THE PHILOSOPHICAL PROGRAMMER

REFLECTIONS ON THE MOTH IN THE MACHINE

DANIEL KOHANSKI

Contents

Title Page

Copyright Notice

Epigraphs

Dedication

Preface

PART I: A PHILOSOPHICAL INTRODUCTION

1. Beyond the Cuckoo Clock

2. Is There an Aesthetic of Programming?

3. The Ethical Quotient

PART II: THE STRUCTURE OF THE COMPUTER

4. Types of Computers

5. The Parts of a Computer

6. The Construction of Memory

7. “On a Clear Disk You Can Seek Forever”

8. A Brief Interruption

9. Operating Systems

PART III: FUNDAMENTAL TOOLS OF PROGRAMMING

10. The Language of the Machine

11. Forms of Data Definition

12. Classes and Types of Statements

13. The Functional Program
14. A Short Commentary
15. Algorithms and Objects

PART IV: THE PROGRAMMER'S TRADE

16. The Moth in the Machine
17. The Real World Out There
18. The Limitations of Design
19. Programming as Abstraction and Reflection

Appendix: Number Representations
Acknowledgments
Glossary
Selected Bibliography
About the Author
Copyright

Insofar as the laws of mathematics refer to reality, they are not certain. And insofar as they are certain, they do not refer to reality.

Albert Einstein

In the design of programming languages one can let oneself be guided primarily by considering “what the machine can do.” Considering, however, that the programming language is the bridge between the user and the machine—that it can, in fact, be regarded as his tool—it seems just as important to take into consideration “what Man can think.”

Edsger W. Dijkstra

For Jean

And for Pat, who should have lived to see this.

Preface

In less than three generations, the computer has gone from being a scientific curiosity and military secret to becoming the defining technology of the age. In the last fifteen years alone, its status has changed from the prized possession of a university or corporation to that of a common household appliance—and one that is sometimes replaced every year or two as new developments make the current one obsolete. In the last five years, books have flooded the marketplace describing in detail what some computer product does, or how to write programs in some particular language on some specific machine. Yet there has been very little discussion about what programming *is* and about the meaning of programs for our lives.

This book tries to look at that question, and to look at it from a philosophical perspective. The starting point for any philosophical inquiry, particularly in the modern age, is from the stand-point of humanity: What is the impact of programming on the human beings who write programs, who use programs, and who in some way are affected by what programmers do? Another approach is to look for common structures in our activities; applied to programming, such an inquiry examines what a program is and how it is put together, and what principles of construction we can discover through this examination.

To do this, I have looked at programming and

computers not only as they are now but as they used to be. We build upon the past, and the computer's past—short as it is—explains much about the way we have come to work with it today. The conventions of programming are by no means intuitive, and they are as much the result of accidents of history as they are the product of deliberate design. Understanding how the conventions developed will help us to remember them when intuition fails.

So, although this is not a "how-to" book and will not teach anyone how to write a program, it will look into what the writing of programs is all about. As such, I hope it will be useful to beginning programmers who need a solid foundation on which to start construction, as well as to my more experienced colleagues in the field, who will benefit from a review of skills that can sometimes become too much a matter of rote.

Most of this book is also for the general reader, someone who wants an idea of what computers are and what programmers do, and why we do it this way. Some chapters are necessarily more technical than others, and I have tried to organize them so that the beginning parts of each will give a general idea of what is going on. A nontechnical reader can skim through the rest of the chapter with no real loss of the sense of the whole.

The first part of the book, then, is a philosophical introduction to computers and programming; that is, it explores what a computer is and what its impact is, and discusses some of the problems that it poses for those who must master its intricacies. The second part describes how the computer is seen by the programmer, while the third goes into some detail about the tools that programmers use. The final part, again a philosophical inquiry, looks at the problems and opportunities the computer offers to

humanity as a whole.

