SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*
Sections *Introduction, The Fallacy of Software Engineering, Software Engineering as Pseudoscience*

This extract includes the book's front matter and part of chapter 7.

These sections include a brief, non-technical history and analysis of the idea of software engineering and its mechanistic fallacies.

The entire book, each chapter separately, and also selected sections, can be viewed and downloaded at the book's website.

www.softwareandmind.com

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

Published by Andsor Books, Toronto, Canada (January 2013)
www.andsorbooks.com

For disclaimers see pp. vii, xv–xvi.

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

Printed on acid-free paper.

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 xiii

Introduction Belief and Software 1
Modern Myths 2
The Mechanistic Myth 8
The Software Myth 26
Anthropology and Software 42
Software Magic 42
Software Power 57

Chapter 1 Mechanism and Mechanistic Delusions 68
The Mechanistic Philosophy 68
Reductionism and Atomism 73
Simple Structures 92
Complex Structures 98
Abstraction and Reification 113
Scientism 127

Chapter 2 The Mind 142
Mind Mechanism 143
Models of Mind 147

Tacit Knowledge 157
Creativity 172
Replacing Minds with Software 190

Chapter 3 Pseudoscience 202
The Problem of Pseudoscience 203
Popper's Principles of Demarcation 208
The New Pseudosciences 233
The Mechanistic Roots 233
Behaviourism 235
Structuralism 242
Universal Grammar 251
Consequences 273
Academic Corruption 273
The Traditional Theories 277
The Software Theories 286

Chapter 4 Language and Software 298
The Common Fallacies 299
The Search for the Perfect Language 306
Wittgenstein and Software 328
Software Structures 347

Chapter 5 Language as Weapon 368
Mechanistic Communication 368
The Practice of Deceit 371
The Slogan "Technology" 385
Orwell's Newspeak 398

Chapter 6 Software as Weapon 408
A New Form of Domination 409
The Risks of Software Dependence 409
The Prevention of Expertise 413
The Lure of Software Expedients 421
Software Charlatanism 440
The Delusion of High Levels 440
The Delusion of Methodologies 470
The Spread of Software Mechanism 483

Chapter 7 Software Engineering 492
Introduction 492
The Fallacy of Software Engineering 494
Software Engineering as Pseudoscience 508

Structured Programming 515
The Theory 517
The Promise 529
The Contradictions 537
The First Delusion 550
The Second Delusion 552
The Third Delusion 562
The Fourth Delusion 580
The *GO TO* Delusion 600
The Legacy 625
Object-Oriented Programming 628
The Quest for Higher Levels 628
The Promise 630
The Theory 636
The Contradictions 640
The First Delusion 651
The Second Delusion 653
The Third Delusion 655
The Fourth Delusion 657
The Fifth Delusion 662
The Final Degradation 669
The Relational Database Model 676
The Promise 677
The Basic File Operations 686
The Lost Integration 701
The Theory 707
The Contradictions 721
The First Delusion 728
The Second Delusion 742
The Third Delusion 783
The Verdict 815

Chapter 8 From Mechanism to Totalitarianism 818
The End of Responsibility 818
Software Irresponsibility 818
Determinism versus Responsibility 823
Totalitarian Democracy 843
The Totalitarian Elites 843
Talmon's Model of Totalitarianism 848
Orwell's Model of Totalitarianism 858
Software Totalitarianism 866

Index 877

Preface

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. This process started three centuries ago, is increasingly corrupting us, and may well destroy us in the future. The book discusses all stages of this degradation.

The book's epigraph, about Newspeak, will become clear when we discuss the similarity of language and software (see, for example, pp. 411–413).

Throughout the book, the software-related arguments are also supported with ideas from other disciplines – from philosophy, 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. Often, the connection between the traditional issues and the software issues is immediately apparent; but sometimes its full extent can be appreciated only in the following sections or chapters. If tempted to skip these discussions, remember that our software delusions can be recognized only when investigating the software practices from this broader perspective.

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. 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.

There is some repetitiveness in the book, deliberately introduced in order to make the individual chapters, and even the individual sections, reasonably independent. Thus, while the book is intended to be read from the beginning, you can select almost any portion and still follow the discussion. An additional benefit of the repetitions is that they help to explain the more complex issues, by presenting the same ideas from different perspectives or in different contexts.

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 usually 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); and the plural, "elites," is used when referring to several entities, or groups of entities, within such a body. Thus, although both forms refer to the same entities, the singular is employed when it is important to stress the existence of the whole body, and the plural when it is the existence of the individual entities that must be stressed. The plural is also employed, occasionally, in its normal sense – a group of several different bodies. Again, the meaning is clear from the context.

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 plan to publish, in source form, some of the software applications 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.

Introduction

My task in this chapter is to show that the body of theories and activities known as software engineering forms in reality a system of belief, a pseudoscience. This discussion is in many ways a synthesis of everything we learned in the previous chapters: the model of simple and complex structures, the two mechanistic fallacies, the nature of software and programming, the structures that make up software applications, the mechanistic conception of mind and software, the similarity of software and language, the principles of demarcation between science and pseudoscience, the incompetence of the software practitioners, and the corruption of the software elite. There are brief summaries here, but bear in mind that a good understanding of these topics is a prerequisite for appreciating the present argument, and its significance.

In chapter 6 we examined the three stages in the spread of mechanistic software concepts: the domain of programming, the world of business, and our personal affairs (see pp. 486–491). And we saw that, while the first stage is now complete, the others are still unfolding. Judged from this perspective, the present chapter can also be seen as a study of the *first* stage. Since this stage involves events that took place in the past, its study can be exact and

objective. We can perhaps still delude ourselves about the benefits of software mechanism in our offices or in our homes, but we cannot in the domain of programming; for, we can *demonstrate* the absurdity of the mechanistic theories, and the resulting incompetence and corruption.

To perform a similar study for the other two stages, we would have to wait a few decades, until they too were complete. But then, it would be too late: if we want to prevent the spread of software mechanism in other domains, we must act now, by applying the lessons of the first stage.

The similarities of the three stages are not accidental. It is, after all, the same elite that is controlling them, and the same software concepts that are being promoted. Common to all stages is the promise to replace human minds with software: with the methods and systems supplied by an authority. And this plan is futile, because mechanistic concepts can replace only the *simple* aspects of human intelligence. The plan, thus, has little to do with enhancing our capabilities. It is in reality a new form of domination, made possible by our mechanistic delusions and our increasing dependence on computers.

