The MasterMinds series:

These concise and accessible books present cutting-edge ideas by leading thinkers in a highly readable format, each title a crystallization of a lifetime's work and thought.

Other books in the MasterMinds series include:

Finding Flow by MIHALY CSIKSZENTMIHALYI

After God: The Future of Religion by DON CUPITT

Extraordinary Minds by HOWARD GARDNER

Future contributors include:

STEWART BRAND

JOHN MADDOX

JOHN SEARLE

SHERRY TURKLE

Praise for BasicBooks' Science Masters series:

"This is good publishing. PBS, eat your heart out."
—*Kirkus Reviews*

"Aimed at busy, nonmathematical readers, this precise series evinces solid quality control and begins under highly favorable auspices."
—*A. L. A. Booklist*

"If this standard is maintained, the Science Masters series looks set to play a major role in the responsible popularization of sciences."
—*New Scientist*

MACHINE BEAUTY

ELEGANCE AND THE HEART OF TECHNOLOGY

DAVID GELERNTER

MasterMinds

BasicBooks

The Master Minds Series is a global publishing venture consisting of original books written by leading thinkers and published by a worldwide team of publishers assembled by John Brockman. The series was conceived by Anthony Cheetham of Orion Publishing and John Brockman of Brockman Inc., a New York literary agency, and developed in coordination with BasicBooks.

The Master Minds name and marks are owned and licensed to the publisher by Brockman Inc.

Published by BasicBooks,
A Subsidiary of Perseus Books, L.L.C.

Designed by Elliott Beard

Library of Congress Cataloging-in-Publication Data

Gelernter, David Hillel.
Machine beauty : elegance and the heart of technology / David Gelernter. — 1st ed.
p. cm. — (MasterMinds)
Includes index.
ISBN 0-465-04516-2
1. Human-computer interaction. 2. Computer software—Human factors. 3. Computers—Design and construction. I. Title. II. Series.
QA76.9.H85G46 1997
004'.01—dc21 97-14613
CIP

98 99 00 01 02 ❖/RRD 10 9 8 7 6 5 4 3 2

For my boys and my Jane

Beauty is crucial to software also. Most computer technologists don't like to discuss it, but the importance of beauty is a consistent (if sometimes inconspicuous) thread in the software literature. A 1996 essay collection edited by Terry Winograd, for example, brought together a series of papers focused on the proposition that design (as in "industrial design") is important to software. In the influential 1990 essay collection *The Art of Human-Computer Interface Design*, software prophet Ted Nelson claimed that "the integration of software cannot be achieved by a committee. . . . It must be controlled by dictatorial artists." In our 1990 programming language textbook, Suresh Jagannathan and I addressed "the enormous significance of aesthetics in the design of languages, indeed of software in general," and "the intimate connection between good engineering and aesthetic insight." Reflecting in 1978 on the seminal programming language Algol 60 that he helped design, Alan Perlis wrote that "this language proved to be an object of stunning beauty . . . a rounded work of art."

Beauty is more important in computing than anywhere else in technology. And where computers are concerned, the beauty paradox is especially acute.

Beauty is important in engineering terms because software is so complicated. Complexity makes programs hard to build and potentially hard to use; beauty is the ultimate defense against complexity. Beauty is our most reliable guide, also, to achieving software's ultimate goal: to *break free of the computer*, to break free *conceptually*. Software is stuff unlike any other. Cyberspace is unlike any physical space. The gravity that holds the imagination back as we cope with these strange new items is the computer itself, the old-fashioned physical machine. Software's goal is to escape this gravity field, and every key step in software history has been a step away from the computer, toward *forgetting* about the machine and its physical structure and limitations—forgetting that it can hold only so many bytes, that its memory is made

of fixed-size cells, that you refer to each cell by a numerical address. Software needn't accept those rules and limitations. But as we throw off the limits, what guides us? How do we know where to head? Beauty is the best guide we have.

What *is* software? A running program is a kind of machine—a strange kind that gets power and substance from another machine, namely, the computer itself. An executing program is a machine that has been "embodied" by a computer in roughly the sense that a hand puppet is embodied when you slip your hand in. A nonexecuting program is the limp puppet without a hand, an empty shell. Slip a computer inside and it becomes a working software machine: an electric-powered information-transforming machine—in the sense that a clothes washer is an electric-powered clothes-transforming machine.

