SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Introduction: *Belief and Software*
Section *The Software Myth*

This section examines the fallacies of the mechanistic software ideology, and shows how it is preventing 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.

The Software Myth

1

The software myth is the idea of software mechanism – the enactment of mechanistic beliefs through software. If traditional mechanism holds that every phenomenon can be represented with a hierarchical structure, software mechanism holds that every phenomenon can be represented with a hierarchical *software* structure. This is true because, once we reduce a phenomenon hierarchically to its simplest entities, these entities can be emulated by means of simple *software* entities. To represent the original phenomenon, all we have to do then is combine these entities hierarchically, and thereby generate a software structure that corresponds to the structure of entities that is the phenomenon itself.

In particular, the phenomena associated with human knowledge can be represented with software. Since any type of knowledge can be reduced hierarchically to simpler and simpler pieces down to some basic bits of knowledge, by incorporating these bits in a software device we can emulate the original knowledge structure. Then, simply by operating the device, anyone will be able to perform the same tasks as a person who took the time to acquire the actual knowledge.

Software devices, thus, are perceived as substitutes for knowledge, skills, and experience. Whereas in the past we needed much learning and practice in order to attain expertise in a given field, all we need to know now, it seems, is how to operate software devices.

One type of knowledge that we have been trying especially hard to represent with software is *programming* knowledge. If software devices are only now gaining acceptance in our businesses and in our homes, their counterparts in the world of programming have existed since the 1960s. Thus, if the use of software devices as substitutes for expertise still sounds plausible for other types of knowledge, we have already had several decades to assess their value in *programming* work. And, as we will see in chapter 7, the claim that there exist substitutes for programming expertise has proved to be a fraud.

The study of software mechanism in the domain of programming can help us to understand, therefore, the delusion of software devices in general. For, it is the same myth that the elites invoke when promoting knowledge substitutes, whether they address programmers or other workers. Programming is the only domain in which we can, today, actually demonstrate the failure of software mechanism and the dishonesty of the software elites. Thus, we must make the most of this experience. If we understand how the software myth has destroyed the programming profession, we will be in a better position to recognize its dangers, and to prevent it perhaps from destroying other fields of knowledge.

2

The reason it is so tempting to think of software development as a mechanistic process is that software applications are indeed hierarchical structures – modules within modules. No matter how large or complex, it seems that an application can always be depicted as a neat structure of software entities, just as a manufactured object can be depicted as a neat structure of parts and subassemblies.

As we do in manufacturing, therefore, we should break down the process of software development into smaller and smaller parts, until we reach software entities that are easy to program. Then, as in manufacturing, we will be able to create applications of any size and complexity by employing inexperienced workers – workers who, individually, can only program small and simple pieces of software.

This idea, known as *software engineering*, is behind every programming theory of the last forty years. But the idea is wrong. We already saw that software applications are in fact *systems* of hierarchical structures, so the structure of modules that appears to represent an application is merely *one* of the structures that make it up. The software entities that constitute the application possess many attributes: they call subroutines, use database fields, reflect business practices, etc. Since each attribute gives rise to a structure, each structure represents a different aspect of the application: one subroutine and its calls, the uses of one database field, the implementation of one business practice, etc. But because they share their elements (the software entities that constitute the application), these structures are not independent. So the only way to develop applications is by dealing with several structures at the same time – something that only minds can do, and only after much practice.

Thus, while software engineering is said to turn programmers from old-fashioned artisans into modern professionals, its true purpose is the exact opposite: to eliminate the need for programming expertise. And this, the elites believe, can be accomplished by discovering scientific (i.e., mechanistic) programming theories, and by restricting programmers to methodologies and development systems based on these theories. The aim is to separate applications into their constituent structures, and further separate these structures into their constituent elements, at which point programmers will only need to deal with small, isolated software entities. For example, the theory of structured programming claims that the only important structure is the one that represents the application's flow of execution, and that this structure can be reduced to some simple, standard constructs; and the theory of object-oriented programming claims that we can treat each aspect of our affairs as a separate structure, which can then be assembled from some smaller, existing structures.