As we read the present chapter, then, we must do more than just recognize how the mechanistic ideology has destroyed the programming profession. We must try to project this phenomenon onto other fields and occupations, and to imagine what will happen when all of us are reduced, as programmers have been, to mere bureaucrats.

The programming theories have not eliminated the need for programming expertise. All they have accomplished is to *prevent* programmers from developing this expertise, thereby making software development more complicated, more expensive, and dependent on the software elite instead of individual minds. Similarly, the software concepts promoted now for our offices and for our homes serve only to prevent us from developing knowledge and skills, and to increase our dependence on the software elite. What we note is an attempt to reduce all human activities to the simple acts required to operate software devices. But this is an impossible quest. So, like the programmers, we will end up with nothing – neither the promised expedients, nor the expertise to perform those activities on our own.

At that point, society will collapse. A society dominated by a software elite and a software bureaucracy can exist only because the rest of us are willing to support them. It is impossible, however, for *all* of us to be as incompetent and inefficient in our pursuits as the programmers are now in theirs. For, who would support the entire society?

The Fallacy of Software Engineering

1

The term *software engineering* was first used in the late 1960s. It expresses the view that, in order to be as successful in our programming activities as we are in our engineering activities, we must emulate the methods of the engineering disciplines. This view was a response to what became known as the software crisis: the realization that the available programming skills could not keep up with the growing demand for software, that application development took too long, and that most applications were never completed, or were inadequate, or were impossible to keep up to date.

Clearly, the experts said, a new programming philosophy is needed. They likened the programmers of that era to the old craftsmen, or artisans, whose knowledge and skills were not grounded on scientific principles but were the result of personal experience. Thus, concluded the experts, just as the traditional fields have advanced since modern engineering principles replaced the personal skills of craftsmen, the new field of software will advance if we replace personal programming skills with the software equivalent of the engineering principles.

So for the last forty years, the imminent transition from software art to software engineering has been the excuse for every new theory, methodology, development environment, and database system. Here are just a few out of the thousands of statements proclaiming this transition: "Software is applying for full membership in the engineering community. Software has grown in application breath and technical complexity to the point where it requires more than handcrafted practices."[1] "Software development has often been viewed as a highly individualistic art.... The evolution of software engineering in the 1970s and 1980s came from the realization that software development is better viewed as an engineering task ...."[2] "Software engineering is not alone among the engineering disciplines, but it is the youngster. We can learn a great deal by studying the history of other engineering disciplines."[3] "Software development currently is a craft.... Software manufacturing involves transferring the twin

[1] Walter J. Utz Jr., *Software Technology Transitions: Making the Transition to Software Engineering* (Englewood Cliffs, NJ: Prentice Hall, 1992), p. xvii.

[2] Ed Seidewitz and Mike Stark, *Reliable Object-Oriented Software: Applying Analysis and Design* (New York: SIGS Books, 1995), p. 4.

[3] Gerald M. Weinberg, *Quality Software Management*, vol. 1, *Systems Thinking* (New York: Dorset House, 1992), p. 295.

disciplines of standard parts and automated manufacture from industrial manufacturing to software development."[4] "We must move to an era when developers design software in the way that electronic engineers design machines."[5] "Software engineering is modeled on the time-proven techniques, methods, and controls associated with hardware development."[6] "Calling programmers 'software engineers' emphasizes the parallel between developing computer programs and developing mechanical or electronic systems. Many practices that have long been associated with engineering … have increasingly been adopted by data processing professionals."[7] "We as practitioners must change. We must change from highly skilled artisans to being software manufacturing engineers."[8] "We now have tools and techniques that enable us to do true software engineering.… With these tools we can build software factories.… We have, working today, the basis for grand-scale engineering of software."[9]

The first thing we note in the idea of software engineering is its circularity. Before formulating programming theories based on engineering principles, we ought to determine whether software can indeed be developed with the methods we use to build cars and appliances. There are many human activities, after all, for which these methods are known to be inadequate. In chapter 2 we saw that, from displaying ordinary behaviour to practising a difficult profession, our acts are largely *intuitive*: we use unspecifiable knowledge and skills, rather than exact methods. This is true because most phenomena we face are complex; and for complex phenomena, our natural, non-mechanistic mental capabilities *exceed* the exact principles of science and engineering. Thus, whether this new human activity – programming – belongs to one category or the other is what needs to be determined. When the software theorists *start* their argument by claiming that programming must be practised

[4] Stephen G. Schur, *The Database Factory: Active Database for Enterprise Computing* (New York: John Wiley and Sons, 1994), p. 9.

[5] James Martin, *Principles of Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: PTR Prentice Hall, 1993), p. 40.

[6] Roger S. Pressman, *Software Engineering: A Practitioner's Approach* (New York: McGraw-Hill, 1982), p. 15.

[7] L. Wayne Horn and Gary M. Gleason, *Advanced Structured COBOL: Batch and Interactive* (Boston: Boyd and Fraser, 1985), pp. 2–3.

[8] Sally Shlaer, "A Vision," in *Wisdom of the Gurus: A Vision for Object Technology*, ed. Charles F. Bowman (New York: SIGS Books, 1996), p. 222.

[9] James Martin, *An Information Systems Manifesto* (Englewood Cliffs, NJ: Prentice Hall, 1984), p. 37.

as an engineering activity, they start by assuming the very fact which they are supposed to prove.

While evident in each one of the foregoing quotations, the circularity is even better illustrated by the following passage: "This book is written with the firm belief that software development is a science, not an art, and should be managed as any other engineering project. For our purposes we will define 'software engineering' as the practical application of engineering principles and methods …."[10] The author, thus, starts by admitting that the idea of software engineering is based on a *belief*. Then, he adds that software development should be managed as "any other" engineering project; so he treats as *established fact* the belief that it is a form of engineering. Finally, he *defines* software engineering as a form of engineering, as if the preceding statements had demonstrated this relationship.

Here is another example of this question-begging logic: "In this section we delineate software engineering and the software engineer.… The first step in the delineation is to establish a definition of software engineering – based upon the *premise* that software engineering is engineering – that will serve as a framework upon which we can describe the software engineer."[11] The authors, thus, candidly admit that they are *assuming* that fact which they are supposed to determine; namely, that software development is a form of engineering. Then, after citing a number of prior definitions that claim the same thing (also without proof), and after pointing out that there are actually some important differences between programming and the work of the engineer, the authors conclude: "Software engineering, in spite of the abstract nature and complexity of the product, is *obviously* a major branch of engineering."[12] The word "obviously" is conspicuously out of place, seeing that there is nothing in the two pages between the first and second quotation to prove that software development is a form of engineering.

