SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 6: *Software as Weapon*
Section *Software Charlatanism*

This extract includes the book's front matter and part of chapter 6.

Copyright © 2013 Andrei Sorin

The digital book and extracts are licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives International License 4.0.

This section shows that the development systems and methods promoted by the software elites are based on mechanistic fallacies and cannot provide the benefits claimed for them.

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

Copyright ©2013 Andrei Sorin
Published by Andsor Books, Toronto, Canada (January 2013)
www.andsorbooks.com

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, without the prior written permission of the publisher. However, excerpts totaling up to 300 words may be used for quotations or similar functions without specific permission.

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

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.

Software Charlatanism

Software exploitation, we saw, plays on our mechanistic delusions; namely, on the belief that problems requiring complex knowledge can be broken down into simpler problems, which can then be solved mechanistically. When the software charlatans tempt us with the promise of easy answers to difficult problems – answers in the form of software devices – what they do is tempt us to commit the mechanistic fallacies, reification and abstraction. For, only if we commit these fallacies will we believe that software devices can be a substitute for the complex, non-mechanistic knowledge required to solve those problems.

In the present section, we will examine how the mechanistic delusions manifest themselves in various software-related activities; that is, how the two mechanistic fallacies lead to *software* delusions, and how these delusions are being exploited by the software elites. The world of programming, in particular, consists almost entirely of delusions, and we will study them in detail in chapter 7. Here, the programming delusions are mentioned briefly, just to show how they are contributing to delusions in other software-related activities.

The Delusion of High Levels

1

Recall our discussion in "Software Structures" in chapter 4. Software applications are complex structures, because they can be viewed from different perspectives. Each aspect of an application is one of its processes, one of the simple structures that make up the complex structure. Thus, each subroutine together with its uses, each database field or memory variable together with the associated operations, each business rule or programming method, can be seen as a simple structure. But these structures are not independent. Although in our imagination we can separate them, in reality they share their elements (the software entities that make up the application), so they interact.

Take, for example, subroutines.[1] The use of subroutines is usually shown as part of the application's flow-control logic – those neat diagrams said to define precisely and completely the flow of execution. Subroutines, though, give rise to *additional* relations between software elements – relations that are not seen

[1] As in chapter 4, I subsume under "subroutine" any software entity that is used in several parts of the application – modules, functions, procedures, objects, etc. (see p. 353).

in a diagram that represents their calls. Each subroutine, along with its uses and effects, forms a hierarchical structure. One way to depict this structure is as follows: the application (the top element) branches into two categories, those software elements that are affected by the subroutine and those that are unaffected; then, the first category branches into more detailed ones, on one or more levels, reflecting the various ways the subroutine affects them, while the second category branches into more detailed ones reflecting the various reasons the subroutine does *not* affect them; finally, the categories branch into the low-level elements – the blocks of statements and the individual statements that make up the application.

It is not difficult to see that, if we do this for different subroutines, the resulting structures will be different. The terminal elements, however, will be the same; for, in all structures, the terminal elements are the application's statements. Thus, since these structures share their elements, they are bound to interact. So there is no one structure that can represent *all* the subroutines. The only way to represent the subroutines and their calls accurately is as a system of interacting structures. The neat diagram that claims to represent the flow of execution does indeed reduce the application to a simple hierarchical structure, but it does this by distorting reality; specifically, by showing only *some* of the relations that subroutines create between software elements (the calls themselves).

Each subroutine, thus, constitutes *one* aspect of the application, one way of viewing it. But there are other ways yet of viewing the application, besides the use of subroutines. We can view it, for example, from the perspective of each set of related data entry operations, or each shared variable or database field, or each business practice implemented in the application. Like subroutines, each one of these processes constitutes one aspect of the application, and can be represented with a hierarchical structure that describes its effects on the software entities that make up the application. And, along with the structures generated by subroutines, these structures share their terminal elements (the software entities we find at the low levels).

A software application, then, is all these structures, and the difficulties we face in programming are due largely to having to deal with many of them at the same time. Clearly, if a software element is shared by several structures, we must take into account all of them if we are to program the element correctly. Most software deficiencies are due to overlooking the links between structures – links caused by this sharing of elements: the programmer takes into account the effect of an operation on some of the structures, but fails to recognize its effect on others. We must accept the fact that software applications are non-mechanistic phenomena: the interactions between their constituent structures are too complex to represent with mechanistic means like rules,

diagrams, or mathematics. To put it differently, the approximations possible with mechanistic models are rarely accurate enough for software applications.

If we forget that applications are complex phenomena, we may think of programming as simply identifying all the structures and interactions, dealing with them separately, and then combining the results. But this would amount to reducing a complex structure to simple ones, and we already know that this is impossible. Even *thinking* of these structures separately is an illusion, since they can only exist as *interacting* structures. We may refer to them informally as separate structures (this is the only way to discuss the issue), but we must bear in mind that they exist only in our imagination; what exists in reality is the complex structure, the whole phenomenon. Thus, if we create the application by dealing with those structures separately, we will likely end up with a *different* phenomenon – a different application. What we will notice in practice is that the application does not work as we expected, and the reason is that some of the interactions are wrong, or missing.

Programming, therefore, requires the capacity for complex structures. So it requires a human mind, because only minds can develop the complex knowledge structures that can mirror the complex phenomena of programming. As programmers, we are expected to combine in our mind the various aspects of an application. We are expected, thus, to develop knowledge that cannot be precisely specified (with rules or diagrams, for instance). And we *can* develop this type of knowledge, because minds can process complex structures.

Nor is this an unreasonable demand. What we are expected to do in programming is no different from what we have to do most of the time simply to live a normal life. Recall the analysis of stories (pp. 350–351). We are all expected to combine in our mind the various aspects of a story that we hear or read. This, in fact, is the only way to understand it, because it is impossible to specify precisely and completely all the knowledge embodied in a story. We understand a story by *discovering* this knowledge. Authors can convey to us, through the expedient of words and sentences, almost any ideas, situations, arguments, or feelings. Complex knowledge that exists in one person's mind can be reproduced with great accuracy in other minds by means of language. We derive *some* of this knowledge from the individual structures – from sentences, and from each aspect of the story. Most of the knowledge, however, inheres not in individual structures but in their *interactions*. We discover these interactions by processing the structures. We develop the complex knowledge that is the story by unconsciously combining its structures in our mind, and combining them also with the knowledge structures already present in the mind. Linguistic communication is possible because we have the capacity to combine knowledge structures, and because we already possess some knowledge that is the same as the author's.

If we tried to express by means of independent structures all the knowledge we derived from a story, we would find it an impossible task, because a complex structure cannot be reduced to simple ones. What we would lose is the interactions. Even if we managed somehow to identify all the elements and all the aspects of the story, we could not identify all their interactions. To put this differently, if we could reduce the story to a set of precise specifications, we could program a computer to understand that story exactly as *we* do – a preposterous notion; or, the fact that a machine cannot understand stories as *we* do proves that stories, and linguistic communication generally, involve complex structures.

And so do software applications. The software theories, then, are wrong when claiming that applications can be programmed with the methods used in building physical structures. In manufacturing and construction we can restrict ourselves to mechanistic methods because we purposely restrict our physical structures to neat hierarchies of parts and subassemblies. But software applications are more akin to stories than to physical structures. This is true because we employ software, as we do language, to represent the world, and our concerns and affairs, which consist of complex phenomena. The potency of software, like that of language, lies in its ability to generate complex structures. If we restricted ourselves to mechanistic methods, as we do in manufacturing and construction, we would use only a fraction of this potency; and we would be unable to create linguistic or software structures that represent the world, or our concerns and affairs, accurately.

We can detect in the individual structures *some* of the knowledge that constitutes the application. But much of this knowledge inheres in the *interactions* between structures. We cannot specify it precisely and completely, therefore, any more than we can specify precisely and completely all the knowledge that inheres in a story. If we reduce this knowledge to precise specifications – as we are obliged to do when strictly following mechanistic programming methods – we must necessarily leave some of it out, and the application will not work as we expected.

It is impossible to create adequate software applications with mechanistic methods for the same reason it is impossible to create or understand *stories* with mechanistic methods. But we know that we can, by relying on the non-mechanistic capabilities of our mind, create and understand complex linguistic structures. We also can, therefore, by relying on the same non-mechanistic capabilities, create complex *software* structures; namely, applications that mirror the complexity of the world. This is the meaning of programming expertise.

We are so easily deceived by the mechanistic software theories because we like their promise. The promise, essentially, is that methods and devices simple enough to be used by almost anyone can be a substitute for programming expertise. And we believe this promise because we fail to see that to accept it means to commit the two mechanistic fallacies, reification and abstraction.

The software theories appear to make programming easier because they treat applications, or the activities involved in creating applications, as separable into independent parts: database operations, display operations, reporting operations, and so on. There are indeed many aspects to an application, as we saw, but these are rarely independent structures. The software theories invite us to reify programming and applications because, once we have independent structures, they can tempt us to start from higher levels of abstraction within each structure. By the time we commit both fallacies, what is left of programming is indeed easy. But it is easy because the concept of programming, and the resulting software, were impoverished: many of the functions we could implement before are no longer possible.

All software theories, in the final analysis, make the same claim: the task of programming can be simplified by starting the development process from higher-level software elements; and we can accomplish this by allowing various expedients – methodologies, software tools, built-in operations – to act as substitutes for the knowledge, experience, and work necessary for creating the lower levels. But this would be possible only if applications consisted of independent structures.

Let us briefly examine how this claim manifests itself in some of the theories and devices promoted by the software elites. The theory known as structured programming encourages us to view the application as a neat hierarchical structure of software elements, which can then be developed independently, one level at a time; but we can do this only if we take into account just one aspect of the application – namely, the sequence of execution of its elements – and ignore the links that the other aspects cause between the same elements. The relational database theory claims that databases can be designed as independent structures, interacting only at high levels with the structures formed by the other aspects of the application; but this is true only in very simple situations. Many development environments provide built-in operations for the user interface, claiming in effect that these operations can be separated from the other aspects, or that the interactions between them occur only at high levels; but in most applications the user interface interacts with the other processes at the low level of statements and variables. Systems like spreadsheets and report writers claim that users can create simple applications without programming – by combining instead some high-level, built-in operations; but even simple applications involve aspects that interact at low levels, and require

therefore programming in one form or another. The concept of ready-made components or objects is based on the assumption that business systems of any complexity can be "built" from high-level software elements, just as cars and appliances are built from prefabricated subassemblies; but the assumption is wrong, because, unlike physical components, software components must also interact at the lower levels that are their internal operations.

2

To demonstrate the fallacy of high starting levels, let us analyze a specific situation. A common requirement, found in most business applications, is to access individual fields in database files. The application's user may need to see the phone number or the outstanding balance of a customer, or the quantity in stock of a certain part; or he may need to modify the address of a customer, or the description of a part. Very often, these operations involve more than one file; for example, a customer is displayed together with its outstanding invoices, or a part together with its sales history. Typically, the user specifies some values to identify the records: customer number, invoice number, range of dates, etc. The program displays certain fields from those records, and the user is perhaps permitted to modify some of them. These can be isolated fields (thus giving the user the opportunity to see or modify any files and fields in the database), but most often they are groups of fields logically associated with specific functions: inventory control, financial information, shipping activity, etc. If we also include such options as adding new records and deleting existing ones, we may refer to this category of operations as *file maintenance* operations.

Now, file maintenance operations constitute fairly simple programming tasks. Moreover, much of this programming is very similar in all applications. So it is tempting to conclude that we can replace the programming of file maintenance operations with a number of high-level software elements – some built-in procedures, for example. We should then be able to generate any file maintenance operation by combining these high-level elements, rather than starting with the individual statements and operations of a traditional programming language. I want to show, though, that despite the simplicity and repetitiveness of file maintenance programming, it is impossible to start from higher-level elements.

The illusion of high levels arises when we perceive software as a combination of separable structures, or aspects. There are at least two aspects to the file maintenance operations: database operations and user interface operations. So, to keep the discussion simple, let us assume that these two aspects are the only important ones. If we think of each aspect separately, it is quite easy to

imagine the higher levels within each structure, and to conclude that we can start from higher levels. We may decide, for example, that most database operations can be generated by starting with some built-in procedures that let us access specific records and fields; and most interface operations, with some built-in procedures that display individual fields and accept new values. Thus, by specifying a few parameters (file and field names, index keys, display coordinates, etc.), we should be able to generate, simply by invoking these procedures, most combinations of database operations, and most combinations of interface operations.

We commit the fallacy of abstraction, however, if we believe that the alternatives possible when starting from higher levels are about the same as those we had before. The temptation of high levels is so great that we are liable to perceive an application as simpler than it actually is, just so that we can rationalize the reduced flexibility. It takes much experience to anticipate the consequences of the restriction to high levels. For, it is only later, when the application proves to be inadequate, when important requirements cannot be met and even simple details are difficult to implement, that the impoverishment caused by abstraction becomes evident.

But abstraction became possible only through reification – only after separating the two structures, database and user interface. And reification causes its own kind of impoverishment. We can indeed view file maintenance operations from the perspective of either the database or the interface operations, but only in our imagination. In reality, the file maintenance operations consist of *both* the database and the interface operations. Separately, these operations can indeed be represented as simple structures, because we can identify most of their elements and relations. But when part of an application, these operations interact, giving rise to a complex structure. It is this complex structure that constitutes the real file maintenance operations, not the two imaginary, reified simple structures.

Reification impoverishes the complex structure by destroying the interactions between its constituent structures. When we lose the interactions, we also lose many alternatives for the top element of the complex structure. Each alternative at this level represents a particular file maintenance operation, which may be required by some application. Certain interactions are still possible, of course – those that can be generated by combining the high-level, built-in procedures. The alternatives resulting from these interactions are the file maintenance operations that can be implemented even after reification and abstraction. Most interactions, however, take place at low levels, so they can only be implemented with such means as statements, variables, and conditions; that is, with programming languages. And these interactions are no longer possible once we lose the lower levels. The two fallacies, thus,

contribute together to the impoverishment of the complex structure that is the application. They are usually committed together, and it is seldom possible, or necessary, to analyze them separately.

To appreciate why it is impossible to eliminate the low levels, all we have to do is think of the details that a programmer faces when implementing a typical file maintenance operation. Thus, the user may want to see only some of the fields at first, and then various other fields depending on the values present in the previous ones; or he may need to scan records, forward or backward, rather than ask for specific ones; in one application the user may want to see detailed information, in another only a summary; in one situation some of the fields may always be modified, in another the fields may be modified only under certain conditions; in some applications, modifying a field must produce a change in other fields, and perhaps in other files too; and so on.

Clearly, the number of possible requirements, even for relatively simple operations like file maintenance, is practically infinite. But the important point is that this variety, and the details that make up these requirements, entail the low levels of *both* the database and the interface operations. To implement a particular requirement, therefore, we need not only low-level software elements in both kinds of operations, but elements that can be *shared* by these operations; in other words, exactly what abstraction and reification would *prevent* us from creating. For example, to display a field depending on the value of another field, we must formulate conditional statements involving particular fields and display operations; and to display details from one file along with the summary of another, we must create a small piece of software that reads records, accesses fields, performs calculations and comparisons, and displays values.

Each requirement reflects a particular file maintenance operation; each one is, therefore, an alternative value for the top element of the complex structure formed by the interaction of the database and display operations. If we agree that a programmer must be able to implement *any* file maintenance operation, and hence to generate *any* alternative, it is obvious that he must be able to create and combine all the low-level elements forming the database operations, and all the low-level elements forming the display operations (and probably other low-level elements and operations too). The use of low levels helps us avoid both fallacies: it lets us generate all the alternatives within each structure, *and* the alternatives resulting from the interaction of the two structures.

If you still have doubts about the importance of the low levels, look at it this way: if just *one* low-level element is not available, at least one file maintenance operation will be impossible to implement; and if this alternative happens to be required, the application will be inadequate. Each alternative of the top element is the result of a unique combination of elements at the lower levels.

So, the only way to ensure that *any* alternative can be implemented is to retain the low-level elements.

Since each alternative is unique, no matter how many alternatives we have already implemented, or are available through built-in procedures, the next application may still have to be programmed starting with low-level elements. Only naive and inexperienced practitioners believe that they can have the versatility of the low levels while being involved only with high levels. In reality, the simplicity promised for high-level operations is achieved precisely by reducing the number of alternatives. It takes the experience of many applications to recognize in a given situation whether we can or cannot give up the low levels.

Note that we never question the need for low-level elements in the case of language. We may well *think* of the various aspects of a story separately; but we all agree that, if we want to retain the freedom to express any idea, we must start with the low-level elements of each aspect – and, moreover, with elements that can be *shared* by these aspects. Only words fulfil both requirements. No one would seriously claim that there exist methods or devices which enable us to start with ready-made sentences and still express any idea.

3

File maintenance was only an example, of course. A major application comprises *thousands* of aspects, most of them more involved than a database or display operation. Besides, we seldom encounter situations where only two aspects interact, as in our simplified file maintenance example. Even there, to discuss realistic situations we had to consider, in addition to the database and display operations, various business practices. These practices are themselves aspects of the application, so they add to the number of structures that must interact. We saw this, for instance, when I mentioned the small piece of software that accesses records and fields, performs calculations and comparisons, and displays values: a small element comprising just a few statements must be shared, nevertheless, by several processes – database, display, and one or more business practices – because this is the only way to implement an operation that involves these processes.

It is hardly necessary, therefore, to demonstrate the need for low levels in real applications, in situations involving thousands of aspects, after showing the need for them even in situations with two aspects. Rather, what I want to show is how the mechanistic fallacies, and the software delusions they engender, lead to software charlatanism. All forms of software exploitation are based, ultimately, on the delusion of high levels that we have just examined.

❖

The deception starts when we are offered some software that promises to enhance our capabilities; namely, software that will allow us to accomplish tasks requiring knowledge that we, in fact, lack. We are promised, in other words, that simply by operating a software device we will be in the same position as those whose skills and experience exceed ours. The promise, it must be emphasized, is not that we will quickly acquire the missing knowledge. On the contrary, the promise is specifically that we don't need to learn anything new: the power to perform those tasks resides in the device itself, so all we need to know is how to operate it.

As programmers, we are offered various tools, development environments, and database systems. We are told that these devices will enable us to create, quickly and easily, applications which otherwise would take us a long time to program, or which are too difficult for us to program at all. The promise, therefore, is that these devices will function as substitutes for programming expertise: through them, we will achieve the same results as programmers who have been developing and maintaining applications successfully for many years.

As users, we are offered various productivity systems, or office systems. We are told that these devices will solve our business problems directly, eliminating the need for programming. Or, we are offered ready-made applications or pieces of applications, and we are told that they will enable us to manage our business just as we would with custom applications created specially for us.

For programmers as for users, the promises are supported with the explanation that the software devices offer higher levels of abstraction: they simplify development by allowing us to start from higher-level elements, bypassing the difficult and time-consuming task of creating the lower levels. The higher-level elements appear in a variety of forms, but, essentially, they are built-in operations or ready-made pieces of software.

No matter what form the higher levels take, the underlying assumption is the same: the work involved in creating a software application is similar to a manufacturing project, so the application can be seen as a neat structure of things within things – parts, modules, subassemblies. Thus, as in manufacturing, the larger the building blocks, the faster we will complete the project. We should avoid programming, therefore, and start instead with the largest modules and subassemblies available: software entities that already contain the lower-level parts. The use of large building blocks benefits us in two ways: by speeding up the software manufacturing process, and by demanding lower skills. Less time, less knowledge, and less experience are needed to assemble a software structure from modules, than to design and build it

from basic components. In the extreme case of ready-made applications, the manufacturing process is eliminated altogether: the starting level is then the top element itself, the complete application.

If software exploitation begins with the lure of high levels, the next stage is, needless to say, the disappointment. As programmers, we still cannot create complex and reliable applications; as users, we still cannot manage our affairs as we hoped. The software devices do provide the promised higher levels, and we can perhaps accomplish some tasks that we could not have accomplished without them. What we find, rather, is that the higher levels are rarely beneficial. If we want to start from higher levels, we must give up the flexibility afforded by the low levels. If we want the benefits of built-in operations and ready-made modules, of less work and easier challenges, we must be content with a fraction of the alternatives otherwise possible. Unfortunately, only rarely is this practical: only rarely can we restrict our affairs to the few alternatives provided by the software devices. The greater the promised benefits, the higher must be the starting levels, and the more severe the reduction in alternatives. The deception, thus, consists in promoting the benefits of higher starting levels while masking their concomitant drawbacks.

We adopt these devices and become dependent on them because we are seduced by slogans like "powerful" and "easy to use." We fail to see that these two qualities are contradictory: easy-to-use devices can be powerful only if we redefine power to mean, not the ability to implement *any* operations, but the ability to implement *some* operations easily. Clearly, if ease of use is claimed, all the power must inhere in the devices themselves, in their built-in capabilities. This means that they may perform well those operations that are built in, but they cannot perform other operations at all; and no device can have all conceivable operations built in.

The only way to implement *any* operations that may be required is by starting with low-level elements. So the software charlatans must provide the low levels if we are to use their devices at all. Their challenge, therefore, is how to reinstate the low levels, and how to make us start from these low levels, while we continue to believe that we are working at high levels. And they do it by implementing the low-level features *within* the high-level environment, as *extensions* to the high-level operations.

The low levels were always available to us – in the form of traditional programming languages, for example. So, if low levels are what we need, there is nothing the elites can give us that we did not have all along. The theories and methodologies, the programming tools and fourth-generation languages, the database and reporting systems, serve in reality the same purpose: they provide some of the low-level elements we need, and the means to link software structures, while pretending to be high-level environments.

The third stage in the process of exploitation, then, is the reinstatement of the low levels. To make their devices useful, the elites must restore the very concept that the devices were meant to supersede. Any device that does *not* provide this functionality is eventually abandoned and forgotten, even by the naive people who believed the original claims, simply because it is useless. (The full-fledged CASE environments, which actually tried to materialize the fantasy of creating entire applications "without writing a single line of code," are an example.)

We will waste no time, thus, examining the devices that do *not* restore the low levels. Let us treat them simply as fraudulent products, no different from the other forms of deception employed by charlatans to exploit gullible people – weight-loss contraptions, back-pain remedies, money-making schemes, and the like. As explained earlier, it is not the traditional means of exploitation, but the new form of domination, that concerns us: the use of software to consume our time and prevent us from gaining knowledge and experience. Eliminating the low levels and then restoring them in a different and more complicated form is an important factor in this domination, as it permits the elites to destroy software freedom and to establish the dependence on their devices. And we are fooled by these charlatans because the devices are based on software theories invented in universities, and described as "scientific."

Recall the principles of demarcation between science and pseudoscience, which we studied in chapter 3. Mechanistic software theories claim that we can create applications by starting with high-level elements. So, when this idea proves to be worthless and the charlatans "enhance" their devices by restoring the low-level capabilities, what they do in reality is turn the falsifications of those theories into new features. And this, we saw, is the stratagem through which fallacious theories are rescued from refutation. Thus, mechanistic software theories are intrinsically pseudoscientific.

Here are examples of software devices that were enhanced by restoring the low levels: The so-called fourth-generation languages started by promising us a higher level than the traditional, third-generation languages; but the only way to make them practical was by restoring, one by one, the features found in the traditional languages (loops, conditions, individual variables, etc.). The relational database systems started by claiming that the database can be treated as separate structures, interacting only at high levels with the other structures of the application; but they became practical only after adding countless new features, and whole programming languages, in order to restore the low-level links between these structures (for instance, the capability to access individual records directly, through file scanning loops). Systems like report writers and spreadsheets started by claiming that their high-level features are adequate for our requirements; but they ended up incorporating many traditional features,

and even programming languages, in order to provide the low-level operations needed in real-world situations.

To summarize, high-level environments that restore the low levels exploit us in two ways. First, we get to depend on some new and complicated programming methods, arising from the idea of using low-level elements as an extension to high-level ones. The traditional method – creating high-level elements from low-level ones – is simple and natural; it follows a concept we all understand intuitively, and confers complete programming freedom. The new methods, on the other hand, are contrived – absurd and unnecessary; their purpose is to maintain the illusion of high levels, and to induce dependence on proprietary development systems. Second, these systems provide only *a few* of the low-level features available through the traditional methods, only the minimum necessary to fool us. So they remain, essentially, high-level environments, lacking the versatility of a general-purpose language. Each low-level feature is presented as a powerful enhancement, and this obscures the fact that these features are merely a more complicated version of features we always had – in the traditional programming languages.

4

The charlatans promise us power, but all they can give us is higher levels of abstraction. So, once they persuade us to adopt their devices, they must restore the essential low levels, while continuing to promote the devices as high-level environments. It is not too much to say, then, that the only real difference between these devices is how they mask the deception; namely, how they prevent us from noticing that we are working, in fact, at *low* levels.

It is precisely because their chief purpose is to deceive us – to persuade us that they possess some important qualities – that the devices end up so complicated and inefficient. The so-called non-procedural languages, for instance, are promoted with the claim that we only need to tell the computer now *what* to do, not *how* to do it. This sounds like a novel programming concept, as if we could almost talk to the computer and tell it what we need done. In reality, these languages merely incorporate a number of high-level elements in the form of built-in operations. And this concept is available in any programming language in the form of libraries of subroutines: ready-made functions providing levels of abstraction that are as high as we want.

But, whether we use subroutines or non-procedural languages, some of our starting elements must still be at low levels, because it is impossible to implement all conceivable requirements by relying entirely on ready-made, high-level elements. The non-procedural languages look impressive when all

we need is one of their built-in operations (these are the examples we see in textbooks and in advertisements, of course), but are more awkward than a traditional language in any other situation. This is true because, in order to make them appear as novel concepts, their authors must make them different from the traditional programming languages; and this also makes them more complicated.

It is common in these languages, for example, when what we need is not just one of the built-in operations, to find long and tangled statements. A procedure that in a traditional language involves conditions, loops, and the use of variables, may turn up in a non-procedural language, in an extreme case, as *one* statement. But, while being perhaps shorter than the procedure, the statement is not, in fact, a higher-level entity: since its clauses must be specified with precision and in detail, they do not constitute a higher level of abstraction.

Thus, in SQL (the most popular database language), we often see statements containing more than a dozen lines, when a number of related database operations must be specified together. These long statements can become extremely complicated, as the programmer is forced to cram into one expression a whole series of related operations. Instead of the familiar structure of loops and conditions found in traditional languages, and which an experienced programmer understands intuitively, we now have an artificial and unwieldy set of specifications. But because the definitions, loops, and conditions are no longer manifest, this complicated piece of software is unlike a traditional language, so we can delude ourselves that what we are doing is no longer programming: "we are only telling the computer what to do, not how to do it."

What we are telling the computer is, however, the same as before. First, the level of abstraction is about the same as in a traditional language (this, after all, is why we needed SQL, why we could not simply use the high-level relational operations). Second, the resulting statements are still the reflection of many structures, which interact and must be kept in the mind simultaneously. In other words, all the difficulties we had before are still there; and because we wanted to *avoid* programming, we must now cope with these difficulties through programming means that are more complicated and less efficient than the traditional ones. (We will examine the SQL fraud in "The Relational Database Model" in chapter 7; see pp. 808–815.)

Let us look at another concept that promises higher levels and, instead, makes programming more complicated. This concept is based on the belief that specifying an operation by selecting it from a list of options, rather than by typing a command or a statement, represents a higher level of abstraction.

Most development environments have features based on this delusion. Like the non-procedural languages, creating applications by selecting things from lists is seen as a novel, high-level concept: all we do now, it seems, is tell the computer what we need, and it automatically generates pieces of software for us, even the entire application.

In reality, whether we select options or write statements, our starting elements must be a combination of high-level operations and low-level ones. Thus, even when we communicate with the system by selecting built-in operations, we must create the application's structures – its unique processes, or aspects – and the links between structures. For, if this were not the case, if our contribution were limited to making selections, the only applications we could create would be random and useless combinations of built-in operations.

With a traditional language, we tell the computer what to do by formulating statements, definitions, and expressions. With the new concept, we are shown lists of options, and options within options, and we tell the computer what to do by selecting entries from these lists. This creates the illusion that we are not programming, that all we must know is how to *select* things. What we must know, though, is the same as before; only the way we apply this knowledge is different.

In principle, one can specify anything by selecting options, but only with trivial requirements is this method more expedient than typing statements. The devices are promoted, however, for *all* applications. Clearly, no one would adopt them if told that they are meant only for novices, or only for solving simple and isolated problems. Thus, the devices must maintain the illusion that, no matter how complex the requirements, all we ever do is make selections; and this is why they end up making programming *more* difficult. But if we believe that one can accomplish more with these devices than without them, we will agree to perform any acts, no matter how illogical, just so that we can use them.

We encounter this delusion, for instance, in the development environments called *visual*, and in those called *by example*. Thus, the concept *query by example* claims to give users the means to perform certain database operations without programming. The concept sounds as if, instead of formulating queries, all we had to do now is show the system some examples of what we need. In reality, since there is no way for a database system to know what we need without being given precise information, we must provide the same specifications as before; and, because we wanted to avoid the traditional method of formulating queries, we end up with a more complicated one.

Thus, to tell the system which records to read, instead of expressing through a traditional language a condition like "products with price less than 100 and quantity in stock greater than 10," we must perform a number of selections: we

select from a list of files "product," then from a list of fields "price," then from a list of relational operators "less," then the value 100, then three more selections for "quantity," "greater," and 10, and finally, from a list of logical operators, "and." Even such trivial acts as the entry of a numeric value like 100 can be reduced to a process of selections: we have all seen systems where a number is displayed for us, and we are expected to increment or decrement it with the mouse until it reaches the desired value. This method takes longer than simply typing the value, but it is an important part in the delusion of high levels: we are now only *selecting* a value, not *specifying* it.

It ought to be obvious that in order to select the right field, operation, or value we must know what these notions are, must appreciate the consequences of each selection and of our particular combination of selections, must understand the significance of operations like "less" or "and" when applied to database fields and records, and so on. Also, the query is meaningless as an isolated function; it is part of an application, so we must be aware at the same time of the application's other aspects, and of the other uses of those files, records, and fields. In other words, to select the right things we must deal with details and with interacting structures, so the new method does not represent a higher level of abstraction: we must have almost the same programming skills as when specifying those things with statements.

The knowledge that is no longer required – remembering what operations are available, for instance, or the correct format of a statement – is the *easy* part of programming, the *mechanistic* knowledge. These devices impress ignorant practitioners, who lack even this basic knowledge (and are unaware of the required *complex* knowledge lying beyond it), and who, therefore, believe that a substitute for *it* is all they will ever need in order to create applications. Experienced programmers refuse to use these devices, not because they cling to the old methods, as the propaganda tells us, but because they recognize how insignificant their benefits are.

The devices, thus, introduce elaborate procedures as a substitute for the *simple* knowledge involved in programming, but they cannot replace the difficult, *complex* knowledge, which can only develop through personal experience. The immensity of the environment, and the endless novelties that must be assimilated in order to use it, mask the fact that it is still our own skills, not the device, that solve the difficult programming problems. In the end, all the device does is sit between us and our applications, forcing us to express our requirements in more complicated ways than we would through traditional programming. Moreover, despite its low-level features, the device still prevents us from implementing all conceivable alternatives.

5

The most flagrant manifestation of software mechanism, thus, is the obsession with ways to avoid programming. Serious programming is indeed a difficult pursuit, but so are other professions. And it is only in programming that the main preoccupation of practitioners has become the *avoidance* of the knowledge, skills, and activities that define their profession. The ignorance pervading the world of programming is so great that the obsession with ways to avoid programming forms its very ideology. The irrationality of this obsession can be observed in this strange phenomenon: as programmers and managers are taught that programming must be avoided at all costs, they end up accepting with enthusiasm any theory or system that claims to eliminate the need for programming, even when this makes application development *more* difficult.

It is important to remember the origin of this stupidity: our mechanistic culture, and the software delusions it has engendered. For, only if we perceive software applications as mechanistic phenomena will we attempt to break down applications into independent structures and to start from higher-level elements; and only then will we accept the software devices that promise to help us in these attempts. The devices, we saw, provide higher starting levels for isolated software structures. The development environments through which they do it, no matter how novel or sophisticated, serve only to deceive us, to prevent us from noticing that all we are getting is some built-in operations. So the higher levels are nothing but a proprietary implementation of a simple and well-known programming concept – subroutines.

Were they not blinded by their mechanistic delusions, software practitioners would easily recognize that the programming aids are only replacing their simple, mechanistic activities, and that successful application development entails *non-mechanistic* knowledge. One can attain non-mechanistic knowledge only through personal experience. Thus, as long as they are guided by mechanistic beliefs and seek progress through programming substitutes, the software practitioners deprive themselves of the opportunity to gain this experience. They are trapped, therefore, in a vicious circle: the only knowledge they believe to be required is the mechanistic knowledge they are trying to replace with devices; consequently, they interpret each disappointment, not as evidence of the need for additional, non-mechanistic knowledge, but as a shortcoming of the particular device they are using; so, instead of gaining the additional knowledge through programming, they merely look for another device, and repeat the whole process in a slightly different way.

It is worth repeating these facts, because they are perhaps not as obvious as

they appear here. How else can we explain the failure of society to notice the incompetence of our programmers? Endless justifications are being suggested to explain why we must disregard, in the case of programmers, notions that we accept implicitly in any other profession; particularly, the need for personal experience in the tasks defining the profession. For programmers, we have redefined the idea of experience to mean experience in using substitutes for experience.

And so it is how the delusion of software mechanism has given rise to that famous phrase, "without writing a single line of code." When referring to a programming substitute, this phrase is a promise; namely, that the device will permit us to create applications, or pieces of applications, without any programming. This promise is seen as the most desirable quality of a software device, and software companies will do almost anything in order to realize it – even invent, as we saw previously, devices that make application development more difficult. What matters is only the claim that we no longer have to "write code" (write, that is, statements or instructions).

It is not surprising, of course, to see this phrase employed for devices addressing software *users* – office workers, managers, amateur developers, and the like. Since no device can allow someone without programming knowledge to perform tasks requiring programming, the claim is a fraud. But we can understand the *wish* of naive people to have such a device, and consequently their exploitation by charlatans. What is surprising is to see the same phrase employed for devices addressing *programmers* – those individuals whom one would expect to possess programming expertise (and hence to have no use for such devices), to be proud of their programming capabilities, and even to enjoy programming.

The fact that the software charlatans employ the same means of deception in both cases ought to draw attention to the absurdity of our software culture: individuals whom we all consider professional programmers have in reality about the same knowledge, ambitions, and expectations as average computer users; like mere users, their chief preoccupation is to improve, not their *programming* skills, but their skills in *avoiding* programming.

And it is not just the software companies that foster these delusions. Researchers in universities participate by inventing mechanistic software theories, the business media by promoting worthless software concepts, corporations by employing programmers who rely on aids and substitutes, governments by permitting the software bureaucracy to exploit society, and in the end, each one of us by accepting this corruption. For, simply by doing nothing, by continuing to worship the software elites and to depend on the software bureaucrats, we are in effect supporting them. The cost of the mechanistic software delusions (probably exceeding one trillion dollars a year

globally) is passed in the end to society, to all of us. So, just by doing nothing, we are in effect paying them, each one of us, thousands of dollars every year, and helping them in this way to increase their domination.

6

In chapter 4 we discussed Jonathan Swift's criticism of the mechanistic ideology that was sweeping the scientific world at the end of the seventeenth century; in particular, his attack on the mechanistic language theories (see pp. 317–318). The idea that there is a one-to-one correspondence between language and knowledge, and the idea that languages can be studied, designed, and improved as we do machines, were seen in Swift's time as a foregone conclusion, and were defended by pointing to the successes of mechanism in the natural sciences. Thus, even though the mechanistic theories of language were mere speculations, most scientists were taking them seriously. To ridicule these beliefs, Swift has his hero, Gulliver, describe to us the language machine invented by a professor at the Grand Academy of Lagado.[2]

The machine is a mechanical device that contains all the words of the English language, and their inflections. By manipulating a number of cranks, the operator can instruct the machine to generate random combinations of words. And by selecting those combinations that constitute valid phrases and sentences, the professor explains, any person intelligent enough to operate the machine – intelligent enough, that is, to turn the cranks – can produce any text in a particular field of knowledge. Thus, a person with no knowledge of philosophy, or history, or law, or mathematics, can now write entire books on these subjects simply by operating the machine.

The professor emphasizes that his invention is not meant to help a person acquire new knowledge, but on the contrary, to enable "the most ignorant person" to write in any field "without the least assistance from genius or study."[3] The machine, thus, will allow an ignorant person to generate any text without having to know anything he does not already know. And this is possible because the person will generate the text (as we say today in programming) "without writing a single line."

Now, one could certainly build such a machine, even with the mechanical means available in the seventeenth century. Swift is not mocking the technical aspects of the project, but the belief that the difficulty of developing ideas is the mechanical difficulty of combining words. If we hold this belief, we will

[2] Jonathan Swift, *Gulliver's Travels and Other Writings* (New York: Bantam Books, 1981), pp. 180–183. [3] Ibid., p. 181.

inevitably conclude that a machine that helps us to manipulate words will permit us to perform the same tasks as individuals who possess knowledge, talent, and experience.

It is obvious that the quality of the discourse generated by a language machine depends entirely on the knowledge of the operator. The machine can indeed produce any text and any ideas, but only by randomly generating all possible combinations of words. So, in the end, it is still the human operator that must decide which combinations constitute intelligent sentences and ideas. Although it appears that the machine is doing all the work and the person is merely operating it, in reality the machine is replacing only the *mechanical* aspects of language and creativity.

Thus, a person using the machine will not accomplish anything that he could not accomplish on his own, simply by writing. Now, however, since he is only *selecting* things, it can be said that he is generating ideas "without writing a single line." Whatever the level of intelligence of a person, it is in fact more difficult to generate a piece of text by operating this machine than by directly writing the text. But if we believe that it is the *mechanical* acts involved in writing that make writing difficult, or if we have to employ as writers individuals known to be incapable of writing, we might just decide that language machines make sense.

Returning to our software delusions, we indeed believe that the difficulty of programming lies in its *mechanical* aspects, in combining pieces of software; and, what is worse, we indeed have to employ as programmers individuals known to be incapable of programming. So we have decided that *programming* machines make sense.

The similarity between Swift's hypothetical language aid and our real *programming* aids is striking. We note, in both cases, devices that address ignorant people; assure them that they don't need to know anything they don't already know; promise them the power to perform tasks that require, in fact, much knowledge; and reduce their involvement to a series of selections.

The similarity is not accidental, of course. We already know that our software delusions and our language delusions stem from the same belief; namely, the belief that the elements of software structures and language structures correspond on a one-to-one basis to the elements that make up reality. So we must not be surprised that devices based on *software* delusions end up just like the device invented by a satirist to mock the *language* delusions.

Swift was trying to demonstrate the absurdity of the mechanistic language theories by exposing their connection to the belief that a mechanical device can replace human knowledge. But today, through the mechanistic *software* theories, we are actually attempting to realize this fantasy: we are building *software* devices to replace human knowledge – programming knowledge, in

particular. Concepts that were only academic speculations in Swift's time, easily ridiculed, have become a reality in our time in the world of software. The kind of device that three centuries ago was only a fantasy – a satirical exaggeration of a delusion – is actually being built today by software companies, and is being used by millions of people.

❖

There is no better way to illustrate the essence of software charlatanism than by imagining how the professor from Lagado would design his language machine today. He would make it a software device, of course, rather than a mechanical one. And, as a matter of fact, it is quite easy to design a software system that allows anyone – including persons who are normally unable to express themselves – to produce books in any domain "without writing a single line." To imagine this device, all we have to do is combine the concepts implemented in Swift's language machine with those implemented in our software systems.

The promise, thus, would be the familiar claim that the only thing we need to know is how to operate the device, and that this knowledge can be acquired in a short time. To operate the mechanical language machine, all they did was turn cranks; to operate a modern language machine, all we would do is "point and click" with a mouse. We are assured, in both cases, that the power of the device is ours to enjoy "at a reasonable charge, and with a little bodily labour,"[4] and only by making selections: we would never have to write a single sentence. The phrase we would use today is "powerful yet easy to use."

Let us examine some of the possibilities. Instead of typing words, we can have the system display them for us in the form of selections within selections. If, for example, we need the sentence "the dog runs," we first select the grammatical function by clicking on *noun*; this displays a list of noun categories, and we select *animal*; within this category we select *domestic*, and finally *dog*; what is left is to click on *singular* and *definite article*. Then, for "runs" we select the grammatical function *verb*, which displays a list of verb categories; we select *action*, within this category we select *motion*, and finally *run*; we then click on *present tense*, *third person*, and *singular*, and the complete sentence is displayed for us.

The popular expedient of *icons* could be profitably employed to help even illiterate persons to use this system: if tiny pictures were used to depict words, categories, and grammatical functions (a picture of a dog for "dog," an animal together with a man for "domestic," a running figure for "run," one and two

[4] Ibid.

objects for "singular" and "plural," etc.), even those of us who never learned to read and write could benefit from the power of language machines.

It is obvious that, with such a system, anyone could generate text on any subject without writing a single line. And future versions could introduce even more powerful features – built-in sentences, for instance. Instead of words, we would be able to select entire sentences, and even entire paragraphs, from lists of alternatives representing classes and categories of topics. With only a little practice, anyone would then be able to generate page after page of exquisite text just by pointing and clicking.

The more elaborate this imaginary language system becomes, the easier it is to recognize its similarity to our software systems – our programming aids, in particular. But, while few people would be deceived by a language machine, the whole world is being deceived by the software charlatans and their application development machines. Not even illiterates could be persuaded to try a device that promises to replace writing skills. But the most important individuals in society – decision makers working in universities, corporations, and governments – keep trying one software theory after another, and one programming substitute after another, convinced that a device can replace *programming* skills.

Recall also our discussion in "The Software Theories" in chapter 3. Scientists believe that a device based on selections can replace human knowledge because they see intelligence and creativity, not as the indeterministic phenomena they are, but as the process of selecting a particular mental act from a predetermined range of alternatives. So they conclude that it is possible to *account* for human knowledge. Specifically, they claim that high-level forms of intelligence can be described with mathematical precision as a function of some low-level mental elements: all grammatically correct sentences that a person can utter can be predicted from the individual words, all behaviour patterns can be explained as a combination of some simple bits of behaviour, and all social customs can be described in terms of some basic human propensities.

This idea leads to the belief that we can incorporate in a device – in a software system, for instance – the low-level elements, and the methods used to derive from them the high-level ones. The device would then be a substitute for intelligence: by selecting and combining the high-level elements generated by the device, anyone would be able to perform the same tasks as a person who generates *in his mind* high-level elements starting with low-level ones. The device would replace, in effect, the experience of a person who took the time to develop whole knowledge structures, starting from low levels.

The fallacy, we saw, lies in the belief that the alternatives created when starting with high-level elements are about the same as those possible when starting from low levels. In reality, we would be limited to a small fraction of the possible alternatives. The impoverishment is caused by abstraction, but also by reification, because when we lose the low levels we also lose the links between the particular knowledge structure that is our immediate concern, and all the other structures present in the mind. This impoverishment explains why mechanistic theories of mind can represent only *some* aspects of human intelligence, and why ignorant persons equipped with software devices can accomplish only *some* of the tasks that experienced persons can with their minds alone.

7

Whether addressing programmers or software users, an *honest* development system simply provides low-level elements and the means to combine them so as to create the higher levels. The low levels come (for programmers, at least) in the form of general-purpose programming languages; and, when practical, higher levels are available through existing subroutines. Systems that provide *only* high levels, and claim that it is possible to create any application in this manner, are dishonest: they invariably end up reinstating the low levels in a different, and more complicated, form. These systems are for programming what language machines are for writing: not useful tools, but means of deception and exploitation. Their purpose, we saw, is to induce ignorance and dependence, by consuming our time and preventing us from improving our skills.

Honest systems allow us to create the higher levels on our own, and to select any subroutines we like. With honest systems, therefore, we can choose any combination of low-level elements and built-in operations. Dishonest systems provide an environment with high starting levels, and add the low levels as a special feature. The software charlatans have reversed, in effect, the principles of programming: instead of a simple system based on low levels, where we can create the high levels independently, they give us a complicated environment based on high levels, where the low levels are provided as "enhancements." What we had all along in any programming language – the low levels – is presented now as a new and powerful feature of their high-level environment. Instead of programming being the standard development method, and the high levels a natural outcome, they make the high levels the standard method, and turn programming into a complicated extension.

Clearly, if we use a general-purpose development system, if we want to

create original applications, and if these applications require a particular level of detail and functionality, our lowest-level elements must be the same mixture of variables, conditions, loops, and statements no matter what development method we use.

The software charlatans prefer environments based on high levels because this is how they can induce dependence. A system based on low levels and subroutines leaves us free to develop and maintain our applications in any way we like. The dishonest systems lure us with the promise of high starting levels, but must include the low levels anyway. They lose, therefore, the only benefit they could offer us. But, because we trusted them and based our applications on their high levels, we will now depend forever on them and on the software companies behind them. While no dependence is possible when using traditional development methods, it is quite easy to control our work, our time, our knowledge, and our expectations through systems based on high levels. For, instead of simply developing applications and expanding our programming skills, we are now forced to spend most of our time with the problems generated by the systems themselves, with complicated concepts, with special languages, and with their endless changes.

The only time a high-level system is justified is when its functions cannot be effectively implemented as subroutines. This is the case, typically, in systems meant for highly specialized applications. Thus, operations involving indexed data files can be added as subroutines to any language. They are more convenient when implemented in the form of statements (as in COBOL), but it would be silly to adopt a new language, or a whole development environment, just for this reason. On the other hand, the features found in an advanced file editing system cannot be simply added to a language as subroutines, because, by its very nature, the editing system must have its own environment (windows, commands, special use of the keyboard, etc.). And what an honest system does, in this case, is make it as easy as possible to transfer the files to and from other systems.

It is worth repeating here that "subroutine" refers to a broad range of high-level software elements, including functions, procedures, subprograms, and the like, which may be explicit or implicit. This term refers, thus, to any elements that can be implemented as a *natural extension* of a general-purpose programming language. The subroutines that perform file operations, for example, are implemented by way of functions in a language like C, but we see them as ordinary statements in a language like COBOL. The important point is that the foundation of the application be a general-purpose language, not the high-level

entities of a development environment. The level of this language may vary, depending on the application; thus, parts of the application, if restricted to narrow, specific domains, can often be developed in a higher-level language.

And I refer to individual statements, conditions, iterations, etc., as "low-level" software elements only because they are lower than subroutines, or built-in operations, or the high-level functions provided by development environments. But these "low-level" elements are what we find, in fact, in general-purpose languages (like COBOL and C) called "high-level" (to distinguish them from assembly languages, which use true low-level elements).

This confusion in terminology is due to the software mechanists, who have distorted the meaning of low and high levels by claiming that it is possible to raise forever the level of the starting elements. Thus, the term "fourth generation" (4GL) was coined for the languages provided by development environments, and "third generation" for the traditional high-level languages, in order to make environments look like an inevitable evolution. Assembly languages were declared at the same time to be "second generation," and machine languages, which use even lower-level elements, "first generation."

The level of these languages, however, has little to do with an advance in programming concepts. Thus, the first three "generations" are still in use today, and will continue to be, because the lower levels are the only way to implement certain types of operations. It is true that, historically, we started with low-level languages and only later invented the high-level ones; but this doesn't prove that there can exist general-purpose languages of even higher levels. And it is true that, in most programming tasks, we were able to replace low-level languages with high-level ones without reducing the functionality of the resulting applications; but it doesn't follow that we can repeat this success, that we can develop the same applications starting from even higher levels.

Everyone agrees that it is more efficient to start from higher levels, and that we should use the highest-level entities that are practical in a given situation. But, as we saw earlier, for typical business applications this level cannot be higher than what has been called third generation. Consequently, the fourth generation is not, relative to the third, what the third is relative to the second. (Thus, the most advanced features that can be added naturally to a second-generation language will not turn it into a third-generation one; but most features found in fourth-generation languages can be added naturally to any third-generation one.) While we may agree that the first three generations represent a certain progression in programming concepts, the fourth one is a fraud. Not coincidentally, it was only when the fourth one was introduced that the term "generation" was coined; formerly we simply had "low-level" and "high-level" languages.

It is precisely because no further "generations" are possible beyond the

third one (in the case of general-purpose languages and general business applications) that the software mechanists were compelled to reverse the principles of programming; that is, to provide low levels within a high-level environment, instead of the other way around. To put it differently, the only way to make a 4GL system practical is by reinstating the traditional, third-generation concepts; but, to maintain the illusion of higher starting levels, the software companies must provide these concepts from within the 4GL environment.

A fourth-generation language, in the final analysis, is merely a third-generation language (assignments, iterations, conditions, etc.) plus some higher-level features (for display, reporting, etc.), bundled together in a complicated development environment. A programmer can enjoy the same blend of low and high levels by starting with traditional languages (COBOL, C, etc.) and adding subroutines and similar features, created by himself or by others.[5]

8

As an example of development environments, let us examine the communications systems. If what we need in our business applications is high-level operations in the domain of communications (say, transferring data under various protocols between computers, or converting files from one format to another), nothing could be simpler than providing these operations in the form of subroutines. We could then develop the applications in any programming language we like, and invoke these operations simply by specifying a number of parameters.

Needless to say, this is *not* how the popular communications systems make their operations available. What programmers are offered is a whole environment, where the operations are invoked interactively. Then, because the interactive method is impractical when the operations must be part of an application, these systems also provide a "powerful feature": a programming language. (To further distract us, euphemisms like "scripts," "macros," or "command files" are employed to describe the resulting programs.) In short, we are taken back to the lower levels of traditional programming. But we

[5] In forty years of programming – from simple utilities and applications to large data management systems and business systems – I have never encountered a situation where I could benefit from a commercial development environment. Even when the project calls for a higher-level programming method, I find it more expedient to implement my own, simple, customized environment (by means of "third-generation" and "second-generation" languages) than to depend on those monstrous systems sold by software companies.

already had programming languages; all we wanted was a few high-level communications operations. Instead, we must get involved with, assimilate, and then become dependent on, yet another system, another language, another software company, and the related documentation, newsletters, seminars, websites, version changes, bug reports, and so on.

Most of these activities are spurious, in that they are caused, not by the communications operations we needed, but by the environment we were forced to adopt in order to have these operations. And, what is worse, the languages that come with these environments are more primitive and less efficient than the general-purpose languages we already had. Only ignorant programmers, of course, can be deceived by this fraud; true professionals recognize that these systems are unnecessary, that their sole purpose is to prevent programming freedom. The popularity of development environments, and the ease with which practitioners can be persuaded to depend on them, demonstrates therefore the incompetence that pervades the world of programming. It is in the interest of the software companies to maintain this incompetence. Thus, by providing environments instead of honest development systems, they ensure that programmers waste their time with spurious activities instead of expanding their knowledge and experience.

As explained previously, programmers are deceived by the development environments because they trust the mechanistic software theories, which claim that it is possible to create applications by starting with high-level software entities. While this may work in narrow, specialized fields, or when the details are unimportant, it is rarely true for general business applications. Systems based on high levels are dishonest, therefore, because they make claims that cannot possibly be met.

