule. After Basic was grafted on, VB was one of Microsoft's more successful initial releases. In contrast, Windows 3.0 shipped more than a year late, and ever since it has been notoriously handicapped by its profuse quantities of vestigial prototype code.

In general, nontechnical managers erroneously value completed code—regardless of its robustness—much higher than design, or even the advice of those who wrote the code. A colleague, Clay Collier, who creates software for in-car navigation systems, told me this story about one system that he worked on for a large Japanese automotive electronics company. Clay developed—at his client's behest—a prototype of a consumer navigation system. As a good prototype should, it proved that the system would work, but beyond that the program barely functioned. One day the president of the Japanese electronics company came to the United States and wanted to see the program demonstrated. Clay's colleague—we'll call him Ralph—knew that he could not deny the Japanese president; he would have to put on a demo. So Ralph picked the president up at LAX in a car specially equipped with the prototype navigation system. Ralph knew that the prototype would give them directions to their offices in Los Angeles, but

nothing else had been tested. To Ralph's chagrin, the president asked instead to go to a specific restaurant for lunch. Ralph was unfamiliar with the restaurant and wasn't at all confident that the prototype could get them there. He crossed his fingers and entered the restaurant's name, and to his surprise, the computer began to issue driving instructions: "Turn right on Lincoln," "Move into the left lane," and so on. Ralph dutifully followed as the president ruminated silently, but Ralph began to grow more uneasy as the instructions took them into increasingly unsavory parts of town. Ralph's anxiety peaked when he stopped the car on the computer's command and the passenger door was yanked open. To Ralph's eternal relief, the valet at the desired restaurant had opened it. A smile broke across the president's face.

However, the very success of this prototype demonstration backfired on Ralph. The president was so impressed by the system's functioning that he commanded that Ralph turn it into a product. Ralph protested that it was just a feasibility proof and not robust enough to use as the foundation for millions of consumer products. The president wouldn't hear of it. He had seen it work. Ralph did as he was told, and eight long years later his company finally shipped the first working version of the product. It was slow and buggy, and it fell short of newer, younger competitors. The *New York Times* called it "clearly inferior."

The expertise and knowledge that Ralph and his team gained by building the prototype *incorrectly* was far more valuable than the code itself. The president misunderstood that and, by putting greater value on the code, made the entire company suffer for it.

⌘

If you define the boundaries of a development project only in terms of deadlines and feature lists, the product might be delivered on time, but it won't be desired. If, instead, you define the project in terms of quality and user satisfaction, you will get a product that users want, and it won't take any longer. There's an old Silicon Valley joke that asks, "How do you make a small fortune in software?" The answer, of course, is, "Start with a large fortune!" The hidden costs of even well-managed software-development projects are large enough to give Donald Trump pause. Yacht racing and drug habits are cheaper in the long run than writing software without the proper controls.

# Index

## *C*

## *D*

## *E*



## *F*

## *G*



## *H*



## *I*

## *J - K*

## *L*



## *M*



## *N*

## *O - P*

## *Q - R*

# *S*



# *T*

## *U*



## *V - W*



## *X - Y - Z*

## *Alan Cooper*

As a software inventor in the mid-70s, Alan got it into his head that there *must* be a better approach to software construction. This new approach would free users from annoying, difficult, and inappropriate software behavior by applying a design and engineering process that focuses on the user first, and silicon second. Using this process, engineering teams could build better products faster by doing it right the first time.

His determination paid off. In 1990 he founded Cooper, a technology product design firm. Today, Cooper's innovative approach to software design is recognized as an industry standard. Over a decade after Cooper opened its doors for business, the San Francisco firm has provided innovative, user-focused solutions for companies such as Abbott Laboratories, Align Technologies, Discover Financial Services, Dolby, Ericsson, Fujitsu, Fujitsu Softek, Hewlett Packard, Informatica, IBM, Logitech, Merck-Medco, Microsoft, Overture, SAP, SHS Healthcare, Sony, Sun Microsystems, the Toro Company, Varian, and VISA. The Cooper team offers training courses for the Goal-Directed® interaction design tools they have invented and perfected over the years, including the revolutionary technique for modeling and simulating users called *personas*, first introduced to the public in 1999 via the first edition of *The Inmates*.

In 1994, Bill Gates presented Alan with a Windows Pioneer Award for his invention of the visual programming concept behind Visual Basic, and in 1998 Alan received the prestigious Software Visionary Award from the Software Developer's Forum. Alan introduced a taxonomy for software design in 1995 with his best-selling first book, *About Face: The Essentials of User Interface Design*. Alan and co-author Robert Reimann published a significantly revised edition, *About Face: The Essentials of Interaction Design*, in 2003.

Alan's wife, Susan Cooper, is President and CEO of Cooper. They have two teenage sons, Scott and Marty, neither of whom is a nerd. In addition to software design, Alan is passionate about general aviation, urban planning, architecture, motor scooters, cooking, model trains, and disc golf, among other things. Please send him email at `inmates@cooper.com` or visit Cooper's Web site at `www.cooper.com`.