But each theory, while presented as a revolution in programming concepts, is in reality very similar to the others. This is true because they are all based on the same fallacy; namely, on the assumption that software and programming are mechanistic phenomena, and can be studied with the principles of reductionism and atomism. Ultimately, the naive idea of software engineering is a reflection of the ignorance that the academics and the practitioners suffer from. They remain ignorant because they waste their time with worthless theories: they are forever trying to explain the phenomena of software and programming through the mechanistic myth. It is not an exaggeration to say that, for the last forty years, their main preoccupation has been this absurd search for a way to reduce software to mechanics. The preoccupation is also reflected in their vocabulary: programmers call themselves "engineers," and refer to programming as "building" or "constructing" software.

The programming theories, thus, are mechanistic delusions, because they attempt to represent complex phenomena mechanistically. What is worse, instead of being abandoned when found to be useless, they are turned by their defenders into pseudosciences. Here is how: Since neither the academics nor the practitioners are willing to admit that their latest theory has failed, they continue to praise it even as they struggle against its deficiencies. They deny the endless falsifications, and keep modifying the theory in the hope of making it practical. While described as new features, the modifications serve in fact to mask the falsifications: they reinstate the traditional, *non-mechanistic* programming concepts – precisely those concepts that the theory had attempted to eliminate. In the end, the theory's exact, mechanistic principles are forgotten altogether. Its defenders, though, continue to promote it by invoking the benefits of mechanism. Then, after perpetrating this fraud for a number of years, another mechanistic theory is invented and the same process is repeated.

So the software workers are not the serious professionals they appear to be, but impostors. Whether they are academics who invent mechanistic theories, or software companies that create systems based on these theories, or programmers who rely on these systems, very little of what they do is genuine. They appear to be dealing with important issues, but most of these issues are senseless preoccupations engendered by their mechanistic delusions: since our problems rarely have simple, mechanistic answers, there is no limit to the specious activities that one can contrive when attempting to solve them mechanistically.

The mechanistic software ideology, thus, is the perfect medium for incompetents and charlatans, as it permits them to engage in modern, glamorous, and profitable activities while doing almost nothing useful. The software practitioners have become a powerful bureaucracy, exploiting society while appearing to serve it. Less than 10 percent (and often less than 1 percent) of their work has any value. Their main objective is not to help us solve our problems through software, but on the contrary, to create new, software-related problems; in other words, to make all human activities as complicated and inefficient as they have made their own, programming activities.

At the top of this bureaucracy are the software elites – the universities and the software companies. It is these elites that control, ultimately, our software-related affairs. And they do it by promoting mechanistic software concepts: since we believe in mechanism, and since their theories and systems are founded on mechanistic principles, we readily accept their elitist position. But if software mechanism is generally useless, their theories and systems are fraudulent, and their elitist position is unwarranted.

3

Three ingredients are needed to implement totalitarianism: a myth, an elite, and a bureaucracy. And the spread of totalitarianism is caused by an expansion of the bureaucracy: larger and larger portions of the population change from their role as citizens, or workers, to the role of bureaucrats; that is, from individuals who perform useful tasks to individuals whose chief responsibility is to practise the myth.

A characteristic of totalitarianism, thus, is this continuous increase in the number of people whose beliefs and acts are a reflection of the myth. Rather than relying on common sense, or logic, or some personal or professional values, people justify their activities by invoking the myth. Or, they justify them by pointing to certain ideas or theories, or to other activities; but if these in their turn can only be justified by invoking the myth, the original activities are specious.

