SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*

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

Copyright © 2013, 2019 Andrei Sorin

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

This chapter analyzes the mechanistic fallacies inherent in the idea of software engineering, and exposes the pseudoscientific nature of the mechanistic programming theories.

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

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

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

The free digital book is a complete copy of the print book, and is licensed under the Creative Commons Attribution-NoDerivatives International License 4.0. You may download it and share it, but you may not distribute modified versions.

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 xiii

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

Chapter 1 **Mechanism and Mechanistic Delusions 68**
The Mechanistic Philosophy 68
Reductionism and Atomism 73
Simple Structures 90
Complex Structures 96
Abstraction and Reification 111
Scientism 125

Chapter 2 **The Mind 140**
Mind Mechanism 141
Models of Mind 145

Tacit Knowledge 155
Creativity 170
Replacing Minds with Software 188

Chapter 3 Pseudoscience 200
The Problem of Pseudoscience 201
Popper's Principles of Demarcation 206
The New Pseudosciences 231
  The Mechanistic Roots 231
  Behaviourism 233
  Structuralism 240
  Universal Grammar 249
Consequences 271
  Academic Corruption 271
  The Traditional Theories 275
  The Software Theories 284

Chapter 4 Language and Software 296
The Common Fallacies 297
The Search for the Perfect Language 304
Wittgenstein and Software 326
Software Structures 345

Chapter 5 Language as Weapon 366
Mechanistic Communication 366
The Practice of Deceit 369
The Slogan "Technology" 383
Orwell's Newspeak 396

Chapter 6 Software as Weapon 406
A New Form of Domination 407
  The Risks of Software Dependence 407
  The Prevention of Expertise 411
  The Lure of Software Expedients 419
Software Charlatanism 434
  The Delusion of High Levels 434
  The Delusion of Methodologies 456
The Spread of Software Mechanism 469

Chapter 7 Software Engineering 478
Introduction 478
The Fallacy of Software Engineering 480
Software Engineering as Pseudoscience 494

Structured Programming 501
The Theory 503
The Promise 515
The Contradictions 523
The First Delusion 536
The Second Delusion 538
The Third Delusion 548
The Fourth Delusion 566
The *GOTO* Delusion 586
The Legacy 611
Object-Oriented Programming 614
The Quest for Higher Levels 614
The Promise 616
The Theory 622
The Contradictions 626
The First Delusion 637
The Second Delusion 639
The Third Delusion 641
The Fourth Delusion 643
The Fifth Delusion 648
The Final Degradation 655
The Relational Database Model 662
The Promise 663
The Basic File Operations 672
The Lost Integration 687
The Theory 693
The Contradictions 707
The First Delusion 714
The Second Delusion 728
The Third Delusion 769
The Verdict 801

Chapter 8 From Mechanism to Totalitarianism 804
The End of Responsibility 804
Software Irresponsibility 804
Determinism versus Responsibility 809
Totalitarian Democracy 829
The Totalitarian Elites 829
Talmon's Model of Totalitarianism 834
Orwell's Model of Totalitarianism 844
Software Totalitarianism 852

Index 863

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.

CHAPTER 7

Software Engineering

Introduction

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

In chapter 6 we examined the three stages in the spread of mechanistic software concepts: the domain of programming, the world of business, and our personal affairs (see pp. 472–477). And we saw that, while the first stage is now complete, the others are still unfolding. Judged from this perspective, the present chapter can also be seen as a study of the *first* stage. Since this stage involves events that took place in the past, its study can be exact and

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

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

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

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

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

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

The Fallacy of Software Engineering

1

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[12] Ibid., p. 11 (italics added).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

By the mid-1960s, most software concepts that are mechanistic and also practical had been discovered. But the theorists could not accept the fact that the easy and dramatic advances were a thing of the past, that we could not expect further improvements in programming productivity simply by adopting a new method or principle. They were convinced that similar advances would take place in the future, that there exist many other mechanistic concepts, all waiting to be discovered. To pick just one example, they noticed the increase in programming productivity achieved when moving from low-level to high-level languages, and concluded that other languages would soon be invented with even higher levels of abstraction, so the same increase in productivity would be repeated again and again. (The notion of "generations" of languages, still with us today, reflects this fantasy; see pp. 452–453.)

