SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 6: *Software as Weapon*
Section *A New Form of Domination*

This section shows that the software elites are promoting mechanistic concepts in order to prevent independence and expertise in software-related activities.

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

Designed and typeset by the author with text management software developed by the author and with Adobe FrameMaker 6.0. Printed and bound in the United States of America.

Acknowledgements

Excerpts from the works of Karl Popper: reprinted by permission of the University of Klagenfurt/Karl Popper Library.

Excerpts from *The Origins of Totalitarian Democracy* by J. L. Talmon: published by Secker & Warburg, reprinted by permission of The Random House Group Ltd.

Excerpts from *Nineteen Eighty-Four* by George Orwell: Copyright ©1949 George Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1949 Harcourt, Inc. and renewed 1977 by Sonia Brownell Orwell, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *The Collected Essays, Journalism and Letters of George Orwell*: Copyright ©1968 Sonia Brownell Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1968 Sonia Brownell Orwell and renewed 1996 by Mark Hamilton, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *Doublespeak* by William Lutz: Copyright ©1989 William Lutz, reprinted by permission of the author in care of the Jean V. Naggar Literary Agency.

Excerpts from *Four Essays on Liberty* by Isaiah Berlin: Copyright ©1969 Isaiah Berlin, reprinted by permission of Curtis Brown Group Ltd., London, on behalf of the Estate of Isaiah Berlin.

Library and Archives Canada Cataloguing in Publication

Sorin, Andrei
Software and mind : the mechanistic myth and its consequences / Andrei Sorin.

Includes index.
ISBN 978-0-9869389-0-0

1. Computers and civilization. 2. Computer software – Social aspects.
3. Computer software – Philosophy. I. Title.

QA76.9.C66S67 2013 303.48'34 C2012-906666-4

Don't you see that the whole aim of Newspeak is to narrow the range of thought?… Has it ever occurred to you … that by the year 2050, at the very latest, not a single human being will be alive who could understand such a conversation as we are having now?

George Orwell, *Nineteen Eighty-Four*

Disclaimer

This book attacks the mechanistic myth, not persons. Myths, however, manifest themselves through the acts of persons, so it is impossible to discuss the mechanistic myth without also referring to the persons affected by it. Thus, all references to individuals, groups of individuals, corporations, institutions, or other organizations are intended solely as examples of mechanistic beliefs, ideas, claims, or practices. To repeat, they do not constitute an attack on those individuals or organizations, but on the mechanistic myth.

Except where supported with citations, the discussions in this book reflect the author's personal views, and the author does not claim or suggest that anyone else holds these views.

The arguments advanced in this book are founded, ultimately, on the principles of demarcation between science and pseudoscience developed by philosopher Karl Popper (as explained in "Popper's Principles of Demarcation" in chapter 3). In particular, the author maintains that theories which attempt to explain non-mechanistic phenomena mechanistically are pseudoscientific. Consequently, terms like "ignorance," "incompetence," "dishonesty," "fraud," "corruption," "charlatanism," and "irresponsibility," in reference to individuals, groups of individuals, corporations, institutions, or other organizations, are used in a precise, technical sense; namely, to indicate beliefs, ideas, claims, or practices that are mechanistic though applied to non-mechanistic phenomena, and hence pseudoscientific according to Popper's principles of demarcation. In other words, these derogatory terms are used solely in order to contrast our world to a hypothetical, ideal world, where the mechanistic myth and the pseudoscientific notions it engenders would not exist. The meaning of these terms, therefore, must not be confused with their informal meaning in general discourse, nor with their formal meaning in various moral, professional, or legal definitions. Moreover, the use of these terms expresses strictly the personal opinion of the author – an opinion based, as already stated, on the principles of demarcation.

This book aims to expose the corruptive effect of the mechanistic myth. This myth, especially as manifested through our software-related pursuits, is the greatest danger we are facing today. Thus, no criticism can be too strong. However, since we are all affected by it, a criticism of the myth may cast a negative light on many individuals and organizations who are practising it unwittingly. To them, the author wishes to apologize in advance.

Contents

Preface

This revised version (currently available only in digital format) incorporates many small changes made in the six years since the book was published. It is also an opportunity to expand on an issue that was mentioned only briefly in the original preface.

Software and Mind is, in effect, several books in one, and its size reflects this. Most chapters could form the basis of individual volumes. Their topics, however, are closely related and cannot be properly explained if separated. They support each other and contribute together to the book's main argument.

For example, the use of simple and complex structures to model mechanistic and non-mechanistic phenomena is explained in chapter 1; Popper's principles of demarcation between science and pseudoscience are explained in chapter 3; and these notions are used together throughout the book to show how the attempts to represent non-mechanistic phenomena mechanistically end up as worthless, pseudoscientific theories. Similarly, the non-mechanistic capabilities of the mind are explained in chapter 2; the non-mechanistic nature of software is explained in chapter 4; and these notions are used in chapter 7 to show that software engineering is a futile attempt to replace human programming expertise with mechanistic theories.

A second reason for the book's size is the detailed analysis of the various topics. This is necessary because most topics are new: they involve either entirely new concepts, or the interpretation of concepts in ways that contradict the accepted views. Thorough and rigorous arguments are essential if the reader is to appreciate the significance of these concepts. Moreover, the book addresses a broad audience, people with different backgrounds and interests; so a safe assumption is that each reader needs detailed explanations in at least some areas.

There is some deliberate repetitiveness in the book, which adds only a little to its size but may be objectionable to some readers. For each important concept introduced somewhere in the book, there are summaries later, in various discussions where that concept is applied. This helps to make the individual chapters, and even the individual sections, reasonably independent: while the book is intended to be read from the beginning, a reader can select almost any portion and still follow the discussion. In addition, the summaries are tailored for each occasion, and this further explains that concept, by presenting it from different perspectives.

❖

The book's subtitle, *The Mechanistic Myth and Its Consequences*, captures its essence. This phrase is deliberately ambiguous: if read in conjunction with the title, it can be interpreted in two ways. In one interpretation, the mechanistic myth is the universal mechanistic belief of the last three centuries, and the consequences are today's software fallacies. In the second interpretation, the mechanistic myth is specifically today's mechanistic *software* myth, and the consequences are the fallacies *it* engenders. Thus, the first interpretation says that the past delusions have caused the current software delusions; and the second one says that the current software delusions are causing further delusions. Taken together, the two interpretations say that the mechanistic myth, with its current manifestation in the software myth, is fostering a process of continuous intellectual degradation – despite the great advances it made possible.

The book's epigraph, about Newspeak, will become clear when we discuss the similarity of language and software (see, for example, pp. 409–411).

Throughout the book, the software-related arguments are also supported with ideas from other disciplines – from the philosophies of science, of mind, and of language, in particular. These discussions are important, because they show that our software-related problems are similar, ultimately, to problems that have been studied for a long time in other domains. And the fact that the software theorists are ignoring this accumulated knowledge demonstrates their incompetence.

Chapter 7, on software engineering, is not just for programmers. Many parts (the first three sections, and some of the subsections in each theory) discuss the software fallacies in general, and should be read by everyone. But even the more detailed discussions require no previous programming knowledge. The whole chapter, in fact, is not so much about programming as about the delusions that pervade our programming practices, and their long history. So this chapter can be seen as a special introduction to software and programming; namely, comparing their true nature with the pseudoscientific notions promoted by the software elite. This study can help both programmers and laymen to understand why the incompetence that characterizes this profession is an inevitable consequence of the mechanistic software ideology.

The book is divided into chapters, the chapters into sections, and some sections into subsections. These parts have titles, so I will refer to them here as *titled* parts. Since not all sections have subsections, the lowest-level titled part in a given place may be either a section or a subsection. This part is, usually, further divided into *numbered* parts. The table of contents shows the titled parts. The running heads show the current titled parts: on the right page the lowest-level part, on the left page the higher-level one (or the same as the right page if there is no higher level). Since there are more than two hundred numbered parts, it was impractical to include them in the table of contents. Also, contriving a short title for each one would have been more misleading than informative. Instead, the first sentence or two in a numbered part serve also as a hint of its subject, and hence as title.