A totalitarian bureaucracy can be seen as a pyramid that expands downward, at its base. The elite, which forms its apex, uses the myth to establish the system's ideology and to recruit the first bureaucrats – the first layer of the pyramid. Further layers are then added, and the pyramid becomes increasingly broad and deep, as more and more categories of people cease living a normal life and join the bureaucracy. Thus, as the pyramid expands, fewer and fewer people are left who perform useful activities; and the closer an individual is to the top of the pyramid, the greater the number of senseless, myth-related preoccupations that make up his life.

Since the lower layers support the higher ones, the model of a pyramid also explains how social power is distributed under totalitarianism: each layer exploits the layers that lie below it, and the elite, at the top of the pyramid, exploits the entire bureaucracy. Thus, the closer we get to the top, the more power, influence, and privileges we find. In addition, the bureaucracy as a whole exploits the rest of society – those individuals and institutions that have not yet joined it.

The totalitarian ideal is that *all* people in society join the bureaucracy and restrict themselves to myth-related activities. But this, clearly, cannot happen; for, who would support them all? In the initial stages of the expansion, when enough people are still engaged in useful activities, the elite and the bureaucrats can delude themselves that their ideology is working. As more and more people join the bureaucracy, however, the useful activities decline and the system becomes increasingly inefficient. Eventually, the inefficiency reaches a point where society can no longer function adequately, and collapses. It is impossible to attain the totalitarian ideal – a bureaucracy that comprises the entire society.

It should be obvious, then, why the software myth can serve as the foundation of a totalitarian ideology. Since the essence of totalitarianism is endless expansion, the ideology must be based on an idea that appeals to every individual in society. And few ideas can match software in this respect. As we will see in chapter 4, software is comparable only to language in its versatility and potency. Thus, even when employed correctly, without falling prey to mechanistic delusions, software can benefit almost anyone. But when perceived as a *mechanistic* concept, its utopian promise becomes irresistible. The promise, we saw, is that software devices can act as substitutes for knowledge, skills, and experience. So, simply by operating a software device, we will be able to perform immediately tasks that would otherwise require special talents, or many years of study and practice. The promise of the software myth, thus, exceeds even the most extravagant promises made by the old political or religious myths. Consequently, an elite can dominate and exploit society through the software myth even more effectively than the political and religious elites did through the other myths, in the past.

❖

The expansion of the software bureaucracy parallels the spread of computers; and even a brief analysis of this expansion (later in this section) will reveal the process whereby various categories of people are turned into bureaucrats. All it takes is a blind belief in the software myth – something that the elite is fostering through propaganda and indoctrination. Then, judged from the perspective of the myth, activities that are in fact illogical, or inefficient, or wasteful, are perceived as important and beneficial; and the incompetents who engage in these activities are perceived as professionals.

Ignorance, therefore, is what makes the belief in a myth, and hence the expansion of a bureaucracy, possible. An individual who took the time to develop expertise in a certain field cannot also develop irrational beliefs in the same field, as that would contradict his personal experience. Thus, in addition to its versatility and potency, it is its novelty that makes software such a good subject for myth. We allowed an elite to assume control of our software-related affairs without first giving ourselves the time to discover what is the true nature of software. And the elite saw software, not as a complex phenomenon, but as a mechanistic one; in other words, not as a phenomenon that demands the full capacity of the mind, but as one that requires only mechanistic thinking.

Because of this delusion, we have remained ignorant: we depend on software while lacking the skills to create and use software intelligently. Instead of developing software expertise, we wasted the last forty years struggling with the worthless theories and methodologies promoted by the elite. Under these conditions, the emergence of irrational beliefs was inevitable. The software myth, thus, is a consequence of our mechanistic culture and our software ignorance.

4

The first workers to be turned into software bureaucrats were the programmers themselves. From the start, the theorists assumed that programming can be reduced to some simple and repetitive acts, similar to those performed by assembly-line workers in a factory. So, they concluded, programmers do not require lengthy education, training, and practice. If we develop software applications as we build appliances, all that programmers need to know is how to follow certain methods, and how to use certain aids – methods and aids based, like those in manufacturing, on the principles of reductionism and atomism. And to improve their performance later, all we need to do is improve the methods and aids.