Communications systems are only one kind of environment, of course. If we are to depend on development environments for our high-level operations, we will also need systems for display, for user interface, for database operations, for graphics, for reporting, for system management, etc. – each one with its own language, documentation, newsletters, seminars, bugs, changes, and so on.[6]

Development environments must include programming languages because their high-level operations, no matter how impressive they may be on their own, are only useful when combined with *other* operations. An application is not simply a series of high-level operations. The operations provided by one

[6] Thus, software reseller Programmer's Paradise boasts on its catalogue cover, "20,000+ software development tools" (for example, the issue Nov–Dec 2008). Perhaps 1 percent of them are genuine programming tools. The rest are environments and the endless aids needed to deal with the problems created by environments. Individuals who need such tools are not true programmers, but a kind of users: just as there are users of accounting systems and inventory management systems, *they* are users of development systems.

system are related to those provided by another, and also to the operations developed specifically for that application. The relations between these operations occur mainly at low levels, so they must be implemented through conditions, loops, statements, and variables; in other words, through the same low-level elements as those found in the traditional programming languages. Like the language machine we examined previously, the environments promise us high levels, but provide in reality the same mixture of levels we had all along. To develop a given application we need the same knowledge as before, but applying that knowledge is now much more difficult.

The complications created by this charlatanism are so great that a new kind of system had to be invented, whose only purpose is to help programmers and users connect the operations of the other systems or transfer data from one system to another; its only purpose, thus, is to solve the problems created by the idea of software environments. These new systems come, of course, with their own environments, languages, procedures, documentation, newsletters, seminars, bugs, changes, and so on. Another kind of system engendered by software charlatanism is the one meant to standardize the operations provided by other systems – to sit above them, as it were, and make their diverse operations available in a common format. Every software company tries to establish *its* system as the standard one, but this struggle merely results in even more facts, languages, procedures, documentation, reviews, etc., that programmers must assimilate.

These complications, to repeat, are a result of the reversal of programming principles: instead of starting with low-level elements and creating the higher levels freely, programmers are forced to develop applications starting with high-level elements. The low levels are then provided only *through* the development environments, and *through* the high levels, thus establishing the dependence.

A system based on low levels and subroutines *also* offers the benefits of high-level elements, and without inducing any dependence. After all, we already have many programming languages – languages better than those we must learn with each development environment; and through these languages, we can create software levels that are as low or as high as we want. Software companies do not promote environments because our general-purpose languages are inadequate, but because traditional concepts like subroutines would not allow them to control our work and our applications as do these environments. Were the high-level operations provided simply as subroutines, our general-purpose languages would provide everything we need to relate them and to create the higher levels. So instead of large software companies, and instead of our incessant preoccupation with their systems, we would simply have independent programmers giving us subroutines, and

independent programmers creating and maintaining applications. When we realize how much power the software companies attain through the concept of development environments, it is easy to understand why they like the mechanistic software theories: these theories provide the ideological justification for reversing the traditional programming principles, and hence for their environments.

❖

The ultimate consequence of programming incompetence, then, is the domination of society by the software elite. Programmers are expected to be mere bureaucrats, operators of software devices; so they are not accountable for their applications as other professionals are for their work. If the responsibility of programmers is limited to the use of development systems, it is, in effect, the software companies behind these systems that control the resulting applications.

Consider this language analogy: We can be praised or blamed for what we say because we are free to create any sentences and express any ideas. But if we lived in a society where sentences and ideas could only be produced with some language machines supplied by an elite, the conception of responsibility and knowledge would be different. If everyone believed that language machines are the only way to express ideas and to communicate, we would be judged, not by what we say, but by how skilled we are at operating the machines. Moreover, if the only thing we knew were how to operate these machines, there would be very little intelligent discourse in society. But that would be considered a normal state of affairs. In the end, the only knowledge possible would be the ready-made sentences and ideas built into these machines by the elite.

It is obvious that an elite could dominate society if it could prevent us from developing linguistic competence, and if it could consume our time with worthless linguistic theories and devices. We have no difficulty understanding this for language, but we are allowing it to happen through software. Software and language, though, fulfil similar functions. Thus, incompetence and charlatanism in software will have, in the end, the same consequences as they would in language. If we allow ignorance and exploitation in our software pursuits, by the time we depend on software as much as we depend today on language our society will be completely dominated by the software elite.

Through careful indoctrination, the knowledge of corporate managers is shaped to serve the interests of the software companies. They are encouraged, not to help their organizations use computers effectively, but on the contrary, to make the use of computers as complicated and expensive as possible, and to accept the dependence on software companies. This is accomplished by

promoting the mechanistic software ideology. An important factor in this ideology is the reliance on software devices – programming aids, development environments, ready-made pieces of software – in preference to the expertise and work of individuals. But software devices can only replace the *simple* aspects of programming. The complex problems remain unsolved, forcing everyone to search for newer devices, in a never-ending process. So the mechanistic ideology guarantees programming incompetence, and hence a perpetual preoccupation with software devices.

❖

To summarize, the only thing that software companies can give us is higher levels of abstraction for our software elements; and this is precisely what *cannot* help us. The higher levels come in the form of built-in operations that address isolated software structures (that is, individual aspects of an application). They cannot help us because these structures must interact at low levels, and when we start with high-level elements we can no longer implement the links between structures. Starting from high levels impoverishes the complex structure that is the application (by reducing the number of alternatives for the value of the top element). This impoverishment is caused both by abstraction (reducing the alternatives within each structure) and by reification (severing the links between structures). As a result, we can implement only a fraction of the possible combinations of elements, and our applications are not as useful as software can be.

Systems based on high starting levels can be beneficial for creating *simple* applications, especially if these applications are in narrow domains (statistics, for example, or text editing, or graphics), and if they are only weakly linked to other applications. They are useless for creating *general* applications, though, because in this case we cannot give up the lower levels. The mechanistic software theories, and the development environments based on them, assume that the various types of operations that make up an application (database, display, user interface, etc.) can be implemented as independent processes. But this is a fallacy, because most operations in an application must interact, and the interactions must take place at low levels. This is why any attempt to implement general applications through high-level systems leads in the end to the reinstatement of the low levels, in a more complicated way.

The conclusion must be that we don't need software companies. Practically all software supplied by these companies – development environments, system and database management tools, ready-made applications – is based on the notion of high levels. It is true that only large companies can create and support these complicated systems, but if high levels cannot help us, then these

systems, while impressive, are worthless. At the same time, the kinds of systems that *can* help us – customized applications, libraries of subroutines, simple tools based on low levels – are precisely what can be created by individual programmers.

This fact is no more a coincidence than the equivalent fact that language machines supplied by large companies could not replace the linguistic performance of individual persons. For software as for language, even the most sophisticated knowledge substitutes can replace just the simple, mechanistic aspects of knowledge. Only with our minds, through personal experience, and by starting from low levels of abstraction, can we hope to attain the complex knowledge needed to solve our problems.

The Delusion of Methodologies

1

So far we have discussed the development of applications mainly from the perspective of programmers. Let us see now how the mechanistic delusions affect the expectations of the *users* of applications – those individuals whose needs are to be embodied in the new software.

When developing a new application, managers familiar with the relevant business practices cooperate with analysts and programmers. The resulting software, thus, will reflect not only programming skills, but also the knowledge and experience of users. And the mechanistic theories and methodologies expect these individuals to express their knowledge and their requirements precisely, completely, and unambiguously; that is, to reduce knowledge and requirements to a form that can be used by analysts and programmers to develop the application. We will examine this absurdity in a moment, but first let us briefly discuss the alternative.

Instead of developing custom software, users can procure ready-made (or what is known as packaged, or canned) applications. With this alternative, the application is available immediately, thus bypassing the lengthy and difficult stages of design and programming. From what we have already discussed, though, it should be obvious that packaged applications are part of the same delusion as all ready-made, or built-in, pieces of software: the delusion of high levels, the belief that one can accomplish the same tasks by starting from high-level software elements as when starting from low-level ones. This delusion finds its ultimate expression in the idea of ready-made applications: the starting level is then the top element itself, the complete application, and the impoverishment is total. From the infinity of alternatives possible for the top

element when *programming* the application, we are now left with only one alternative: the particular combination of operations built into the package by its designers. Most packages include options, of course, for some of their built-in operations. But combinations of options still provide only a fraction of the combinations of operations that may be required by an organization. So, in the end, packages remain a poor substitute for custom applications.

Organizations are tempted by the promise of packaged applications because they underestimate the limitations they will face later, when they get to depend on this kind of software. And even when the users realize that the package permits them to implement only *some* operations, and addresses only *some* of their needs, they still fail to appreciate the real consequences of inflexible software. What they tend to forget is that their needs and practices evolve continually, so their software applications must evolve too. It is difficult enough to judge whether a certain application can answer our *current* needs (the only way to be absolutely sure is by running it live, by *depending* on it); but it is impossible to assess its usefulness for the next ten years, simply because we cannot know *what* our needs will be. No one can predict the changes that an organization will face in the future. How, then, can anyone expect a piece of software that is based on a particular combination of built-in processes and operations to cope with such changes?

Note that it is not the *quality* of the application that is at issue here: no matter how good and useful it is today, and even if the company supporting it will bring it up to date regularly in the future, it will always be a *generic* piece of software, designed to answer only that subset of needs common to many organizations; it cannot possibly adapt to the specific needs of every one of them.

It is quite incredible, thus, to see so many organizations depend on packaged software; and they do, not just for minor applications, but also for their important business needs. Most packages fail, of course, so we must not be surprised at the frequency with which these organizations try new ones. The failure of a package rarely manifests itself as major deficiencies, or as software defects. What we see typically is a failure to answer the needs of its users, something that may only become evident months or years after its adoption. Since this type of failure is so common, the reason why organizations continue to depend on packages is, clearly, not their usefulness, but the incompetence of the software practitioners: if programmers lack the skills necessary to create and maintain applications, ready-made software, however unsatisfactory, becomes an acceptable expedient.

More subtle and more harmful than the inadequacy of an application is the tendency of users to lower their expectations in order to match its limitations. In other words, instead of rejecting an inadequate application, they modify the

way they conduct their affairs so as to be able to use it. To help them rationalize this decision, the software elites stress the importance of adopting the latest "technologies" – relational databases, object-oriented environments, graphic user interface, client-server systems, and so forth. Ignorant users are impressed and intimidated by these concepts, so they end up interpreting the application's shortcomings as modern and sophisticated features which they don't yet appreciate. Thus, instead of objectively assessing the application's usefulness, they merely judge it by how closely it adheres to the software innovations promoted by the elites, even if these innovations are worthless. So, in the end, the application appears indeed to satisfy their requirements; but this is because they agreed to replace their true requirements with spurious ones.

2

Having established that packages are rarely a practical alternative for serious applications, let us return to the subject of software *development*. Developing their own applications is what many organizations must do, even if lacking the necessary skills, because this is the only way to have adequate software.

An application, we recall, consists of many structures, all sharing the same software entities (see "Software Structures" in chapter 4). These structures are the various *aspects* of the application – the processes implemented in it. Each structure, thus, is one way of viewing the application; and it is this system of interacting structures that constitutes the actual application. Although in our imagination we can treat each aspect as a separate structure, the only way to create the application is by dealing with several structures at the same time. This is true because most entities in a piece of software – most statements and modules – are affected by several aspects of the application, not just one. When writing a statement, for example, it is seldom sufficient to think of only *one* logical structure; we may well perceive a particular structure as the most important, but the same statement is usually an element in other structures too. It is this capability of software entities to be part of several structures simultaneously, and hence link them, that allows software applications to mirror our affairs. This capability is important because our affairs consist of processes and events that already form interacting structures.

If this is what software applications actually are, let us review what the software theories assume them to be. Applications, the theories tell us, must be developed following a *methodology*. Although many methodologies have been proposed, all are ultimately founded on the same fallacy; namely, the belief that it is possible to reduce a software application to a *definition*. The definition of an application is a set of specifications (formal descriptions, flowcharts, block

diagrams, and the like) believed to represent, *precisely and completely*, the actual software. Methodologies, thus, are a manifestation of the mechanistic belief – the belief that a complex structure (the software application, in this case) can be reduced to simple ones.

To define an application, users and analysts spend many hours discussing the requirements – the business practices that are to be embodied in the application. This activity is known as analysis and design, and the methodologies prescribe various steps, which, if rigorously followed, are said to result in a complete definition; namely, a definition that represents the application as precisely as drawings and specifications represent a house or a car. It is believed, thus, that a set of mechanistic expedients can capture all the knowledge inhering in a complex phenomenon: the structures that make up the application, their interactions, and their effects when the application is running.

The reason we start with a definition, of course, is that we prefer to work with specifications rather than the statements of a programming language. Deficiencies, for example, are easier to correct by modifying the definition than by rewriting software. Thus, we are told, if we follow the methodology, we should be able to create the entire application in the form of a definition, and then simply translate the definition into a programming language. To put this differently, the methodologies claim that it is possible to represent an application with expedients other than the software itself – expedients that are simpler than software, and accessible to users and programmers alike. Although simpler than the actual application, these expedients represent it completely and precisely. The definition *is*, in effect, the application.

The fallacy of this claim ought to be obvious: if it were possible to express by means of diagrams, flowcharts, etc., all the details of the application, we wouldn't need programming languages. For, a compiler could then translate the definition itself into the machine language, and we wouldn't need to write the programs. In reality, definitions are simpler than programs precisely because they do *not* include all the details that the programs ultimately will.

So definitions are *imprecise* and *incomplete* representations of the application. They are useful only because people can interpret them, because people can add some of their own knowledge when converting them into software. One reason why definitions are simpler than programs, thus, is that they need not be perfect. An error in the program can render the application useless, but in the definition it is harmless, and may even go unnoticed. The impossibility of translating automatically definitions into software proves that definitions are incomplete, faulty, and ambiguous, and require human minds to interpret and correct them.

But an even more important reason why definitions are simpler than

programs is that they represent *separately* the software structures that make up the application. The difficulty in programming, we saw, is dealing with several structures simultaneously. Our programming languages permit us to create software entities that can be shared by diverse structures, and this is why it is possible to develop useful applications. In a definition, on the other hand, we usually specify each structure separately: the business practices, the database relations and operations, the display and report layouts – we strive to represent each one of these processes clearly, so we separate them. Even if we *wanted* to relate them in the definition it would be difficult, because the diagrams, flowcharts, and descriptions we use in definitions are not as versatile as programming languages. Definitions are simpler than programs, thus, because most specifications do not share their elements, as software structures do. What this means is that a definition *cannot* represent the application precisely and completely. So the methodologies are wrong when claiming that definitions are important.

The fallacy of definitions is easy to understand if we recall the concept of simple and complex structures. A definition is, in effect, the reification of a complex structure (the application) into its constituent simple structures. It is, thus, an attempt to reduce a complex phenomenon to a mechanistic representation. This can be done, as we know, only when the separated structures can usefully approximate the actual phenomenon. In the case of software phenomena, this can be done for trivial requirements. For typical business applications, however, mechanistic approximations are rarely accurate enough to be useful. In the end, we like software definitions for the same reason we like all other mechanistic concepts: because of their promise to reduce complex problems to simple ones. Definitions are indeed simpler than the applications they represent, but they are simpler because they are only approximations.

Thus, since applications cannot be represented accurately by any means other than the programs themselves, the conclusion must be that definitions are generally irrelevant to application development. They may have their uses, but their importance is overrated. No definition can be complete and accurate, and an application created strictly from a definition is useless. Application development cannot be reduced to a formal activity, as the software theorists say. Since no one can specify or even envisage all the details, and since most details will change anyway (both before and after the application is completed), it is futile to seek a perfect set of specifications. Some brief and informal discussions with the users are all that an experienced programmer needs in order to develop and maintain an application.

The failure of the mechanistic concepts in the early days were so blatant that the software gurus had to modify their methodologies again and again. The invention of new methodologies, thus, became a regular spectacle in the world of programming, and there were eventually nearly as many methodologies as there were gurus. (Most methodologies are known by the name of their creators, a practice borrowed apparently from the world of fashion design.)

Some methodologies tried to eliminate the rigidity of the traditional development phases, and introduced notions like prototyping and stepwise refinements; others attempted to modify the traditional roles played by users, analysts, and programmers. But, in the end, no matter how different they may appear to the casual observer, all methodologies are alike. And they are alike because they all suffer from the same fallacy: the belief that indeterministic phenomena – the applications, and their development and use – can be treated as mechanistic processes. The idea of methodologies, thus, is just another manifestation of the belief that programming expertise can be replaced with some easy skills – the skills needed to follow rules and methods.

The similarity between the various methodologies is betrayed by the trivial innovations their creators introduce in an effort to differentiate themselves. For example, they use pretentious terms to describe what are in fact ordinary features, in order to make these features look like major advances. But most ludicrous is their preoccupation with the graphic symbols employed in diagrams, as if the depiction of processes, operations, and conditions with one symbol rather than another could significantly alter the outcome of a development project. For example, the traditional rectangular boxes are replaced with ovals, or with a shape resembling a cloud, or a bubble, or one known as a bubtangle (a rectangle with rounded corners). And we must remember that these idiocies are discussed with great seriousness in books and periodicals, and are taught in expensive courses attended by managers and analysts from the world's most prestigious corporations.

Programming methodologies, thus, are like the development environments we discussed previously: they provide elaborate systems to replace the *easy* aspects of programming, those parts demanding mechanistic knowledge; but they cannot replace what are the most important and the most difficult aspects, those parts demanding complex knowledge. Since the same knowledge is required of people to create a serious application whether or not they use a methodology, the methodologies, like the development environments, are in the end a fraud. They are another form of software exploitation, another way for the software elites to prevent expertise and to induce dependence on systems and devices which they control.

When a methodology appears successful, its contribution was in fact insignificant. For, why should some techniques that work for one organization

fail to work for others? It is the people, obviously, that made the difference. When people have the necessary knowledge, they will develop applications with or without a methodology; and when they lack this knowledge, no methodology can help them. Development environments, we saw, promise programmers and users higher levels of abstraction, and then trick them into working at low levels, as before. Similarly, methodologies promise them simpler, high-level concepts, and then demand the same skills as before. In both cases, this charlatanism complicates the development process, so inexperienced practitioners are even less likely to succeed. Besides, they waste their time now assimilating worthless concepts, instead of using it to improve their skills by creating and maintaining applications.

3

The delusion of methodologies and definitions is reflected in the distorted attitude that everyone has toward the subject of *maintenance*. Software maintenance is the ongoing programming work needed to keep an application up to date. And all studies agree that, for most business applications, this work over the years exceeds by far the work that went into the initial development. We should expect the theorists, therefore, to propose more solutions to the problems arising in maintenance than to those arising during development. What we find, though, is the exact opposite: all theories and methodologies deal with the creation of new applications, and barely mention the subject of maintenance. Moreover, we find the same distorted attitude among corporate managers: maintenance is treated as incidental work, is avoided whenever possible, and is relegated to the least experienced programmers.

In reality, the obsession with new applications is a reaction to the problem of programming incompetence: because programmers cannot keep the existing applications up to date, new ones must be developed. But without proper maintenance the new ones quickly fall behind, so the users find themselves in the same situation as before. At any given time, then, companies are either installing new applications, or struggling with the current ones and looking forward to replacing them. The software elites encourage this attitude, of course, as it enhances their domination. They present the latest fads – fourth-generation or object-oriented systems, CASE tools or relational databases, graphic interface or distributed computing – as revolutionary advances, and as solutions to the current problems. Their applications are inadequate, the companies are told, because based on old-fashioned software concepts. They must replace them with new ones, based on these advances.

So the preoccupation with new applications helps everyone to rationalize

the failure of maintenance. It takes great skills to modify a live application quickly and reliably: much programming experience, and a good understanding of the existing functions. In contrast, creating a new application from a definition, as the methodologies recommend, is relatively easy. It is easy because the neat definition is only a simplified version of the actual application. As we saw, definitions can only *approximate* the true, complex needs. But the belief that the next application can be precisely defined inspires everyone with confidence, so a new development project always looks like a wise decision.

To put this differently, practitioners prefer a new application to maintenance because new projects make self-deception possible. A methodology permits them to create, instead of the *required* application, an imaginary, simpler one: the application matching a neat definition and their limited skills. And when *that* application proves to be inadequate, the practitioners still do not suspect their practices. They blame the changing requirements, or the imperfection of the original specifications. They refuse to see these facts as a reality they must cope with, as the very essence of business software. So, instead of accepting the facts, they continue to claim that their practices are sound, and that precise definitions are possible. In other words, if reality does not match the mechanistic software principles, something is wrong with reality.

In new development projects, then, self-deception helps practitioners to deny their failures and to cling to the easy, mechanistic concepts. And they dislike maintenance because, in this type of work, self-deception cannot help them. Each maintenance project is relatively small and well-defined, so it is harder to replace it with an imaginary, simpler one. Ultimately, in maintenance work it is harder to find excuses for failures.

We note a marked discrepancy between the *perception* and the *reality* of applications. On the one hand, everyone strives to create a perfect application – by following a strict methodology, and by using the latest development systems. It is far more expensive to modify the software itself later, we are told, so we must eliminate the imperfections in the design stage. This is why definitions are important. On the other hand, all studies show that less than 5 percent of new applications are adequate. The others must be modified if they are to be used at all, and many are so different from the actual requirements that they must be abandoned. Moreover, even those that are adequate must immediately start a process of ongoing modifications, simply because business requirements change constantly.

Thus, whether it is the original differences (due largely to the fact that no definition can reflect the actual requirements) or the future ones (due to the

normal, unpredictable changes in requirements), it is obvious that modifying business applications is an essential programming activity. Yet, for over forty years, all theories and methodologies have been attempting to create "perfect" applications; that is, applications matching some fixed specifications, and requiring as few changes as possible. In reality, all software changes are alike – whether due to faulty specifications, or varying user preferences, or the need for additional features, or the adoption of new business rules, or some external factors. So, if we must be able to deal with endless changes in any case, the idea of a perfect application is meaningless, and there is no point in trying to design one initially.

It is wrong, in fact, even to think of maintenance as modifying the application. The role of business software is to satisfy, at any given time, the current needs. An application, therefore, must be seen as that particular software system which accomplishes this. Business needs change constantly, so the application must change too. Thus, rather than first developing an application and then maintaining it, it is better to think of this work as a continuous, never-ending development.

❖

We find further evidence of the distorted attitude toward maintenance in the notion of application *life cycles*. All experts agree that applications cannot last more than a few years. So, even while encouraging us to create a new one, they warn us to prepare for its demise. Borrowed from biology, the idea of life cycles holds that software resembles live things, so the existence of an application can be divided into stages: birth (definition of requirements), growth (development and testing), maturity (normal operation), and death (obsolescence). Each application represents a cycle, and is followed by another one, and then another one, forever.

But this is an absurd idea, contrived specifically in order to justify the need for new applications. Software, by its very nature, is modifiable. In principle, then, an application never needs to be replaced; it only needs to be kept up to date. Everyone acknowledges the need for changes, and acknowledges also the inability of programmers to implement them. So the idea of life cycles was introduced as a compromise: every few years, a new application is created in order to implement *together* all the changes that should have been implemented *one at a time* in the past. The theorists and the practitioners can now defend the lack of proper maintenance, and hence the need for a new application, by invoking the idea of software life cycles. This logic, however, is circular; for, the idea of life cycles was itself an invention, a response to the incompetence that prevents proper, *ongoing* maintenance.

Instead of trying to eradicate the incompetence, everyone looks for ways to rationalize it.

Business software can fulfil its promise only if it is as changeable as the business issues themselves: inflexible business software can be as bad as inflexible business practices. Thus, replacing the whole application from time to time is a poor substitute for the ability to satisfy new needs as soon as they arise. So the ultimate price we pay for distorting the subject of maintenance is having to depend on perpetually inadequate applications. This is true because, even though an inadequate application is eventually replaced, it reaches that condition gradually, one unsatisfied requirement at a time. This means that it was *always* inadequate, even in its period of normal use. The difference between that period and the time when it is actually replaced is only in the *degree* of inadequacy; namely, how far it is from the users' actual needs, how many unsatisfied requirements have accumulated to date.[1]

4

The delusion of methodologies and definitions is also demonstrated by the failure of CASE (Computer-Aided Software Engineering, see pp. 535–536). The elimination of programming from the process of application development was seen by most theorists as the undisputed next step in development tools, as the ultimate benefit of software engineering. Ambitious CASE systems were promoted for a number of years with the claim that managers and analysts could now create directly, without programming, applications of any complexity – simply by manipulating block diagrams, flowcharts, and the like, on a computer display. The system would guide them in creating the definition, and would then translate the definition automatically into software.

The belief that an application can be generated automatically is a logical consequence of the belief that a definition can represent all the knowledge embodied in an application. (Could definitions do that, automatic programming would indeed be possible.) The CASE fantasy, thus, was born from the

[1] The longest I maintained one of my applications is thirty years (until the manufacturing company using it ceased production). This was a complex, integrated business system, which combined all the computing needs of that company. At any given time there was a list of requirements, some of them urgent; but I always implemented them, so no one ever saw the need for new applications. The system kept growing, and was eventually a hundred times larger than it had been in the first year, due to countless new functions; but no one perceived these developments as new applications. Most work, though, was in modifying existing parts (replacing or adding features and details). Again, a properly maintained application never needs to be replaced, because it always has what the users need.

concepts of methodologies and definitions that we have just discussed – concepts which *continue* to dominate the programming theories, despite the failure of CASE. No one seems to realize that, if CASE evolved from these concepts, its failure proves the fallaciousness of these concepts too. Let us analyze this connection.

Even when following a methodology, people do more than implement rules and standards. The software created by programmers contains more than what the analysts specified in their definition, and the definition created by analysts contains more than what the users specified in their requirements. Each individual involved in the development of the application has the opportunity to add some personal knowledge to the project, but this is largely an unconscious act. Simply to *understand* a set of requirements or specifications, the person must *interpret* them; that is, he must combine the knowledge found in the document with some previous knowledge, present in his own mind. For, if this were not the case, if the only thing that analysts and programmers did were follow rules and methods, then a person who knows nothing about software or about a particular company, but who can follow rules and methods, could also develop applications.

The knowledge missing from the formal requirements and specifications, and hence contributed by individuals, varies from general facts on computers and software to details specific to their organization, and from common business practices to the knowledge shared by people living in a particular society. It is precisely because most people already possess this kind of knowledge that we take it for granted and do not include it in instructions and documents. Recall also that the most important part contributed by human minds constitutes *non-mechanistic* knowledge: not isolated knowledge structures, but the complex structure that is their totality. The capacity for non-mechanistic knowledge must be provided by human minds because it cannot exist in simple structures like instructions or diagrams.

Thus, all the people involved in the development of an application may be convinced that they are following the rules prescribed by the methodology, while depending on personal knowledge and experience to fill in the missing pieces, or to resolve the ambiguities and inconsistencies found in specifications. If the application is successful, they will praise the methodology, convinced that it was the principles of software engineering that led to their success. Most likely, they will not realize that it was in fact their own minds that provided the most important part (the non-mechanistic knowledge), and that the principles, theories, and methods addressed only the simple part (the mechanistic aspects of the project).

Clearly, if the methodology provides only mechanistic principles while our activities are mostly non-mechanistic, the only way to use a methodology is by

taking its practical parts and ignoring or overriding the rest. People may be convinced that they are *following* the methodology, when they are using it *selectively*. So it is not too much to say that, to develop an application successfully, people must work *against* the methodology: if they rigorously followed the mechanistic principles, they would never complete the application. Thus, when a software project is successful, this is not *due* to the methodology but *despite* it.

And it is during programming that people make the greatest contribution. For it is in programming, more than in any other activity, that people have the need and the opportunity to override the rules imposed by a mechanistic methodology. So it is the programmers – more than the managers with their specifications, or the analysts with their definitions – that must use the non-mechanistic capabilities of their minds. We can perhaps delude ourselves in the early stages of development that specifications and definitions represent the application completely and precisely. But if we want to have a useful application, we must permit human minds to deal at some point with the missing pieces, with the ambiguities, and with the inconsistencies. It is during programming, therefore – when the application is created and tested, when it must mirror reality if it is to be used at all – that the delusions of formal methodologies and precise definitions, of neat diagrams and flowcharts, must come to an end.

It should be obvious, then, why CASE failed. The CASE systems were based on methodologies: they literally incorporated some of the popular methodologies, thus allowing managers and analysts who wished to follow a particular methodology to do so through a software system rather than on their own. The system could now *force* people to follow the methodology, eliminating the temptation to omit or modify some of the steps – what was believed to be the chief cause of development failures. Since the methodology was now part of the development environment, the experts claimed, anyone could enjoy its benefits; and since the resulting specifications and definitions were stored in the computer, the system could use them to generate the application automatically, eliminating the programming phase altogether.

CASE failed because it eliminated the opportunities that people had to *override* the methodologies and definitions. By automating the development process, CASE made it impossible for people to contribute any knowledge that conflicted with the mechanistic software theories. They could only use now trivial, mechanistic knowledge, which is insufficient for developing serious applications. What CASE eliminated – what the software mechanists thought

was the cause of development failures – was in fact the very reason why methodologies and definitions appeared occasionally to work: the contribution made by people when, out of frustration, and perhaps unconsciously, they were using their non-mechanistic capabilities to override the methods, rules, and specifications. Thus, the failure of CASE proves that people normally contribute to the development process a kind of knowledge – non-mechanistic knowledge – that cannot be replaced with formal methodologies and theories.

There is another way to look at this. A CASE environment is logically equivalent to a traditional development environment where the users, the analysts, and the programmers follow a methodology *rigorously*; where analysis and design, specifications and definitions, theories of programming and testing, are all implemented exactly as dictated by the principles of software engineering; where everyone refrains from *interpreting* the specifications or the definitions; where no one uses personal knowledge to add details to the formal documents, or to resolve ambiguities and inconsistencies. A CASE environment is equivalent to all this because, when the methodologies and programming theories are part of the development system, people are *forced* to follow them rigorously.

Logically, then, the only difference between a CASE environment and a traditional environment is the non-mechanistic knowledge contributed by people – the knowledge that *cannot* be incorporated in a CASE system. So, if CASE failed, we must conclude that this knowledge plays a critical part in a development project. With traditional development methods, when people possess this knowledge the project is successful, and when they do not the project fails. In a CASE environment, people had no opportunity to use this knowledge, whether they possessed it or not; so the result was the same as when people used traditional development methods *and* lacked this knowledge. The promoters of CASE did not recognize the need for this knowledge. They believed that mechanistic knowledge suffices for developing applications; and, since mechanistic knowledge can be embodied in software devices, they believed that the contribution made by people can be reduced to the knowledge required to operate these devices.

The main purpose of this argument, you will recall, is not to show the absurdity of CASE, but to show how the failure of CASE demonstrates the fallaciousness of all methodologies and definitions – which, in turn, demonstrates the fallaciousness of all mechanistic software theories. For, it is software mechanism – the belief that applications consist of independent structures, which can be fully and precisely specified – that is the fundamental delusion. This delusion leads to the delusion that programming expertise can be replaced with rules and methods, which then leads to the notion of methodologies and definitions, and eventually to CASE. The CASE systems merely implemented

formally what the theories had been claiming all along, what practitioners had been trying before to do manually. So the only logical explanation for the failure of CASE is that these theories are invalid.