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

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

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

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

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

3

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

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

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

The software theorists note the higher levels of knowledge attained by

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

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

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

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

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

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

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

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

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

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

Software Engineering as Pseudoscience

1

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

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

These definitions are correct insofar as they describe the programming principles recommended by the software theorists. But we have no evidence that it is possible to develop actual software applications by adhering to these principles. We saw in the previous section that the very term "software engineering" constitutes a circular definition, since it was adopted without determining first whether programming is indeed a form of engineering; it was adopted because the software theorists *wished* programming to be like engineering (see pp. 481–483). And the same circularity is evident in the

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2

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

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

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

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

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

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

Any hypothesis can form the basis of a delusion, and hence a pseudoscience. So we should not be surprised that the popular *mechanistic* hypothesis has been such a rich source of delusions and pseudosciences. Because of their similarity to the traditional pseudosciences, I have called the delusions engendered by the mechanistic hypothesis *the new pseudosciences* (see pp. 201–202). Unlike the traditional ones, though, the new pseudosciences are pursued by respected scientists, working in prestigious universities.

Let us recall how a mechanistic delusion turns into a pseudoscience (see pp. 202–203, 231–233). The scientists start by committing the fallacy of reification: they assume that a model based on one isolated structure can provide a useful approximation of the complex phenomenon, so they extract that structure from the system of structures that make up the actual phenomenon. In complex phenomena, however, the links between structures are too strong to be ignored, so their model does not represent the phenomenon closely enough to be useful. What we note is that the theory fails to explain certain events or situations. For example, if the phenomenon the scientists are studying involves minds and societies, the model fails to explain certain behaviour patterns, or certain intelligent acts, or certain aspects of culture.

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

A theory can be said to work when it successfully explains and predicts; if it must be expanded continually because it fails to explain and predict some events, then, clearly, it does *not* work. Defending a mechanistic theory looks like scientific research only if we regard the quest for mechanistic explanations as an indisputable method of science. Thus, the mechanists end up doing in the name of science exactly what pseudoscientists do when defending *their* theories. When expanding the theory, when making it agree with an increasingly broad range of situations, what they do in effect is annul, one by one, its original claims; they make it less and less precise, and eventually render it worthless (see pp. 223–224).

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

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

3

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

Recall how a mechanistic software theory turns into a pseudoscience. Software applications are systems of interacting structures. The structures that make up an application are the various *aspects* of the application. Thus, each software or business practice, each file with the associated operations, each subroutine with its calls, each memory variable with its uses, forms a simple structure. And these structures interact, because they share their elements – the various software entities that make up the application (see pp. 345–346).

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

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

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

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

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

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

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

Structured Programming

Structured programming occupies a special place in the history of software mechanism. Introduced in the 1970s, it was the first of the great software theories, and the first one to be described as a *revolution* in programming principles. It was also the first attempt to solve the so-called software crisis, and it is significant that the solution was seen, even then, not in encouraging programmers to improve their skills, but in finding a way to eliminate the *need* for skills.

Thus, this was the first time that programming expertise was redefined to mean expertise in the use of substitutes for expertise – methods, aids, or devices supplied by a software elite. This interpretation of expertise was so well received that it became the chief characteristic of all the theories that followed.

Another common characteristic, already evident in structured programming, is the enthusiasm accompanying each new theory – an enthusiasm that betrays the naivety of both the academics and the practitioners. Well-known concepts – the hierarchical structure, the principles of reductionism and atomism – are rediscovered again and again, and hailed as great advances, as the beginning of a *science* of programming. No one seems to notice that, not only are these concepts the same as in the previous software delusions, but they are the same as in every mechanistic delusion of the last three centuries.