Part I

A PHILOSOPHICAL INTRODUCTION

1

BEYOND THE CUCKOO CLOCK

The degree to which we make and use tools is one of the qualities that distinguishes us from the rest of the animal kingdom. Even so simple a thing as a pointed rock gave our hominid ancestors new powers: They could use it to cut up meat, and could even kill game that had previously been too strong, too fast, or simply too big for their unaided bodies to handle. A dead limb might become a lever to pry up a boulder that their muscles by themselves could never budge. The essence of a tool is that it extends our reach, multiplies the power of our limbs, improves our eyesight, and in countless other ways increases our ability to manipulate and control the world around us. The development of new tools is one of the ways in which we measure the progress of our species from hominid to human.

Until recently (that is, recently even by historical standards), tools were used primarily to extend the physical powers of our bodies. We use the hammer to increase the power of our fist; we use the plow to help break up soil too hard for our fingers; we use sacks to carry more seed than our hands can hold. All of these require our physical presence and our physical involvement, and for all of them, no matter how much they magnify the power of our muscles, they are limited to providing an incremental increase of the enabling muscle power.

Now consider an example of a different kind of tool: the medieval clock. It was driven not by the constant action of our muscles but by the movement of weights and counterweights; once set in motion, it could function for hours or days with no further human intervention. Whereas other tools altered our environment, the clock altered our perception of the environment instead. Much of this was done deliberately; monks in their medieval monasteries had committed themselves to saying their prayers at precisely defined times of the day, and to miss even once was to put their immortal souls in peril. Rather than guess at the hour of Prime (6 A.M.), they developed ever more sophisticated instruments to announce the time for them. Thus they left the natural time of circadian rhythm and daily course of the sun, choosing instead to be guided by instruments which they had devised but over which they had somewhat less than complete control—for of course they could not arbitrarily change the time once it had been established. In a small way, this tool exerted control over the toolmaker.

Because tools such as the clock expand the power of our minds rather than our muscles, it becomes part of their function to give us guidance and even control. The human mind cannot measure time with the accuracy of even a cuckoo clock, so we have learned to rely on timepieces to determine the proper time and will even cite them as authority for our actions. When the cuckoo comes out to sound the hour, we assume that it is correct. That is to say, we depend on the clockmaker's skill to provide us with an accurate device. And as with other tools of this sort, as their complexity increases it becomes harder and harder for the ordinary, unskilled user to fix them when they go wrong, or sometimes even to be aware that they have gone wrong at all.

The modern computer is even more a tool of the mind than is the cuckoo clock. Although its appendages can manipulate physical objects with great precision and dexterity, the computer itself is essentially a tool that extends the power of our thoughts. But it goes far beyond the cuckoo clock and any other mind tool we have invented thus far because, unlike any of them, it is a general-purpose tool that can be transformed almost at whim into nearly anything that our minds can conceive.

Yet there are still some ways in which the cuckoo clock resembles the computer. It processes raw data—the movement of weights or a spring or a pendulum—into useful information: the time. Once started and set, it operates for the most part without human intervention. It makes decisions, such as when to send the cuckoo out to sound the hour, on its own, again without human intervention once the clockmaker has designed and built its gears. The cuckoo clock and its contemporaries are in fact early examples of what today would be called an analog computer.

Analogs are representations of objects in the world. An analog computer processes these representations as continuous streams of information—a flow rather than a set of discrete intervals. It is the difference between a sweep second hand and one that jumps from second to second; the sweep movement is an exact analog of all the infinite instants of time. What makes the cuckoo clock a kind of analog computer is its processing of the continuous swings of its pendulum or the smooth rise and fall of its weight and counterweight.

Analog computers are not widely used these days. While they can handle infinitely varied input, they are limited in their ability to preserve and manipulate this data.