Figures are numbered within chapters, but footnotes are numbered within the lowest-level titled parts. The reference in a footnote is shown in full only the first time it is mentioned within such a part. If mentioned more than once, in the subsequent footnotes it is abbreviated. For these abbreviations, then, the full reference can be found by searching the previous footnotes no further back than the beginning of the current titled part.

The statement "italics added" in a footnote indicates that the emphasis is only in the quotation. Nothing is stated in the footnote when the italics are present in the original text.

In an Internet reference, only the site's main page is shown, even when the quoted text is from a secondary page. When undated, the quotations reflect the content of these pages in 2010 or later.

When referring to certain individuals (software theorists, for instance), the term "expert" is often used mockingly. This term, though, is also used in its normal sense, to denote the possession of true expertise. The context makes it clear which sense is meant.

The term "elite" is used to describe a body of companies, organizations, and individuals (for example, the software elite). The plural, "elites," is used when referring to several entities within such a body.

The issues discussed in this book concern all humanity. Thus, terms like "we" and "our society" (used when discussing such topics as programming incompetence, corruption of the elites, and drift toward totalitarianism) do not refer to a particular nation, but to the whole world.

Some discussions in this book may be interpreted as professional advice on programming and software use. While the ideas advanced in these discussions derive from many years of practice and from extensive research, and represent in the author's view the best way to program and use computers, readers must remember that they assume all responsibility if deciding to follow these ideas. In particular, to apply these ideas they may need the kind of knowledge that, in our mechanistic culture, few programmers and software users possess. Therefore, the author and the publisher disclaim any liability for risks or losses, personal, financial, or other, incurred directly or indirectly in connection with, or as a consequence of, applying the ideas discussed in this book.

The pronouns "he," "his," "him," and "himself," when referring to a gender-neutral word, are used in this book in their universal, gender-neutral sense. (Example: "If an individual restricts himself to mechanistic knowledge, his performance cannot advance past the level of a novice.") This usage, then, aims solely to simplify the language. Since their antecedent is gender-neutral ("everyone," "person," "programmer," "scientist," "manager," etc.), the neutral sense of the pronouns is established grammatically, and there is no need for awkward phrases like "he or she." Such phrases are used in this book only when the neutrality or the universality needs to be emphasized.

It is impossible, in a book discussing many new and perhaps difficult concepts, to anticipate all the problems that readers may face when studying these concepts. So the issues that require further discussion will be addressed online, at *www.softwareandmind.com*. In addition, I plan to publish there material that could not be included in the book, as well as new ideas that may emerge in the future. Finally, in order to complement the arguments about traditional programming found in the book, I have published, in source form, some of the software I developed over the years. The website, then, must be seen as an extension to the book: any idea, claim, or explanation that must be clarified or enhanced will be discussed there.

A New Form of Domination

The Risks of Software Dependence

I have stated that the elites can control knowledge by means of software just as they can by means of language, but I must clarify this point. I am not referring here to the *direct* use of software to control, acquire, or restrict knowledge. This discussion is *not* concerned with such well-known software dangers as allowing an authority to decide what information is to be stored in databases, or allowing the centralized collection of information about individuals. Nor is it concerned with the use of deceptive language in software propaganda, as in calling everything "technology," or "solution." Important as these dangers are, they are insignificant compared to the dangers we face when an elite controls *the way we create and use software.*

The other dangers we understand, and if we understand them we can perhaps deal with them. But we have yet to understand what it means for a society to depend on software as much as it depends on language; and consequently, we do not realize that it is just as important to prevent an elite from controlling software as it is to prevent one from controlling language. This ignorance can be seen in the irresponsible attitudes of our political leaders, of our corporations and educational institutions, and ultimately of every one of us: we are watching passively as the software elites are increasing their power and control year after year; and we continue to trust and respect them, even as they are creating a world where the only thing left for us to do is to operate their devices.

Thus, while the other forms of software abuse would lead to familiar forms of exploitation, what concerns us here is *a new form of domination*. We are facing a new phenomenon, a new way to control knowledge and thought, which could not exist before we had computers. The dependence of a society on software is a new phenomenon because software and programming are new phenomena. We have been inventing tools for millennia, but the computer is unique in that it is programmable to a far greater extent than any other tool we have had. Software, therefore, is what gives the computer its potency; and the act of programming is what controls this potency. No other human activity – save the use of language – is as far-reaching as programming, because no other activity involves something as potent as software.

This is the first time since humans developed languages that we have invented something comparable in scope or versatility. Software resembles language more than anything else: both systems permit us to mirror the world in our minds and to communicate with it. At the same time, software is sufficiently different from language to mask the similarity (we can easily invent new programming languages, for example, but not natural ones). As a result, we fail to appreciate the real impact that software has on society, and the need for programming expertise and programming freedom. And we have fallen victim to the fallacies of the software myth: the belief that software is a kind of product, and that software applications must be built as we build appliances; the belief that we need elaborate tools for these manufacturing projects, and hence a software industry to supply these tools; the belief that development methodologies and environments can be a substitute for programming expertise; the belief that it is better to program and maintain complex business systems by employing large teams of inexperienced programmers, analysts, and managers, instead of one professional; and the belief that the highest programming skills that human minds can attain, and that society needs, are those possessed by the current practitioners.

❖

It is not surprising that we are unprepared for the consequences of programming, since we did not take sufficient time to learn what programming really is.

Originally, we set out merely trying to develop a particular kind of machine – a fast, programmable calculator. Instead, we stumbled upon a system that gives us a whole new way to use our minds, to communicate, to represent the world. But we continue to regard programming as we did originally, as an extension to the engineering effort required to build the hardware; that is, as an activity akin to manufacturing, and which must be performed in the same fashion. We still fail to see that the skills needed to program computers are more akin to those needed to use language. Programming projects cannot be neatly broken down, like manufacturing activities, into simpler and simpler tasks. Programming skills, therefore, entail a capacity for complex structures. They can be acquired only through lengthy exposure to the phenomena arising from developing and maintaining large and complex applications.

There can be little doubt that within a few decades humans will interact with the world around them by means of software as much as they do now by means of language. Software lends itself to this task just like language, and there is no reason why we should not depend on our newly discovered programming capabilities, just as we depend on our linguistic capabilities. We must ensure, however, the right conditions: first, programmers must have the same competence with software as normal humans have now with language; and second, the activities involving programming and software must be, like those involving language, free from controls or restrictions. A society that allows an elite to control its software and programming faces the same danger as a society that allows its *language* to be controlled: through language or through software, the elite will eventually control all knowledge and thought, and will reduce human minds to the level of machines.

The form of domination that we are studying here can emerge, therefore, when a society depends on software but lacks the necessary programming competence. As Orwell points out, an elite could achieve complete control through language only by forcing us to replace our language with an impoverished one, like Newspeak, which demands no intelligence. In our present-day society, exploitation by way of language is necessarily limited, because we are all competent language users. Few programmers, however, attain a level of software competence comparable to our linguistic competence. Accordingly, the world of programming is already an Orwellian world: it resembles a society that lacks linguistic competence.

Our dependence on programming aids, and on the organizations behind them, stems from the incompetence of programmers. Programmers need these devices because they are not permitted to attain the level of programming competence of which human minds are naturally capable. But programming aids are only poor substitutes for programming expertise, because, unlike minds, they can only deal separately with the various aspects of programming.

As a result, applications based on these substitutes cannot represent the world accurately, just as statements in Newspeak cannot.

And, just as people restricted to Newspeak cannot realize how limited their knowledge is, *we* cannot realize how limited our *programming* knowledge is, because this is the only kind of programming we have. Just as the people in Orwell's society are forced to depend on the linguistic tools provided by their elite, and their knowledge is shaped and restricted by these tools, our programmers are forced to depend on the devices provided by the software companies, and their knowledge is similarly shaped and restricted. Only mechanistic software concepts, only beliefs that reinforce the software myth, can enter their minds. Programming expertise for them means expertise in the use of substitutes for programming expertise.