Structured programming was the chief preoccupation of practitioners and academics in the 1970s and 1980s. And, despite the occasional denial, it continues to dominate our programming practices. The reason this is not apparent is our preoccupation with more recent theories, more recent revolutions. But, even though one theory or another is in vogue at a given time, the principles of structured programming continue to be obeyed as faithfully as they were in the 1970s. The GOTO superstition, for example, is as widespread today as it was then.

Finally, it is important to study structured programming because it was this theory that established the software bureaucracy, and the culture of programming incompetence and irresponsibility. The period before its introduction was the last opportunity our society had to found a true programming profession. For, once the bureaucrats assumed control of corporate programming, what ensued was inevitable. It was the same academics and gurus who invented the following theories, and the same programmers and managers who accepted them, again and again. The perpetual cycle of promises and disappointments – the cycle repeated to this day with each new methodology, programming aid, or development environment – started with structured programming. Since the same individuals who are naive enough to accept one theory are called upon, when the theory fails, to assess the merits of the next one, it is not surprising that the programming profession has become a closed, stagnating culture. Once we accepted the idea that it is not programming expertise that matters but familiarity with the latest substitutes for expertise, it was inevitable that precisely those individuals *incapable* of expertise become the model of the software professional.

The Theory

1

To appreciate the promise of structured programming, we must take a moment to review the programming difficulties that prompted it. Recall, first, what is the essence of software. The operations that make up a program are organized in logical constructs – mostly conditions and iterations – which reflect the relations between the processes and events we want to represent in software. A typical software module, therefore, is not just a series of consecutive operations, but a combination of blocks of operations that are executed or bypassed, or are executed repeatedly, depending on various run-time conditions. Within each block, we can have, in addition to the consecutive operations, further conditions and iterations. In other words, these constructs can be nested: a module can have several levels of conditions within conditions or iterations, and iterations within conditions or iterations. Blocks can be as small as one operation, or statement, but usually include several. And if we also remember that certain operations serve to invoke other modules at run time, then, clearly, applications of any size can be created in this manner.[1]

The conditional construct consists of a condition (which involves, usually, values that change while the application is running) and two blocks; with this construct, the programmer specifies that the first block be executed when the condition is evaluated as *True*, and the second block when evaluated as *False*. (In practice, one of the two blocks may be empty.) The iterative construct consists of a condition (which involves usually values that change from one iteration to the next) and the block that is to be executed repeatedly; with this construct, the programmer specifies that the repetition continue only as long as the condition is evaluated as *True*. (Iterative software constructs are known as *loops*.) Conditions and iterations are *flow-control* operations, so called because they control the program's flow of execution; that is, the sequence in which the other kinds of operations (calculations, assigning values to variables, displaying data, accessing files, etc.) are executed by the computer.

Modifying the flow of execution entails "jumping" across blocks of operations: jumping forward, in order to prevent the execution of an unwanted

[1] Some definitions: *Block* denotes here any group of consecutive, related operations (not just the formal units by this name in block-structured languages). *Operation* denotes the simplest functional unit, which depends on the programming language (one operation in a high-level language is usually equivalent to several, simpler operations in a low-level language). *Statement* denotes the smallest executable unit in high-level languages. Since structured programming and the other theories discussed here are concerned mainly with high-level languages, "statement" and "operation" refer to the same software entities.

block, or jumping backward, in order to return to the beginning of a block that must be repeated. These jumps are necessary because the sequence of operations that make up a program constitutes, essentially, a one-dimensional medium. Physically (in the program's listing, or when the program resides in the computer's memory), all possible operations must be included, and they can appear in only one sequence. At run time, though, the operations must be executed selectively, and in one sequence or another, depending on how the various conditions are evaluated. In other words, the *dynamic* sequence may be very different from the *static* one. The only way to execute the operations differently from their static sequence is by instructing the computer at various points to jump to a certain operation, forward or backward, instead of continuing with the next one. For example, although the two blocks in a conditional construct appear consecutively (both in the listing and in memory), only one must be executed; thus, the first one must be bypassed when the second one is to be executed, and the second one must be bypassed when the first one is to be executed.