(If a running program is an information processor, does that mean it is just like the brain? After all, the brain is an information processor, too, right? Wrong: the brain is no mere information processor, it is a *meaning creator*—and meaning creation is a trick no computer can accomplish. The brain is a lump of hardware artfully arranged so as to produce an I—to create the illusion that some entity inside you is observing the world that your senses conjure up. That rose over there merely triggered, when you saw it, a barrage of neuron firings in your brain. But you have the sensation that some entity—namely, *you*, not to put too fine a point on it—actually *saw* the rose. Computers, so far as we can tell, are capable of no such trick. You can build a sophisticated digital rose-recognition system, wave a rose in front of it, and thereby bring about lots of electrical activity; and perhaps after a while some words will appear on a screen—"rose recognition accomplished" or "damn, what a rose!"—but no one and nothing has had the sensation of having *seen* anything. And no computer scientist has any reason to believe that any computer ever *will* have such a sensation, or any other sensation. Granted there is no reason in principle why you couldn't build a machine that

shares with the brain this remarkable capacity; but there is also no reason to suppose you could do it without reproducing the brain itself.)

A running program is often referred to as a *virtual machine*—a machine that doesn't exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in technology history, and is a crucial step in the evolution of ideas about software. To come up with it, scientists and technologists had to recognize that a computer running a program isn't merely a washer doing laundry. A washer is a washer whatever clothes you put inside, but when you put a new program in a computer, it becomes a new machine.

The virtual machine idea clarifies an important problem—what exactly do programmers do? What activity are they engaged in when they make a program? Are they technical writers? (Their job is to compose what might be mistaken for technical documents.) Are they mathematicians? (The "documents" they create tend to be full of mathematical notation and might be mistaken for equations or proofs.) Neither: they are machine designers. They need talent and training of the sort that makes for structural engineers, automobile designers, or (in a general way) architects—not for writers or mathematicians. A program is a blueprint for a virtual machine—a blueprint that gets converted into the thing itself (the executing program, the "embodied" virtual machine) automatically when you hand it to a computer.

Here is a first entry in my guide to the nonexistent Museum of Beautiful Computing:

The virtual machine: *a way of understanding software that frees us to think of software design as machine design.*

Beauty is decisively important to computer technologists because, first, virtual machines are always in danger of drowning in complexity. Hardware machines are held in check by physical reality. Allow such a machine to get too

complicated and no one will be able to afford it, or it will be so heavy it will stave in the floor, or use so much power it will burn up. But software builders don't need to assemble materials or worry about power supplies, heat dissipation, weight, drag, toxicity. So they go wild; a single programmer alone at his keyboard can improvise software machines of fantastic or even incomprehensible complexity. Imagine what kind of palaces people would live in if all you needed to do were to draw a blueprint, hand it to a machine, and see the structure realized automatically at the cost of a few drips of electricity. The most complex machines in the world today are made of software, and the "average" software machine—the typical word processor or spaceship game or operating system—is enormously complicated, too. A modern TV may contain half a million bytes of software (a byte is the size of a single alphabetical character inside the computer); a typical modern car has a 30,000-line program inside.

This huge complexity is responsible for software's permanent crisis: if you build a big enough program, it is almost impossible to make it come out right. Studies show that the average commercial software project takes 50 percent longer than it was supposed to, and one project in four is abandoned. Your only hope is to keep the number of serious bugs low enough so that your program is more or less okay most of the time. The "beta test" is the industry's admission of failure—the procedure whereby a product that is known to be flawed, but is nonetheless as good as the manufacturer can make it, is handed to expert users in hopes they will find some of the remaining bugs. (Beta testing was developed originally by kings, emperors, and potentates of the ancient world, whose field engineers would taste each dish on the menu before the big man tried it himself. If the taster keeled over, the beta test concept had proved itself yet again and a new tester was hired on the spot.)

A new airport is scheduled to open in Denver in the fall of 1993; fall '93 comes and goes, months pass, and it is still

closed—because the software that is supposed to control the baggage-delivery system doesn't work. A Defense Department satellite tumbles into oblivion because of buggy software. In 1987 an announcement is made in California that two existing, correctly working programs will be merged—the driver registration and car registration systems—and some components added to allow people to use the finished product directly from kiosks. The new system is supposed to be finished by 1993. Then it is supposed to be finished by 1998. Then it is canceled, six years and forty some-odd million tax dollars after work began, because the task turns out to be in effect impossible. There are many similar stories.

To get out of the crisis, two steps are necessary: programmers need to be better trained, and software builders need to concentrate on making reusable blueprints and frameworks instead of reusable little pieces.