A telephone circuit can carry the nuances and tones of a human voice in all its infinite varieties, but as the voice travels from switch to switch on its way to its destination, it loses more and more of its quality until, sent far enough, it can become unrecognizable gibberish. The power of the modern computer comes in large part from its processing of data as a set of discrete units—as digital information. Digital technology analyzes the continuous voice curve and assigns a number to various points on the curve. These numbers are then transmitted to the receiving end, which reconverts them into sounds that, while not a perfect match for the original, provide a close approximation. Digital technology trades perfect representation of the original data for a perfect preservation of the representation. To return to our clock example, an analog clock may use a sweep second hand which might be exactly accurate, but which cannot be exactly described. On the other hand, one can look at the face of a modern digital timepiece and describe exactly what time it reads, though such a clock is accurate, at best, only to the nearest second.

Digital technology provides far more flexibility in toolmaking than the analog world can. The cuckoo clock can decide when to send out the cuckoo, but that is about it. The microcomputer in a modern answering machine, however, can tell us what time a call came in, how long the message ran, and perhaps even where it came from. In theory, we could build an analog device to do these things as well. But it would not be able to do anything else unless we disassembled it and rebuilt it for its new task. The digital computer, however, uses for its instructions a set of discrete numbers rather than a series of gears or other analog devices. Thus the advantage of the digital computer is that to change what it does, we need only change these

instruction numbers—its programming—and start it up again. If we want to add new features to the answering machine, in many cases we can do so simply by reprogramming it.

It is this idea of programming that gives the modern computer its power. Merely by changing a set of written instructions we create new tools out of the same physical device: we can turn the computer into a clock, an adding machine, a typewriter, or even a chess or Scrabble player. For the first time, words themselves have become tools which *in and of themselves* cause things to happen. Manual dexterity need no longer be a requirement for a physical creation; instead, it is the ingenuity of the programmer's thinking that is crucial to bringing an idea to fruition. In this new era, the primary creative force is becoming less and less the hand and more and more the word.

Words—instructions to a computer—have become tools largely because of another major component of the modern computer: its memory. Early computing devices were driven by numbers punched as holes in cards or paper tape; even today, a Jacquard loom will weave a cloth pattern as directed by a punched card, and player pianos will play music according to the notches on a drum. The Harvard Mark I, which started operation in 1944, followed instructions punched on holes in paper tape to produce gunnery tables for the U.S. Navy. All such devices, however, had a common limitation: they could not change the instructions or the data on their own. If any change was needed—a new formula for a table, for example—the entire tape had to be punched all over again.

By the time the Mark I was running, engineers had already recognized this restriction and were working to overcome it by developing methods of storing numbers in a

modifiable medium—that is, memory. Early forms of memory included mercury delay lines, vacuum tubes, and magnetic drums. The ENIAC, another prototype computer built during World War II, used 18,000 vacuum tubes—a phenomenal number for the time. By 1954, core memories —small doughnut-shaped magnets on a wire grid—were in use in commercial computers, and the transistor was replacing the vacuum tube. It was now possible to store large groups of numbers that could be easily manipulated at electronic speed.

The EDVAC project, successor to the ENIAC, inspired another breakthrough, one that now defines the modern computer: *the idea of a program as data.* The ENIAC was programmed, as some of its predecessors had been, by setting wire plugs in a board. This method not only made reprogramming a time-consuming chore, but it also made programs slow to execute. The mathematician John von Neumann, assisting the ENIAC and EDVAC projects in 1944–45, documented a new approach that has since been likened to the invention of the wheel: storing the instructions in the computer's memory along with the data.[1]

Storing the program in memory meant that the new computer could access each instruction at the same electronic speed with which it accessed the data. But it also meant that the computer could treat instructions as though they were themselves data—that is, numbers to be manipulated. This ability is the basis of all modern programming. A programmer writes a program—a set of instructions—using words that more or less resemble English (or other natural language) and enters these words into the computer as data. A program called a *compiler* then examines these words and converts them into numeric instructions, which the computer circuitry can execute. It

then puts these new numbers into memory, and the computer executes them. A program can even alter its own instructions in response to changing circumstances, which is a basic prerequisite of artificial intelligence. Just as we change and refine our actions as we learn more about our environment, so the computer is capable of altering its programming as it acquires more data.