It is the programmer's task to design the intricate network of jumps that, when the application is running, will cause the various operations to be executed, skipped, or repeated, as required. To help programmers (and the designers of compilers) create efficient machine code, computers have a rich set of *low-level* flow-control features: conditional and unconditional jump instructions, loop instructions, repeat instructions, index registers, and so forth. These features are directly available in low-level, assembly languages, and their great diversity reflects the important role that flow-control operations play in programming. In high-level languages, the low-level flow-control features are usually available only as part of complex, built-in operations. For example, a statement that compares two character strings in the high-level language will use, when translated by the compiler into machine code, index registers and loop instructions.

The jump operation itself is provided in high-level languages by the GO TO statement (often spelled as one word, GOTO); for example, GO TO L2 tells the computer to jump to the statement following the label L2, instead of continuing with the next statement. While it is impossible to attain in high-level languages the same versatility and efficiency as in assembly languages, the GO TO statement, in conjunction with conditional statements and other features, allows us to create all the flow-control constructs we need in those applications normally developed in high-level languages.

Programmers, it was discovered from the very beginning, cannot easily visualize the flow of execution; and, needless to say, without a complete understanding of the flow of execution it is all but impossible to design an application correctly. What we note is software defects, or "bugs": certain

operations are not executed when they should be, or are executed when they shouldn't be.

Serious applications invariably give rise to intricate combinations of flow-control constructs, simply because those affairs we want to represent in software consist of complex combinations of processes and events. It is the *interaction* of flow-control constructs – the need to keep track of combinations of constructs – that poses the greatest challenge, rather than merely the *large number* of constructs. Even beginners can deal successfully with *separate* constructs; but nested, interacting constructs challenge the skills of even the most experienced programmers. The problem, of course, is strictly a human one: the limited capacity of our mind to deal with combinations of relations, nestings, and alternatives. The computer, for its part, will execute an involved program as effortlessly as it does a simple one. Like other skills, though, it is possible to improve our ability to deal with structures of flow-control constructs. But this can only be accomplished through practice: by programming and maintaining increasingly complex applications over many years.

The flow-control constructs, then, are one of the major sources of programming errors. The very quality that makes software so useful – what allows us to represent in our applications the diversity and complexity of our affairs – is necessarily also a source of programming difficulties. For, we must *anticipate* all possible combinations, and describe them accurately and unambiguously. Experienced programmers, who have implemented many applications in the past, know how to create flow-control constructs that are consistent, economical, streamlined, and easy to understand. Inexperienced programmers, on the other hand, create messy, unnecessarily complicated constructs, and end up with software that is inefficient and hard to understand.

Business applications must be modified continuously to keep up with the changing needs of their users. This work is known as maintenance, and it is at this stage, rather than in the initial development, that the worst consequences of bad programming emerge. For, even if otherwise successful, badly written applications are almost impossible to keep up to date. This is true because applications are usually maintained by different programmers over the years, and, if badly written, it is extremely difficult for a programmer to understand software developed and modified by others. (It was discovered, in fact, that programmers have difficulty understanding even *their own* software later, if badly written.)

Without a good understanding of the application, it is impossible to modify it reliably. And, as in the initial development, the flow-control constructs were found to be especially troublesome: the more nestings, jumps, and alternatives there are, the harder it is for a programmer to visualize, from its

listing, the variety of run-time situations that the application was meant to handle. The deficiencies caused by incorrect modifications are similar to those caused by incorrect programming in the initial development. For a live application, however, the repercussions are far more serious. So, to avoid jeopardizing their applications, businesses everywhere started to limit maintenance to the most urgent requirements, and the practice of developing new applications as a substitute for keeping the existing ones up to date became widespread. The ultimate consequences of bad programming, thus, are the perpetual inadequacy of applications and the cost of replacing them over and over.