By preventing programming competence, then, an elite can use software to control and exploit society, just as language could be used if we lacked *linguistic* competence. The programming aids, and the resulting applications, form a complex world that parallels the real world but has little to do with it. Their chief purpose is to support a large software bureaucracy, and to prevent the emergence of a body of competent and responsible programmers. And if the software bureaucrats no longer deal with the real world, we have to reshape our own views to match theirs. To the extent that our society depends on software, and hence on this software bureaucracy, we *all* live in an Orwellian world: we are all forced to perceive our work, our values, our expectations, our responsibilities, in ways that serve the interests of the software elites.

It is unlikely that Orwell's extreme form of mind control through language can ever happen in the real world, but this is unimportant. Orwell's world is a model, not a prophesy. We must appreciate its value as model, therefore, rather than feel self-complacent because it cannot happen. And when we study it, we recognize that its importance is growing as our dependence on software is growing, because this dependence increases the possibility of an elite controlling our minds through software as the elite in the model does through language.

In our current software culture, the degree of control that an elite can attain through software is not limited by an existing condition, as control through language is limited by our linguistic competence; it rests solely on how much we depend on software. The reason the software elites do not have complete control over our minds today is not our software competence, but the fact that we do not yet depend completely on software. And if our dependence on software is growing, by the time we depend on software as much as we depend now on language it will be too late to do anything. At that point, to use Orwell's model, we will live in a world where Newspeak finally replaces English. Our chief means of thinking, of communicating, of representing the world, will be a simple system requiring only mechanistic knowledge – not because software structures cannot involve complex knowledge, but because there will be no one to create or use the kind of software that requires the full capacity of the mind.

Dependence on software, coupled with software ignorance and programming incompetence – this is what the software elites are trying to achieve. They are persuading us to give up our dependence on knowledge and skills (means through which we *know* how to become competent) and to replace it with a dependence on software (means which they control, and through which they can *prevent* us from becoming competent).

The Prevention of Expertise

1

We probably fail to recognize software domination because the idea of mind control through software is so incredible. Before we had software, only political organizations could carry out such a totalitarian project. And we have yet to accept the fact that an elite can control society through software as effectively as a political elite could through traditional means of domination.

To understand this danger, we must make the most of what we know today. We cannot afford merely to wait and see, because the resulting conditions would likely be irreversible. We must study, for example, the similarity between the role of software in society and that of language. Since we all agree that language can be used to control and restrict thought, we must ensure complete software freedom even if we still cannot see clearly how the software elites can turn software into a means of domination. We should simply assume that they will use software as they would language, had they the opportunity to control language as they do software.

Even more importantly, we must study those aspects of society that are *already* controlled by the software elites: those aspects that form the world of programming itself. Studying the world of programming affords us a glimpse of the future, of the time when the entire society will be controlled by these elites. It was easy to degrade the notion of programming expertise because, this being a new field, there were no established values, as there are in the traditional professions. As a result, the highest level of expertise we believe to be needed in programming is one that in other professions would be considered the level of novices.

❖

We have been involved with software for more than half a century, so by now we could have had a sufficient number of expert programmers; namely, men and women whose skills represent *the utmost that human minds can attain in the domain of programming*. This is how we define expertise in other professions, and this is what we should expect of programmers. Instead, what we find is a software *bureaucracy*: a social system whose chief doctrine is the *prevention* of programming expertise.

We have programmers who are incapable of performing anything but small and isolated programming tasks, and who are not even expected to do more. We have managers who read "success stories" in childish computer publications, and search for ready-made applications and other programming substitutes instead of allowing their own programmers to gain the necessary skills. We have professors and gurus who teach the principles of "software engineering" – which claim that programming is like manufacturing, so what we need is unskilled labourers who know only how to assemble "prefabricated software components." We have software companies that supply practitioners with an endless series of "software tools" – elaborate development and business systems that promise to eliminate the need for programming. And, addressing the incompetence engendered by this corrupt culture, there are thousands of books, magazines, newspapers, brochures, advertisements, catalogues, trade shows, newsletters, courses, seminars, and online sources, all offering "solutions."

Few people realize that this whole bureaucracy could be replaced with a relatively small number of real, expert programmers. This is true because only a fraction of the work performed by the current practitioners is actually useful; that is, directly related to the creation and maintenance of applications. Most of their work consists in solving the problems generated by their dependence on aids and substitutes.

We have no equivalent bureaucracy in other professions. We have surgeons, pilots, engineers, musicians, military commanders, writers, repairmen, and so forth. And we understand that, to reach expertise in a difficult profession, an individual needs many years of education, training, and practice, a sense of personal responsibility, and perhaps special talents as well. We don't attempt to replace a surgeon with a dozen ignorant individuals, and defend the decision by claiming that the work of a surgeon can be broken down into simpler tasks, as in manufacturing.

We don't do this in other professions because we took the time to determine what is the *highest* level that human beings can attain in those fields. We made *that* level our definition of expertise, and we measure everyone's performance against that level. We understand that the more difficult the profession, the longer it takes to attain expertise, and the fewer the individuals who can succeed; and we give these individuals the time and the opportunity to develop their capabilities. We never tried to contend with this problem by reducing our expectations, as we do in programming. We never concluded that, given the urgent need for surgeons, the answer is to debase the definition of expertise to match the level of the available, of the inexperienced, individuals.

We treat programming differently from other professions because this serves the interests of the software elites. In just a few years, an unprecedented propaganda system has made the software myth the greatest system of belief in history, and we now take for granted in the domain of programming, notions that we would dismiss as absurd in other domains. The software practitioners have become a powerful, self-serving bureaucracy, but we continue to regard them as saviours. The reason we fail to see that they are *exploiting* society, not serving it, is that we have no other form of programming as measure. Programming controlled by a bureaucracy is the only programming we know, the only kind we have ever had.

An important element of the software myth is the belief that the typical work currently performed by programmers represents the highest level of competence we should expect in this profession. And if they have difficulty with their applications, it is not greater programming knowledge that they need, but more programming aids and substitutes. Thus, individuals with just a year or two of training and experience – which consist largely in the use of aids and substitutes – have reached the highest knowledge expected of them. The only knowledge they will acquire from then on is how to use the *future* aids and substitutes. This doctrine fits well within the ideology promoted by the software elites, as it ensures continued incompetence among programmers. It also ensures the complete dependence of programmers, and of those using their applications, on the software companies supplying the aids and substitutes. Lastly, this doctrine serves to weaken the programmers' sense of responsibility: what they perceive as their main concern is the problems generated by the aids and substitutes, rather than the real social or business problems that software is supposed to solve.

As a result, no matter how many years of practice programmers have behind them, their real programming experience remains as it was after the first year or two. This is true because the aids and substitutes limit their work to simple and isolated bits of programming, whereas successful application development demands the capacity to deal with many software processes simultaneously. This incompetence also explains why most applications are inadequate, and why most programming work consists in replacing existing applications, which programmers cannot keep up to date.

❖

If we can benefit from studying the similarity of software and language, and from studying the world of programming and the delusions of the software myth, then we can benefit even more from studying these topics together; specifically, from studying the link between the mechanistic ideology and the incompetence of programmers.

We should regard the world of programming as the result of an unintended social experiment: an attempt to replace human expertise with software. The experiment has failed, but we can learn a great deal from this failure. We must create, to begin with, the social and business environment where a body of expert programmers can evolve. The software elites are doing everything in their power to prevent this, of course, since widespread programming incompetence is a critical factor in their plan of domination. A true programming profession will not only stop the flow of trillions of dollars from society to the software elites and bureaucrats, but will lead to better software and, ultimately, greater benefits from computers.

Moreover, by abolishing the software bureaucracy we will prevent the software elites from corrupting other aspects of society. For they are using the power gained from controlling the world of software, to degrade other professions and occupations just as they have degraded programming. If allowed to continue, they will soon force us all to depend on knowledge substitutes instead of our minds. Like programmers, we will all be reduced to the level of novices. As programmers do now, we will all live in a world where expertise is neither possible nor necessary, where the only thing left to do is to operate the devices supplied by the software elites.