This fallacy – defining a concept in terms of the concept itself – is known as *circular definition*. Logically, the theorists ought to start by investigating the nature of programming, and to adopt the term "software engineering" only after determining that this activity is indeed a form of engineering. They start, however, with the *wish* that programming be like engineering, and their definition ends up reflecting this wish rather than reality. Invariably, the theorists *start* by calling the activity "software engineering," and *then* set out searching for an explanation of this activity! With such question-begging reasoning, their conclusion that software development is a form of engineering

[10] Ray Turner, *Software Engineering Methodology* (Reston, VA: Reston, 1984), p. 2.

[11] Randall W. Jensen and Charles C. Tonies, "Introduction," in *Software Engineering*, eds. Randall W. Jensen and Charles C. Tonies (Englewood Cliffs, NJ: Prentice Hall, 1979), p. 9 (italics added). [12] Ibid., p. 11 (italics added).

is not surprising. Nor is it surprising that the same experts who promote the idea of software engineering also promote absurd theories like structured programming or object-oriented programming: we can hardly expect individuals who fall victim to an elementary logical fallacy to invent sensible theories.

❖

The second thing we note in the idea of software engineering is a distortion of facts. When the theorists liken the current programmers to the old craftsmen, they misrepresent both the spirit and the tradition of craftsmanship. The craftsmen were highly skilled individuals. They developed their knowledge over many years – years of arduous training as apprentices, followed by years of practice as journeymen, and further experience as masters. The craftsmen were true experts, in that they knew everything that could be known in their time in a particular field. Another way to describe their expertise is by saying that they were expected to attain the highest level of proficiency that human minds can attain in a given domain.

When likening programmers to craftsmen, the software theorists imply that the knowledge and experience that programmers have in their domain is similar to the knowledge and experience that craftsmen had in theirs; they imply that programmers know everything that can be known in the domain of software, that they have attained the utmost that human minds can attain in the art of programming. But is this true?

Let us recall what kind of "craftsman" was the programmer of the 1960s and 1970s – the period when this comparison was first enunciated. The typical worker employed by a company to develop software applications had no knowledge whatever of computers, or electronics, or engineering, and only high-school knowledge of such software-related subjects as science, logic, and mathematics. Nor was he required to have any knowledge of accounting, or manufacturing, or any other field related to business computing. Most of these individuals drifted into programming, as a matter of fact, precisely because they had no skills, so they could find no other job. Moreover, to become programmers, all they had to do was attend an introductory course, measured in *weeks*. (In contrast, the training of engineers, nurses, librarians, social workers, etc., took years. So, compared with other occupations, programmers knew nothing. Programming was treated, thus, not as a profession, but as *unskilled labour*. This attitude never changed, as we will see throughout the present chapter. Despite the engineering rhetoric, programmers are perceived as the counterpart, not of engineers, but of assembly-line workers.)

Not only did programmers lack any real knowledge, but they were prevented from gaining any real experience. Their work was restricted to trivial

programming tasks – to small and isolated pieces of an application – and no one expected them to ever create and maintain whole business systems. After a year or two of this type of work, they were considered too skilled to program, so they were promoted to the position of systems analyst, or project manager, or some other function that involved little or no programming. Because it was deemed that their performance was the highest level attainable by an average person, many such positions were invented in an attempt to turn the challenge of software development from a reliance on programming skills to a reliance on management skills; that is, an attempt to create and maintain software applications through a large organization of incompetents, instead of a small number of professionals.[13]

From the beginning, then, the programming career was seen, not as a lifelong plan – a progression from apprentice to master, from novice to expert – but as a brief acquaintance with programming on the way to some other career. Programmers were neither expected nor permitted to expand their knowledge, or to perform increasingly demanding tasks. Since it was assumed that dealing with small and isolated programming problems represents the highest skill needed, and since almost anyone could acquire this skill in a few months, being a programmer much longer was taken as a sign of failure: that person, it was concluded, could not advance past the lowly position of programmer. The programming career ended, in effect, before it even started. Programming became one of those dubious occupations for which the measure of success is how soon the practitioner ceases to practise it. Thus, for a programmer, the measure was how soon he was promoted to a position that did *not* involve programming.

The notion of craftsmanship entailed, of course, more than just knowledge and experience. It was the craftsman's devotion to his vocation, his professional pride, and a profound sense of responsibility, that were just as important for his success. By perceiving programming as a brief phase on their way to some other occupation, it was impossible for programmers to develop the same qualities. Thus, even more than the lack of adequate knowledge and experience, it is the lack of these qualities that became the chief characteristic of our programming culture.

[13] While the whole world was mesmerized by the software propaganda, which was portraying programmers as talented professionals, the few sociologists who conducted their own research on this subject had no difficulty discovering the reality: the systematic deskilling of programmers and the bureaucratization of this profession. The following two works stand out (see also the related discussion and note 2 in "The Software Myth" in the introductory chapter, pp. 34–35): Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer-Verlag, 1977); Joan M. Greenbaum, *In the Name of Efficiency: Management Theory and Shopfloor Practice in Data-Processing Work* (Philadelphia: Temple University Press, 1979).

So the occupation of programming became a haven for mediocre individuals, and for individuals with a bureaucratic mind. Someone who previously could do nothing useful could now hold a glamorous and well-paid position after just a few weeks of training – a position, moreover, demanding only the mechanical production of a few lines of software per day. Actually, it soon became irrelevant whether these lines worked at all, since the inadequacy of applications was accepted as a normal state of affairs. All that was required of programmers, in reality, was to conform to the prescripts laid down by the software elite. It is not too much to say that most business applications have been created by individuals who are not programmers at all – individuals who are not even apprentices, because they are not *preparing* to become programmers, but, on the contrary, are looking forward to the day when they will no longer have to program.

In conclusion, if we were to define the typical programmer, we could describe him or her as the exact opposite of a craftsman. Since the notion of craftsmanship is well understood, the software theorists must have been aware of this contradiction when they formulated their idea of software engineering. Everyone could see that programmers had no real knowledge or experience – and, besides, were not *expected* to improve – while the craftsmen attained the utmost knowledge and experience that an individual could attain. So why did the theorists liken programmers to craftsmen? Why did they base the idea of software engineering on a transition from craftsmanship to engineering, when it was obvious that programmers were not at all like the old craftsmen?