These, then, were the difficulties that motivated the search for better programming practices. As we saw earlier, these difficulties were due to the incompetence of the application programmers. But the software theorists were convinced that the solution lay, not in encouraging programmers to improve their skills, but in discovering methods whereby programmers could create better applications while remaining incompetent. And, once this notion was accepted, it was not hard to invent such a method. It was obvious that inexperienced programmers were creating bad flow-control constructs, and just as obvious that this was one of the reasons for programming inefficiency, software defects, and maintenance problems. So it was decided to prevent programmers from creating their own flow-control constructs. Bad programmers can create good applications, the theorists declared, simply by restricting themselves to the existing flow-control constructs. This restriction is the essence of structured programming.

Most high-level languages of that time already included statements for the basic flow-control constructs, so all that was needed to implement the principles of structured programming was a change in programming style. Specifically, programmers were asked to use one particular statement for conditions, and one for iterations. What these statements do is eliminate the need for explicit jumps in the flow of execution, so the resulting constructs – which became known as the *standard* constructs – are a little simpler than those created by a programmer with GOTO statements. The jumps are still there, but they are now implicit: for the conditional construct that selects one of two blocks and bypasses the other, we only specify the condition and the two blocks, and the compiler generates automatically the bypassing jumps; and for a loop, the compiler generates automatically the jump back to the beginning of the repeated block.

Since we *must* use jumps when we create our own flow-control constructs,

what this means is that, if we restrict ourselves to the standard constructs, we will never again need explicit jumps. Or, expressing this in reverse, simply by avoiding the GOTO statement we avoid the temptation to create our own flow-control constructs, and hence inferior applications. GOTO, it was proclaimed, is what causes bad programming, so it must be avoided.

Now, good programmers use the standard constructs when suitable, but do not hesitate to modify them, or to create their own, when specialized constructs are more effective. These constructs *improve* the program, therefore, not complicate it. Everyone could see that it is possible to use GOTO intelligently, that only when used by incompetents does it lead to bad programming. The assumption that programmers cannot improve remained unchanged, however. So the idea of eliminating the need for expertise continued to be seen as an important principle, as the only solution to the software crisis.

The main appeal of structured programming, thus, is that it appears to eliminate those programming situations that demand skills and experience. This, it was hoped, would reduce application development to a routine activity, to the kind of work that can be performed by almost anyone.

Substitutes for expertise are always delusions, and structured programming was no exception. First, it addressed only *one* aspect of application development – the design of flow-control constructs. Bad programmers, though, do *everything* badly, so even if structured programming could improve this one aspect of their work, other difficulties would remain. Second, structured programming does not really eliminate the need for expertise even in this one area. It was very naive to believe that, if it is possible *in principle* to program using only the standard constructs, it is also possible to develop *real* applications in this fashion. This idea may look good with the small, artificial examples presented in textbooks, but is impractical for serious business applications.

So, since programmers still have to supplement the standard constructs with specialized ones, they need the same knowledge and experience as before. And if they do, instead, restrict themselves to the standard constructs, as the theory demands, they end up complicating *other* aspects of the application. The aspects of an application are the various structures that make it up, and the difficulty of programming is due to the need to deal with many of these structures at the same time. Structured programming succeeds perhaps in simplifying the flow-control structure, but only by making the other structures more involved. And, in any case, programmers still need the capacity to deal with interacting structures. We will study these fallacies in detail later.

2

So far we have examined the *informal* arguments – the praise of standard flow-control constructs and the advice to avoid GOTO. These arguments must be distinguished from the *formal* theory of structured programming, which emerged about 1970. The reason we are discussing both types of arguments is that the formal theory never managed to displace the informal one. In other words, even though it was promoted by its advocates as an exact, mathematical theory, structured programming was in reality just an assortment of methods – some sensible and others silly – for improving programming practices. We will separate its formal arguments from the informal ones in order to study it, but we must not forget that the two always appeared together.

The informal tone of the early period is clearly seen in E. W. Dijkstra's notorious paper, "Go To Statement Considered Harmful."[2] Generally acknowledged as the official inauguration of the structured programming era, this paper is regarded by many as the most famous piece of writing in the history of programming. