SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*
Section *Object-Oriented Programming*

This section analyzes the theory of object-oriented programming and its mechanistic fallacies, and shows that it is a pseudoscience.

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

www.softwareandmind.com

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

Published by Andsor Books, Toronto, Canada (www.andsorbooks.com)
First edition 2013. Revised 2019.

For disclaimers see pp. vii, 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

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

George Orwell, *Nineteen Eighty-Four*

Disclaimer

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

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

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

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

Contents

Preface

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

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

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

A second reason for the book's size is the detailed analysis of the various topics. This is necessary because most topics are new: they involve either

entirely new concepts, or the interpretation of concepts in ways that contradict the accepted views. Thorough and rigorous arguments are essential if the reader is to appreciate the significance of these concepts. Moreover, the book addresses a broad audience, people with different backgrounds and interests; so a safe assumption is that each reader needs detailed explanations in at least some areas.

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

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

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

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

Chapter 7, on software engineering, is not just for programmers. Many parts

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

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

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

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

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

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

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

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

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

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

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

Object-Oriented Programming

The Quest for Higher Levels

Mechanistic software theories attempt to improve programming productivity by raising the level of abstraction in software development; specifically, by introducing methods, languages, and systems where the starting elements are of a higher level than those found in the traditional programming languages. But the notion of higher starting levels is a delusion. It stems from the two mechanistic fallacies, reification and abstraction: the belief that we can separate the structures that make up a complex phenomenon, and the belief that we can represent a phenomenon accurately even while ignoring its low-level elements.

The similarity of software and language, we saw, can help us to understand this delusion. We cannot start from higher levels in software development for the same reason we cannot start with ready-made sentences in linguistic communication. In both cases, when we ignore the low levels we lose the ability to implement details and to link structures. The structures are the various *aspects* of an idea, or of a software application. In language, therefore, we must start with words, and create our own sentences, if we want to be able to express *any* idea; and in programming, we must start with the traditional software elements, and create our own constructs, if we want to be able to implement *any* application.

In a simple structure, the values displayed by the top element reflect the combinations of elements at the lower levels. So, the lower the starting elements, the more combinations are possible, and the larger is the number of alternatives for the value of the top element. In a *complex* structure even more values are possible, because the top element is affected by several interacting structures. Software applications are complex structures, so the impoverishment caused by starting from higher levels can be explained as a loss of both combinations and interactions: fewer combinations are possible between elements within the individual structures, and fewer interactions are possible between structures. As a result, there are fewer possible values for the top element – the application. (See "Abstraction and Reification" in chapter 1.)

While starting from higher levels may be practical for simple applications, or for applications limited to a narrow domain, for general business applications the starting level cannot be higher than the one found in the traditional

programming languages. Any theory that attempts to raise this level must be "enhanced" later with features that restore the low levels. So, while praising the power of the high levels, the experts end up contriving more and more *low-level* expedients – without which their system, language, or method would be useless.

We already saw this charlatanism in the previous section, when non-standard flow-control constructs, and even GOTO, were incorporated into structured programming. But because structured programming was still based on the traditional languages, the return to low levels was not, perhaps, evident; all that the experts had to do to restore the low levels was to annul some of the restrictions they had imposed earlier. The charlatanism became blatant, however, with the theories that followed, because these theories restrict programming, not just to certain constructs, but to special development systems. Consequently, when the theories fail, the experts do not restore the low levels by returning to the traditional programming concepts, but by reproducing some of these concepts *within* the new systems. In other words, they now *prevent* us from regaining the freedom of the traditional languages, and *force* us to depend on their systems.

In the present section, we will see how this charlatanism manifests itself in the so-called object-oriented systems; then, in the next section, we will examine the same charlatanism in the relational database systems. Other systems belonging to this category are the fourth-generation languages and tools like spreadsheets and database query, which were discussed briefly in chapter 6 (see pp. 441–442, 444–445, 452–453).

If we recall the language analogy, and the hypothetical system that would force us to combine ready-made sentences instead of words, we can easily imagine what would happen. We would be unable to express a certain idea unless the system happened to include the required sentences. So the experts would have to offer us more and more sentences, and more and more methods to use them – means to modify a sentence, to combine sentences, and so forth. We would perceive every addition as a powerful new feature, convinced that this was the only way to have language. We would spend more and more time with these sentences and methods, and communication would become increasingly complicated. But, in the end, even with thousands of sentences and features, we would be unable to express ourselves as well as we do now, simply by combining words.

While it is hard to see how anyone could be persuaded to depend on a system that promises higher starting levels in language, the whole world is being fooled by the same promise in software. And when this idea turns out to be a delusion, we continue to be fooled: we agree to depend on these systems even as we see them being modified to reinstate the low levels.

At first, the software experts try to enhance the functionality of their system by adding more and more high-level elements: whenever we fail to implement a certain requirement by combining existing elements, they provide some new ones. But we need an infinity of alternatives in our applications, and it is impossible to provide enough high-level elements to generate them all. So the experts must also add some *low-level* elements, similar to those found in the traditional languages. By then, their system ceases to be the simple and elegant high-level environment they started with; it becomes an awkward mixture of high and low levels, built-in functions, and odd software concepts.

And still, many requirements remain impossible or difficult to implement. There are two reasons for this. First, the experts do not restore *all* the low-level elements we had before; and without enough low-level elements we cannot create all the combinations needed to implement details and to link the application's structures. Second, the low-level elements are provided as an artificial extension to the high-level features, so we cannot use them freely. Instead of the simple, traditional way of combining elements – from low to high levels – we must now use some contrived methods based on high-level features.

In conclusion, these systems are fraudulent: not only do they fail to provide the promised improvement (programming exclusively through high-level features), but they make application development even more difficult than before. Their true purpose is not to increase productivity, but to maintain programming incompetence and to prevent programming freedom. The software elites force us to depend on complicated, expensive, and inefficient development environments, when we could accomplish much more with ordinary programming languages. (We discussed the fallacy of high-level starting elements in "The Delusion of High Levels" in chapter 6.)

The Promise

Like structured programming before it, object-oriented programming was hailed as an entirely new approach to application development: "OOP – Object-Oriented Programming – is a revolutionary change in programming. Without a doubt, OOP is the most significant single change that has occurred in the software field."[1] "Object technology … represents a major watershed in the history of computing."[2] "Object-oriented technology promises to produce

[1] Peter Coad and Jill Nicola, *Object-Oriented Programming* (Englewood Cliffs, NJ: PTR Prentice Hall, 1993), p. xxxiii.

[2] Paul Harmon and David A. Taylor, *Objects in Action: Commercial Applications of Object-Oriented Technologies* (Reading, MA: Addison-Wesley, 1993), p. 15.

a software revolution in terms of cost and quality that will rival that of microprocessors and their integrated circuit technologies during the 1980s."[3] "The goal is not just to improve the programming process but to define an entirely new paradigm for software construction."[4] "Object orientation is … the technology that some regard as the ultimate paradigm for the modelling of information, be that information data or logic."[5] "The *paradigm shift* we'll be exploring … is far more fundamental than a simple change in tools or terminology. In fact, the shift to objects will require major changes in the way we think about and use business computing systems, not just how we develop the software for them."[6]

Thus, while structured programming had been just a revolution, object-oriented programming was also *a new paradigm*. Finally, claimed the theorists, we have achieved a breakthrough in programming concepts.

If the promise of structured programming had been to develop and prove applications mathematically, the promise of object-oriented programming was "reusable software components": employing pieces of software the way we employ subassemblies in manufacturing and construction. The new paradigm will change the nature of programming by turning the dream of software reuse into a practical concept. Programming – the "construction" of software – will be simplified by systematically eliminating all repetition and duplication. Software will be developed in the form of independent "objects": entities related and classified in such a way that no one will ever again need to program a piece of software that has already been programmed. One day, when enough classes of objects are available, the development of a new application will entail little more than putting together existing pieces of software. The only thing we will have to program is the *differences* between our requirements and the existing software.

Some of these ideas were first proposed in the 1960s, but it was only in the 1980s that they reached the mainstream programming community. And it was in the 1990s, when it became obvious that structured programming and the structured methodologies did not fulfil their promise, that object-oriented programming became a major preoccupation. A new madness possessed the universities and the corporations – a madness not unlike the one engendered

[3] Stephen Montgomery, *Object-Oriented Information Engineering: Analysis, Design, and Implementation* (Cambridge, MA: Academic Press, 1994), p. 11.

[4] David A. Taylor, *Object-Oriented Technology: A Manager's Guide* (Reading, MA: Addison-Wesley, 1990), p. 88.

[5] John S. Hares and John D. Smart, *Object Orientation: Technology, Techniques, Management and Migration* (Chichester, UK: John Wiley and Sons, 1994), p. 1.

[6] Michael Guttman and Jason Matthews, *The Object Technology Revolution* (New York: John Wiley and Sons, 1995), p. 13.

by structured programming in the 1970s. Twenty years later, we hear the same claims and the same rhetoric: There is a software crisis. Software development is inefficient because our current practices are based, like those of the old craftsmen, on personal skills. We must turn programming into a formal activity, like engineering. It is concepts like standard parts and prefabricated subassemblies that make our manufacturing and construction activities so successful, so we must emulate these concepts in our programming activities. We must build software applications the way we build appliances and houses.

Some examples: "A major theme of object technology is *construction from parts*, that is, the fabrication, customization, and assembly of component parts into working applications."[7] "The software-development process is similar in concept to the processes used in the construction and manufacturing industries."[8] "Part of the appeal of object orientation is the analogy between object-oriented software components and electronic integrated circuits. At last, we in software have the opportunity to build systems in a way similar to that of modern electronic engineers by connecting prefabricated components that implement powerful abstractions."[9] "Object-oriented techniques allow software to be constructed of *objects* that have a specified behavior. Objects themselves can be built out of other objects, that in turn can be built out of objects. This resembles complex machinery being built out of assemblies, subassemblies, sub-subassemblies, and so on."[10]

For some theorists, the object-oriented idea goes beyond software reuse. The ultimate goal of object-oriented programming, they say, is to reduce programming to mathematics, and thereby turn software development into an exact, error-free activity. Thus, because they failed to see why the earlier idea, structured programming, was mistaken despite its mathematical aspects, these theorists are committing now the same fallacy with the object-oriented idea. Here is an example: "For our work to become a true engineering discipline, we must base our practices on hard science. For us, that science is a combination of mathematics (for its precision in definition and reasoning) and a science of

[7] Daniel Tkach and Richard Puttick, *Object Technology in Application Development* (Redwood City, CA: Benjamin/Cummings, 1994), p. 4.

[8] Ed Seidewitz and Mike Stark, *Reliable Object-Oriented Software: Applying Analysis and Design* (New York: SIGS Books, 1995), p. 6.

[9] Meilir Page-Jones, *What Every Programmer Should Know about Object-Oriented Design* (New York: Dorset House, 1995), p. 66.

[10] James Martin, *Principles of Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: PTR Prentice Hall, 1993), pp. 4–5.

information. Today we are starting to see analysis methods that are based on these concepts. The Shlaer-Mellor method of OOA [object-oriented analysis], for example, is constructed as a mathematical formalism, complete with axioms and theorems. These axioms and theorems have been published as 'rules'; we expect that as other methods become more fully developed, they, too, will be defined at this level of precision."[11]

And, once the analysis and design process is fully formalized, that elusive dream, the automation of programming, will finally be within reach. With the enormous demand for software, we can no longer afford to squander our skills constructing software by hand. We must alter the way we practise programming, from *handcrafting* software, to operating machines that make software for us: "We as practitioners must change. We must change from highly skilled artisans to being software manufacturing engineers.... We cannot afford to sit in front of our workstations and continue to build, fit, smooth, and adjust, making by hand each part of each subassembly, of each assembly, of each product.... How far away is this future? Not very far.... Our New Year's resolution is to continue this effort and, working with commercial toolmakers, to put meaningful automation in your hands by year's end. I think we can do it."[12]

Thus, the mechanistic software ideology – the belief that software development is akin to manufacturing, and the consequent belief that it is not better programmers that we need but better methods and tools – did not change. What was perceived as a shift in paradigms was in reality only a shift in preoccupations, from "structured" to "object-oriented."