The answer is that the theorists held the principles of software mechanism as unquestionable truth. They noticed that the programming practices were both non-mechanistic and unsatisfactory, and concluded that the only way to improve them was by making them mechanistic. This question-begging logic prevented them from noticing that they were making contradictory observations; namely, that programmers were incompetent, and that they were like the old craftsmen. Both observations seemed to suggest the idea of software engineering as solution, when in fact the theorists had accepted that idea implicitly to begin with. The alternative solution – a culture where programmers can become *true* software craftsmen – was never considered.

Barry Boehm,[14] in a paper considered a landmark in the history of software engineering, manages to avoid the comparison of programmers to craftsmen only by following an even more absurd line of logic. He notes the mediocrity of programmers, and concludes that the purpose of software engineering must

[14] Barry W. Boehm, "Software Engineering," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE) – paper originally published in *IEEE Transactions on Computers* C-25, no. 12 (1976): 1226–1241.

be, not to create a body of skilled and responsible professionals, but on the contrary, to develop techniques whereby even incompetent workers can create useful software: "For example, a recent survey of 14 installations in one large organization produced the following profile of its 'average coder': 2 years college-level education, 2 years software experience, familiarity with 2 programming languages and 2 applications, and generally introverted, sloppy, inflexible, 'in over his head,' and undermanaged. Given the continuing increase in demand for software personnel, one should not assume that this typical profile will improve much. This has strong implications for effective software engineering technology which, like effective software, must be well-matched to the people who must use it."[15]

Boehm, evidently, doesn't think that we ought to determine first whether programming is, in fact, the kind of skill that can be replaced with hordes of incompetents trained to follow some simple methods. Note also his idea of what "effective" software generally must do: not help people to develop their minds, but keep them at their current, mediocre level. This remark betrays the paternalism characteristic of the software elite: human beings are seen strictly as operators of software devices – devices which they, the experts, are going to design. Thus, the "easy-to-use" software environments upon which our work increasingly depends today, both as programmers and as users, are, clearly, the realization of this totalitarian vision.

2

The absurdities we have just examined are the type of fallacies one should indeed expect to find in a mechanistic culture like ours. But we cannot simply dismiss them. For, if we are to understand how the pseudoscience of software engineering grew out of the mechanistic ideology, we must start by studying this distortion of the notions of programming and craftsmanship.

The theorists who promoted the idea of software engineering had, in fact, very little programming experience. They were mostly academics, so their knowledge was limited to textbook cases: small and isolated programming problems, which can be depicted with neat diagrams and implemented by way of rules and methods. Their knowledge was limited, in other words, to software phenomena simple enough to represent with exact, mechanistic models. A few of these theorists were mathematicians, so their preference for formal and complete explanations is understandable.

And indeed, some valuable contributions were made by theorists in the

[15] Ibid., p. 67 n. 3.

1950s and 1960s, when the field of software was new, and useful mechanistic concepts were relatively easy to come by: various algorithms (methods to sort tables and files, for instance) and the principles of programming languages and compilers are examples of these contributions.

The importance of the mechanistic concepts is undeniable; they form, in fact, the foundation of the discipline of programming. Mechanistic models, however, can represent only simple, isolated phenomena. And consequently, mechanistic software concepts form only a small part of programming knowledge. The most important part is the *complex* knowledge, the capacity to deal with many software phenomena simultaneously; and complex knowledge can only develop in a mind, through personal experience. We need complex programming knowledge because the phenomena we want to represent in software – our personal, social, and business affairs – are themselves complex. Restricted to mechanistic concepts, we can correctly represent in software only phenomena that can be isolated from the others.

So it was not so much the search for mechanistic theories that was wrong, as the belief that *all* programming problems are mechanistic. The theorists had no doubt that there would be future advances in programming concepts, and that these advances would be of the same nature as those of the past. They believed that the field of software would eventually be like mathematics: nothing but neat and exact definitions, methods, and theories.

This conviction is clearly expressed by Richard Linger et al.,[16] who refer to it as a "rediscovery" of the value of mathematics in software development. They note that the early interest in mathematical ideas faded as software applications increased in complexity, that the pragmatic aspects of programming seem more important than its mathematical roots. But they believe this decline in formal programming methods to be just a temporary neglect, due to our failure to appreciate their value: "Thus, although it may seem surprising, the rediscovery of software as a form of mathematics in a deep and literal sense is just beginning to penetrate university research and teaching, as well as industry and government practices.… *Of course, software is a special form of mathematics* …."[17]

The authors continue their argument by citing approvingly the following statement made by E. W. Dijkstra (the best-known advocate of "structured programming"): "As soon as programming emerges as a battle against unmastered complexity, it is quite natural that one turns to that mental discipline whose main purpose has been for centuries to apply effective structuring to

[16] Richard C. Linger, Harlan D. Mills, and Bernard I. Witt, *Structured Programming: Theory and Practice* (Reading, MA: Addison-Wesley, 1979), pp. vii–viii.

[17] Ibid., p. viii (italics added).

otherwise unmastered complexity. That mental discipline is more or less familiar to all of us, it is called Mathematics. If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed the most effective way to come to grips with complexity, we have no choice any longer: *we should reshape our field of programming in such a way that, the mathematician's methods become equally applicable to our programming problems, for there are no other means*."[18]

The delusion of software mechanism is clearly seen in these claims. What these theorists see as complexity is not at all the *real* complexity of software – the complexity found in the phenomena I call complex structures, or systems of interacting structures. They consider "complex," systems that are in fact *simple* structures, although perhaps very large structures. They praise the ability of mathematics to master this "complexity"; and indeed, mechanistic methods can handle simple structures, no matter how large. But it is not this kind of complexity that is the real problem of programming. The theorists fail to see that it is quite easy to deal with this kind of complexity, and it is easy precisely because we have the formal, exact tools of mathematics to master it. The reason why practitioners neglect the mathematics and continue to rely on informal methods is that, unlike the professors with their neat textbook examples, *they* must deal with the *real* complexity of the world if they are to represent the world accurately in software. And to master *that* complexity, the formal methods of mathematics are insufficient.

Note the last, emphasized sentence, in each of the two quotations above. These confident assertions clearly illustrate the morbidity of the mechanistic obsession. The theorists say that software is, "of course," a form of mathematics, but they don't feel there is a need to prove this claim. Then, they assert just as confidently that "we should reshape" programming to agree with this claim, treating now an unproven notion as established fact. In other words, since the mechanistic theories do not seem to reflect the reality of programming, we must modify reality to conform to the theories: we must restrict our software pursuits to what *can* be explained mechanistically. Instead of trying to understand the *true* nature of software and programming, as real scientists would, these theorists believe their task is simply to enforce the mechanistic doctrine. The idea that software and programming can be represented mathematically is *their* delusion; but they see it as their professional duty to make us all program and use computers in this limited, mechanistic fashion.