Each instruction that the computer executes is stored in a particular location in its memory, and the computer normally proceeds from one instruction to the next. However, there are some instructions which will alter the flow of execution based on some condition—for instance, if the value of a piece of data exceeds a defined threshold, the computer will be directed to execute a different set of instructions than it would have otherwise. These types of instructions are called *conditional branch instructions,* and they give the computer the ability to make decisions. In a rocket guidance system, for example, the computer is constantly receiving new data—the current position, speed, and angle—and making decisions based on that data to increase or decrease the fuel flow to the various thrusters that keep the rocket on target.

Although we often speak of the computer as making decisions, in reality it is the programmer who decides, or rather, it is the programmer who determines the conditions under which the computer will execute one or another set of instructions. The computer does nothing on its own, but only what some programmer has told it to do. If the programmer gives incorrect instructions, the computer will blithely follow them. In this respect the computer is no more than an infinitely more complex cuckoo clock, which is only as accurate as the skill of the clockmaker. But where a faulty cuckoo clock might result only in a missed

appointment, the consequences of a failure in a computer program can be drastic indeed.

The cuckoo clock affected the lives of those who depended on it although it provided only a single piece of information (the correct time), and even the earliest computers quickly became indispensable for solving all kinds of mathematical problems. But the ultimate difference between the computer and the cuckoo clock lies in the computer's generalized ability to process almost any kind of information, not just the value of a formula or the time of day. Stock transactions and bank transfers can be instantly ordered and credited. Telephones can be made to hold calls, to transfer or reject calls, to remember who called while the line was busy, all by prior instruction to a computer. Photographs and movies can be enhanced, colorized, and even completely altered through computer graphics. Pacemakers can adjust a heart's rhythm instantly in response to the changing state of the heart muscles. Our era has barely begun to explore the possibilities of the computer, and yet it is already known as the Computer Age.

But each new use of the computer's abilities means a new demand on the programmer's talents. The early days of simple computation of a mathematical formula have long since given way to complex manipulations of vast quantities of data, with each innovation giving rise to demands for more—and all of it is controlled by some programmer's instructions to a machine that is both blindingly fast and witheringly intolerant of error. This almost inhuman requirement for accuracy in computer programming has both philosophical and practical ramifications. The next chapters and Part IV explore some of these issues, while Parts II and III explain some of the means by which we

ease the burden of dealing with them.

2

IS THERE AN AESTHETIC OF PROGRAMMING?

A program is a set of detailed instructions given to a computer to perform a specific task. The computer will take no action without such instructions. In this respect, the computer is no different from a shovel. Just as a shovel will do nothing on its own, and performs its function only when someone picks it up and shoves it into the dirt, so a computer does nothing unless we give it orders. It matters not whether the motivating force is muscle power or typed commands; what is important is the motivation, which in all cases comes from the human mind.

It is therefore appropriate to speak of computer programming, as we speak of all other human activities, as having an aesthetic aspect. While aesthetics might be dismissed as merely expressing a concern for appearances, its encouragement of elegance does have practical advantages. Even so prosaic an activity as digging a ditch is improved by attention to aesthetics; a ditch dug in a straight line is both more appealing and more useful than one that zigzags at random, although both will deliver the water from one place to the other. Getting a computer program to deliver its intended result is a far more complex task than digging a ditch from a well, and attention to the aesthetic aspects ought to be an essential part of the process.

To begin with, there are aesthetic concerns inherent in the design of a program. Every program is the expression in computer language of a series of actions that the computer needs to take in order to solve a problem. Each such action or set of actions, when described in theoretical terms, is called an *algorithm.* While there may be a seemingly endless number of algorithms that can be used to solve a problem, some are more efficient, more elegant —more aesthetically correct—than others.

Consider, for example, the designing of a house. Did the architect put the bathroom near the bedrooms, or at the other end of the hall? And are there enough bathrooms for the people who will be living there? Is the kitchen near enough to the bathrooms to use the same water supply, or will extra money need to be spent to lay more pipes? Are there useless corners and dead ends? How much sunlight will each room have? There is not necessarily one right answer to any of these questions—it may be that putting the bathroom near the bedrooms justifies the cost of extra pipe—but it is clear that there are better designs and there are worse designs. The better designs are more aesthetically pleasing because they are more efficient and take more considerations into account.

The same is true with programming design. Although we use algorithms instead of blueprints, the same types of questions need to be raised: Is this piece of the structure necessary? Will it get in the way of other parts? Is it too far from—or too close to—other steps in the process? How much weight should be given to each part in order to achieve an overall balance? Is there another, more efficient way to reach the same solution? And, of course, what can be done to ensure that this is a correct solution?

Suppose I want to look up a name in a telephone

directory. I could start at the beginning of the book, and examine every name to see if it matches the one I am looking for. If I find it, then I can stop; otherwise I get to the end of the book and know that the name is not in there.

While it will work, such an algorithm—such a means of solving the problem—is hardly elegant or efficient. A much better approach to finding the name is to open the book near to where the first letter of the name is likely to be, then to go backwards or forwards in the book according to whether the name I want is before or after the name in the book that I am pointing to right now. I repeat this process until I either find the name, or else find two names next to each other, such that the name that I want would have been between them; in this case, the name is not in the book. This method yields its result—found or not found—much faster than the first one. It is much less taxing, and there is an aesthetic appeal to it with which the brute force method of the first technique cannot compete.

It is also readily understandable to a human being. Aesthetics, let us admit, mean nothing to a computer. And elegance in programming is by no means a guarantor of efficiency. It must be constantly borne in mind, however, that programs are not written solely to be understood by computers, but by people as well. While the computer may well be able to execute a poorly designed program and produce the correct answer, it is often difficult to determine from an examination of the program that it *is* the correct answer. Badly designed programs are notoriously error-prone, are likely to be slower, and are often referred to by derisive nicknames such as “spaghetti code”—for the logic paths resemble nothing so much as a bowl of spaghetti and are even harder to untangle. The very programmer who wrote such a piece of code might find it hard to trace its

logic a week or a month later on. Moreover, even if the program does the proper job today, tomorrow may bring a new set of requirements. Programs constantly evolve, or to be more precise, the uses of a program evolve, and it is the programmer who performs the evolution. The requirements of modern programming assignments are such that they place an almost inhuman—and certainly inhumane—burden of perfection on fallible human beings, who will find an all-but-impossible job that much harder if the program's design is not perceptible to them. Attention to aesthetics improves human perception.

Another area of programming where aesthetics is involved is in the structure of the data. Take the previous example of the telephone book. The more efficient algorithm only worked because the data was organized alphabetically. A phone book in random order would indeed require the first, brute force algorithm of examining every entry, and a company that produced such a book would probably not last long. Most computer data structures are more complex than a phone book, and attention to the aesthetic aspects of their design will result both in more efficient processing of the data and faster understanding by the human programmers who must work with them.

The requirement for elegance involves not just the design of the algorithm and the internal data layout but the design of the input and output as well. The organization of information being entered, and the way the answers appear on the screen or the printed page, make a great deal of difference in their ability to be understood by the human beings who use them, although once again, it means nothing to the computer. Lives have been lost because a computer operator could not make sense of the data displayed.[1]

This aspect of aesthetics often goes by the name *ergonomics.* The science of ergonomics studies how people relate to the machines they use and tries to make such use more comfortable and more efficient. The design of the office chair is one obvious application of ergonomics. But it also applies to the computer programs that people use. A program, as I shall discuss later in detail, generally interacts with human beings at various points, particularly in accepting raw data as input and producing its results as finished output.

At every airline ticket counter there is a computer terminal. The airline agent, using a keyboard, fills in various fields on the screen—name of the passenger, destination, checked luggage, number of screaming children, and so on. The computer then checks its files to validate the reservation, makes a seat assignment if one was not already made, and prints out the boarding pass and baggage stickers. In processing all this, the computer—that is to say, the computer program, which is ultimately to say the computer programmer—must accept and display this information in such a way that the ticket agent can easily digest it and pass it on. Boarding passes, to take but one small part of the example, have (or should have) the seat number printed in large type in a blank area, which is easier for the passenger to see.

In the early days of computing, aesthetics was a luxury programmers felt they could ill afford. Space and time were at a premium, computers were slow, and any trick that programmers could play with the design, any extra space they could squeeze out of the data, was a savings well earned. While computer costs have dropped dramatically, this short-cut mentality still endures. Moreover, computer programmers are just as susceptible as everyone else to

failing to take the long view; all too often an algorithm, a technique, or a data layout that was meant to be a temporary quick-and-dirty fix becomes etched in stone—or at least in silicon.

A dramatic example of this insufficient attention to consequences is the potential for disaster presented by the year 2000 problem. Early computers, and the punched cards they used for input, were very short on space and often used a two-digit field to represent the year. Everyone understood that “50” meant “1950.” But no one stopped to think that a computer program that interpreted “50” as “1950” would also interpret “00” as “1900” instead of 2000—or if some did think about it, they thought that a program written in 1950 would be long gone by the turn of the century.

Many of these early programs and the computers they ran on are indeed relics of the past. But the algorithms and the data layouts that were developed for them still haunt us today; each succeeding incarnation of these programs was written to carry on the program of before, and the habits of an earlier generation propagated their way through new generations of programmers. Now we are faced with a horrendous effort to track down, upgrade, test, and install every single program—and there are millions—that relies on knowing what year it really is. The IRS, the Social Security Administration, even so simple an object as an elevator which is programmed to shut down if it has not had maintenance in the past six months, all are vulnerable because of our carelessness in restricting our representation of a year to a single century.

A greater concern for the long-term consequences of our casual programming decisions would have gone a long way toward minimizing the problems the year 2000 is

causing us. While such a concern is not normally classified as an aesthetic principle, one of the qualities of elegance is its longevity—an acknowledgment that our constructions may well live longer than we first thought and should therefore be designed for the long term. There is also an aesthetic factor in recognizing that one's design may, even must, evolve to meet unanticipated challenges. Growth—for programs as well as for living things—can be orderly, well designed, elegant—or it can be ungainly and grotesque, a cancerous mass. The inexorability of time should have taught us in no uncertain terms that such growth *will* occur; our only choice is whether to guide it or to be consumed by it.

A working list of the aesthetic principles of programming might read as follows:

1) *The program has an elegant design.* Each step in the design follows logically from the previous one, and the flow from step to step is in the same direction. Each step performs one task.

2) *The program evolves.* A program that is written for one purpose will often be used to handle related but different situations, and the design must allow for expansion. The program must be flexible enough to grow, and to grow neatly.

3) *The program will last longer than you think.* It is often tempting to write an ungainly, sloppy "quick fix" to solve an urgent problem. But it often happens that the same program will be needed for other similar problems, or it is discovered that it can be used as the foundation for a larger program. Once this occurs, changing the original design of the program is effectively impossible. Better to start off right in the first place.[2]

4) *The program has a limited life span.* No matter how much thought is given to design, expansion, and new

requirements, eventually a program will become so ungainly and so overburdened with tasks that it becomes error-prone and difficult to maintain. A programmer must be prepared to recognize this when it occurs and to write a new program rather than pile more complexities onto an old one.

5) *The data is well laid out.* The data is organized so that it can be quickly and accurately accessed for the purposes of the program, and so that it is readily understood by anyone who needs to write code to manipulate it. The data structure also allows for expansion.

6) *The program and data structures are explained.* Each algorithm, each step in the program, has comments and other documentation that describe what it does in clear and concise prose. Each part of the data layout includes a description of what it is used for. Any shortcuts or other tricks are thoroughly explained and well marked. This should be the most obvious principle of all. All too often it is not, and failure to observe it costs the industry untold hours of lost productivity.

All of the ideas I have discussed here are well known to many programmers; the current crop grew up with structured programming, well-defined data layouts, and in recent years have become familiar with object-oriented principles. True, it is rare for us to call them by the term “aesthetics.” Yet I group them together under this heading because, as I said in the beginning, aesthetics is a profoundly human concept that speaks to peculiarly human needs. A programmer must always keep in mind that other human beings will use this code, will have to maintain, evolve, and ultimately replace this code, and will be profoundly affected in all aspects of life by what this code does or fails to do. Calling it aesthetics reminds us of the human dimension.

3

THE ETHICAL QUOTIENT

Ethics may be understood to mean a set of philosophical principles that govern our conduct, with the primary focus being on human interaction. Any object constructed or used by human beings in a way that affects other human beings therefore carries with it elements of ethical concern. While this applies to any of the tools we have created and used over the years, there are characteristics unique to computers and computer programming that present us with ethical challenges unlike any we have ever faced before. In brief, these include the qualities of magnification, precision, alienation, believability, malleability of information, impermeability, and autonomy of operation.

By *magnification* I mean the power of any tool to magnify our personal abilities. But unlike earlier tools, or even what we are accustomed these days to thinking of as tools, the computer magnifies our thoughts more than our muscles, and does so to a greater degree than any previous tool ever could. Whereas a bookkeeper in a firm keeps track of as many customer accounts as can be humanly read and updated in a day, a programmer can direct a computer to monitor thousands or millions of accounts in that same period of time. A sailor might watch two, three, or four planes on a radar screen in an effort to determine if they are hostile or peaceful; a computer processing that same data might be able to track dozens of

planes at once. The local cop on the beat may know the faces and habits of a few dozen local criminals; the computer back at the precinct house is collecting data about thousands of them all over the city. Government clerks plod slowly and sometimes inaccurately through income tax returns, while a computer can check a hundred taxpayers' calculations in a fraction of a second.

But magnification of thought in this way means that mistaken and malicious thoughts are also magnified. The computer possesses no moral compass, no more than any other tool; it cannot distinguish good from evil or truth from error unless we carefully instruct it to do so. An unscrupulous programmer could write a bookkeeping or banking program to transfer money from unsuspecting customers into a private account. Or an outside thief might discover an unanticipated contingency in the programming and use it as a license to steal. A program to help decide whether an approaching aircraft is hostile might have inadvertently left out a display of altitude changes—a crucial element in determining a plane's intentions. A database on criminals might not allow an operator to delete entries where the accused was found innocent, or might confuse two people who have similar names, or might not bother to distinguish between an arrest warrant for murder and a twenty-year-old parking violation. Outdated IRS computers, while still faster than a human clerk, are not fast enough to keep up with the flood of tax returns, nor are they always updated accurately to include changes in the tax code. The computer magnifies all our thoughts uncritically, both for good and for ill. It is we human beings who must choose which thoughts the computer follows and how it follows them.

The difficulty we face in doing so is partly explained by

the next ethical conundrum the computer poses: the need for *precision.* Instructions to a computer must be precise in every detail, with no ambiguity, or the program will go off in some unintended direction or even fail completely just when it is needed most. Every contingency must be planned for in advance; the computer has no ability to make new decisions on its own, but can decide only in accordance with instructions we have already given it. There have been considerable advances in programming techniques designed to provide just this sort of precision and to make allowances for each possible situation. Yet completeness remains an unreachable goal as the increasingly complex demands we make on computers and their programs generate an exponential increase in the details associated with them. Any program of any size and consequence requires constant monitoring and maintenance.

In addition, the tendency of the computer to isolate programmers and operators from the people impacted by their actions fosters a new degree of *alienation,* which increases the problems created by its magnification of our imprecise thoughts. We are accustomed to making ethical decisions about how we use our tools according to the immediate and visible consequences of those decisions. But when we are separated from the people affected by our actions, this distance—this alienation—often provides us with an excuse to overlook the connection. This temptation long predates the computer, of course. A building contractor using inferior materials in the expectation of being long gone from the scene when the walls collapse is but one infamous example. The facility with which the computer alienates perpetrator from victim, however, adds a new dimension to an ancient concern. We no longer see a real person, only numbers and names on a computer

screen. People who would never have the temerity to rob passers-by on the street have no compunctions about using a computer to steal their credit reputations instead—and do far more damage in the process. Negligence in keeping a database up-to-date or carelessness in data entry can have disastrous consequences for the person whose record has been mishandled. It has happened that a person is arrested again and again on the basis of the same mistaken information because the original error in the database was never corrected or the dismissal of the case was never entered. Such a cavalier attitude is credible because the database operator never deals with anything but streams of data that all seem the same, so the operator does not connect them with the real people they represented.

The phenomenon of alienation works on the programmer even more than on operators or users. Instead of seeing the program as a real instrument affecting the lives of real people, programmers sometimes see it as a game, a challenge to their ingenuity. The alienating quality of the computer permits us to overlook the human consequences of a programming decision or error. An assignment to link scattered databases in different computers, for example, becomes nothing more than a problem to be solved; the programmer does not notice the resulting diminution of privacy and the increased opportunities for mischief that can result.

That we can allow ourselves to be so blind to the consequences of the collection of personal information is a result of how the computer approaches this data. To the computer and its programs, data about a person is no different from data about the physical world. Both are quantities to be manipulated according to mathematical formulas. The alienating quality of the computer is such that

it can reduce a living person to nothing more than numbers in a machine.

Alienation in the form of mediation—which is to say, the computer as mediator or intermediary—works on the user as well. People who would be unfailingly polite to strangers on the street do not hesitate to hurl insults at them on the Internet. The computer separates people from the person at the other end; they are not insulting a fellow human being, only a message on the screen. Again, this is a phenomenon with ancient roots, but one that is magnified by the ability of the computer to isolate us from those who are affected by our actions. The anonymity that the computer offers is not always as complete as it allows us to pretend—the computer may well keep records of our actions that we never know about until a much later confrontation—but the semblance, at least, of anonymous escape presents a great temptation for both deliberate and careless harm. It is ironic that the same Internet that allows us to interact with a wide variety of people that we might never otherwise meet also allows (and in some sense even encourages) behavior that alienates all who are not exactly of like mind.

Alienation and its concomitant anonymity figure in yet another aspect of the computer that raises ethical concerns, and that is its *believability.* The development of the computer and modern society have each been encouraged by each other in a kind of symbiosis; each growth spurt by one part stimulates a matching leap by the other part. As a result, our society and its institutions have become so large and so complex that without computers they would instantly dissolve. Imagine for a moment trying to run the Social Security system, or a modern bank, or even a warehouse, without computers. But this same

dependence has resulted in a necessity to believe what the computer tells us: whether the Social Security recipient is alive or dead, or how much money is in a customer's account, or whether we have sufficient stock on hand to fill an order. Because the computer has allowed us to collect and maintain far more information than we could before, it has also made us almost totally dependent on it to produce valid information. The computer simply provides us with too much information too fast for us to do anything more than assume it is correct.

This assumption of correctness has been further strengthened by our acute awareness of the computer's arithmetic abilities. We can easily verify the results when a computer calculates 2 plus 2. But when we ask it to multiply 9,785.63 by 10,348.27, we are not inclined to question it when it reports the answer as 101,264,341.3601. We expect the computer to compute, in the original meaning of the word: to perform arithmetic calculations, and to do them flawlessly.[1]

But now we have transferred this aura of infallibility from calculation to information, although there is no guarantee of its correctness beyond that of the skill of the operator who entered the data and the programmer who wrote the instructions to manipulate it. If the National Crime Information Center computer reports an outstanding warrant, the police are far more likely to believe it than the protests of the hapless victim who claims they are arresting an innocent man. Unexpunged parking violation tickets, theft and assumption of another's identity, carelessness in cleaning up case backlogs—all have resulted in citizens who were going about their lawful business being detained and forced to prove their innocence against the word of the computer.[2] We have gone beyond dependence on the