This shift is also reflected in the accompanying rhetoric: as all the claims and promises made previously for structured programming were now being made for object-oriented programming, old slogans could be efficiently reused, simply by replacing the term "structured" with "object-oriented." Thus, we now have object-oriented techniques, object-oriented analysis, object-oriented design, object-oriented methodologies, object-oriented modeling, object-oriented tools, object-oriented user interface, object-oriented project management, and so forth.

There is one striking difference, though: the use of the term "technology." While structured programming was never called a technology, expressions like

[11] Sally Shlaer, "A Vision," in *Wisdom of the Gurus: A Vision for Object Technology*, ed. Charles F. Bowman (New York: SIGS Books, 1996), pp. 219–220.

[12] Ibid., pp. 222–223. These statements express perfectly that absurd, long-standing wish of the software theorists – to reduce software to mechanics: the "parts" that we build, fit, etc., in the quotation are *software* parts; and the "toolmakers" are making *software* tools, to be incorporated into software machines (development systems), which will then automatically make those parts for us.

"object technology" and "object-oriented technology" are widespread. What is just another programming concept is presented as a *technology*. But this is simply part of the general inflation in the use of "technology," which has affected all discourse (see "The Slogan 'Technology'" in chapter 5).

❖

To further illustrate the object-oriented propaganda, let as analyze a few passages from a book that was written as a guide for managers:[13] "We see object-oriented technology as an important step toward the industrialization of software, in which programming is transformed from an arcane craft to a systematic manufacturing process. But this transformation can't take place unless senior managers understand and support it."[14] This is why "this guide is written for managers, not engineers":[15] for individuals who need not "know how to program a computer or even use one."[16] The guide, in other words, is for individuals who can believe that, although they know nothing about programming, they will be able to decide, just by reading a few easy pages, whether this new "technology" can solve the software problems faced by their organization.

Taylor continues by telling us about the software crisis, in sentences that could have been copied directly from a text written twenty years earlier: development projects take longer than planned, and cost more; often, the resulting applications have so many defects that they are unusable; many of them are never completed; those that work cannot be modified later to meet their users' evolving needs.[17] Then, after describing some of the previous attempts to solve the crisis (structured programming, fourth-generation languages, CASE, various database models), Taylor concludes: "Despite all efforts to find better ways to build programs, the software crisis is growing worse with each passing year.... We need a new approach to building software, one that leaves behind the bricks and mortar of conventional programming and offers a truly better way to construct systems. This new approach must be able to handle large systems as well as small, and it must create reliable systems that are flexible, maintainable, and capable of evolving to meet changing needs.... Object-oriented technology can meet these challenges and more."[18]

The object-oriented revolution will transform programming in the same way the Industrial Revolution transformed manufacturing. Taylor reminds us how goods were produced earlier: Each product was a unique creation of a particular craftsman, and consequently its parts were not interchangeable with

[13] David A. Taylor, *Object-Oriented Technology: A Manager's Guide* (Reading, MA: Addison-Wesley, 1990). [14] Ibid., p. iii.
[15] Ibid., p. vii ("engineers," of course, means programmers). [16] Ibid.
[17] Ibid., pp. 1–2. [18] Ibid., pp. 13–14.

those of another product, even when the products were alike. Goods made in this fashion were expensive, and their quality varied. Then, in 1798, Eli Whitney conceived a new way of building rifles: by using standard parts. This greatly reduced the overall time and cost of producing them; moreover, their quality was now uniform and generally better. Modern manufacturing is based on this concept.[19]

The aim of object-oriented technology is to emulate in programming the modern manufacturing methods. It is a radical departure from the traditional approach to software development – a paradigm shift, just as the concept of standard parts was for manufacturing: "Two hundred years after the Industrial Revolution, the craft approach to producing material goods seems hopelessly antiquated. Yet this is precisely how we fabricate software systems today. Each program is a unique creation, constructed piece by piece out of the raw materials of a programming language by skilled software craftspeople.... Conventional programming is roughly on a par with manufacturing two hundred years ago.... This comparison with the Industrial Revolution reveals the true ambition behind the object-oriented approach. The goal is not just to improve the programming process but to define an entirely new paradigm for software construction."[20]

Note, throughout the foregoing passages, the liberal use of terms like "build," "construct," "manufacture," and "fabricate" to describe software development, without any attempt to prove first that programming is similar to the activities performed in a factory. Taylor doesn't doubt for a moment that software applications can be developed with the methods we use to build appliances. It doesn't occur to him that the reason we still have a software crisis after all these years is precisely this fallacy, precisely because all theories are founded on mechanistic principles. He claims that object-oriented programming is different from the previous ideas, but it too is mechanistic, so it too will fail.

This type of propaganda works because few people remember the previous programming theories, and even fewer understand the reason for their failure. The assertions made in these passages – presenting the latest theory as salvation, hailing the imminent transition of programming from an arcane craft to an engineering process – are identical to those made twenty years earlier in behalf of structured programming. And they are also identical to those made in behalf of the so-called fourth-generation languages, and CASE. It is because they didn't study the failure of structured programming that the theorists and the practitioners fall prey to the same delusions with each new idea.

Also identical is calling incompetent programmers "skilled software craftspeople" (as in the last quotation), or "highly skilled artisans" (as in a previous

[19] Ibid., pp. 86–87. [20] Ibid., p. 88.

quotation, see p. 619). We discussed this distortion earlier (see pp. 483–485). The same theorists who say that programmers are messy and cannot even learn to use GOTO correctly (see pp. 605–607) say at the same time that programmers have attained the highest possible skills (and, hence, that new methods and tools are the only way to improve their work). Although absurd – because they are contradictory, and also untrue – these claims are enthusiastically accepted by the software bureaucrats with each new theory. Thus, at any given time, and just by being preoccupied with the latest fantasies, ignorant academics, managers, and programmers can flatter themselves that they are carrying out a software revolution.

The Theory

1

Let us examine the theory behind object-oriented programming. Software applications are now made up of *objects*, rather than modules. Objects are independent software entities that represent specific processes. The *attributes* of an object include various types of data and the operations that act on this data. The objects that make up an application communicate with one other through *messages*: by means of a message, one object invokes another and asks it to perform one of the operations it is capable of performing. Just as in calling traditional subroutines, a message may include parameters, and the invoked object may return a value. So it is this structure of objects and messages that determines the application's performance, rather than a structure of modules and flow-control constructs, as was the case under structured programming.

Central to the concept of objects is their hierarchical organization. Recall our discussion of hierarchical structures and levels of abstraction (in "Simple Structures" in chapter 1). When we move up from one level to the next, the complexity of the elements increases, because one element is made up of several lower-level elements. At each level we extract, or abstract, those attributes that define the relation between the two levels, and ignore the others; so the higher-level element retains only those attributes that are common to all the elements that make it up. Conversely, when we move down, each of the lower-level elements possesses all the attributes of the higher-level element, plus some new ones. There are more details as we move from high to low levels, and fewer as we move from low to high levels. Thus, the levels of a hierarchy function as both levels of complexity and levels of abstraction.

We saw how the process of abstraction works in classification systems. Take, for example, a classification of animals: we can divide animals into wild and

domestic, the domestic into types like dogs, horses, and chickens, the dogs into breeds like spaniel, terrier, and retriever, and finally each breed into the individual animals. Types like dogs, horses, and chickens possess *specific* attributes, and in addition they *share* those attributes defining the higher-level element to which they all belong – domestic animals. Similarly, while each breed is characterized by specific attributes, all breeds share those attributes that distinguish them as a particular type of animal – dogs, for instance. Finally, each individual animal, in addition to possessing some unique attributes, shares with others the attributes of its breed.

Just like the elements in the classification of animals, software objects form a hierarchical structure. The elements at each level are known as *classes*, and the attributes relating one level to the next are the data types and the operations that make up the objects. A particular class, thus, includes the objects that possess a particular combination of data types and operations. And each class at the next lower level possesses, in addition to these, its own, unique data types and operations. The lower the level, the more data types and operations take part in the definition of a class. Conversely, the higher the level, the simpler the definition, since each level retains only those data types and operations that are common to all the classes of the lower level. So, as in any hierarchical structure, the levels in the classification of software objects also function as levels of abstraction.

This hierarchical relationship gives rise to a process called *inheritance*, and it is through inheritance that software entities can be systematically reused. As we just saw, the classes that make up a particular level *inherit* the attributes (the data types and operations) of the class that forms the next higher level. And, since the latter inherits in its turn the attributes of the next higher level, and so on, each class in the hierarchy inherits the attributes of all the classes above it. Each class, therefore, may possess many inherited attributes in addition to its own, unique attributes.

The process of inheritance is, obviously, the process of abstraction observed in reverse: when following the hierarchy from low to high levels, we note the *abstraction* of attributes (fewer and fewer are retained); from high to low levels, we note the *inheritance* of attributes (more and more are acquired).

Through the process of inheritance, we can create classes of objects with diverse combinations of attributes without having to define an attribute more than once. All we need to do for a new class is define the *additional* attributes – those that are not possessed by the higher-level classes. To put it differently, simply by defining the classes of objects hierarchically, as classes within classes, we eliminate the need to duplicate attributes: a data type or operation defined for a particular class will be inherited by all the classes below it. So, as we extend the software hierarchy with lower and lower levels of classes, we will

have classes that, even if adding few attributes of their own, can possess a rich set of attributes – those of all the higher-level classes.

The classes are only templates, *definitions* of data types and operations. To create an application, we generate replicas, or instances of these templates, and it is these instances that become the *actual* objects. All classes, regardless of level, can function as templates; and each one can engender an unlimited number of actual objects. Thus, only in the application will the data types and operations defined in the class hierarchy become real objects, with real data and operations.

2

These, then, are the principles behind the idea of object-oriented programming. And it is easy to see why they constitute a new programming paradigm, a radical departure from the traditional way of developing applications. It is not the idea of software reuse that is new, but the idea of taking software reuse to its theoretical limit: in principle, we will never again have to duplicate a programming task.

We always strove to avoid rewriting software – by copying pieces of software from previous applications, for example, and by relying on subroutine libraries. But the traditional methods of software reuse are not very effective. Their main limitation is that the existing module must fit the new requirements perfectly. This is why software reuse was limited to small pieces of code, and to subroutines that perform some common operations; we could rarely reuse a *significant* portion of an application. Besides, it was difficult even to *know* whether reusable software existed: a programmer would often duplicate a piece of software simply because he had no way of knowing that another programmer had already written it.

So code reuse was impractical before because our traditional development methods were concerned largely with *programming* issues. Hierarchical software classes, on the other hand, reflect our affairs, which are themselves related hierarchically. Thus, the hierarchical concept allows us to organize and relate the existing pieces of software logically, and to reuse them efficiently.

The object-oriented ideal is that all the software in the world be part of one giant hierarchy of classes, related according to function, and without any duplication of data types or operations. For a new application, we would start with some of the existing classes, and create the missing functions in the form of new classes that branch out of the existing ones. These classes would then join the hierarchy of existing software, and other programmers would be able to use them just as we used the older ones.

Realistically, though, what we should expect is not *one* hierarchy but a large number of *separate* hierarchies, created by different programmers on different occasions, and covering different aspects of our affairs. Still, because their classes can be combined, all these hierarchies together will act, in effect, as one giant hierarchy. For example, we can interpret a certain class in one hierarchy, together perhaps with some of its lower-level classes, as a new class that branches out of a particular class in another hierarchy. The only deviation from the object-oriented ideal is in the slight duplication of classes caused by the separation of hierarchies.

The explanation for the exceptional reuse potential in the object-oriented concept is that a class hierarchy allows us to start with software that is just *close*, in varying degrees, to a new requirement – whereas before we could only reuse software that fitted a new requirement *exactly*. It is much easier to find software that is *close* to our needs than software that *matches* our needs. We hope, of course, to find some *low-level* classes in the existing software; that is, classes which already include most of the details we have to implement. But even when no such classes exist, we can still benefit from the existing software. In this case, we simply agree to start from slightly higher levels of abstraction – from classes that resemble only broadly our requirements – and to create a slightly larger number of new classes and levels. Thus, regardless of how much of the required software already exists, the object-oriented approach guarantees that, in a given situation, we will only perform the minimum amount of work; specifically, we will only program what was not programmed before.

Let us take a specific situation. In many business applications we find data types representing the quantity in stock of various items, and operations that check and alter these values. Every day, thousands of programmers write pieces of software that are, in the end, nothing but variations of the same function: managing an item's quantity in stock. The object-oriented approach will replace this horrendous duplication with one hierarchy of classes, designed to handle the most common situations. Programmers will then start with these classes, and perhaps add a few classes of their own to implement some unique functions. Thus, the existing classes will allow us to increment and decrement the quantity, interpret a certain stock level as too high or too low, and the like. And if we need an unusual function – say, a history of the lowest monthly quantities left in stock – we will simply add to the hierarchy our own class, with appropriate data types and operations, just for this one function.

Clearly, we could have a hierarchy of this kind for every aspect of our work. But we could also have classes for entire processes, even entire applications. For example, we could have a hierarchy of specialized classes for inventory management systems. Then, starting with these classes, we could quickly create any inventory management application: we would take some classes

from low levels and others from high levels; we would ignore some classes altogether; and we would add our own classes to implement details and unusual requirements. We could even combine classes from several inventory management hierarchies, supplied by different software vendors.

This is how the experts envisage the future of application development: "The term software industrial revolution has been used to describe the move to an era when software will be compiled out of reusable components. Components will be built out of other components and vast libraries of such components will be created."[1] "In the not-too-distant future, it will probably be considered archaic to design or code *any* application from scratch. Instead, the norm will be to grab a bunch of business object classes from a gigantic, worldwide assortment available on the meganet, create a handful of new classes that tie the reusable classes together, and – *voilà*! – a new application is born with no muss, no fuss, and very little coding."[2]

Programming as we know it will soon become redundant, and will be remembered as we remember today the old manufacturing methods. The number of available object classes will grow exponentially, so programmers will spend more and more time combining existing classes, and less and less time creating new ones. The skills required of programmers, thus, will change too: from knowing how to create new software, to knowing what classes are available and how to combine them. Since the new skills can be acquired more easily and more quickly, we will no longer depend on talented and experienced programmers. The object-oriented paradigm will solve the software crisis, therefore, both by reducing the time needed to create a new application and by permitting a larger number of people to create applications.

The Contradictions

1

We recognize in the object-oriented fantasy the software variant of the language fantasies we studied in chapter 4. The mechanistic language theories, we saw, assume that it is possible to represent the world with a simple hierarchical structure. Hence, if we invent a language that can itself be represented as a hierarchical structure, we will be able to mirror the world perfectly in language: the smallest linguistic elements (the words, for example) will mirror

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

[2] Michael Guttman and Jason Matthews, *The Object Technology Revolution* (New York: John Wiley and Sons, 1995), p. 76.

the smallest entities that make up the world; and the relations between linguistic elements will mirror the natural laws that govern the real things. The hierarchical structure of linguistic elements will then correspond on a one-to-one basis to the hierarchical structure of real objects, processes, and events. By combining sentences in this language as we do operations in mathematical logic, we will be able to explain any phenomenon. Thus, being logically perfect and at the same time a perfect picture of the world, a language designed as a simple hierarchical structure will allow us to represent and to understand the world.

From the theories of Leibniz, Dalgarno, and Wilkins in the seventeenth century to those of Russell and Carnap in the twentieth, the search for a logically perfect language has been one of the most persistent manifestations of the mechanistic myth. The fallacy, we saw, is not so much in the idea of a logically perfect language, as in the belief that such a language can accurately mirror the world. It is quite easy, in fact, to design a language in the form of a hierarchical structure, and to represent in it the entities and levels of abstraction that exist in the world. The problem, rather, is that there are *many* such structures – many different ways to represent the world – all correct and relevant.

The entities that make up the world possess many attributes, and are therefore connected through many structures at the same time, one structure for each attribute. Thus, if our language is to represent reality accurately, the linguistic elements too must be connected through more than one structure at the same time. The language mechanists attempt to find one classification, or one system, that would relate all objects, processes, and events that can exist in the world. But this is a futile quest. Even a simple object has many attributes – shape, dimensions, colour, texture, position, origin, age, and so forth. To place it in *one* hierarchy, therefore, we would have to choose *one* attribute and ignore the others. So, if we cannot represent with one hierarchy even ordinary objects, how can we hope to represent the more complex aspects of the world?

It is precisely because they are *not* logically perfect that our *natural* languages allow us to describe the world. Here is how: We use words to represent the real things that make up the world. Thus, since the real things share many attributes and are linked through many structures, the words that represent those things will also be linked, in our mind, through many structures. The words that make up a message, a story, or an argument will form one structure for each structure formed by the real things.

The mechanistic language theories fail to represent the world accurately because their elements can be connected in only one way: they attempt to represent with *one* linguistic structure the *system* of structures that is the world. The mechanists insist on a simple structure because this is the only way to

have a deterministic system of representation. But if the world is a complex structure, and is therefore an indeterministic phenomenon, any theory that attempts to represent it through deterministic means is bound to fail.

❖

Since it is the same world that we have to represent through language and through software, what is true for language is also true for software. To represent the world, the software entities that make up an application must be related through many structures at the same time. If we restrict their relations to one hierarchy, the application will not mirror the world accurately. Thus, whether we classify all the existing software entities or just the entities of one application, we need a system of interacting structures. *One* structure, as in the object-oriented paradigm, can only represent the relations created by *one* attribute (or perhaps by *a few* attributes, if shared by the software entities in a limited way).

Recall our discussion of complex structures in chapter 1 (pp. 98–102) and in chapter 4 (pp. 354–361). We saw that any attempt to represent several attributes with one structure results in an incorrect hierarchy. Because the attributes must be shown *within one another*, all but the first will be repeated for each branch created by the previous ones; and this is not how entities possess attributes in reality.

Only when each attribute is possessed by just *some* of the entities can they all be included in one hierarchy. Here is how this can be done, if we agree to restrict the attributes (figure 1-6, p. 101, is an example of such a hierarchy): the class of all entities is shown as the top element, and one attribute can be shared by all the entities; on the basis of the values taken by this attribute, the entities are divided into several classes, thereby creating the lower level; then, in each one of these classes the entities can possess a second attribute (but they must all possess the same attribute, and this attribute cannot be shared with entities from the other classes); on the basis of the values taken by this attribute, each class is then divided into third-level classes, where the entities can possess a third attribute, again unique to each class; and so on. (On each level, instead of one attribute per class, we can have a set of several attributes, provided they are all unique to that class. The set as a whole will act in effect as one attribute, so the levels and classes will be the same as in a hierarchy with single attributes.)

The issue, then, is simply this: Is it possible to restrict software entities to the kind of relations that can be represented through a strict hierarchical structure, as described above? Do software entities possess their attributes in such a limited way that we can represent all existing software with one structure? Or, if not all existing software, can we represent at least each individual application

with one structure? As we saw, the answer is *no*. To mirror the world, software entities must be related through all their attributes at the same time; and these attributes, which reflect the various processes implemented in the application (see pp. 345–346), only rarely exist *within one another*. Only rarely, therefore, can software entities be classified or related through *one* hierarchical structure. Whether the classification includes all existing software, or just the objects of one application, we need a *system* of structures – perhaps as many structures as there are attributes – to represent their relations.

The benefits promised by the object-oriented theory can be attained *only* with a simple hierarchical structure. Thus, since it assumes that the relations between software entities can be completely and precisely represented with one structure, the theory is fundamentally fallacious.

❖

Let us recall some of the hierarchies we encountered in previous chapters. The biological classification of animals – classes, orders, families, genera, species – remains a perfect hierarchy only if we agree to take into account just *a few* of their attributes, and to ignore the others. We deliberately limit ourselves to those attributes that *can* be depicted within one another; then, obviously, the categories based on these attributes are related through a strict hierarchy. This classification is important to biologists (to match the theory of natural evolution, for instance); but we can easily create other classifications, based on other attributes.

The distinction between wild and domestic, for example, cannot be part of the biological classification. The reason is that those attributes we use to distinguish an animal as wild or domestic cannot be depicted *within* those attributes we use to distinguish it as mammal, or bird, or reptile; nor can the latter attributes be depicted *within* the former. The two hierarchies overlap. Thus, horses and foxes belong to different categories (domestic and wild) in one hierarchy, but to the same category (class of mammals) in the other; chickens and dogs belong to the same category (domestic) in one hierarchy, but to different categories (birds and mammals) in the other. Clearly, if we restricted ourselves to the biological classification we wouldn't be able to distinguish domestic from wild animals. Each classification is useful if we agree to view animals from one perspective at a time. But only a system of interacting structures can represent *all* their attributes and relations: a system consisting of several hierarchies that exist at the same time and share their terminal elements, the individual animals.

Similarly, organizations like corporations and armies can be represented as a strict hierarchy of people only if we take into account *one* attribute – the role

or rank of these people. This is the hierarchy we are usually concerned with, but we can also create hierarchies by classifying the people according to their age, or gender, or height, or any other attribute. Each classification would likely be different, and only rarely can we combine two hierarchies by depicting one attribute *within* the other.

For example, only if the positions in an organization are gender-dependent can we combine gender and role in one hierarchy: we first divide the people into two categories, men and women, and then add their various roles as lower levels *within* these two categories. The final classification is a correct hierarchy, with no repetition of attributes. It is all but impossible, however, to add a *third* attribute to this hierarchy without repetition; that is, by depicting it strictly *within* the second one. We cannot add a level based on age, for instance, because people of the same age are very likely found in more than one of the categories established by the various combinations of gender and role.

Recall, lastly, the structure of subassemblies that make up a device like a car or appliance. This structure too is a strict hierarchy, and we can build devices as hierarchies of things within things because we purposely design them so that their parts are related mainly through one attribute – through their role in the construction and operation of the device. The levels of subassemblies are then the counterpart of the levels of categories in a classification hierarchy. But, just as entities can be the terminal elements in many classifications, the ultimate parts of a device can be the terminal elements of many hierarchies.

The hierarchy we are usually concerned with – the one we see in engineering diagrams and in bills of material, and which permits us to build devices as levels of subassemblies – is the structure established by their physical and functional relationship. But we can think of many other relations between the same parts – relations based on such attributes as weight, colour, manufacturer, date of manufacture, life expectancy, or cost. We can classify parts on the basis of any attribute, and each classification would constitute a different hierarchy. Besides, only rarely do parts possess attributes in such a way that we can depict their respective hierarchies as one *within* another. Only rarely, therefore, can we combine several hierarchies into one. (Parts made on different dates, for example, may be used in the same subassembly; and parts used in different subassemblies may come from the same manufacturer.)

The promise of object-oriented programming is *specifically* the concept of hierarchical classes. This concept is well-suited for representing our affairs in software, the experts say, because the entities that make up the world are themselves related hierarchically: "A model which is designed using an object-

oriented technology is often easy to understand, as it can be directly related to reality."[1] "The object-oriented viewpoint attempts to more closely reflect the natural structure of the problem domain rather than the implicit structure of computer hardware."[2] "OOP [object-oriented programming] enables programmers to write software that is organized like the problem domain under consideration."[3] "One of the greatest benefits of an object-oriented structure is the direct mapping from objects in the problem domain to objects in the program."[4] "OOP design is less concerned with the underlying computer model than are most other design methods, as the intent is to produce a software system that has a natural relationship to the real world situation it is modelling."[5] "Object orientation … should help to relate computer systems more closely to the real world."[6] "The intuitive appeal of object orientation is that it provides better concepts and tools with which to model and represent the real world as closely as possible."[7] "The models we build in OO [object-oriented] analysis reflect reality more naturally than the models in traditional systems analysis.… Using OO techniques, we build software that more closely models the real world."[8]

But, as we saw, the entities that make up the world are related through *many* hierarchies, not one. How, then, can software entities related through one classification mirror them accurately? The software mechanists want to have both the simplicity of a hierarchical structure and the ability to mirror the world. And in their attempt to realize this dream, they commit the fallacy of reification: they extract *one* structure from the complex phenomenon, expecting this structure alone to provide a useful approximation.

Now, it is obvious that hierarchical software classes allow us to implement such applications as the process of assembling an appliance, or the positions held by people in an organization, or the biological classification of animals.