Thus, instead of trying to understand the true nature of software and programming, the theorists *assumed* them to be mechanistic phenomena; and the programming profession was founded upon this assumption. Using the mechanistic myth as warrant, programming expertise was redefined as expertise in the use of theories, methodologies, and development aids; in other words, expertise in the use of substitutes for expertise. So what was required of programmers from then on was not programming skills, but merely familiarity with the latest substitutes.

If expertise is the highest level attainable by human minds in a given domain, and incompetence the lowest, programmers were neither expected nor permitted to attain a level much higher than incompetence. And, as society needed more and more software, everyone was convinced that what we needed was more and more of this kind of programmers. The alternative – promoting expertise and professionalism, allowing individuals to develop the highest possible skills – was never considered.

The effects of this ideology can be seen in the large number of software failures: development projects abandoned after spending millions of dollars, critical business needs that remain unfulfilled, applications that are inadequate or unreliable, promises of increased savings or efficiency that do not materialize. Statistics unchanged since the 1970s show that less than 5 percent of programming projects result in adequate applications. What these statistics do not reveal is that even those applications that are adequate when new cannot be kept up to date (because badly written and badly maintained), so they must be replaced after a few years. The statistics also do not reveal that, with inexperienced programmers, it costs far more than necessary to create even those applications that are successful. And if we remember also the cost of the additional hardware needed to run badly written applications, it is safe to say that, for more than forty years, society has been paying in effect one hundred dollars for every dollar's worth of useful software.[1]

The conclusion ought to be that the mechanistic assumption is wrong: programming expertise is not the kind of knowledge that can be replaced with methods or devices, so personal skills and experience remain an important factor. The answer to the software failures is then simply to recognize that, as is the case in other difficult professions, to become a proficient programmer one needs many years of serious education, training, and practice.

In our mechanistic software culture, however, this idea is inadmissible; and someone who suggests it is accused of clinging to old-fashioned values, of resisting science and progress. The only accepted answer to the software failures is that we need, not better programmers, but better theories, methodologies, and development aids. If the previous ones failed, we are told, it is because they did not adhere faithfully enough to the mechanistic ideology; so the next ones must be even more mechanistic. In other words, the only permissible solutions to the problem of programming incompetence are those derived from the mechanistic myth – the same solutions that were tried in the past, and which *cause* in fact the incompetence. No matter how many failures we witness, the mechanistic ideology is never questioned.

The mechanistic software concepts cause incompetence because they are specifically intended as *substitutes* for programming expertise. Thus, it is not surprising that programmers who rely on these substitutes do not advance past the level of novices: they are *expected* to remain at this level.

So the incompetence of programmers, and the astronomic cost of software, are a direct consequence of the mechanistic myth. For the first time, a mechanistic delusion is powerful enough to affect the entire society. Previously, it was only in universities that individuals could pursue a mechanistic fantasy, in the guise of research; and the failure of their projects had little effect on the rest of society. Through software, however, the pursuit of mechanistic fantasies became possible everywhere. Unlike the mechanistic theories in

[1] Software expenses, and computing expenses generally, are usually called "technology investments," or "technology solutions," and are seen therefore as an asset (rather than a liability) regardless of their real value. This is one reason why the enormous cost of software is not obvious. Deceptive language is an important tool in the marketing of worthless products and services, as it helps ignorant decision makers to rationalize expenses. (We will see in chapter 5 how the slogan "technology" is used for this purpose.)

psychology, sociology, or linguistics, the mechanistic *software* theories are not limited to academic research. Being applicable to business computing, they spread throughout society, and degraded the notions of expertise and responsibility in business just as mechanistic research had degraded these notions in universities. Just as the academics perceive their responsibility to be, not the discovery of useful theories but the pursuit of mechanistic ideas, programmers perceive their responsibility to be, not the creation of useful applications but the use of mechanistic software methods.

Millions of individuals are engaged, thus, not in programming but in the pursuit of mechanistic fantasies. Probably no more than 1 percent of the programming activities in society represent useful work; that is, work benefiting society in the way the work of doctors does. We find ourselves today in this incredible situation because programming is a new profession, without established standards of expertise. We allowed the software elite to persuade us that this profession must be based on mechanistic principles, so the standard of expertise became, simply, expertise in mechanistic software concepts. Had we tried first the alternative – giving programmers the time and opportunity to develop the highest knowledge and skills that human beings can attain in this new profession – we would easily recognize the absurdity of the mechanistic concepts, and the incompetence of those who restrict themselves to such concepts. It is only because we take software mechanism as unquestionable truth that we accept the current programming practices as a normal level of expertise. And if we consider this level normal, it is natural to accept also the resulting cost and the failures.

Also, with so many programmers around, new types of supervisors had to be created: more and more employees were turned into software bureaucrats – project managers, systems analysts, database administrators – to oversee the hordes of programmers who, everyone agreed, could not be trusted to develop applications on their own. Again, no one questioned this logic. If the programmers were deemed incompetent and irresponsible, the answer should have been to improve their training. Instead, it was decided to adopt, for software development, the assembly-line methods used in manufacturing; namely, to treat programmers as unskilled workers, and to develop applications by relying on management expertise rather than programming expertise.

So for every few programmers there was now a manager, and for every few managers a higher manager. But the manufacturing methods are inadequate for programming, because software applications are not neat hierarchical structures of subassemblies. Consequently, turning software development into factory-type work did not solve the problem of programming incompetence. It only increased the software bureaucracy, and hence the cost of software, and the failures. (Sociological studies of the programming profession, conducted in the 1970s, show that the main goal of corporate management was not so much to improve programming practices, as to repress the programmers' attitudes and expectations. For example, the theory of structured programming was promoted as the means to turn programming into an exact activity, and programmers into skilled professionals, while its true purpose was the opposite: to *deskill* programmers; specifically, to eliminate the need and opportunity for programmers to make important decisions, and to give management complete control over their work.[2])

Finally, as the benefits expected from mechanistic software concepts are not materializing, new types of bureaucrats must be constantly invented as a solution to the incompetence of programmers. Thus, companies have now employees with absurd titles like architect, systems integrator, data analyst, business intelligence analyst, and report developer. While justified by invoking the growing complexity of business computing, and the growing importance of information technology, the task of these new bureaucrats is in reality to do what programmers should be doing; that is, create and maintain business applications. What masks this fact is that, instead of programming, they try to accomplish the same thing through various end-user tools, or by putting together ready-made pieces of software. But the idea that we can create useful applications in this fashion is based on the same delusions as the idea that programming expertise can be replaced with methods and aids. So it only adds to the complexity of business computing, while the real problems remain unsolved. This is interpreted, though, as a need for even more of the new bureaucrats, in a process that feeds on itself.

❖

A major role in the spread of the software bureaucracy is played by the organizations that *create* the knowledge substitutes – the software companies. These companies form the elite, of course. But in addition to propagating the mechanistic software ideology, they function as employers; and in this capacity, they are turning millions of additional workers into software bureaucrats.

[2] See Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer-Verlag, 1977); see also Joan M. Greenbaum, *In the Name of Efficiency: Management Theory and Shopfloor Practice in Data-Processing Work* (Philadelphia: Temple University Press, 1979). It must be noted, though, that, while ground-breaking and as important today as they were in the 1970s, these studies treat the deskilling of programmers as part of the traditional conflict between management and labour. Their authors were unaware of the fallacies of software mechanism, and that theories like structured programming do not, in fact, work. Thus, the delusion that programmers must be a kind of factory workers – because programming is a kind of manufacturing – constitutes a sociological phenomenon that has yet to be studied.

From just a handful in the 1960s, the software companies have grown in number and in size to become an important part of the economy. And they accomplished this simply by invoking the myth of software mechanism. For, their software and services can be justified only if we accept unquestioningly the mechanistic ideology. Thus, only if we agree that software development is a form of manufacturing will we accept the resulting incompetence, and hence the aids and substitutes supplied by these companies as the answer. Or, putting this in reverse, if we had professional programmers instead of the current practitioners, less than 1 percent of the software supplied by these companies would be needed at all.

What this means is that countless organizations, while operating as legitimate businesses under the banner "technology," are actually engaged in the making and marketing of mechanistic software fantasies. So their employees, no matter how good they may be in these activities – programming, research, management, administration, selling – are not performing work that is truly useful. They belong, therefore, to the software bureaucracy.

The programmers who work for these companies hold a special place in the bureaucracy. They are, in general, better prepared and more experienced than the application programmers. But if their job is to develop the useless systems sold by the software companies, their talents are wasted. These systems may appear impressive to their users, but they cannot replace good applications, nor the expertise needed to create good applications. So, if these systems cannot be a substitute for expertise, the work of those who create them is just as senseless as the work of those who use them. We are witnessing, therefore, this absurd situation: our better programmers are employed to create, not the custom applications that society needs, but some generic applications, or some substitutes for the knowledge required to create custom applications. Instead of helping to eradicate the software bureaucracy, our universities prepare programmers for the software companies, thereby *adding* to the bureaucracy. For, by catering to the needs of software bureaucrats, the system programmers are reduced to bureaucrats themselves.

❖

A different kind of software companies are the enterprises run by the individuals known as industry experts, or gurus. Unlike the regular software companies, the gurus earn their fame personally – as theorists, lecturers, and writers. Their role, however, is similar: to promote the ideology of software mechanism. So they are part of the elite. Also like the software companies, their existence is predicated on widespread programming incompetence and an ever-growing bureaucracy.

Although they seldom have any real programming experience (that is, personally creating and maintaining serious business applications), the gurus confidently write papers and books on programming, publish newsletters, invent theories and methodologies, lecture, teach courses, and provide consulting services. Their popularity – the fact that programmers, analysts, and managers seek their advice – demonstrates, thus, the ignorance that pervades the world of programming. To appreciate the absurdity of this situation, imagine a similar situation in medicine: individuals known to have no medical training, and who never performed any surgery, would write and lecture on operating procedures; and real surgeons, from real hospitals, would read their books, attend their courses, and follow their methods.

While unthinkable in other professions, we accept this situation as a logical part of our *programming* culture. The reason it seems logical is that it can be justified by pointing to the software myth: if what we perceive as programming expertise is familiarity with theories, methodologies, and software devices, it is only natural to respect, and to seek the advice of, those who know the most in this area. So the gurus are popular because they always promote the latest programming fads – which, at any given time, are what ignorant practitioners believe to be the cure for their current difficulties.

❖

Programming was only the first profession to be destroyed by the software myth. Once we agreed to treat programmers as mere bureaucrats, instead of insisting that they become proficient and responsible workers, the spread of the software bureaucracy was inevitable. Every aspect of the degradation that is currently occurring in other professions can be traced to the incompetence of programmers. For, as we increasingly depend on computers and need more and more software applications, if the programmers are unreliable we must find other means to develop these applications. We already saw how new types of managers, and new types of software workers, were invented to deal with the problem of programming incompetence. This did not help, however. So the problem spread beyond the data-processing departments, and is now affecting the activities of software *users*.

Little by little, to help users perform the work that should have been performed by programmers, various software aids have been introduced. They vary from simple programming environments derived from database or spreadsheet systems (which promise users the power to implement their own applications) to ready-made applications (which promise users the power to eliminate programming altogether). These aids, however, are grounded on the same mechanistic principles as the development aids offered to programmers, so they suffer from the same fallacies. If the substitutes for expertise cannot help programmers, we can hardly expect them to help amateurs, to create useful applications.

Workers everywhere, thus, are spending more and more of their time doing what only programmers had been doing before: pursuing mechanistic software fantasies. Increasingly, those who depend on computers must modify the way they work so as to fit within the mechanistic software ideology: they must depend on the inferior applications developed by inexperienced programmers, or on the childish applications they are developing themselves, or on the generic, inadequate applications supplied by software companies. The world of business is being degraded to match the world of programming: other workers are becoming as inefficient in their occupations as programmers are in theirs; like the programmers, they are wasting more and more of their time dealing with specious, software-related problems.

But we perceive this as a normal state of affairs, as an inevitable evolution of office work and business management. Because we believe that the only way to benefit from software is through the mechanistic ideology, we are now happy to adopt this ideology in our own work. As software users, we forget that the very reason we are preoccupied with software problems instead of our real problems is the incompetence and inefficiency caused by the mechanistic ideology in programming. So, by adopting the same ideology, we end up replicating the incompetence and inefficiency in other types of work. In other words, we become software bureaucrats ourselves.

❖

Thus, because we do not have a true programming profession, workers with no knowledge of programming, or computers, or engineering, or science are increasingly involved in the design and creation of software applications. And, lacking the necessary skills, they are turning to the knowledge substitutes offered by the software companies – which substitutes address now all people, not just programmers. So, as millions of amateurs are joining the millions of inexperienced practitioners, the field of application development is becoming very similar to the field of consumer goods. A vast network of distribution and retail was set up to serve these software consumers, and a comprehensive system of public relations, marketing, and advertising has emerged to promote the knowledge substitutes: books, periodicals, brochures, catalogues, newsletters, trade shows, conventions, courses, seminars, and online sources.

The similarity to consumer goods is clearly seen in the editorial and advertising styles: childish publication covers; abundance of inane terms like “powerful,” “easily,” “solution,” and “technology”; the use of testimonials to demonstrate the benefits of a product; prices like $99.99; and so on. Thus, while discussing programming, business, efficiency, or productivity, the promotion of the software devices resembles the promotion of cosmetics, fitness gadgets, or money-making schemes. Also similar to consumer advertising are the deceptive claims; in particular, promising ignorant people the ability to perform a difficult task simply by buying something. The software market, thus, is now about the same as the traditional consumer market: charlatans selling useless things to dupes.

Again, to appreciate the absurdity of this situation, all we have to do is compare the field of programming with a field like medicine. There is no equivalent, in medicine, of this transformation of a difficult profession into a consumer market. We don't find any advertisers or retailers offering knowledge substitutes to lay people and inexperienced practitioners who are asked to replace the professionals.

This transformation, then, has forced countless additional workers to join the software bureaucracy. For, if what they help to sell is based on the idea that software devices can replace expertise, and if this idea stems from the belief in software mechanism, all those involved in marketing the knowledge substitutes are engaged in senseless activities.

❖

Finally, let us recall that it is precisely those institutions which ought to encourage rationality – our universities – that beget the software delusions. Because they teach and promote only *mechanistic* software concepts, the universities are, ultimately, responsible for the widespread programming incompetence and the resulting corruption.

In the same category are the many associations and institutes that represent the world of programming. The ACM and the IEEE Computer Society, in particular – the oldest and most important – are not at all the scientific and educational organizations they appear to be. For, while promoting professionalism in the use of computers, and excellence in programming, their idea of professionalism and excellence is simply adherence to the mechanistic ideology. Thus, because they advocate the same concepts as the universities and the software companies, these organizations serve the interests of the elite, not society.

If this sounds improbable, consider their record: They praise every programming novelty, without seriously verifying it or confirming its usefulness. At any given time, they proselytize the latest programming "revolution," urging practitioners to join it: being familiar with the current software concepts, they tell us, is essential for advancement. In particular, they endorsed the three pseudoscientific theories we examine in chapter 7, and conferred awards on scientists who upheld them. As we will see, not only are these theories fallacious and worthless, but the scientists used dishonest means to defend them; for example, they claimed that the theories benefit from the rigour and precision of mathematics, while this is easily shown to be untrue. Thus, instead of exposing the software frauds, the ACM and the IEEE Computer Society help to propagate them.

What these organizations are saying, then, is exactly what every software guru and every software company is saying. So, if they promote the same values as the *commercial* enterprises, they are not responsible organizations. Like the universities, their aim is not science and education, but propaganda and indoctrination. They may be sincere when using terms like "professionalism" and "expertise," but if they equate these terms with software mechanism, what they do in reality is turn programmers into bureaucrats, and help the elite to exploit society.

5

The foregoing analysis has shown that our mechanistic software culture is indeed a social phenomenon that is causing the spread of a bureaucracy, and hence the spread of totalitarianism. Every one of the activities we analyzed can be justified only through the software myth – or through another activity, which in its turn can be justified only through the myth or through another activity, and so on. The programmers, the managers, the academics, the gurus, the publishers, the advertisers, the retailers, the employees of software companies, and increasingly every computer user – their software-related activities seem logical only if we blindly accept the myth. As soon as we question the myth, we recognize these activities as what they actually are: the pursuit of mechanistic fantasies.

So the expansion of software-related activities that we are witnessing is not the expansion of some useful preoccupations, but the expansion of delusions. It is not a process of collective progress in a new field of knowledge – what our software-related affairs *should* have been – but a process of degradation: more and more people are shifting their attention from their former, serious concerns, to some senseless pursuits.

In chapter 8 we will study the link between the mechanistic ideology and the notion of individual responsibility; and we will see that a mechanistic culture leads inevitably to a society where people are no longer considered responsible for their acts. The road from mechanism to irresponsibility is short. The belief in mechanism tempts us to neglect the natural capabilities of our minds, and to rely instead on inferior substitutes: rather than acquiring knowledge, we acquire devices that promise to replace the need for knowledge. We accomplish by means of devices less than we could with our own minds, and the devices may even be wrong or harmful, but no one blames us. Our responsibility, everyone agrees, is limited to knowing how to operate the devices.

Today, the incompetence and irresponsibility are obvious in our software-related activities, because these activities are dominated by mechanistic beliefs. But if we continue to embrace software mechanism, we should expect the incompetence and irresponsibility to spread to other fields of knowledge, and to other professions, as our dependence on computers is growing.

A society where all activities are as inefficient as are our software-related activities cannot actually exist. We can afford perhaps to have a few million people engaged in mechanistic fantasies, in the same way that we can afford to have an entertainment industry, and to spend a portion of our time with idle amusements. But we cannot, all of us, devote ourselves to the pursuit of fantasies. Thus, if the spread of software mechanism is causing an ever-growing number of people to cease performing useful work and to pursue fantasies instead, it is safe to predict that, at some point in the future, our society will collapse.

To avert this, we must learn all we can from the past: we must study the harm that has *already* been caused by software mechanism, in the domain of programming. In programming we have been trying for more than forty years to find substitutes for expertise, so we have enough evidence to *demonstrate* the absurdity of this idea, and the dishonesty of those who advocate it.

Despite its failure in programming, it is the same idea – replacing minds with software – that is now being promoted in other domains. And it is the same myth, software mechanism, that is invoked as justification, and the same elites that are perpetrating the fraud. So, in a few years, we should expect to see in other domains the same corruption we see today in programming, the same incompetence and irresponsibility. One by one, all workers will be reduced, as programmers have been, to software bureaucrats. As it has been in programming, the notion of expertise will be redefined everywhere to mean expertise in the use of substitutes for expertise. As programmers are today, we will all be restricted to the methods and devices supplied by an elite, and prevented from developing our minds.

Thus, if we understand how the mechanistic delusions have caused the incompetence and irresponsibility found today in the domain of programming, we will be able perhaps to prevent the spread of these delusions, and the resulting corruption, in other domains.