2

To understand the concept of software domination, we must start by recalling what we learned in chapter 2 about the mind. We can acquire the most diverse kinds of knowledge and skills: using language, recognizing faces, playing games, programming computers – the kind of knowledge we all share simply by living in a society, as well as specialized knowledge related to individual lifestyles and occupations. But all knowledge and skills, ultimately, involve our mind's capacity for complex structures. When exposed to a new phenomenon, and hence to the new knowledge embodied in that phenomenon, we start by noting the *simple* structures that make it up, the patterns and regularities. What we note, in other words, is those aspects that can be represented with facts, rules, and methods. Being limited to simple structures, our performance at this point resembles that of a software device. We progress from novice to expert by being exposed to that phenomenon repeatedly. This permits our mind to discover, not only more structures, but also the *interactions* between structures; and this in turn permits it to create a replica of the *complex* structures that make up the phenomenon. Thus, when we reach expertise our performance exceeds that of software devices, which are forever restricted to simple structures.

We communicate with the world around us through our senses, which receive information in the form of simple structures (patterns of symbols and sounds, for instance). Complex structures, therefore, can exist only in the phenomenon itself and in the mind; we cannot acquire them through our senses directly from the phenomenon, or from another mind. Consequently, the only way to attain expertise in a given domain is by giving our mind the opportunity to create the complex structures which reflect the phenomena of that domain. And this the mind can do only through repeated exposure to those phenomena; in other words, through personal experience.

Human acts require the capacity for complex structures because most phenomena we face consist of interacting structures. They consist of entities (objects, persons, processes, events) that have many attributes, and belong therefore to many structures at the same time – one structure for each attribute. To put this differently, we can always view ourselves and our environment from different perspectives, while the entities that constitute our existence are the same. So the entities form many structures; but the structures interact, because they share these entities. We rarely find a structure – a particular aspect of our life – whose links to the other structures are so weak that it can be extracted from the complex whole without distorting it or the others.

This recapitulation was necessary in order to remind ourselves of the conclusion reached in chapter 2, and its significance. We note that most mental processes, most knowledge and skills, involve *complex* structures. And we note also that software devices are based on *simple* structures. As substitutes for human intelligence, therefore, software devices are useful only for the rare tasks that can be represented with simple structures; specifically, those tasks that can be separated from others.

On the one hand, then, practically everything we do involves the full capacity of the mind, and cannot be broken down into simpler mental processes. On the other hand, we agree to depend more and more, in almost every domain, on software devices – which attempt to eliminate the need for expertise by reducing knowledge to simple structures. How can we explain this contradiction?

Our software delusions stem from our mechanistic delusions. Our most popular theories of mind claim that human intelligence can be represented with mechanistic models – models based on precise diagrams, rules, methods, and formulas. And, even though these theories keep failing, we also believe now that it is possible to represent intelligence with mechanistic *software* models. Thus, the promoters of mind mechanism can claim, for the first time, to have *actual* devices – software devices – that emulate human intelligence. Anyone with a computer can now perform any task, including tasks requiring knowledge that he lacks. All he needs to do is purchase a software device which contains that knowledge.

We have always used tools to simplify tasks, or to improve our performance; so the idea that a device can enhance certain types of knowledge and skills, or help us perform some tasks faster or better, is not new. If we view software as a device of this kind, the claims are easily justified: the computer, with the programs that run on it, is the most versatile tool we have ever invented; and it can enhance our capabilities in many tasks.

The software claims, though, do not stop at the kind of claims traditionally advanced for devices. The claims are extended to encompass *intelligent* acts; that is, acts involving non-mechanistic knowledge, and hence complex knowledge structures. But devices can represent only *simple* structures. So, to replace those acts with software, we must first separate the complex knowledge structure into several simple ones.

Software domination, thus, starts when we are tempted to commit the fallacy of reification. We believe the claim that knowledge and skills can be replaced with software devices because we already believe that intelligent acts can be broken down into simpler intelligent acts. This belief tempts us to reify the phenomenon of intelligence, and commit therefore, with software, the fallacy already committed by the mechanistic theories of mind: the separated knowledge structures are no longer what they were when part of the complex knowledge; they lose the interactions, so even when we manage to represent them faithfully with software, the knowledge embodied in them is not the same as the original, complex knowledge.

But reification is only the first step. Now that we have independent structures, we are tempted to start from higher levels of abstraction within each structure as we replace it with software. We can be more productive, the experts tell us, if we avoid the low levels of software and start with higher-level elements – with elements that already contain the lower levels. Thus, we also commit the second fallacy, abstraction: we believe that we can accomplish the same tasks as when starting with low-level elements. Starting from higher levels impoverishes the structure by reducing the number of alternatives for the value of the top element; that is, the top element of a software structure that is already a reified, and hence inaccurate, representation of the real knowledge. What this means in practice is that an inexperienced person will accomplish by means of software devices only a fraction of what an experienced person will with his mind alone.

The two fallacies can be seen clearly in the domain of programming. We are told that the most effective way to develop applications is by starting from high levels of abstraction. Specifically, we should avoid programming as much as possible, and use instead software entities that already include many elements: ready-made applications (or, at least, ready-made modules and components), and the built-in operations provided by development tools and environments. To benefit from these high levels, however, we must view our applications, mistakenly, as separable software processes. Each business or software practice, each case of shared data or operations, is a process; and each process represents one aspect of the application, one structure (see "Software Structures" in chapter 4). These structures exist at the same time and use the same software entities, so it is their totality that constitutes the application. If we separate them, we may indeed manage to program each structure starting from higher-level elements; but the resulting application will reflect only poorly the original requirements. We *can* create applications that mirror reality, but only if we have the expertise to start from low levels and to deal with all the processes together.

3

We are now in a position to understand the concept of software domination. The software elites are exploiting our mechanistic delusions; specifically, our belief that software can be a substitute for non-mechanistic knowledge. We see software successfully replacing human minds in *some* tasks, and we trust the elites when they promise us similar success in other tasks. We believe that there is only a quantitative, not a qualitative, difference between the knowledge involved in mechanistic and non-mechanistic tasks, and we allow the elites to exploit this belief.

So the mechanistic delusions act like a trap. If we believe that a given task can be replaced with software, we do not hesitate to depend on that software and on the organization behind it. We enthusiastically devote our time to that software, instead of using it to gain knowledge. But if the task is non-mechanistic, the time we devote to software will be wasted. By luring us with the promise of immediate answers to our complex problems, the elites prevent us from developing our minds. So, not only does software fail to solve those problems, but it also prevents us from gaining the knowledge whereby we *could* solve them. In the end, we have nothing – neither a useful software expedient, nor the knowledge needed to perform the task on our own. We are caught, thus, in the software variant of the traditional mechanistic trap: we believe in the existence of mechanistic solutions for non-mechanistic problems, so we restrict ourselves to precisely those methods that cannot work.

We get to depend on software because the promise is so enticing. The promise is, essentially, that software can solve important problems in a particular field by acting as a substitute for the knowledge needed in that field. So, instead of taking the time to acquire that knowledge, we can solve the problems right away, simply by buying and operating a piece of software. And the reason we believe this promise is that we see similar promises being fulfilled in tasks like calculations and data processing, where software does indeed allow novices to display the same performance as experts. For tasks involving interacting knowledge structures, however, the promise cannot be met. Software can be a substitute only for knowledge that can be neatly broken down into smaller, simpler, and independent pieces of knowledge – pieces which in their turn can be broken down into even smaller ones, and so on – because this is the only way to represent knowledge with software.

When we depend on software in situations involving complex knowledge structures, what we notice is that it does not work as we expected. But, while software fails to provide an answer, we never question the mechanistic assumptions. We recognize its inadequacy, but we continue to depend on it. And this dependence creates a new *kind* of problems: software-related problems. We don't mind these new problems, though, and we gladly tackle them, because we think that by solving *them* we will get the software to solve our original problems. Most software problems involve isolated knowledge structures, and have therefore fairly simple, mechanistic solutions. Since these solutions often entail additional software, they generate in their turn software problems, which have their own solutions, and so on.

The more software problems and solutions we have, the more pleased we are with our new preoccupations: all the time we seem to be *solving* problems. So we spend more and more time with these software problems, when in fact they are spurious problems generated by spurious preoccupations, in a process that feeds on itself. We interpret our solutions to software problems as progress toward a solution to our real problems, failing to see that, no matter how successful we are in solving software problems, if the original problems involve complex knowledge we will never solve *them* with software. In the end, content with our software preoccupations, we may forget the real problems altogether.

❖

Bear in mind that it is not software dependence in itself that is dangerous, but the combination of this dependence with software ignorance. There is nothing wrong in depending on software when we possess software expertise. It is only when we separate software structures from various knowledge structures, and when we restrict ourselves to high-level elements, that our dependence on software can be exploited; in other words, when our software knowledge is at a mechanistic level.

Recall the language analogy. We attain linguistic competence by starting with *low-level* linguistic elements – morphemes and words, with all their uses and meanings – and by creating language structures that interact with various knowledge structures present in the mind. This is how we form the complex mental structures recognized as intelligence. No linguistic competence or intelligence would be possible if we had to create our language structures starting with ready-made sentences and ideas, or if we treated them as independent structures. Similarly, software expertise can be attained only by starting with *low-level* software elements, and by treating the software structures that we create with our programs, not as independent structures, but as part of the complex structures that make up our affairs. And this is precisely what the software elites are *preventing* us from doing.

We are exploited through software because of the belief that it is possible to benefit from software without having to develop software expertise. We understand why we could not benefit from *language* without having linguistic competence, but we fail to see this for software. When we trust the elites and get to create and use software in the manner dictated by them, we end up with a combination of several weaknesses: programming incompetence; failure to solve our problems with software, because of the inadequacy of our applications; a growing preoccupation with spurious, software-related problems; and a perpetual dependence on the elites for solutions to both the real and the spurious problems – a dependence that is futile in any case, because only we, with our minds, can hope to accomplish those tasks that require complex knowledge. It is not difficult for the elites, then, to exploit us.

The Lure of Software Expedients

1

Software domination is based on a simple stratagem: consuming people's time. Forcing us to waste our time is the simplest way to keep us ignorant, since, if we spend this time with worthless activities, we cannot use it to improve our minds. The software elites strive to keep us ignorant because only if ignorant can normal people be turned into bureaucrats, into automatons.

Our time, in the end, is all we have, our only asset. Whether we count the hours available in a day or the years that constitute a life, our time is limited. What we do with our time, hour by hour, determines what we make of our lives. We can squander this time on unimportant pursuits, or use it to expand as much as we can our knowledge and skills. In the one case we will accomplish whatever can be done with limited knowledge, and we will probably live a dull life; in the other case we will make the most of our minds, and we have a good chance to live a rich life and to make a contribution to society.

It is not too much to say that, as individuals living in a free society, we are, each one of us, responsible for a human life – our own – and we have an *obligation* to make the most of it. Like freedom itself, realizing our human potential is a *right* we all have; but, just like freedom, this right is also a *duty*, in that we are all responsible for its preservation by defending it against those who want to destroy it. Specifically, we must strive to expand our minds *despite* the attempts made by an elite to keep us ignorant. Only thus, only when each individual *and* each mind counts, can the idea of freedom survive.

Conversely, preventing an individual from realizing his or her potential, from making the most of his or her mind, amounts in effect to an attempt to destroy a life; so it must be considered a crime nearly as odious as murder. Seen from this perspective, our software elites could be described as criminal organizations, since forcing us to squander our time is one of the principles of their ideology.

The software elites consume our time by creating an environment where our activities are far below our natural mental capabilities. When we depend on their concepts and devices, we end up spending most of our time acquiring isolated bits of knowledge, or solving isolated problems. Being simple and mechanistic, these activities do not allow us to create complex knowledge structures in our minds – the kind of knowledge that constitutes skills and experience. We can recognize this in that our capabilities do not progress on a scale from novice to expert, as they do in the traditional fields of knowledge. (Thus, no matter how many of these problems we solve, the next one will demand about as much time and effort.) In any case, mechanistic concepts cannot help us to solve our *complex* problems; so we are wasting our time both when acquiring the mechanistic knowledge and when using it.

Without exception, the software devices are presented as simple, easy to use, requiring little knowledge, and demanding an investment of just a few minutes, or perhaps a few hours. It is a sign of our collective naivety that, in a world which is becoming more complex by the day, we believe in the existence of some devices that can provide immediate answers to our problems, and that this power is ours to enjoy just for the trouble of learning how to operate them. This childish belief can be understood only by recognizing it as the software variant of our eternal craving for salvation: the performance of some simple acts, we think, will invoke the assistance of fabulous powers. This is the same belief that permitted so many other elites to exploit us in the past.

Many of these devices are indeed simple, just as their promoters claim. But they are simple precisely because they are generally useless, because they *cannot* solve our complex problems. As we saw, they are based on the delusion that complex structures – our problems, and the knowledge required to solve them – can be broken down into simple structures, and hence replaced with mechanistic expedients like software devices. The mechanistic knowledge required to use the software, together with the mechanistic capabilities of the software itself, is believed to provide a practical substitute for the complex knowledge required to solve the problems. But no combination of simple structures can replace a complex knowledge structure, because this kind of knowledge can develop only in a mind, and only after much learning and practice.

❖

When we trust the software elites, we are exploited more severely and in more ways than we think. For, the waste of time and the prevention of knowledge reinforce each other, impoverishing our lives and degrading our minds to the point where we can no longer understand what is happening to us. This is an important point, and we must dwell on it.

We saw that the promise of immediate solutions to our problems tempts us to depend on mechanistic software expedients – ready-made pieces of software, or software tools and aids. Complex knowledge becomes both unnecessary and impossible, as all we need to do is select and combine various operations within the range of alternatives permitted by a device. We agree to depend on mechanistic expedients because we believe this is the most effective way to accomplish a given task, when in fact most tasks require *non-mechanistic* knowledge.

But it is important to note that there is a second process at work here: if we spend our time engaged in activities requiring only mechanistic thinking, we lose the *opportunity* to develop complex, non-mechanistic knowledge. We have the *capacity* for non-mechanistic knowledge, but we will not take the time to *develop* it as long as we believe that the simpler, mechanistic knowledge suffices. So our knowledge remains at mechanistic levels, and we continue to depend on the mechanistic expedients, even though they are inferior to our own capabilities.

What distinguishes human minds from devices is the capacity to develop complex knowledge. But, as we saw in chapter 2, the only way to attain complex knowledge is through personal experience: by engaging in activities that require that knowledge, by being exposed repeatedly to the complex phenomena that embody it. Thus, in some domains we need many years of learning and practice to attain the knowledge levels recognized as expertise. Mechanistic knowledge, on the other hand, can be acquired fairly quickly. (See pp. 155–157.)

The promise of mechanistic solutions to complex problems is, then, a trap. When inexperienced, and hence limited to mechanistic knowledge, the mechanistic expedients do indeed exceed our skills. It is only later, when we develop non-mechanistic knowledge, that we will outperform them. But we will never reach that level if we get to depend on mechanistic expedients from the start, because this very dependence deprives us of the opportunity to develop non-mechanistic knowledge.

This degradation – restricting us to mechanistic thinking, to a fraction of our mental capabilities – is the goal of the software elites when tempting us with mechanistic expedients. The prevention of non-mechanistic knowledge is a critical element in their plan of domination, because they must ensure that we remain inferior to their devices, and hence dependent on them.

We can choose only one of the two alternatives: either pursue activities demanding mechanistic knowledge (because they are easy and immediately accessible), or take the time to develop non-mechanistic knowledge. Mechanistic knowledge (following rules and methods, operating a software device) we can quickly acquire at any time, while non-mechanistic knowledge (the experience to perform complex tasks, the creativity to solve important problems) requires many years of learning and practice.

The software elites encourage us to choose the first alternative. This choice brings immediate rewards and is hard to resist, but it restricts us forever to mechanistic thinking. To prefer the second alternative, we must appreciate the potential of our minds. This choice amounts, in effect, to an investment in ourselves: we decide to forgo some easy and immediate benefits, and, instead, take the time to develop our minds. But we can make this choice only if we already have an appreciation of non-mechanistic knowledge, only if we realize how much more we can accomplish later, when we attain this type of knowledge. And we can develop this appreciation only if, when young or when novices in a particular field, we note around us both mechanistic and non-mechanistic knowledge, and learn to respect those who possess the latter – because their skills exceed ours by far.

The software elites, however, are creating a culture that fosters mechanistic thinking – a culture where non-mechanistic capabilities offer no benefits, as we are all expected to stay at about the same skill level. More and more, in one occupation after another, the only thing we have to know is how to use a software system. The notions of expertise, creativity, professionalism, and responsibility are being degraded to mean simply the skill of following methods and operating devices. As we depend increasingly on mechanistic knowledge alone, non-mechanistic knowledge is becoming redundant: we have fewer and fewer opportunities to either develop it or use it.

By creating a culture where all we need is mechanistic knowledge, the elites make it impossible for us to discover the superiority of *non-mechanistic* knowledge. We are trapped in a vicious circle: we start by being inexperienced and hence limited to mechanistic knowledge; at this point our performance is inferior to their devices, so the elites easily persuade us that the only way to improve is by using devices; as we get to depend on devices, the only knowledge we acquire is the mechanistic knowledge required to operate them; so our skills remain below the level of devices, and we believe that we must continue to depend on them. The only way to escape from this trap is by developing non-mechanistic knowledge, and thus becoming superior to the devices. But this takes time, and time is precisely what we do not have if we squander it on a preoccupation with devices. As long as we trust the elites, therefore, we are condemned to using only the *mechanistic* capabilities of our minds; we are condemned, in other words, to staying at novice levels forever.

Let me put this differently. To control our life, the software elites must induce a state of permanent ignorance and dependence. And they achieve this by persuading us to trust their mechanistic expedients – concepts, theories, methods, devices – while these expedients can rarely solve our real problems. Consuming our time by keeping us preoccupied with their expedients is a critical factor in the process of domination, because the elites must prevent us from using this time to develop our minds. And promoting worthless expedients is an integral part of this process: they wouldn't give us useful ones even if they could. Only expedients that do *not* work can be employed to consume our time; only by *not* solving our problems can they add to our spurious, software-related preoccupations. No domination would be possible if we were asked to depend on the elites only in those few situations where their expedients are indeed superior to our minds (that is, where a complex phenomenon can be usefully approximated with simple structures).

Promoting mechanistic expedients, thus, ensures our continued dependence in two ways at once: by restricting our knowledge and skills to levels even lower than those attained by the expedients, and by consuming our time with the endless preoccupations generated by the expedients.

2

It may be useful to recall our software preoccupations, although we are already spending so much time with them that they are well known. Installing new software, seeking technical support, downloading updates, studying lists of "frequently asked questions," checking the latest notes on a website, reading computer magazines, discovering "undocumented features," running virus protection utilities, printing online manuals, trying to get different pieces of software to work together – these are some of the activities we must perform when involved with software. But these are only the incidentals. We must also include the time required to learn to use the software (the features and options we have to assimilate, how to specify and combine them, keeping up with changes from one version to the next), and the time we take to actually use it, once we get to depend on it.

These activities require almost exclusively mechanistic knowledge: they consist of isolated and fairly simple tasks, which cannot help us to develop an important body of knowledge or skills. We note this in that almost everyone, regardless of age or experience, has to deal with the same kind of problems; and almost everyone manages to solve them. We also note it in that, no matter how many of these problems we faced in the past, we will still face similar ones in the future. In other words, the proportion of time we must devote to software-related problems does not decrease significantly with experience.

Anyone who encountered software-related problems is familiar with the feeling of satisfaction experienced when finally uncovering the answer. The answer is usually a simple fact; for instance, learning that we must select one option rather than another in a particular situation. But instead of being outraged that we had to spend time with an activity so simple that we could have performed it as children, we perceive it as an essential aspect of our work, so we believe that we have learned something important. Although we don't think of this activity as a form of amusement, we experience the satisfaction of solving a puzzle. And even if it is true that we must now spend a great part of our time solving puzzles instead of addressing real problems, it is significant that these are *trivial* puzzles, demanding only a fraction of our mental capabilities. Clearly, there is no limit to the number of software-related puzzles that we can find, and hence the time we must take to deal with them, if we agree to depend on concepts and products that cannot solve our real problems to begin with.

❖

Any activity, method, or tool entails some incidental preoccupations, so we cannot expect to benefit from software without investing some time; and we may even have to spend part of this time dealing with trivial issues. Thus, what I am trying to show here is *not* that our collective preoccupation with software is too great relative to the benefits we derive from it. Such deficiency we could attribute to the novelty of software and to our inexperience. We could then conclude that this condition is transient, and that we will eventually become as able in our software pursuits as human beings can be – just as we have become in other domains.

What I am trying to show, rather, is that this interpretation is wrong, that our incompetence is getting worse not better, that our software preoccupations do not reflect a natural process of intellectual evolution. On the contrary: the incompetence is deliberately fostered by the software elites as part of a monstrous plan of domination founded on our mechanistic delusions, and made possible by our growing dependence on computers – a plan whose goal is to degrade the mind of every human being on earth.

Our continued ignorance in software-related matters – programming, in particular – is essential in this plan of domination, because software knowledge, like linguistic knowledge, is related to all other types of knowledge. *Software* ignorance and dependence, thus, are only the means to bring about *total* ignorance and dependence. It is in order to induce this collective ignorance and dependence that the software elites are exploiting our mechanistic delusions, and the consequent software delusions. For, as long as we believe that knowledge and skills can be replaced with mechanistic concepts, we will inevitably conclude that we must depend on organizations that produce devices based on these concepts – just as we depend on organizations that produce appliances, detergents, or electricity.

In reality, to succeed in software-related activities – programming, in particular – we need *skills*. And, like other skills, these new skills depend on our own capabilities and experience. Also like other skills, they demand the full capacity of the mind, and, to attain expertise, many years of learning and practice.

The software elites are promoting the notion of knowledge substitutes precisely because these substitutes are worthless. It is precisely because they cannot replace skills, and hence fail to solve our problems, that we constantly need new ones and spend so much time with them. By consuming our time with the petty preoccupations generated by these substitutes, the elites are preventing us from developing skills, thus ensuring our continued incompetence and dependence.

Were we permitted, as society, to develop software skills as we develop skills in other domains – were we permitted, in other words, to attain the highest level that human minds can attain in software-related matters – the issues of incompetence and waste of time would not even arise. We would then be, quite simply, as good in these new skills as we can possibly be; and we would take as much time with our software preoccupations as is justifiable. This is how we progressed in other domains, and there is no reason to view software and programming differently. It is unlikely that we have already reached the highest level, or that we are advancing in that direction, since we are using now mostly *mechanistic* knowledge; in other domains, it is our *non-mechanistic* capabilities that we use when we attain expertise. The software elites can persuade us to prefer their mechanistic substitutes to our own minds only because, as society, we have no idea how good we can actually be in software-related matters: we never had the opportunity to find out.

All skills – whether easy or difficult, acquired early in life or later – entail the same mental processes. Interpreting visual sensations, recognizing social contexts, diagnosing diseases, playing musical instruments, flying airplanes, repairing appliances, teaching children, managing warehouses – we can acquire almost any skill, but the only way to acquire it is by performing the activities involved in that skill, and by allowing our mind to discover the complex knowledge structures which constitute that skill. For no other skills can we find an elite that prevents us from using the full capacity of our mind, or forces us to use methods and devices instead of expanding our knowledge, or redefines expertise to mean expertise in the use of substitutes for expertise. From all the skills we can acquire, only those associated with software and programming seem to have engendered such notions, and the reason is simple: these skills are *more complex* than the others, and hence misunderstood. They are so complex, in fact, that they permit us to use our mind and view the world in entirely new ways. They are comparable in scope only to our linguistic skills.

So it is the complexity itself that allows charlatans to deceive us. We became infatuated with software too quickly, without taking the time to appreciate the true meaning and the implications of software knowledge. We still do not understand what can happen when a society depends on software while software is controlled by an authority. This is why we continue to accept the absurd notions promoted by the software elites; in particular, the notion that we must depend, in all software-related affairs, on systems produced by software companies. It is precisely because software knowledge is so difficult that we are tempted by theories which tell us that we can enjoy the benefits of software without taking the time to develop software knowledge.

3

Our preoccupation with *ease of use* deserves a brief analysis. Software – applications, development systems, utilities – is always promoted with the claim that it is easy to use. I want to show, however, that the belief in easy-to-use software is a mechanistic delusion. The notion "easy to use" is, strictly speaking, meaningless.

Like any tool or device, a piece of software cannot be any easier to use than whatever effort is required to accomplish a given task. The only sensible claim, therefore, is that it is *well-designed*. A lathe, for example, even if well-designed, is necessarily more difficult to use than a chisel. And so it is with software: all we can expect of a particular business application, or a particular development tool, is that it be well-designed. Once this requirement is fulfilled, the notion "easy to use" becomes irrelevant: that software will be as easy or as difficult to use as software can be in a particular situation.

Now, we see the claim "easy to use" for *all* types of software – for business and for home, for programmers and for end users. We never see software described, for example, with the warning that we need much knowledge, or many months of study and practice, in order to enjoy its features. Thus, as we are becoming dependent on software in practically everything we do, if all this software is also easy to use, we reach the absurd conclusion that human beings will never again have to face a challenging situation.

The delusion of easy-to-use software becomes clearer if we recall the other quality commonly claimed for software – *power*. Just as all software devices are said to be easy to use, they are also said to be powerful. The two qualities are often claimed together, in fact, as in the famous phrase "powerful yet easy to use." By combining the two qualities, the following interpretation presents itself: we believe that software devices embody a certain power, and we perceive ease of use as the ease of invoking this power.

The only power that can inhere in a software device is its built-in operations; that is, higher levels of abstraction for our starting elements. And it is the higher levels that also make the device easy to use. The power and ease of use are illusory, however: high starting levels make the device convenient when our needs match the built-in operations, but awkward or useless otherwise.

We saw this with the language analogy: we have *less* power when starting with ready-made sentences; we must start with *words* if what we want is the capability to express any conceivable idea. Similarly, if true software power is the capability of a system to handle any situation, the only way to have this power is by starting with *low-level* entities. Like claiming ease of use, therefore, claiming power for a software device is nonsensical: what is usually described as power is the exact opposite of software power. Moreover, since ease of use can be attained only by providing higher starting levels, and hence by *reducing* the power of the device, claiming both power and ease of use at the same time is especially silly.

Putting all this together, it is obvious that the software elites want us to think of software as an assortment of devices that have the power to solve our problems, while all *we* have to do is *use* them. The only thing left for us to do from now on is operate software devices; and this we can learn in a matter of hours. This notion is absurd, as we just saw, but we enthusiastically accept it. The elites are plainly telling us that we will no longer have the opportunity to use the full capacity of our minds, that our sole responsibility will be to perform tasks so simple that anyone can learn to perform them in a short time. But instead of being outraged, we welcome this demotion; and, to rationalize it, we interpret our diminished responsibility as a new kind of expertise.

❖

The elites also claim that software devices will enhance our creativity, by taking over the dull, routine activities. With higher starting levels, the elites tell us, we can reach the top element of a given structure much sooner. Why waste our time and talents with the details of the low levels, when the functions built into these devices already include all the combinations of low-level elements that we are likely to need? When starting from low levels we squander our superior mental capabilities on trivial and repetitive tasks; let the computer perform this tedious work for us, so that we have more time for those tasks demanding creativity. Just as successful managers and generals deal only with the important decisions and leave the details to their subordinates, we should restrict ourselves to high-level software entities and leave the details to the computer.

It is easy to show the absurdity of these claims. We are told to give up the details, and to use instead ready-made entities, so that we have more time for the important work, more time to be creative. But our work *is* the development of high-level entities from low-level ones. In one occupation after another, the software elites are redefining the concept of work to mean the act of combining the high-level entities provided by their devices. To be creative, however, we must be able to arrive at *any one* of the possible alternatives; and this we can do only by starting with *low-level* entities. Moreover, we are offered software devices in *all* fields of knowledge, so we cannot even hope that the time we perhaps save in one type of work will permit us to be more creative in another.

Returning to the language analogy, if a writer used ready-made sentences instead of creating new ones, starting with words, we would study his work and recognize that he is not being *more* but *less* creative. Clearly, fewer ideas can be expressed by selecting and combining ready-made sentences than by creating our own, starting with words. And this is true for all types of knowledge: the higher the level we start with, the greater the effect of reification and abstraction, and the fewer the alternatives for the top element. So it is absurd to claim that we can be more creative by avoiding the low levels, seeing that it is precisely the low levels that make creativity possible.

Managers and generals who make good decisions only *appear* to start from high levels. In reality, their decisions involve knowledge structures that interact at low levels, at the level of details. But this is largely intuitive knowledge, so all we can observe is the top element of the complex structure; that is, the final decision. (See "Tacit Knowledge" in chapter 2.) They developed their knowledge over many years, by dealing with all structures and all levels, low and high. This is the essence of personal experience. Were their knowledge limited to the high levels, to those selections and combinations that can be observed, then anyone could quickly become a successful manager or general – simply by learning to select and combine some high-level concepts.

This delusion is also the basis of the software devices known as expert systems – one of the sillier ideas in artificial intelligence. Expert systems claim that it is possible to capture, in a specially structured database, the knowledge possessed by a human expert in a given domain. The database consists of answers that the expert provides to certain questions – questions formulated so as to simulate various decision-making situations. Then, for a real problem, simply by interrogating the system, anyone should be able to make the same decisions that the expert would make. The fact that such devices are being considered at all demonstrates the degradation in the notions of expertise and responsibility that we have already suffered. As we saw in "Tacit Knowledge," expertise is the level where a person does *not* have to rely on rules, methods, and databases of facts (see pp. 157–158). Thus, the device can capture only the *mechanistic* aspects of the expert's knowledge; and consequently, a person using it will not emulate an expert but a novice.

Another claim we see is that software devices enhance our creativity by giving us new forms of expression. And this claim, too, is empty. Software does indeed allow us to express ourselves and to view the world in new ways, as does language. But, as in the case of language, we can only enjoy this quality if we develop our structures starting with *low-level* entities. For, only then can we discover all possible interactions between the software structures, between software structures and the other structures that exist in the world, and between software structures and the knowledge structures present in our minds. If we get to depend on software devices, and hence on *high-level* software entities, we will not only fail to develop all possible alternatives in the new, software-related matters, but we will lose alternatives in the knowledge and skills that we had in the past. In the end, we will have *fewer* alternatives than before, *fewer* ways to express ourselves. Thus, far from *enhancing* our creativity, software devices are in fact degrading our minds, by forcing us to spend more and more time with activities requiring largely mechanistic knowledge.

❖

Power and ease of use, thus, are specious qualities. The elites tell us that software devices can have these qualities because they want to replace our traditional conception of expertise with a dependence on these devices. They want us to believe that all the knowledge that matters inheres now in software devices, so all *we* have to know is how to operate them. The implicit promise is that, thanks to these devices, we don't need to know anything that we don't already know – or, at least, anything that we cannot learn in a short time.

So the elites are downgrading our conception of expertise by reducing to a minimum the range from novice to expert. If all we have to know is how to operate software devices, the difference between novice and expert is just the time taken to acquire this knowledge. Where we thought that one needs many years of study and practice to attain expertise in a difficult field, we are told that this type of knowledge is obsolete. The propaganda makes it seem modern, sophisticated, and glamorous to perform a task by operating a software device, and unprofessional or old-fashioned to perform it by using our minds. Consequently, we are according more importance to our methods and tools than we do to the results of our work. Increasingly, we are judging a person's knowledge and skills by his acquaintance with software devices, instead of his actual capabilities and accomplishments. Increasingly, it doesn't even matter what the results are, as the main criterion for assessing a professional activity is whether the person is using the latest software devices.

4

Recall the pseudosciences we studied in chapter 3. I stated there that our software delusions belong to the same tradition, that they are a consequence of the same mechanistic culture. With our software theories we are committing the same fallacies as the scientists who pursue mechanistic theories in psychology, sociology, or linguistics. When we waste our time with the spurious problems generated by our mechanistic software concepts, we are like the scientists who waste their time studying the mechanistic phenomena created by reifying human phenomena. Just as those scientists cannot explain the complex phenomena of mind and society by explaining separately the simple, mechanistic phenomena, *we* cannot solve our complex social or business problems by solving the simple, software-related problems.

These mechanistic delusions I have called *the new pseudosciences*, and we saw that they are similar to the traditional pseudosciences – astrology, alchemy, and the rest. They are similar in that they too are systems of belief masquerading as scientific theories, and they too are based on hypotheses that are taken as unquestionable truth. In the case of the new pseudosciences, the hypothesis is that mechanism can provide useful explanations for complex phenomena – for phenomena involving human minds and societies, in particular. The mechanists are, in effect, today's astrologers and alchemists: respected thinkers who attract many followers, even though their theories do not work.

Before we had software, it was only in the academic world that one could spend years and decades pursuing a mechanistic fantasy. One could hardly afford to fall prey to mechanistic delusions in business, for instance. But through software, the ignorance and corruption engendered by mechanistic thinking is increasingly affecting the entire society: corporations, governments, individuals. Through software, we are all asked now to accept fantastic mechanistic theories – theories that promise to solve our problems with practically no effort on our part. Through software, the entire society is returning to the irrationality of the Dark Ages: we are increasingly guided by dogmas instead of logic, by beliefs instead of reason.

When we believe that a software device can replace knowledge, skills, and experience, we are committing the same mistake as the scientists who believe that mechanistic theories can explain human intelligence and social phenomena. So if all of us now, not just the academics, are wasting our time with pseudoscientific theories, we must ask ourselves: Can we afford this corruption? Can our civilization survive if *all* of us engage in futile mechanistic pursuits? When mechanistic theories fail in the academic world, the harm is limited to a waste of resources, and perhaps a lost opportunity to improve our knowledge through better theories. But what price will we pay if we create a society where *all* theories fail?

As we are modifying our values and expectations to fit the mechanistic software ideology, we are adopting, in effect, mechanistic theories – theories on our capabilities as human beings, or on our responsibilities as professionals. And since these software-based theories suffer from the same fallacies as the traditional mechanistic theories, they too must fail. But what does it mean for *these* theories to fail? Since what they claim is that we can accomplish more by depending on software devices than by developing our minds, a failure of *these* theories means that we are making a wrong decision about ourselves: we mistakenly assume that our minds can be no better than some mechanistic expedients. Thus, when we decide to leave our non-mechanistic capabilities undeveloped and to depend instead on mechanistic expedients, we are causing, quite literally, a reversal in our intellectual evolution: we are choosing to degrade our conception of intelligence to a mechanistic level, and to create a world where there is no need or opportunity to exceed this level.

Let us interpret the new pseudosciences in another way. The equivalent of a world where we depend on software while being restricted to mechanistic software theories is an imaginary world where the *traditional* mechanistic theories – those explaining minds and societies – actually work. Since these theories fail to explain our *real* intelligence and behaviour, in the imaginary world we would have to alter minds and societies to fit the theories. To comply with the linguistic theory of universal grammar, for example, we would restrict our sentences, and the associated thoughts, to what can be depicted with exact diagrams and formulas; similarly, to comply with behaviourism or cognitive science, we would restrict our behaviour and mental acts to patterns that can be precisely explained and predicted; and to comply with the theories of structuralism or the social sciences, we would restrict our institutions, customs, and cultures to activities that can be described mathematically.

These theories reflect the diminished view that mechanists have of human beings – the view that our acts can be explained with precision, because our capabilities are like those of complicated machines. The scientists who invent and promote mechanistic theories wish them to work, of course. But the theories can work only if we are indeed like machines, so we must conclude that these scientists *want* us to be like machines. And if we, the subjects of these theories, also wanted them to work – if we agreed, as it were, to satisfy the wish of their authors – we would have to restrict our capabilities to what these theories can explain and predict. In other words, we would have to mutate into automatons.

What has saved us from this fate so far is not wisdom – for, if we had that wisdom we would have abandoned the mechanistic ideology already – but the fact that none of these scientists had the power to make us conform to their theories. Through software, however, it has finally become possible for the mechanists to realize their dream: a world where human beings can be designed and controlled as successfully as we design and control machines. The world that we can only *imagine* through the traditional mechanistic theories, we are actually creating through our mechanistic *software* theories. Whereas we can still think, learn, speak, and behave while ignoring the mechanistic theories of mind and society, we are forced to create and use software according to mechanistic theories. But if we are to depend on software in all aspects of our life – including those aspects studied by the theories of mind and society – then by following mechanistic software theories we *are*, in effect, mutating into the automatons that the mechanists wish us to be.

Remember, though, that it is not software dependence in itself that is harmful. On the contrary, if we were permitted to use it freely, as we use language, software would *enhance* our mental capabilities, as does language. The danger lies in the dependence on software while software knowledge is restricted to its mechanistic aspects – a policy intended to prevent us from using the full capacity of our minds.

❖

The decision we are making now is more than a choice; it is a commitment. As individuals and as society, we are making a commitment; namely, to invest in software expedients rather than our minds.

As individuals, we reaffirm this commitment when we consent to depend on software devices that are inferior to our own minds; when we spend time solving a specious, software-related problem, instead of expanding our knowledge to deal with the real problem; and when we degrade our conception of professionalism and responsibility, from the utmost that human beings can do, to merely knowing how to use software devices. As society, we reaffirm this commitment when our corporations and governments, instead of encouraging their workers to develop expertise, spend vast amounts of money on projects that increase their dependence on the software elites.

As individuals, if we are wrong, our knowledge in, say, ten years will not be much greater than what it is at present. We will waste that time acquiring worthless bits of knowledge; specifically, knowledge of ways to avoid the need for real knowledge. If we make this choice, of course, we will be unable to recognize our own ignorance in ten years; so for the following ten years we will make the same choice, and so on, and we will remain for the rest of our lives at the present level. As society, if we are wrong, within a few decades we will be where we were centuries ago: in a new dark age, ruled by elites that know how to exploit our ignorance and irrationality.

The decision we are making now is a commitment because we cannot choose both alternatives. If software mechanism is our decision, we will need only mechanistic capabilities; so we will leave our superior, non-mechanistic capabilities undeveloped. If we are wrong, we cannot reverse this decision later: if we choose the mechanistic alternative, in any domain, we will not practise; and practising is the only way to develop non-mechanistic knowledge. If we lose our appreciation of non-mechanistic knowledge, we will forget, in one occupation after another, that we *are* capable of more than just following methods and operating software devices.

This is precisely what has happened in the domain of programming. The superior alternative – personal knowledge and skills – is always available, in principle: any programmer, any manager, any company, could choose to ignore the official software ideology and treat programming as we do the other professions. Yet, despite the evidence that programming aids and substitutes are inferior to human expertise, we continue to trust the software elites and their mechanistic theories. In the domain of programming, we have already lost our appreciation of non-mechanistic knowledge.