[1] Ivar Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, rev. pr. (Reading, MA: Addison-Wesley/ACM Press, 1993), p. 42.

[2] Ed Seidewitz and Mike Stark, *Reliable Object-Oriented Software: Applying Analysis and Design* (New York: SIGS Books, 1995), p. 26.

[3] Peter Coad and Jill Nicola, *Object-Oriented Programming* (Englewood Cliffs, NJ: PTR Prentice Hall, 1993), p. xxxiii.

[4] Greg Voss, *Object-Oriented Programming: An Introduction* (Berkeley, CA: Osborne McGraw-Hill, 1991), p. 30.

[5] Mark Mullin, *Object-Oriented Program Design* (Reading, MA: Addison-Wesley, 1989), p. 5.

[6] Daniel Tkach and Richard Puttick, *Object Technology in Application Development* (Redwood City, CA: Benjamin/Cummings, 1994), p. 17.

[7] Setrag Khoshafian and Razmik Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces* (New York: John Wiley and Sons, 1990), p. 6.

[8] James Martin and James J. Odell, *Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: Prentice Hall, 1992), p. 67.

But these are artificial structures, the result of a design that deliberately restricted the relations between elements to certain attributes: we can ignore the other structures because we *ensured* that the relations caused by the other attributes are much weaker. These structures, then, do not represent the actual phenomenon, but only one aspect of it – an aspect that can be depicted with one hierarchy. So, like any mechanistic concept, hierarchical software classes are useful when the problem can indeed be approximated with one structure.

The object-oriented promise, though, is that the concept of hierarchical classes will help us to implement *any* application, not just those that are *already* a neat hierarchy. Thus, since the parts that make up our affairs are usually related through several hierarchies at the same time, the object-oriented promise cannot possibly be met. Nothing stops us from restricting every application to what *can* be represented with one hierarchy; namely, relations based on one attribute, or a small number of carefully selected attributes. But then, our software will not mirror our affairs accurately.

As we saw under structured programming, an application in which all relations are represented with one hierarchy is useless, because it must always do the same thing (see p. 533). Such an application can have no conditions or iterations, for example. Whether the hierarchy is the nesting scheme of structured programming, or the object classification of object-oriented programming, each element must always be executed, always executed once, and always in the same relative sequence. This, after all, is what we expect to see in any system represented with one hierarchy; for instance, the parts and subassemblies that make up an appliance always exist, and are always connected in the same way.

Thus, after twenty years of structured programming delusions, the software experts started a new revolution that suffers, ultimately, from the same fallacy: the belief that our affairs can be represented with *one* hierarchical structure.

2

What we have discussed so far – the neatness of hierarchical classes, the benefits of code reuse, the idea of software concepts that match our affairs – is what we see in the *promotion* of object-oriented programming; that is, in advertisements, magazine articles, and the introductory chapters of textbooks. And this contrasts sharply with the *reality* of object-oriented programming: what we find when attempting to develop actual applications is difficult, non-intuitive concepts. Let us take a moment to analyze this contradiction.

As we saw, the theorists promote the new paradigm by claiming that it lets us represent our affairs in software *more naturally*. Here are some additional

examples of this claim: "The models built during object-oriented analysis provide a more natural way to think about systems."[9] "Object-oriented programming is built around *classes* and *objects* that model real-world entities in a more natural way.… Object-oriented programming allows you to construct programs the way we humans tend to think about things."[10] "The object-oriented approach to computer systems is … a more natural approach for people, since we naturally think in terms of objects and we classify them into hierarchies and divide them into parts."[11]

The illustrations, too, are simple and intuitive. One book explains the idea of hierarchical classes using the Ford Mustang car: there is a plain, generic model; then, there is a base model and an improved LX model, each one inheriting the features of the generic model but also adding its own; and there is the GT sports model, derived from the LX but with some features replacing or enhancing the LX features.[12] Another book explains the object-oriented concepts using the world of baseball: objects are entities like players, coaches, balls, and stadiums; they have attributes like batting averages and salaries, perform operations like pitching and catching, and belong to classes like teams and bases.[13]

The impression conveyed by the *promotion* of object-oriented programming, thus, is that all we have to do is define our requirements in a hierarchical fashion – an easy task in any event, since this is how we normally view the world and conduct our affairs – and the application is almost done. The power of this new technology is ours to enjoy simply by learning a few principles and purchasing a few tools.

When we study the *actual* object-oriented systems, however, we find an entirely different reality: huge development environments, complicated methodologies, and an endless list of definitions, rules, and principles that we must assimilate. Hundreds of books had to be written to help us understand the new paradigm. In one chapter after another, strange and difficult concepts are being introduced – concepts which have nothing to do with our programming or business needs, but which must be mastered if we want to use an object-oriented system. In other words, what we find when attempting

[9] James Martin, *Principles of Object-Oriented Analysis and Design* (Englewood Cliffs, NJ: PTR Prentice Hall, 1993), p. 3.

[10] Andrew C. Staugaard Jr., *Structured and Object-Oriented Techniques: An Introduction Using C++*, 2nd ed. (Upper Saddle River, NJ: Prentice Hall, 1997), p. 29.

[11] John W. Satzinger and Tore U. Ørvik, *The Object-Oriented Approach: Concepts, Modeling, and System Development* (Cambridge, MA: Course Technology, 1996), p. 11.

[12] Khoshafian and Abnous, *Object Orientation*, pp. 8–10.

[13] Donald G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach* (New York: John Wiley and Sons, 1993), pp. 5–9.

to *practise* object-oriented programming is the exact opposite of what its promotion says.

To make matters worse, the resulting applications are large, unwieldy, and difficult to manage. What can be programmed with just a few statements in a traditional language ends up as an intricate system of classes, objects, definitions, and relations when implemented in an object-oriented environment.

The theorists agree. After telling us that object-oriented programming is a natural, intuitive concept, they tell us that it is in fact difficult, and that it requires much time and effort to learn: "Many experienced and intelligent information systems developers have difficulty understanding and accepting this new point of view."[14] "Those who have programmed before may well find OOP [object-oriented programming] strange at first. It may take a while to forget the ways you have learned, and to [master] another method of programming."[15] "To use OO [object-oriented] technology well, much careful training is needed. It takes time for computer professionals to think in terms of encapsulation, inheritance, and the diagrams of OO analysis and design.… Good use of inheritance and reusable classes requires cultural and organizational changes."[16]

Claiming at the same time that the object-oriented principles are simple and that they are difficult is not as absurd as it sounds; for, in reality, the theorists are describing two different things. When praising the simplicity of these principles, they are referring to the *original* idea – the fantasy of combining and extending hierarchically classes of objects. And indeed, implementing applications as strict hierarchies of objects is easy and intuitive. Very few applications, however, *can* be implemented in this fashion, because very few aspects of the world are mechanistic. So, since most applications must be implemented as *systems* of hierarchies, the original idea was worthless. To make object-oriented programming practical, the means to create multiple, interacting hierarchies had to be restored. But this capability – a natural part of the *traditional* programming concepts – can only be added to an object-oriented system through contrived, awkward extensions. And it is these extensions, as opposed to the simple original idea, that the theorists have in mind when warning us that the object-oriented principles are hard to understand.

The difficulties caused by the object-oriented systems are due, thus, to the reversal of a fundamental programming principle: instead of creating high-level software elements by starting with low-level ones, we are asked to *start*

[14] Satzinger and Ørvik, *Object-Oriented Approach*, p. 3.

[15] David N. Smith, *Concepts of Object-Oriented Programming* (New York: McGraw-Hill, 1991), pp. 11–12.

[16] Martin, *Object-Oriented Analysis*, p. 45.

with high-level elements (classes of objects) and to add, where required, lower-level ones. But this is rarely practical. Only by starting with low-level elements can we create all the elements we need at the higher levels. Starting with low-level elements is, therefore, the only way to implement the interacting structures that make up a serious application. The object-oriented theory claimed that we can start with classes of objects because it assumed that we can restrict ourselves to isolated, non-interacting structures; but then, it was extended to permit us to *link* these structures. So now we must create the interactions by starting with high-level elements, which is much more complicated than the traditional way – starting with low-level ones.

3

If a theory expects us to represent our affairs with one hierarchy, while our affairs can only be represented with a system of interacting hierarchies, we must either admit that the theory is invalid, or modify it. The original object-oriented theory was falsified again and again, every time a programmer failed to represent with a strict hierarchical classification a real-world situation. The experts responded to these falsifications, however, not by doubting the theory, but by *expanding* it: they added more and more "features" to make it cope with those situations that would have otherwise refuted it. The theory became, thus, unfalsifiable. As is usually the case with a pseudoscientific theory, the experts saved it from refutation by turning its falsifications into new features. And it is these features, rather than the original concepts, that constitute the *actual* theory – what is being practised under the object-oriented paradigm.

The new features take various forms, but their ultimate purpose is the same: to help us override the restrictions imposed by the original theory. The *actual* theory, thus, is the set of features that allow us to create *interacting* hierarchies. It is these features, the experts explain, that make the object-oriented paradigm such a powerful concept. In other words, the power of the theory derives from those features introduced in order to bypass the theory. We will examine some of these features shortly.

Structured programming, we recall, became practical only after restoring the means to create multiple, interacting flow-control structures – precisely what the original theory had condemned and claimed to be unnecessary. So, in the end, what was called structured programming was the exact opposite of the original theory. Similarly, the object-oriented concepts became practical only after restoring the means to create multiple, interacting class hierarchies. So what is called now object-oriented programming is the exact opposite of the original idea. To this day, the object-oriented concepts are being promoted by

praising the benefits of strict hierarchical relations, and by demonstrating these benefits with trivial examples. At the same time, the *actual* object-oriented systems are specifically designed to help us *override* this restriction. But if the benefits are attainable only with a single hierarchy, just as the original theory said, the conclusion must be that the *actual* object-oriented systems offer no benefits.

So the object-oriented paradigm is no better than the other mechanistic software theories: it gives us nothing that we did not have before, with the traditional programming concepts and with any programming language. Each time, the elites promise us a dramatic increase in programming productivity by invoking the hierarchical model. Ultimately, these theories are nothing but various attempts to reduce the complex reality to a simple structure: an isolated flow-control structure, an isolated class structure, and so on. And when this naive idea proves to be worthless, the elites proceed to "enhance" the theories so as to allow us to create *complex* structures again: they restore both the lower levels and the means to link structures, which is the only way to represent our affairs in software.

But by the time a mechanistic theory is "enhanced" to permit multiple, interacting structures, the promised benefits – formal methods for reusing existing software, for building applications as we build appliances, for proving their validity mathematically – are lost. Now it is again our minds that we need, our personal skills and experience, because only minds can process complex structures. So we are back where we were before the theory. The theory, and also the methodologies, programming tools, and development environments based on it, are now senseless. They are even detrimental, because they force us to express our requirements in more complicated ways. We are told that the complications are worthwhile, that this is the only way to attain those benefits. But if the benefits were already lost, all we have now is a theory that makes programming more difficult than it was before.

Thus, by refusing to admit that their theory has failed, by repeatedly expanding it and asking us to depend on it, the elites are committing a fraud: they are covering up the fact that they have nothing to offer us; they keep promising us an increase in programming productivity, when in reality they are preventing us from practising this profession and improving our skills.

❖

As we did for structured programming, we will study the object-oriented fantasy by separating it into several delusions: the belief that we can represent our affairs with a neat, hierarchical classification of software entities; the belief that, instead of one classification, we can represent the same affairs by

combining many small, independent classifications; the belief that we can use the object-oriented concepts through traditional programming languages; the belief that we can modify the concepts of abstraction and inheritance in any way we like and still retain their benefits; and the belief that we no longer need to concern ourselves with the application's flow of execution.

Although the five delusions occurred at about the same time, they can be seen, like the delusions of structured programming, as stages in a process of degradation: repeated attempts to rescue the theory from refutation. Each stage was an opportunity for the software experts to recognize the fallaciousness of their theory; instead, at each stage they chose to *expand* it, by incorporating the falsifications and describing them as new features. The stages, thus, mark the evolution of the theory into a pseudoscience (see "Popper's Principles of Demarcation" in chapter 3).

Also as was the case with structured programming, when the object-oriented concepts were being promoted as a revolution and a new paradigm, all five delusions had *already* occurred. Thus, there never existed a serious, practical theory of object-oriented programming. What the experts were promoting was something entirely different: complicated development environments that helped us to create precisely what that theory had claimed to be unnecessary – multiple, interacting software hierarchies.

The First Delusion

The first object-oriented delusion is the belief that we can represent the world with a simple structure of software entities. In fact, only *isolated aspects* of the world can be represented with simple structures. To represent the world accurately we need a *system* of structures. We need, in other words, a complex structure: a set of software entities that belong to several hierarchies at the same time.

The first delusion is akin to the seventeenth-century belief that it is possible to represent all knowledge with one hierarchical structure (see pp. 311–315). What we need to do, said the rationalist philosophers, is depict knowledge in the form of concepts within concepts. The simplest concepts will function as terminal elements (the building blocks of the knowledge structure), while the most complex concepts will form the high levels. Everything that can be known will be represented, thus, in a kind of classification: a giant hierarchy of concepts, neatly related through their characteristics.

It is the principle of abstraction that makes a hierarchical classification possible: at each level, a concept retains only those characteristics common to

all the concepts that make up the next lower level. This relationship is clearly seen in a tree diagram: the branches that connect several elements to form a higher-level element signify the operation that extracts the characteristics shared by those elements; then another operation relates the new element to others from the same level, forming an element of the next higher level, and so on.

Similarly, we believe that it is possible (in principle, at least) to design a giant hierarchy of all *software* entities. This hierarchy would be, in effect, a classification of those parts of human knowledge that we want to represent in software – a subset, as it were, of the hierarchy envisaged by the seventeenth-century philosophers. This idea, whether or not explicitly stated, forms the foundation of the object-oriented paradigm. For, only if we succeed in relating all software entities through one hierarchical structure can the benefits promised by this paradigm emerge. The benefits, we recall, include the possibility of formal, mechanistic methods for reusing and extending software entities.

No hierarchy has ever been found that represents all knowledge. This is because the concepts that make up knowledge are related, not through one, but through *many* hierarchies. Similarly, no hierarchy can represent all software, because the software entities that make up our applications are related through many hierarchies. So these theories fail, not because we cannot *find* a hierarchy, but because we can find *many*, and it is only this system of hierarchies, with their interactions, that can represent the world.

The mechanists are encouraged by the ease with which they discover one or another of these hierarchies, and are convinced that, with some enhancements, that hierarchy will eventually mirror the world. Any one hierarchy, however, can only relate concepts or software entities in one particular manner – based on one attribute, or perhaps on a small set of attributes. So one hierarchy, no matter how large or involved, can only represent *one* aspect of the world.

The theory of object-oriented programming was refuted, thus, even before it was developed. The theorists, however, misinterpreted the difficulty of relating all existing software entities through one giant hierarchy as a problem of management: it is impossible for one organization to create the whole hierarchy, and it is impractical to coordinate the work of thousands of individuals from different organizations. We must simplify the task, therefore, by dividing that hypothetical software hierarchy into many small ones. And this is quite easy to do, since any hierarchical structure can be broken down into smaller structures. For example, if we sever all the branches that connect a particular element to the elements at the lower level, that element will become a terminal element in the current structure, and each lower-level element will become the top element of a new, separate structure.

Thus, concluded the theorists, even if every one of us creates our own, smaller structures, rather than all of us adding elements to one giant structure, the *totality* of software entities will continue to form one giant structure. So the promise of object-oriented programming remains valid.

To save their theory, the advocates of object-oriented programming rejected the evidence that the idea of a giant software hierarchy is a delusion, and in so doing they succumbed to a second delusion.

The Second Delusion

If the first delusion is that it is possible to classify all existing software in one hierarchy, the second delusion – which emerged when this idea failed – is that it is *not* necessary, after all, to restrict ourselves to one classification: we can also create applications formally, as strict hierarchies of software entities, by combining many small, independent, specialized classifications. But this idea is even sillier than the first one. For, could we combine these structures, we would not have had to separate them in the first place. Let us analyze this problem.

The object-oriented theory assumes that each application is a hierarchy of software entities, and that this hierarchy is part of the larger hierarchy that is the classification of all existing software entities. In reality, just like the totality of existing software, each application is a system of interacting hierarchies. An application is indeed part of all existing software, but in an *indeterministic* way; namely, in the way a complex structure is part of a larger complex one, not in the way a simple structure is part of a larger simple one. There are no mechanistic means – no precise, completely specifiable methods – to derive an application from the system of entities that is the classification of all software. And this is why the idea of a formal classification of software entities, and a formal method of software reuse, is fundamentally mistaken.

The second delusion can also be described as the belief that there is a way around the problems created by the first delusion. But it is no easier to create an application by combining several smaller hierarchies, than it is to create one by extracting portions of a larger hierarchy. The difficulty that prevents us from building one hierarchical classification of all software – the need to relate software entities through *many* hierarchies, not one – is also the difficulty that prevents us from building individual applications as single hierarchies.

Let us see how this problem manifests itself in practice. Let us assume that we already have a large number of separate classifications, each one representing an isolated aspect of software applications: display functions,

database functions, one type or another of accounting functions, one style or another of reporting, and the like. But it is impossible to create applications simply by combining these hierarchies; that is, by building a large hierarchy that incorporates somehow the individual ones. For, the only way to combine hierarchies in an object-oriented environment is mechanistically, as one *within* another. This is true because the only way for an element to possess attributes from both element *A* of one hierarchy and element *B* of another hierarchy is through inheritance: we make *A* a lower-level element in the latter hierarchy, thereby allowing it to inherit attributes from *B*.

A particular application may require, for example, display, database, and accounting operations. But even if the three separate hierarchies embodying these operations are complete and correct, even if they include all the details that we are likely to need, they are useless for generating serious accounting applications. The reason is that, in an application, the display operations are not always performed *within* the database or accounting operations; nor are the accounting operations performed *within* the display or database operations, or the database operations *within* the display or accounting operations. What we need is software entities that can invoke the three types of operations *freely*; and we cannot create such entities if restricted to hierarchical combinations. To put this differently, the hierarchical combinations represent only a fraction of all possible relations between the elements of the three structures. Missing are those combinations we would see in a system of *interacting* structures – the kind of system that is impossible to create through object-oriented programming.

We must also bear in mind that it is more than three hierarchies that we have to combine when creating an application. We may be able to represent with one hierarchy such functions as display or database, which are artificial and restricted by our mechanistic computing means in any case. But it is impossible to represent with one hierarchy all our accounting processes, for instance. These processes reflect business, social, and personal affairs, which can only be represented as *interacting* structures of entities. To create a serious accounting application, therefore, we must combine hundreds of different hierarchies, not three; and few of these combinations can be depicted as one hierarchy *within* another.

Another thing to bear in mind is that it doesn't matter whether we start with hierarchies that embody separately the three types of operations – display, database, and accounting – or with hierarchies that are already a combination of these operations. The best approach may well be to have whole accounting hierarchies, each one embodying a certain aspect of accounting. Each hierarchy would include, therefore, not just accounting operations, but also the associated display and database operations. Even then, however, to

create an application we would have to combine these hierarchies by non-mechanistic means, because the various aspects of accounting do not exist as one within another.

The Third Delusion

We saw that the idea of combining several class hierarchies into one is a fallacy. Only very simple applications can be created in this fashion: those for which we can restrict ourselves to hierarchical combinations of elements. This idea, we recall, was thought to be a solution to the failure of the *original* object-oriented idea – which idea was to represent with one hierarchy *all* software, not just individual applications. (And the original idea is, in fact, the only way to derive the benefits promised by the object-oriented paradigm.)

Thus, to deal with the problems created by the first delusion, the theorists felt justified to modify the object-oriented concept; but the new idea is as fallacious as the first, so it became the second delusion. Just as they failed to recognize the first delusion as a falsification of the object-oriented concept, they failed to recognize the second one as a new falsification. And, just as they modified the theory in response to the first delusion, they now introduced additional modifications, to deal with the problems created by the second one.

Because it is impossible to relate software entities freely through one hierarchy, the theorists had to provide the means to build systems of *interacting* hierarchies. All the modifications, then, have one purpose: to enable us to relate software entities through several hierarchies at the same time; in other words, to bypass the restriction to one hierarchy. Faithful to the pseudoscientific tradition, these modifications – which are, in fact, blatant violations of the object-oriented principles – are described as new features, or enhancements. Here we will discuss only the simplest enhancement, the use of traditional programming languages; then, under the fourth and fifth delusions, we will study the others.

The traditional languages do provide, of course, the means to relate software entities freely. Here is how: Each element in the application is affected by various processes (calling certain subroutines, using certain memory variables and database fields, being part of certain practices). Each element is related, therefore, to the other elements affected by the same processes. And we can design these relations – which become ultimately a system of interacting structures – in any way we like. (Software processes were introduced in chapter 4; see pp. 345–346.)

So the simplest way to combine hierarchies is by creating modules, blocks of statements, conditional constructs, and the like, by means of a *traditional* language, and *then* picking whatever classes we need from the various hierarchies. We use the class hierarchies, thus, not as originally intended – as a formal representation of the whole application – but in the manner of subroutine libraries. In this way, any element in the application can inherit attributes from several hierarchies, simply by invoking several classes. So, by using classes as we use subroutines, any element can possess any combination of attributes we need: we are no longer restricted to combining attributes hierarchically, one within another, as stipulated by the object-oriented principle of inheritance.

Recall the earlier problem: combining classes from three hierarchies – display, database, and accounting operations – but *not* as one within another. While impossible under the object-oriented paradigm, this requirement is easily implemented once we extend the use of classes so as to invoke them freely: directly rather than hierarchically, wherever needed, just as we invoke subroutines.

The first modification, then, was to turn the object-oriented concept from a formal, autonomous programming method, supported by special programming languages, into a mere extension to the *traditional* methods and languages. And this was accomplished by adding object-oriented capabilities to some of the popular languages (C and COBOL, for instance). The enhanced variants are known as *hybrid* languages. (The reverse is also true: special languages like Simula and Smalltalk, originally intended as pure object-oriented environments, were later enhanced with traditional capabilities.)

Thus, there are no strict object-oriented languages in existence, simply because one adhering to the object-oriented principles would be totally impractical. The theorists invented a new term to describe what is in reality not a new feature, but the reinstatement of old, well-established concepts: "hybrid" sounds as if these languages added a new quality to the object-oriented principles, when in fact they are a *reversal* of these principles. No one wondered why, if object-oriented programming is the revolutionary concept the experts say it is, we still need to rely on the old languages. The experts praise the power of the object-oriented paradigm, even as everyone can see that this paradigm is useless, and that its power derives from the freedom we regain when reverting to the traditional concepts.

In the end, no application was ever based on the *true* object-oriented principles. Programmers believe that they are practising object-oriented programming, when what they are practising in reality is *traditional* programming – supplemented here and there, when not too inconvenient, with some object-oriented concepts.

The Fourth Delusion

1

The most important "features" and "improvements" added to the object-oriented theory are those that alter the very nature of a hierarchical structure. We saw that the theory had to be modified in order to give us the means to combine class hierarchies, and that using class hierarchies from within a traditional language is the simplest way to accomplish this. But if we had to rely on this method alone to combine hierarchies, we would find little use for the *actual* object-oriented features. All we would have then is some class libraries that, apart from providing perhaps better hierarchical links, are identical to the traditional subroutine libraries.

In order to permit us to relate class hierarchies freely *within* the object-oriented paradigm, the very notion of a class hierarchy had to be modified. In the end, the theory of object-oriented programming was rescued by annulling its most celebrated principle – the restriction to classes related hierarchically through inherited attributes.

Inheritance, we recall, is that property of hierarchical structures whereby an element derives some of its attributes from the higher levels. Thus, in the case of software class hierarchies, each element, in addition to possessing its own attributes, inherits the attributes of the higher-level class – the class to which it is directly subordinate. And, since the latter inherits the attributes of the class to which *it* is subordinate, and so on, each element will possess the attributes of all the classes above it.

This property is not new to the object-oriented theory, but common to all hierarchical systems. This is so obvious, in fact, that inheritance is rarely mentioned as a hierarchical feature. It is the property of *abstraction* that is usually described as the distinguishing quality of hierarchical structures. Abstraction means that, as we move from low to high levels, an element at a given level retains only those attributes that are common to all the elements of the next lower level. Inheritance, therefore, is not a separate quality, but merely the process of abstraction observed in reverse. We can reverse the last sentence, for instance, and say that all the elements at a given level inherit the attributes possessed by the element of the next higher level. Both statements describe the same relationship.

The object-oriented theory, though, presents the property of inheritance as an important and powerful feature. We are left with the impression that this feature is somehow *additional* to the hierarchical relations between software classes. And, once inheritance is perceived as a separate feature, it is only natural to try to enhance it. But this idea is absurd. The property of inheritance

cannot be enhanced; like abstraction, it is implicit in the notion of a hierarchy, a reflection of the relations between the structure's elements. One cannot have a hierarchy where the concept of inheritance is different in any way from its original meaning.

❖

The first modification was to allow a class to *change*, and even to *omit*, an inherited attribute. The capability to add its own, unique attributes remains, but the class no longer needs to possess *all* the attributes possessed by the class of the next higher level. In other words, the attributes of a class, and hence its relations with the other classes, are no longer determined by its position in the class hierarchy. If what we need is indeed a hierarchical relationship with the higher-level classes, we let it inherit all their attributes, as before; but if what we need is a different relationship, we can change or omit some of these attributes.

The attributes of a class are its data types and operations. So what this modification means is that each class in the application can now have any data types and operations we like, not necessarily those inherited from the classes above it.

Attributes, as we know, relate entities by grouping them into hierarchical structures (see "Software Structures" in chapter 4). In a software application, each attribute generates a different structure by relating in a particular way the entities that make up the application. Clearly, then, what has been achieved with the new feature is to eliminate the restriction to one hierarchy. Since classes can now possess *any* attributes, they can be related in any way we want, so they can form many structures at the same time. The structure we started with – the class hierarchy – is no longer the only structure in the application. When we study this structure alone, the application's classes still appear to be related through a neat hierarchy. But if the relations that define the class hierarchy are now optional, if each class can also be related to the others through different attributes, the application is no longer a simple structure; it is a *complex* structure, and the class hierarchy is just one of the structures that make it up.

❖

We can also appreciate the significance of the new feature by imagining that we had to implement the additional relations *without* the ability to change and omit attributes. Thus, for each inherited attribute that we were going to change or omit in a particular class, we would have to go up in the hierarchy, to the level just above the class where that attribute is defined. We would create there

a new class, at the same level as the first one, and identical to it in all respects except for that attribute; in its stead, we would define the *changed* attribute (or we would *omit* the attribute). We would then duplicate, below the new class, the entire section of the hierarchy that lies below the first class. All the lower-level classes here would be identical to those in the original section, but they would inherit the new attribute instead of the original one (or no attribute, if omitted). The application would now be a larger hierarchy, consisting of both the original and the new sections. And in the new section, the counterpart of our original, low-level class would indeed possess the changed attribute (or no attribute), just as we wanted.

With this method, then, we can create classes with changed or omitted attributes but without the benefit of the new feature; that is, without modifying the concept of inheritance. We would have to repeat this procedure, however, for each attribute that must be changed or omitted. So the hierarchy would grow exponentially, because for most attributes we would have to duplicate a section of the hierarchy that is already the result of previous duplications.

It is not the impracticality of this method that concerns us here, though, but the repetition of attributes. Every time we duplicate a section, along with the classes defined in that section we must also duplicate their attributes. Moreover, some of the duplicated attributes will be duplicated again for the next attribute (when we duplicate a section of the new, larger hierarchy), and so on. And we already know that if we repeat attributes, we are creating an incorrect hierarchy: this repetition gives rise to relations that are additional to the strict hierarchical relations, and indicates that we are attempting to represent with one hierarchy a complex structure (see pp. 98–102, 358–360).

What we were trying to accomplish in this imaginary project was to implement through the *original* inheritance concept the kind of relations that we can so easily implement through the *modified* concept, by changing or omitting inherited attributes. Thus, if one method gives rise to a complex structure, the conclusion must be that the other method does too. The non-hierarchical relations may not be obvious when implemented by modifying the concept of inheritance, but we are only deluding ourselves if we believe that the class hierarchy is still the only structure. After all, the very reason for changing and omitting attributes is that we cannot create applications while restricted to one structure. The purpose of the new feature, thus, is to allow us to create multiple, interacting structures.

❖

But even allowing us to change and omit inherited attributes did not make object-oriented programming a practical idea. A second feature had to be

introduced – a second modification to the concept of inheritance. Through this feature, a class can inherit attributes, not just from the higher levels of its own hierarchy, but also from other hierarchies. Called *multiple inheritance*, this feature is seen as an especially powerful enhancement. There are no limitations, of course; a class is not restricted to inheriting only certain attributes from certain hierarchies, or required to inherit *all* the attributes above a certain level. We can now simply add, to any class we want, whichever attributes we need, from any class, from any hierarchy. And this feature can be combined with the first one; that is, after picking the attributes we need, we can change them in any way we like.

Recall the problem we discussed under the second delusion – the need to combine attributes from several hierarchies (database, display, and accounting, for instance). Multiple inheritance is the answer, as we can now select attributes from these hierarchies freely, and thereby create classes with any combination of data types and operations. Without this feature, we saw, the only way to combine attributes is by combining classes: we must employ a traditional language and invoke – in the same module, in the manner of subroutines – classes from several hierarchies.

In conclusion, modifying the concept of inheritance has *downgraded* it: from a formal property of hierarchical structures, to the informal act of copying an attribute from one class to another. And as a result, the relationship between the application's classes has been relaxed: from a strict hierarchy, to multiple and unrestricted connections. If the attributes of a class can be unique, or can be taken from the higher levels, or can be taken from higher levels but changed, or can even be taken from other hierarchies, then what we have is simply classes that can possess *any* attributes. The attributes of a class are no longer determined by its position in the hierarchy, or by the attributes of the other classes.

The theorists continue to use terms like "hierarchy" and "inheritance," but if a class can possess any attributes we like, these terms have lost their original meaning. What they describe now is not a formal class hierarchy, but software entities that possess whatever attributes we need, and are therefore related in whatever ways we need, to implement a particular application. What the modifications have accomplished, in other words, is to restore the programming freedom we had *before* object-oriented programming – the freedom that the new paradigm had attempted to eliminate in its quest for formality and precision.

2

We recognize in the modified concept of inheritance the pseudoscientific stratagem of turning falsifications into features: the theory is saved from refutation by *expanding* it – by incorporating, in the guise of new features, capabilities that were explicitly excluded originally. The original claim was that applications can be developed as strict hierarchies of software classes: either classes that already exist, or classes that can be generated hierarchically from existing ones. The only relations between the classes used in an application, then, would be those established by a hierarchical structure. This restriction is essential if we want to classify and extend software through exact principles, and, ultimately, turn software development into a formal and predictable activity.

The promise, thus, was to turn software development into an activity resembling the design and manufacture of appliances. But this promise can only be fulfilled if software applications, as well as their design and implementation, are restricted to entities and processes that can be represented with isolated hierarchical structures – as are indeed our appliances, and their design and manufacture.

Software applications, though, cannot be developed in this fashion, so the object-oriented theory was refuted. But instead of admitting that it has no practical value, its supporters modified it: they added, in the guise of enhancements, the means to create *multiple* structures – the very feature that the original theory had prohibited. The need to relate software entities through more than one hierarchy is a *falsification* of the object-oriented theory; but the modifications are presented as new and powerful *features* of the theory. These "features" make the theory practical, but they achieve this by contradicting its original principles. It is absurd, therefore, to say that these features enhance the theory, when their very purpose is to bypass the restrictions imposed by the theory.

The fourth delusion, thus, is the belief that what we are practising now, after these modifications, is still object-oriented programming; in other words, the belief that the "power" we gained from the new features is due to the object-oriented principles. In reality, the power derives from *abolishing* these principles, from lifting their restrictions and permitting us to create complex software structures again.

While regaining this freedom, however, we lose the promised benefits. For, those benefits can only emerge if we restrict ourselves to one hierarchy, or perhaps multiple but independent hierarchies – as we do in manufacturing and construction. The theorists praise the benefits of the hierarchical concept, and claim that the object-oriented paradigm is turning programming into a mechanistic activity, but at the same time they give us the means to bypass the mechanistic restrictions. They believe that we can enjoy the promised benefits – formal, exact programming methods – *without* the rigours demanded by the original theory.

So what we are doing after the fourth delusion is merely a more complicated version of what we were doing before the object-oriented paradigm. As was the case with structured programming earlier, what started as an ambitious, formal theory ended up as little more than a collection of programming tips. We are again creating complex software structures, and what is left of the object-oriented principles is just the exhortation to restrict software classes to hierarchical relations, and to avoid other links between them, "as much as possible."

It is indeed a good idea to relate software entities hierarchically. But because our applications consist of multiple, interacting hierarchies, this idea cannot be more than an informal guideline; and, in any case, we can also create hierarchical relations with *traditional* programming means.

In the end, since the idea of independent software structures is a fantasy, the object-oriented theory makes programming more complicated and more difficult, while offering us nothing that we did not already have. We are *not* developing applications through exact, formal methods – the way the experts had promised us. We are creating systems of interacting structures, just as before; so we depend on the non-mechanistic capabilities of our mind, on personal skills and experience, just as before. But by using terms like "objects," "classes," and "inheritance," we can delude ourselves that we are programming under a new paradigm.

The Fifth Delusion

1

The most fantastic object-oriented delusion is undoubtedly the fifth one. The fifth delusion is the belief that we no longer need to concern ourselves with the application's flow of execution: the important relations between the application's objects are those of the class hierarchy, so the relations determining the sequence of their execution can be disregarded.

The application's flow of execution, we recall, was the chief preoccupation of structured programming. The fallacy there was the belief that it is possible to represent applications with *one* flow-control structure. The flow-control structure, according to that theory, is the application's *nesting scheme*: the hierarchical arrangement of modules that makes up the application, plus the hierarchical arrangement of flow-control constructs that makes up each module. And the nesting scheme is depicted by the application's *flow diagram*. The theorists failed to see that the flow diagram depicts *only one* of the nesting schemes; that the *dynamic* structures created by conditional and iterative constructs at run time consist in fact of multiple, overlapping nesting schemes, so the application's flow-control structure is the complex structure that comprises *all* these nesting schemes; and that, moreover, the application's elements are connected through many other *types* of structures – the structures formed by the multitude of software *processes* that make up the application. (Software processes were introduced in chapter 4; see pp. 345–346. The dynamic structures were discussed under structured programming's second delusion; see pp. 542–546.)

The structured programming theory, thus, while mistaken, at least recognized the importance of the flow of execution. The object-oriented theory, on the other hand, ignores it completely. There are no flow diagrams in object-oriented programming. We don't find a single word about conditional and iterative constructs, or about constructs with one entry and exit, or about a restriction to standard constructs. All the problems that structured programming attempted to solve are now neglected. And if an expert mentions them at all, it is only in order to criticize them: It was wrong to represent applications with flow diagrams and flow-control constructs, because these are artificial concepts, designed to match the way *computers* work. These concepts force us to view our affairs unnaturally, and hence develop software that is very different, logically, from the way we deal with the *actual* issues. By replacing the structured programming principles with the concept of class hierarchies, the object-oriented paradigm helps us to build software structures that closely match the real world. Unlike the relations between modules and between flow-control constructs, the relations between software classes are very similar to the way we normally view our affairs.

To verify this claim, let us first recall what are the objects of an application. Each object is an instance of one of the classes defined in the class hierarchy; so the *static* relationship between objects reflects indeed the hierarchical relationship that links the classes. The sequence in which objects are executed, however, is determined, not by the class hierarchy, but by the *messages* they send and receive at run time. An object is executed only when receiving a message from another object in the application. The various operations that an

object is designed to perform are called *methods*, and the particular method selected by the receiving object depends on the parameters accompanying the message. While performing its operations, an object may send messages to other objects, asking those objects to perform some of *their* operations, and so on. Following each message, execution returns to the object and operation that sent the message. Thus, messages, as well as the operations performed in response to messages, are nested hierarchically. And it is this hierarchy of messages and operations – which is *different* from the class hierarchy – that constitutes the application's flow of execution.

So, from the start, we note the same fallacy as in structured programming: the belief that the dynamic structure that represents the application's run-time performance can mirror the static structure of software entities that makes up the application (see pp. 532–533). The static structure – what was the hierarchical flow diagram of modules and constructs in structured programming – is now the hierarchy of classes; and the theorists believe that the neat relations they see in this structure are the only important links between objects. In structured programming, they failed to see the other *types* of structures – those formed by business or software practices, by shared data, and by shared operations; and they also failed to see the multiple *dynamic* flow-control structures. In object-oriented programming, the theorists again fail to see the many types of structures – they believe that each application, and even the totality of existing software, can be represented with one class hierarchy; and they fail to see the flow-control structures altogether, static or dynamic.

It is true that the theorists eventually removed the restriction to one hierarchy. They allowed interacting hierarchies, and they modified the concept of inheritance to create even more interactions. But these ideas contradict the object-oriented principles, negating therefore their benefits. To study the fifth delusion, then, we must separate it from the previous ones: we must assume, with the theorists, that even after modifying the object-oriented principles, even after expanding them to allow complex structures, we can still enjoy the promised benefits. In other words, we must forget that the object-oriented theory has already been refuted. What I want to show here is that the fifth delusion – the failure to deal with the application's flow of execution, and, moreover, the failure to note that it is the same as the flow of execution generated with any other programming method, including structured programming – would alone render the object-oriented theory worthless, even if the previous delusions had not already done this.

2

It is difficult to understand why the theorists ignore the application's flow of execution. For, even a simple analysis reveals that there are just as many deviations from a sequential flow as there were under structured programming. If, for example, we represented with a flow diagram all the conditions, iterations, and object invocations, we would end up with a diagram that looks just like the flow diagrams of structured programming. The theorists discuss the operations performed within each object, and the transfer of control between objects, but they don't see all this as a flow of execution.[1]

Clearly, to perform a particular task the application's elements must be executed by the computer in a specific sequence, no matter what method we use to develop that application. And, since the computer itself cannot be expected to know this sequence, *we* must design it. Now, it ought to be obvious that the relative sequence in which the objects are to be executed cannot be determined solely by the hierarchical relations between classes. This is true because class hierarchies are meant to be used in different applications, so the same objects may have to be executed in a different sequence on different occasions.[2] Thus, if the flow of execution is a critical part of the application's logic but is not determined by the class hierarchy, how are we designing it under the object-oriented paradigm?

There are two parts to the object-oriented flow of execution: *between* objects, and *within* objects. And, despite the new terminology, both parts are practically identical to the flow of execution familiar from earlier forms of programming – namely, between modules and within modules.

Between objects, the transfer of control is implemented by way of messages. And, clearly, sending a message from one object to another is logically and

[1] A half-hearted attempt to deal with the flow of execution is found in the so-called state transition diagrams, used by a few theorists to represent the effect of messages on individual objects. But, like the flow diagrams of structured programming, these diagrams can only depict the *static* aspects of the flow of execution. The *dynamic* aspects (the combined effect of messages in the running application) constitute a complex phenomenon, so they cannot be reduced to an exact, mechanistic representation.

[2] In fact, even if each application had its own class hierarchy, we would need more than a simple hierarchical structure to represent its flow of execution. As we saw under structured programming, if the sequence in which the application's elements are executed was determined solely by their relative position in the hierarchical nesting scheme, the application would be useless, because it would always do the same thing (see p. 533). Similarly now, the sequence in which the objects are executed must be determined by factors other than their relative position in the hierarchical class structure.

functionally identical to invoking a module or subroutine in traditional programming. *Within* objects, we can distinguish between the jump performed in order to select the so-called method (the object's response to a particular message) and the jumps performed by the operations that make up the method. Selecting a method is in effect a conditional flow-control construct (where the condition involves the values received as parameters with the message). Thus, while object-oriented languages may well offer a specialized construct, we could just as easily implement this selection with traditional constructs like IF or CASE. As for the operations that make up the methods, they are, of course, ordinary pieces of software: statements, blocks of statements, conditions, and iterations. These operations, therefore, are as rich in flow-control constructs as are the operations found in traditional languages.

But it is important to note that the messages themselves are, in effect, operations within methods. This is true because a message may be sent from within a conditional or iterative construct that is part of a method. Consequently, the execution of objects in a running application is not one nesting scheme but a *system* of nesting schemes. Just like the modules invoked in structured programming, the nested invocations of objects would form a simple hierarchical structure only if the methods included sequential constructs alone. Just as in structured programming, the purpose of conditional and iterative constructs is to create multiple dynamic nesting schemes (see pp. 541–544).

The role of the flow-control constructs, thus, is to create complex flow-control structures not just within methods, but also between objects. So, when disregarding the effect of the flow-control constructs on the operations within methods, the theorists also disregard their effect on the flow of execution between objects. In the end, not only are the application's objects subject to a flow of execution, but this execution forms a complex structure, just like the execution of modules in structured programming.

To conclude, the flow of execution in an application created through object-oriented programming is identical, for all practical purposes, to the one implemented through structured programming. And the latter, we recall, after annulling the restriction to standard flow-control constructs, was identical to the flow of execution implemented through any other programming method.[3]

[3] The object-oriented flow of execution is, in fact, even more complex than the one in structured programming (because a message may be sent to several objects simultaneously, an object may continue execution while waiting for the reply to a message, etc.). So the number of flow-control structures that we must deal with in our mind is even greater. Moreover, we must remember that the so-called hybrid languages (employed, actually, in all object-oriented systems) provide also the traditional concept of modules and subroutines, thereby adding to the number of flow-control structures.

❖

Both structured programming and object-oriented programming promised to revolutionize software development by restricting applications to a simple hierarchical structure. And when this idea turned out to be a fantasy, both theories were expanded so as to provide the means to create complex software structures again; in particular, complex *flow-control* structures. Thus, like all pseudoscientific theories, they ended up restoring the very features they had excluded in the beginning, and on the exclusion of which they had based their claims. So what we have in the end, after all the "enhancements," is some complicated programming concepts that offer us exactly what we had, in a much simpler form, before the theory. Still, no one sees this reversal as a failure of the theory. The promised benefits, possible *only* if applications are restricted to a simple structure, are now lost. The theory, nevertheless, continues to be promoted with the original claims.

The fifth delusion, thus, is similar to the previous ones: we believe that we can enjoy the benefits promised by the object-oriented paradigm even after annulling the object-oriented principles and reinstating the means to create complex structures. What we are creating now is complex *flow-control* structures. First, by introducing the concept of messages into object-oriented programming, we provide the means to link the application's objects through relations that are different from their relations in the class hierarchy. In other words, the sequence in which the objects are executed by the computer – the hierarchical nesting scheme that is the flow of execution – need not depend on their relative position in the class hierarchy. The application's objects, then, will belong to two different structures at the same time: a class hierarchy and a flow-control hierarchy. Second, by allowing messages to be controlled by conditional and iterative constructs, we turn the flow-control hierarchy itself into a complex structure: not *one* nesting scheme, but a *system* of nesting schemes.

3

Although we are discussing *flow-control* structures, we must not forget that objects, like their counterpart, subroutines, also give rise to a different *type* of structures. If an object is invoked from several other objects in the application, it necessarily links those objects logically. So, like subroutines, objects constitute a special case of shared operations (see pp. 351–354). For each object, we can represent with a hierarchical structure the unique way in which the application's other objects are affected by it. And the relations created by these

structures will be different from those created by the flow-control structures or by the class hierarchies.

It is the concept of messages that makes all the additional structures possible. Without messages, the application's objects would be related only through class hierarchies, the way it was originally intended. So the concept of messages, described as an important object-oriented feature, was introduced specifically in order to override the limitations of the original principles. The theorists ignore completely the relations engendered by messages. They give us the means to link the application's objects through additional structures, but they continue to present the object-oriented concept as if the objects were linked only through class hierarchies. What is the point in designing strict class hierarchies if we are going to relate the same objects in many other ways, by means of messages, while the application is running?

In structured programming, the dream was to reduce the flow of execution to one structure, as this would permit us to represent the running application mathematically. And this idea failed because it was too restrictive, because applications must have *multiple* flow-control structures if they are to represent the world accurately. The object-oriented model is said to be more powerful. But when we examine this power, we find that it derives simply from lifting the restrictions introduced by structured programming; it derives from allowing us to link objects in any way we want, and in particular, to link them from the perspective of the flow of execution in any way we want. (Some of these restrictions had been lifted even under structured programming, when the theorists allowed us to use non-standard constructs and GOTO.)

By disregarding the effect of conditions and iterations, by refusing to draw flow diagrams, and by giving old concepts new names, the software experts managed to persuade us that the application's elements are related only through class hierarchies, so we no longer need to concern ourselves with the sequence of their execution. But, in the end, to create applications we are doing what we had been doing all along. The only real change is calling subroutines "objects," their invocation "messages," and their internal operations "methods."

So the power said to inhere in the object-oriented paradigm does not derive from the new programming concepts, but simply from having more opportunities to create complex software structures. What the theorists did was merely restore some of the programming freedom we had before structured programming, and invent some new terminology. The claim that this freedom is due to the object-oriented paradigm is a fraud. The freedom to connect the application's elements in any way we like is a freedom we always had, through any programming language – and, besides, without having to depend on complicated development environments.

The Final Degradation

1

We saw how, through several delusions, the idea of object-oriented programming was degraded from a strict theory to a set of informal concepts. These concepts, moreover, are practically identical to those we had *before* the theory. But the degradation did not end with those delusions. In addition to the traditional concepts, a number of new features and principles were added over the years to the object-oriented idea. Totally unrelated to the original theory, these enhancements were inspired by various concepts that were being introduced into programming languages in the same period. In other words, any concept found useful was labeled "object-oriented," and was incorporated into this theory too. Thus, the notion of object-oriented programming became increasingly vague, and the terms "object" and "object-oriented" were applied to almost any feature and principle.

The final degradation, then, was the degradation in expectations: from the original idea of finding a formal way to reuse software, to a preoccupation with isolated programming concepts. If the theory was promoted at first with the claim that it would revolutionize programming, in the end, when the revolution did not materialize, the same theory was promoted by praising merely its features and principles. Thus, the benefits of individual programming concepts replaced the benefits originally claimed for the theory, as the ultimate goal of object-oriented programming. Let us briefly study some of these fallacies.

❖

I have already mentioned that the concept of hierarchies, and the related concepts of inheritance and abstraction, were known and appreciated long before the object-oriented theory. The concept of abstraction, in particular, is praised now as if the only way to benefit from it were with classes and objects. We are told, for example, that the object-oriented paradigm allows us to define abstract software entities, and then create actual instances of these entities by adding some lower-level attributes. The instances will differ from one another in their details, while sharing the broader attributes of the original entities.

Abstraction, however, is not peculiar to the object-oriented theory. It is, in fact, a fundamental programming principle. We make use of abstraction in any programming language, and in any programming task. The very essence of programming is to create data and operations of different levels of abstraction. Thus, merely calling subroutines hierarchically, and passing data by means of

parameters, creates in effect levels of abstraction; and merely using variables and fields, which hold entities that differ in value while sharing certain attributes, is, again, a form of abstraction. It would be impossible to program serious applications if we restricted ourselves to software entities that cannot be altered, or extended, or grouped, or used in different contexts; in other words, if we did not make use of the concept of abstraction. Structured programming too, although criticized now, was based on abstraction: the flow-control constructs perform the same function at different levels of nesting.

Another object-oriented concept that is in reality a fundamental programming principle is *information hiding*, or *encapsulation*. We are told that the new paradigm allows us to hide inside an object the details of its operations, so that the other objects may know its capabilities without having to know how they are implemented. One of the benefits of this principle is that if we later modify an object, we won't have to modify also the objects that communicate with it. Object-oriented textbooks praise this principle and show us examples of situations where extensive modifications are avoided through object-oriented programming, alleging that this is the first time we can benefit from it. But the principle is a well-known one, and is found in every programming language (for example, in the use of subroutines and local variables). Experienced programmers always strive to keep software entities independent. Only the terms "information hiding" and "encapsulation" are new.

Along with encapsulation, we are told that keeping the data and the operations that act on it together, as one entity, is a new concept. This, we are told, is more natural than the traditional methods, which treated data and operations as separate entities. Actually, we always designed software in this fashion, when appropriate. And we didn't need a special development environment to do it: we simply ensured that a module uses local variables, or is the only one to use certain global variables. It is absurd to call this well-known programming style a new concept.

The very fact that notions like abstraction and encapsulation, understood and appreciated since the 1950s, are seen as a revolution and a new paradigm demonstrates the ignorance that the theorists and the practitioners suffer from. All that the object-oriented environments do is *formalize* these notions; that is, provide them in the form of built-in features, forcing us to depend on them. But, as we saw, this idea failed. It failed because, no matter how useful the hierarchical model is, we cannot *restrict* ourselves to hierarchical relations. So, in the end, the means to use and relate software entities freely – what we had been doing through traditional programming – had to be restored.

Other claims are even sillier. *Polymorphism* is the principle of implementing an operation in several different ways while providing a common interface. For example, different objects could be designed to print different types of

documents, but this fact would be hidden from the rest of the application; we would always invoke one object, called "print," and the appropriate printing object would be invoked automatically, depending on the type of document to be printed. This is indeed a good programming technique, but what has it to do with the object-oriented theory? Polymorphism is described as one of the most important object-oriented principles, while being in reality a simple and common programming method, easily implemented in any language by means of subroutines and conditional constructs. And even if the concept of classes and objects simplifies sometimes its implementation, this is hardly a programming revolution. The object-oriented propaganda, though, presents this simple principle as if without classes and objects we would have to duplicate pieces of software all over the application every time we had to select one of several alternatives in a given operation.

Overloading is another concept described as an object-oriented principle, while being known, in fact, for a long time. Overloading allows us to redefine the function of a symbol or a name, in order to use it in different ways on different occasions. The operator *plus*, for example, is used with numbers; but we could also use it with character strings, by redefining its function as string concatenation. In a limited form, this feature is available in most programming languages; and, in any case, it can be easily implemented by means of subroutines and conditional constructs. Object-oriented languages do provide greater flexibility, but, again, this is just a language feature, not a programming revolution; and it has nothing to do with the object-oriented theory.

In conclusion, abstraction, information hiding, polymorphism, and the rest, are just a collection of programming principles, which can also be added to a traditional language. And if not directly available in a language, we can implement these principles by adopting an appropriate programming style. The software experts describe these principles as if *they* constituted the object-oriented theory; but if in one form or another we always had them, in what sense is this theory a new paradigm?

It is perhaps easier to implement some of these principles with an object-oriented language (that is, if we overlook the fact that we must first agree to depend on an enormously complex development environment). But this quality is not what the experts had promised us as the benefits of the theory. The promised benefits were not abstraction, encapsulation, or polymorphism, but the "industrialization" of software: the prospect of creating software applications the way we build appliances, through a process akin to the assembly of prefabricated components. It was its promises, not its principles, that made the object-oriented idea popular; the principles were merely the means to attain the promised benefits. In any case, after all the delusions, we no longer have the original theory; what we have now is just a more

complicated way to program. So, since the promised benefits were lost with the original theory, the principles alone are perceived now as the benefits of object-oriented programming.

❖

We saw earlier how structured programming underwent a process of degradation: it started as a formal theory, promising us error-free software; and it ended as a preoccupation with trivial concepts like top-down design, constructs with one entry and exit, and avoiding GOTO. Now we see that a similar process of degradation, from an ambitious theory to a collection of trivial concepts, also affected object-oriented programming.

It is easy to understand the reason for this degradation. When the benefits promised by a theory are not forthcoming (we still don't create applications mathematically, or by assembling prefabricated software components), we can either admit that the theory has failed, or attempt to rescue it. The only way to rescue an invalid theory is by making it unfalsifiable; specifically, by *expanding* it, so that events which would normally falsify it no longer do so. And this can be accomplished by replacing the original principles with broader and simpler ones, which can be easily implemented. Thus, if we redefine structured programming or object-oriented programming to mean just a collection of programming principles, and if some of these principles are useful, then the *redefined* theory is indeed valid.

Both structured programming and object-oriented programming became in the end unfalsifiable, and hence pseudoscientific. Thanks to the various "enhancements," and to their degradation from a formal theory to a collection of principles, they became impossible to refute. Had they retained their original, exact definition, it would be obvious that they failed, simply because we are still not enjoying the claimed benefits. But by reducing them to an assortment of simple and well-known principles, they appear to work even if the claimed benefits never materialize. Indenting statements, expressing requirements hierarchically, information hiding, and the like, are indeed excellent principles; so, if *this* is what the theories are now, it is impossible to criticize them.

2

The degradation of the object-oriented idea can also be seen in the degradation of the *terms* "object" and "object-oriented." We saw earlier how the term "structured" was applied to almost any flow-control construct, and to almost

any software-related activity. For example, the theorists allowed into structured programming any construct that was useful – simply because, after drawing around it a rectangular box with one entry and one exit, it looked like a structured construct. This trick worked so well for structured programming that the theorists repeated it with objects.

In the original theory, objects were formal, precisely defined entities. But the idea of an object has been degraded to such an extent that the term "object" can now be used to designate any piece of software. Such entities as data records, display screens, menus, subroutines, and utilities are called objects – simply because, like objects, they can be invoked, or possess attributes, or perform actions. In other words, we can take any software entity, draw a box around it, and call the result an object.

Even entire programs can be called, if we want, objects. For example, through a procedure called *wrapping*, an old application, or part of an application, written in a traditional language, can instantly become an object.[1] The application itself remains unchanged; but, by "wrapping" it (that is, adding a little software around it so that it can be *invoked* in a new fashion), it can become part of an object-oriented environment: "Wrapper technology … provides an object-oriented interface to legacy code. The wrapped piece of legacy code behaves as an object."[2]

Along with the idea of an object, the object-oriented principles themselves were degraded. Thus, any programming feature, method, or technique that involves hierarchies, or abstraction, or encapsulation, and any development system that includes some of these principles, is called "object-oriented." We can see this degradation in books, articles, and advertising. And, since the use of these terms is perceived as evidence of expertise and modernity, ignorant academics, programmers, and managers employ them liberally in conversation. Thus, "object" and "object-oriented" are now little more than slogans, not unlike "technology," "power," and "solution."

In the end, the definition of object-oriented programming was degraded to the point where the original promises were forgotten altogether, and the criterion of success became merely whether an application can be developed at all through object-oriented concepts (or, rather, through what was left of these concepts after all the delusions). Thus, the success stories we see in the media are not about companies that achieved a spectacular reuse of existing software classes, or managed to reduce formally all their business requirements to a class

[1] See, for example, Daniel Tkach and Richard Puttick, *Object Technology in Application Development* (Redwood City, CA: Benjamin/Cummings, 1994), pp. 113–115.

[2] Ibid., p. 148. Note, again, the slogan "technology": what is in fact a simple programming concept (code wrapping) is presented as something important enough to name a whole domain of technology after it.

hierarchy, but about companies that are merely *using* a system, language, or methodology said to be object-oriented.

An example of this type of promotion is *Objects in Action*.[3] This book includes nineteen case studies of object-oriented development projects, from all over the world. For each project, those involved in its implementation describe in some detail the requirements and the work performed. These projects were selected, needless to say, because they were exceptional.[4] But, while presented as object-oriented successes, there is nothing in these descriptions to demonstrate the benefits of object-oriented programming. The only known fact is that certain developers implemented certain applications using certain object-oriented systems. There is no attempt to prove, for instance, that some other developers, experienced in traditional programming, could *not* have achieved the same results. Nor is there an attempt to understand why thousands of other object-oriented projects were *not* successful. In the end, there is nothing in these descriptions to exclude the possibility that the successes had nothing to do with the object-oriented principles, and were due to other factors (the type of applications, the particular companies where they were developed, unusual programming skills, etc.).

It is when encountering this kind of promotion that we get to appreciate the importance of Popper's idea; namely, that it is not the *confirmations* of a theory that we must study, but its *falsifications* (see "Popper's Principles of Demarcation" in chapter 3). As we just saw, if what we want to know is how useful the object-oriented principles really are, those success stories can tell us nothing. Promoters use success stories as evidence precisely because such stories can always be found and are so effective in fooling people. For, few of us understand why confirmations are worthless. The programming theories, in particular, are always promoted by pointing to isolated successes and ignoring the many failures. Thus, the very fact that the elites rely on this type of evidence demonstrates their dishonesty and the pseudoscientific nature of their theories.

3

The previous theory, structured programming, was promoted with the claim that it provides certain benefits; and we saw that, in fact, these benefits can be attained simply through good programming. In other words, those structured programming principles that are indeed useful can be implemented

[3] Paul Harmon and David A. Taylor, *Objects in Action: Commercial Applications of Object-Oriented Technologies* (Reading, MA: Addison-Wesley, 1993).

[4] This is acknowledged in the book: ibid., p. vii.

without the restrictions imposed by this theory. The motivation for structured programming, therefore, was not a desire to improve programming practices, but the belief that it is possible to get inexperienced programmers to perform tasks that demand expertise. What was promoted as an effort to turn programming into an exact activity was in reality an attempt to raise the level of abstraction in this work, so as to remove both the need and the possibility for programmers to make important decisions.

The software theorists assumed that the skills acquired after a year or two of practice represent the highest level that a typical programmer can attain. Thus, since these programmers create bad software, the conclusion was that the only way to improve their performance is by reducing programming to a routine activity. Anyone capable of acquiring mechanistic knowledge – capable, that is, of following rules and methods – would then create good software.

And this corrupt ideology was also the motivation for object-oriented programming. The true goal was, again, not to improve programming practices, but to raise the level of abstraction, in the hope of getting inexperienced programmers to perform tasks that lie beyond their capabilities. As we saw, those object-oriented principles that are indeed useful – abstraction, code reuse, information hiding, and the like – were always observed by good programmers. Those principles, moreover, can be implemented through any programming language. Just as they do not have to avoid `GOTO` in order to enjoy the benefits of hierarchical flow-control structures, good programmers do not have to use an object-oriented environment in order to create software that is easy to reuse, modify, and extend.

Ultimately, the object-oriented paradigm is merely another attempt to incorporate certain programming principles into development systems and methodologies, so as to allow programmers who are incapable of understanding these principles to benefit from them nonetheless. Just as the operator of a machine can use it to fabricate intricate parts without having to understand engineering principles, the new systems and methodologies would enable a programmer to fabricate software parts without having to understand the principles behind good programming.

Thus, like structured programming before it, object-oriented programming was not an attempt to turn bad programmers into good ones, but to eliminate the *need* for good ones. Each theory claimed to be the revolution that would turn programmers from craftsmen into modern engineers; but, in reality, programmers had neither the skills of the old craftsmen before the theory, nor the skills of engineers after it.

All that mechanistic theories can hope to accomplish is to turn ignorant programmers into ignorant operators of software devices. But we can only

incorporate in devices *mechanistic* principles, while our applications must mirror *non-mechanistic* phenomena. So, to permit programmers to create useful applications, the theories must abandon in the end their restriction to mechanistic principles. They restore in roundabout and complicated ways the low levels of abstraction, and the means to link software structures, thereby bringing back the most challenging aspect of programming – the need to manage complex structures. Thus, not only do these theories fail to eliminate the need for non-mechanistic knowledge, but, by forcing programmers to depend on complicated concepts and systems, they make software development even more difficult than before.

Each time they get to depend on a mechanistic theory instead of simply practising, programmers forgo the only opportunity they have to improve their skills. Their performance remains at novice levels, and they believe that the only way to make progress is by adopting the *next* mechanistic theory. Professional programming, the elites keep telling them, means being familiar with the latest concepts and development systems.

Both structured programming and object-oriented programming are an expression of our mechanistic software ideology – an ideology promoted by universities and by the software companies. It is in the interest of these elites to prevent the evolution of a true programming profession. By redefining programming expertise as the capability to follow methods and to operate devices, the mechanistic ideology has reduced programmers to bureaucrats.