Thus, although it was evident from the beginning that the mechanistic concepts are useful only in isolated situations – only when we can extract a

[18] E. W. Dijkstra, "On a Methodology of Design," quoted ibid. (italics added).

particular software structure, or aspect, from the complex whole – the theorists insisted that the difficulty of programming large and complex applications can be reduced to the easier challenge of programming small pieces of software. They believed that applications can be "built" as we build cars and appliances; that is, as a combination of modules, each module made up of smaller ones, and so on, down to some small bits of software that are easy to program. If each module is kept independent of the others, if they are related strictly as a hierarchical structure, the methods that work with small bits of software – rules, diagrams, mathematics – must hold for modules of any size. The entire application can then be built, one level at a time, with skills no greater than those required to program the smallest parts. All that programmers need to know, therefore, is how to handle isolated bits of software.

So the idea of software engineering is based, neither on personal experience, nor on a sensible hypothesis, but merely on the mechanistic dogma: on the belief that any phenomenon can be modeled through reductionism and atomism.

By the mid-1960s, most software concepts that are mechanistic and also practical had been discovered. But the theorists could not accept the fact that the easy and dramatic advances were a thing of the past, that we could not expect further improvements in programming productivity simply by adopting a new method or principle. They were convinced that similar advances would take place in the future, that there exist many other mechanistic concepts, all waiting to be discovered. To pick just one example, they noticed the increase in programming productivity achieved when moving from low-level to high-level languages, and concluded that other languages would soon be invented with even higher levels of abstraction, so the same increase in productivity would be repeated again and again. (The notion of "generations" of languages, still with us today, reflects this fantasy; see pp. 464–465.)

To make matters worse, just when major improvements in programming concepts ceased, advances in computer hardware made *larger* applications possible. Moreover, continually decreasing hardware costs permitted more companies to use computers, so we needed *more* applications. This situation was called the software crisis. The theorists watched with envy the advances in hardware, which continued year after year while programming productivity stagnated, and interpreted this discrepancy as further evidence that programming must be practised like engineering: if engineering concepts are successful in improving the computers themselves, they *must* be useful for software too.

The so-called software crisis, thus, was in reality the crisis of software mechanism: what happened when the mechanistic principles reached the limit of their usefulness. The crisis was brought about by the software theorists, when they declared that programming is a mechanistic activity. This led to the belief that anyone can practise programming, simply by following certain methods. So the theorists founded the culture of programming incompetence, which eventually caused the crisis. They recognized the crisis, but not its roots – the fallacy of software mechanism. They aggravated the crisis, thus, by claiming that its solution was to treat programming as a form of engineering, which made programming even more mechanistic. Software mechanism became a dogma, and all that practitioners were permitted to know from then on was mechanistic principles.

Deprived of the opportunity to develop complex knowledge, our skills remain at a mechanistic level – the level of novices. Craftsmanship – the highest level of knowledge and skills – is attained by using the mind's capacity for complex structures, while mechanistic thinking entails only simple structures. So what the theorists are promoting through their ideas is not an intellectual advance, but a reversal: from complex to mechanistic thinking, from expertise to mediocrity, from a culture that creates skilled masters to one that keeps programmers as permanent novices.

The software crisis was never resolved, of course, but we no longer notice it. We no longer see as a crisis the inefficiency of programmers, or the astronomic amounts of money spent on software, or the $100-million failures. We are no longer surprised that applications are inadequate, or that they cannot be kept up to date and must be perpetually replaced; we are regularly replacing now, in fact, not just our applications but our entire computing environments. We don't question the need for society to support a large software bureaucracy. And we don't see that it is the incompetence of programmers, and the inadequacy of their applications, that increasingly force other members of society to waste their time with spurious, software-related activities. What was once a crisis in a small section of society has become a normal way of life for the entire society.

The software crisis can also be described as the struggle to create useful applications in a programming culture that permits only mechanistic thinking; specifically, the struggle to represent with simple software structures the complex phenomena that make up our affairs. It is not too much to say that whatever useful software we have had was developed, not *by means* of, but *in spite* of, the principles of software engineering; it was developed *through craftsmanship*, and while fighting the restrictions imposed by our corrupt programming culture. Had we followed the teachings of the software theorists, we would have no useful applications today.

3

The software theorists, we saw, distorted both the notion of craftsmanship and the notion of programming to fit their mechanistic fantasies. They decided *arbitrarily* that programming is like engineering, because they had already decided that future advances in programming principles were possible, and that these advances would be, like those of the past, mechanistic. They likened incompetent programmers to craftsmen because they saw the evolution of practitioners from craftsmen to engineers as a necessary part of these advances. The analogy – an absurdity – became then the central part of the idea of software engineering. Mesmerized by the prospect of building software applications as successfully as engineers build physical structures, no one noticed the falsity of the comparison. Everyone accepted it as a logical conclusion reached from the idea of software engineering, even as software engineering itself was only a wish, a fantasy.

The theorists claimed that programming, if practised as craftsmanship, cannot improve beyond the level attained by an average programmer. But they made this statement without knowing what *real* software craftsmanship is. They saw programmers as craftsmen while programmers lacked the very qualities that distinguished the old craftsmen. Programming, as a matter of fact, is one of those vocations that can benefit greatly from the spirit of craftsmanship – from personal skills and experience – because it requires complex knowledge. If we are to liken programmers to the old craftsmen, we should draw the correct conclusion; namely, that programmers too must have the highest possible education, training, and experience. (And it is probably even more difficult to attain the highest level of expertise in the field of programming than it was in the old fields.)

Had we allowed programmers to develop their skills over many years, to perform varied and increasingly demanding tasks, and to work in ways that enhance their minds, rather than waste their time with worthless concepts – in other words, had we created a programming culture in the spirit of craftsmanship – we would have had today a true programming profession. We would then realize that what programmers must accomplish has little to do with engineering; that mechanistic knowledge (including subjects like mathematics and engineering), crucial though it is, is the *easy* part of programming expertise; that it is the *unspecifiable* kind of knowledge (what we recognize as personal skills and experience) that is the most difficult and the most important part.

The software theorists note the higher levels of knowledge attained by

certain individuals, but they cannot explain this performance mechanistically; so they brand it as "art" and reject it as unreliable. We could always find exceptional programmers; but instead of interpreting their superior performance as evidence that it *is* possible to attain higher levels of programming skills, instead of admitting that the traditional process of skill acquisition is the best preparation for programmers, the mechanists concluded the opposite: that we must *avoid* these individuals, because they rely on personal knowledge rather than exact theories.

Distorting the notions of craftsmanship and programming, however, was not enough. In order to make software mechanism plausible, and to appropriate the term "engineering" for their own activities, the software theorists had to distort the notion of engineering itself. Thus, they praise the principles of engineering, and claim that they are turning programming into a similar activity, while their ideas are, in fact, childish imitations of the engineering principles.

It is easy for the software theorists to delude themselves, since they know even less about engineering than they know about programming. They praise the power and precision of mathematics; and, indeed, the real engineering disciplines are grounded upon exact and difficult mathematical concepts. *Their* theories, on the other hand – when not plain stupid – are little more than informal pieces of advice. Far from having a solid mathematical foundation, the software theories resemble the arguments found in self-help books or in cookbooks more than they do engineering principles. The few theories that are indeed mathematical have no practical value, so they are ignored, or are made useful by being downgraded to informal methods. The most common form of deception, we will see, is to promote a formal theory by means of contrived, oversimplified case studies, while employing in actual applications only the downgraded, informal variant. Thus, whereas real engineering is a practical pursuit, *software* engineering works only with trivial, artificial examples.

The software theorists also misrepresent engineering when they point to the neat hierarchical structures – components, modules, prefabricated subassemblies – as that ideal form of design and construction that programming is to emulate. Because they know so little about engineering, all they see in it is what they wish programming to become, what they believe to be the answer to all programming problems, as if the concept of hierarchical structures were all there is to engineering. They ignore the creativity, the skills, the contribution of exceptional minds; that is, the *non-mechanistic* aspects of engineering, which are just as important as the formal principles and methods.

Clearly, without the non-mechanistic aspects there would be no inventions or innovations, and engineering would only produce neat structures of old things.

The software theorists refuse to acknowledge the informal aspects of engineering because, if they did, they would have to admit that much of programming too is informal, non-mechanistic, and dependent on personal skills and experience. In programming, moreover, our non-mechanistic capabilities are even more important, because, unlike our engineering problems, nearly all the problems we are addressing through software – our social, personal, and business affairs – form systems of interacting structures.

In conclusion, the idea of software engineering makes sense only if we agree to degrade our conceptions of knowledge and skills, of craftsmanship and engineering, of software and programming, to a level where they can all be replaced with the mechanistic principles of reductionism and atomism.

The early software theorists were trained scientists, as we saw, and made a real contribution – at least where mechanistic principles are useful. But it would be wrong to think that *all* software theorists are true scientists. By upholding the mechanistic software ideology, the early theorists established a software culture where incompetents, crackpots, and charlatans could look like experts.

Thus, someone too ignorant to work in the exact sciences, or in the real engineering disciplines, could now pursue a prestigious career in a software-related field. Just as the mechanistic software culture had made it possible for the most ignorant people to become programmers, the same culture allowed now anyone with good communication skills to become a theorist, a lecturer, a writer, or a consultant. Individuals with practically no knowledge of programming, or computers, or science, or engineering became rich and famous simply by talking and writing about software, as they could hypnotize programmers and managers with the new jargon. Also, because defining things as a hierarchical structure was believed to be the answer to all programming problems, anyone who could draw a hierarchical diagram was inventing a new theory or methodology based on this idea. Thousands of books, newsletters, periodicals, shows, and conferences were created to promote these idiocies.

Finally, as the entire society is becoming dependent on software, and hence on ignorant theorists and practitioners, we are all increasingly preoccupied with worthless mechanistic ideas. Thus, the ultimate consequence of the mechanistic software ideology is not just programming incompetence, but a mass stupidity that the world has not seen since the superstitions of the Dark Ages. (If you think this is an exaggeration, wait until we study the GOTO superstition – the most famous tenet of programming science.)

Software Engineering as Pseudoscience

1

Let us start with the *definition* of software engineering. Here are three definitions frequently cited in the software literature: "Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems."[1] "The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them."[2] "The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines."[3]

At first sight, these statements look like a serious depiction of a profession, or discipline. Their formal tone, however, is specious. They may well describe a sphere of activities, but there is nothing in these definitions to indicate the *usefulness*, or the *success*, of these activities. In other words, even if they describe accurately what software practitioners are doing (or ought to be doing), we cannot tell from the definitions themselves whether these activities are essential to programming, or whether they are spurious. As we will see in a moment, an individual can appear perfectly rational in the pursuit of an idea, and can even display great expertise, while the idea itself is a delusion.

These definitions are correct insofar as they describe the programming principles recommended by the software theorists. But we have no evidence that it is possible to develop actual software applications by adhering to these principles. We saw in the previous section that the very term "software engineering" constitutes a circular definition, since it was adopted without determining first whether programming is indeed a form of engineering; it was adopted because the software theorists *wished* programming to be like engineering (see pp. 495–497). And the same circularity is evident in the

[1] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1218.

[2] Barry W. Boehm, "Software Engineering," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE), p. 54 – paper originally published in *IEEE Transactions on Computers* C-25, no. 12 (1976): 1226–1241.

[3] F. L. Bauer, quoted in Randall W. Jensen and Charles C. Tonies, "Introduction," in *Software Engineering*, eds. Randall W. Jensen and Charles C. Tonies (Englewood Cliffs, NJ: Prentice Hall, 1979), p. 9.

definitions just cited: software engineering is "that form of engineering," is "the practical application of scientific knowledge," is "sound engineering principles." In all three cases, the definition merely expresses the unproven idea that software engineering is a form of engineering.

The definition of software engineering, thus, is not the definition of a profession or discipline, but the definition of a wish, of a fantasy. The fallacy committed by the advocates of software engineering is to think that, if it is possible to *define* a set of principles and methods so as to formally express a wish, then we should also be able to practise them and *fulfil* that wish.[4]

Recall what a pseudoscience is: a system of belief masquerading as scientific theory. Accordingly, the various principles, methods, and activities known as software engineering, no matter how rational they may appear when observed individually, form in reality a pseudoscience.

If it is so difficult to distinguish between sensible and fallacious definitions, or between useful and spurious activities, in the domain of programming, it will perhaps help to examine first some older and simpler delusions.

Consider superstitions – the idea that the number 13 brings misfortune, for instance. Once we accept this idea, the behaviour of a person who avoids the number 13 appears entirely rational and logical. Thus, to determine whether a particular decision would involve the value 13, that person must perform correctly some calculations or assessments; so he must do exactly what a mathematician or engineer would do in that situation. When a person insists on redesigning a device that happens to have thirteen parts, or kilograms, or inches, his acts are indistinguishable from those of a person who redesigns that device in order to improve its performance; in both cases, the changes entail the application of strict engineering methods, and the acts constitute the pursuit of a well-defined goal. When the designers of a high-rise building decide to omit the thirteenth floor, they must adjust carefully their plans to take into account the discrepancy between levels and floor numbers above the twelfth floor. And lawyers drawing documents for the units on the high floors must differentiate between their level, which provides the legal designation, and the actual floor number. These builders and lawyers, then, perform the same acts as when solving vital engineering or legal problems.

[4] In "Software Magic" (in the introductory chapter), we studied the similarity between mechanistic software concepts and primitive magic systems; and we saw that magic systems, too, entail the formal expression of wishes and the meticulous practice of the rituals believed to fulfil those wishes.

Note that the activities performed by believers, and by anyone else affected by this superstition, are always purposeful, logical, and consistent. Watching each one of these activities *separately*, a casual observer has no reason to suspect that the ultimate objective is simply to avoid the number 13. In fact, even when we *are* aware of this objective, we may have no way of recognizing the *uselessness* of these activities. Thus, we could formally define these activities as "the practical application of scientific and engineering knowledge to the prevention of misfortune." But, obviously, just because we can define them it doesn't mean that they are effective.

Consider also a system of belief like astrology. Astrologers must follow, of course, the position of the heavenly bodies, and in this activity they behave just like astronomers. The interpretation of these positions involves various principles and methods, some of which have been in use for millennia; so in this activity, too, astrologers must display professional knowledge and skills. A person who cancels a trip because an astrological calculation deems travel hazardous is clearly concerned with safety, and acts no differently from a person who cancels a trip because of bad weather. Astrologers employ certain principles and methods – tables that relate traits to birth dates, for example – to assess the personality of people and to explain their behaviour; but psychologists also use various principles and methods to assess personality and to explain behaviour.

So, as in the case of superstitions, just by watching each activity *separately*, an observer cannot suspect that astrology as a whole is a delusion. Within this system of belief – once we accept the idea that human beings are affected by the position of the heavenly bodies – all acts performed by practitioners and believers appear purposeful, logical, and consistent. A formal definition like "the practical application of astronomic and mathematical principles to the prediction of future events" describes accurately these activities. But being definable doesn't make these activities sensible. As in the case of superstitions, their definition is the definition of a wish.

And so it is for software engineering. Recall the definitions cited earlier: "that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems," etc. We wish programming to be a form of engineering; but, just because we can express this wish in a formal definition, it doesn't follow that the methods and activities described by this definition form a practical pursuit like traditional engineering. We note that millions of practitioners follow the mechanistic theories of software engineering, and each one of these activities appears purposeful, intelligent, and urgent. But no matter how logical these activities are when observed *separately*, the body of activities as a whole can still constitute a delusion. As we will see in the present chapter, the pseudoscientific

nature of the mechanistic software theories is easily exposed when we assess them with the principles of demarcation between science and pseudoscience (see "Popper's Principles of Demarcation" in chapter 3).

2

Pseudosciences are founded on hypotheses that are treated as unquestionable truth. (In the theories we have just examined, the hypothesis is that certain numbers, or the heavenly bodies, influence human affairs.) Scientific theories also start with a hypothesis, but their authors never stop *doubting* the hypothesis. Whereas pseudoscientists think their task is to *defend* their theory, serious workers know that theories must be *tested*; and the only effective way to test a theory is by attacking it: by searching, not for confirmations, but for falsifications. In empirical science it is impossible to *prove* a theory, so confirmations are worthless: no matter how many confirmations we find, we can never be sure that we have encountered all possible, or even all relevant, situations. At the same time, just one situation that falsifies the theory is sufficient, logically, to refute it. The correct approach, therefore, is to accept a theory not because it can be defended, but because it cannot be refuted; that is, not because we can confirm it, but because we cannot falsify it.

It is easy to defend a fallacious theory: all we have to do is restrict our studies to cases that confirm its claims, and ignore those cases that falsify it. Thus, while a scientific theory is required to *pass* tests, pseudosciences appear to work because they *avoid* tests. If we add to this the practice of continually *expanding* the theory (by inventing new principles to cope with the falsifications, one at a time), it should be obvious that almost any theory can be made to look good.

A popular pseudoscientific theory becomes a self-perpetuating belief system, and can reach a point where its validity is taken for granted no matter how fallacious it is. This is because its very growth is seen by believers, in a circular thought process, as proof of its validity. Whenever noticing a failure – a claim that does not materialize, for instance – they calmly dismiss it as a minor anomaly. They are convinced that an explanation will soon be found, or that the failure is merely an exception, so they can deal with it by modifying slightly the theory. They regard the system's size, its many adherents, the large number of methods and formulas, the length of time it has been accepted, as a great logical mass that cannot be shaken by one failure. They forget that the system's great mass was reached precisely because they always took its validity for granted, so they always dismissed its failures – *one at a time*, just as they are now dismissing the new one.

A theory can be seen as a body of provisional conjectures that must be verified empirically. Any failure, therefore, must be treated as a falsification of the theory and taken very seriously. If believers commit (out of enthusiasm, for example) the mistake of regarding any success as confirmation of the theory while dismissing the failures as unimportant, the system is *guaranteed* to grow, no matter how erroneous those conjectures are. The system's growth and popularity are then interpreted as evidence of its validity, and each new failure is dismissed on the strength of this imagined validity, when in fact it is these very failures that ought to be used to *judge* its validity. This circularity makes the theory unfalsifiable: apparently perfect, while in reality worthless.

Pseudosciences, thus, may suffer from only one mistaken hypothesis, only one false assumption. Apart from this mistake, the believers may be completely logical, so their activities may be indistinguishable from true scientific work. But if that one assumption is wrong, the system as a whole is nonsensical.

It is this phenomenon – the performance of activities that are perfectly logical individually even while the body of activities as a whole constitutes a delusion – that makes pseudosciences so hard to detect, so strikingly like the real sciences. And this is why the principles of demarcation between science and pseudoscience are so important. Often, they are the only way to expose an invalid theory.

Any hypothesis can form the basis of a delusion, and hence a pseudoscience. So we should not be surprised that the popular *mechanistic* hypothesis has been such a rich source of delusions and pseudosciences. Because of their similarity to the traditional pseudosciences, I have called the delusions engendered by the mechanistic hypothesis *the new pseudosciences* (see pp. 203–204). Unlike the traditional ones, though, the new pseudosciences are pursued by respected scientists, working in prestigious universities.

Let us recall how a mechanistic delusion turns into a pseudoscience (see pp. 204–205, 233–235). The scientists start by committing the fallacy of reification: they assume that a model based on one isolated structure can provide a useful approximation of the complex phenomenon, so they extract that structure from the system of structures that make up the actual phenomenon. In complex phenomena, however, the links between structures are too strong to be ignored, so their model does not represent the phenomenon closely enough to be useful. What we note is that the theory fails to explain certain events or situations. For example, if the phenomenon the scientists are studying involves minds and societies, the model fails to explain certain behaviour patterns, or certain intelligent acts, or certain aspects of culture.

Their faith in mechanism, though, prevents the scientists from recognizing these failures as a refutation of the theory. Because they take the possibility of a mechanistic explanation not as hypothesis but as fact, they think that only *a few* falsifying instances will be found, and that their task is to *defend* the theory: they search for confirming instances and avoid the falsifying ones; and, when a falsification cannot be dismissed, they *expand* the theory to make it explain that instance too. What they are doing, thus, to save the theory, is *turning falsifications of the theory into new features of the theory.* Poor mechanistic approximations, however, give rise to an *infinity* of falsifying instances; so they must expand the theory again and again. This activity is both dishonest and futile, but they perceive it as research.

A theory can be said to work when it successfully explains and predicts; if it must be expanded continually because it fails to explain and predict some events, then, clearly, it does *not* work. Defending a mechanistic theory looks like scientific research only if we regard the quest for mechanistic explanations as an indisputable method of science. Thus, the mechanists end up doing in the name of science exactly what pseudoscientists do when defending *their* theories. When expanding the theory, when making it agree with an increasingly broad range of situations, what they do in effect is annul, one by one, its original claims; they make it less and less precise, and eventually render it worthless (see pp. 225–226).

Since it is the essence of mechanism to break down complex problems into simpler ones, the mechanistic hypothesis, perhaps more than any other hypothesis, can make the pursuit of a delusion look like serious research. These scientists try to solve a complex problem by dividing it into simpler ones, and then dividing these into simpler ones yet, and so on, in order to reach isolated problems; finally, they represent the isolated problems with simple structures. And in both types of activities – dividing problems into simpler ones, and working with isolated simple structures – their work is indistinguishable from research in fields like physics, where mechanism *is* useful. But if the original phenomenon is non-mechanistic, if it is the result of interacting phenomena, a model based on isolated structures cannot provide a practical approximation. So those activities, despite their resemblance to research work, are in fact fraudulent.

Being worthless as theories, all mechanistic delusions must eventually come to an end. The scientists, however, learn nothing from these failures. They remain convinced that the principles of reductionism and atomism can explain all phenomena, so their next theory is, invariably, another mechanistic delusion. They may be making only one mistake: assuming that any phenomenon can be separated into simpler ones. But if they accept this notion unquestioningly, they are bound to turn their theories into pseudosciences.

3

The purpose of this discussion is to show how easy it is for large numbers of people, even an entire society, to engage in activities that appear intelligent and logical, while pursuing in fact a delusion; in particular, to show that the mechanistic software pursuits constitute this type of delusion. Our software delusions have evolved from the same mechanistic culture that fosters delusions in fields like psychology, sociology, and linguistics. But, while these other delusions are limited to academic research, the software delusions are affecting the entire society.

Recall how a mechanistic software theory turns into a pseudoscience. Software applications are systems of interacting structures. The structures that make up an application are the various *aspects* of the application. Thus, each software or business practice, each file with the associated operations, each subroutine with its calls, each memory variable with its uses, forms a simple structure. And these structures interact, because they share their elements – the various software entities that make up the application (see pp. 347–348).

The mechanistic software theories, though, claim that an application can be programmed by treating these aspects as *independent* structures, and by dealing with each structure separately. For example, the theory of structured programming is founded on the idea that the *flow-control* operations form an independent structure; and the database theories are founded on the idea that the *database* operations form an independent structure.

Just like the other mechanistic theories, the software theories keep being falsified. A theory is falsified whenever we see programmers having to override it, whenever they cannot adhere strictly to its principles. And, just like the other mechanists, the software mechanists deny that these falsifications constitute a refutation of the theory: being based on mechanistic principles, they say, the theory *must* be correct.

So instead of *doubting* the theory, instead of severely testing it, the software mechanists end up *defending* it. First, they search for confirmations and avoid the falsifications: they discuss with enthusiasm the few cases that make the theory look good, while carefully avoiding the many cases where the theory failed. Second, they never cease "enhancing" the theory: they keep expanding it by contriving new principles to make it cope with the falsifying situations as they occur, one at a time.

Ultimately, as we will see in the following sections, all software theories suffer from the two mechanistic fallacies, reification and abstraction: they claim that we can treat the various aspects of an application as independent

structures, so we can develop the application by dealing with these structures separately; and they claim that we can develop the same applications by starting from high levels of abstraction as we can by starting from low levels. The modifications needed later to make a theory practical are then simply a reversal of these claims; specifically, restoring the capability to link structures and to deal with low-level entities. In the end, the informal, traditional programming concepts are reinstated – although in a more complicated way, under new names, as part of the new theory. So, while relying in fact on these informal concepts, everyone believes that it is the theory that helps them to develop applications.

Our programming culture has been distinguished by delusions for over forty years. These delusions are expressed in thousands of books and papers, and there is no question of studying them all here. What I want to show rather is that, despite their variety and their changes over the years, all software delusions have the same source: the mechanistic ideology. I will limit myself, therefore, to a discussion of the most famous theories. The theories are changing, and new ones will probably appear in the future, but this study will help us to recognize the mechanistic nature of any theory.

In addition to discussing the mechanistic theories and their fallacies, this study can be seen as a general introduction to the principles and problems that make up the challenge of software development. So it is also an attempt to explain in lay terms what is the *true* nature of software and programming. Thus, if we understand why the mechanistic ideas are worthless, we can better appreciate why personal skills and experience are, in the end, the most important determinant in software development.