The first issue comes down in significant part to aesthetics. A good programmer can be a hundred times more productive than an average one, easily. The gap has little to do with technical or mathematical or engineering training, much to do with taste, good judgment, aesthetic gifts—and also, to be fair, a quality that has nothing to do with aesthetics: sheer intellectual aggressiveness. And brains don't hurt. But the fact that software's biggest hits are exactly the systems that are repeatedly praised for *elegance*—the Algol 60 language, from which so much modern practice derives; the "object-oriented" programming technique that emerged from Algol in 1967; the Apple desktop, on which the vast majority of computer users rely, in one form or other, today—ought to be a clue to the flummoxed industry that elegance has something to do with good software, that there is a connection somewhere between aesthetics and success.

But the beauty paradox is such that the industry *and* computer science researchers would far rather pursue mathematical solutions, so-called formal methods, than teach programmers about beauty. Mathematics is serious, aesthetics not; hence the

field has been banging its head against the wall since the mid-1970s in an effort to put programming on a mathematical basis, and made such astonishingly little progress you'd imagine it would have drawn certain conclusions. But when mathematical methods fail, the invariable response is, "Bring on more mathematical methods!" A little progress has been made here and there, and mathematics is fine in its place. But it *cannot* be the whole story or even the main one, or we would not be stuck where we are, in a permanent mudbank spinning our wheels. "The hell with mathematics; let's teach our programmers about beauty" is what we ought to hear. Instead we are solemnly informed (in a representative *Scientific American* piece) that "intuition is slowly yielding to analysis as programmers begin using quantitative measurements. . . . The mathematical foundations of programming are solidifying."

"Anyone who wants to analyze the properties of matter in a real problem," Feynman writes,

> might want to start by writing down the fundamental equations and then try to solve them mathematically. [Who needs mere intuition when you can have *analysis*?] Although there are people who try to use such an approach, these people are the failures in this field [on second thought . . .]; the real successes come to those who start from a *physical* point of view, people who have a rough idea where they are going and then begin by making the right kind of approximations.

Perhaps there is something to be said for intuition after all. "The German emphasis on calculations," David Billington warns apropos of bridge building, "was a double-edged sword; it forced designers to think rationally but it also drew them away from forms for which they had no calculations, and thus narrowed the range of structural possibilities."

Not only is your typical software machine hugely complicated on the inside; it offers enormous power and a wide

range of functions as well. A taste for beauty is the technologist's most important ally, also, in his struggle to produce software that people are capable of using effectively.

Beauty determines which virtual machines triumph and which are rejected, left to rust like old cars in weedy meadows. Ugly virtual machines waste the underlying computer's power and, vastly more important, the user's time, but a beautiful program hovers nearby like an attentive, unobtrusive British butler. A beautiful program's way of doing things is so close to your own that creative symbiosis develops, a thought-amplifying feedback loop. You have an idea and the machine accommodates it immediately—no backtalk, no bargaining. The machine's transparency and willingness might even nudge your thinking a step forward.

The software's role is humble and basically passive, but it can amplify your thought—an important accomplishment. One of the field's foremost visionaries, the inventor in the late 1960s of the mouse and the computer window, ran a laboratory with the strange title Augmentation Research Center. According to Douglas Engelbart, computers are tools for "augmenting human intellect."

They can play that role, however, only to the extent they are beautiful. No creative symbiosis is possible with an ugly virtual machine—with a complex or weak program that forces you to bend to its worldview instead of accommodating yours.

In the computer world, beauty is the most important thing there is, and the paradox is this: beauty inspires the best technologists and confuses or outright repels everyone else—at least to begin with.

Ordinary technologists don't understand it, and often behave as if they hate it. They don't, actually; it's just that fancy features and complex, sophisticated functions inspire them—and, as for beauty, they can't see it at all. Because they are unaware of it, they will knock it over and destroy it without batting an eyelash, utterly oblivious.

The public's relationship to beautiful software is more complicated. "Almost no one seems to be able to recognize *good* design," writes Nelson—"except users, and that only sometimes." Not sometimes, *always*—but only in the long term. Consumers invariably go for beauty in the end; right up to that point, many seem to detest it. Why? It is a hard question and no one knows for certain. I will discuss it in these pages. This much is clear: (1) most computer technologists are oblivious to beauty; (2) the best are obsessed with it; (3) the public has a love-hate relationship with beauty in computing; and (4) beauty is the most important quality that exists in the computer world, when all is said and done.

To see this paradox in action, consider the strange story of Apple computers.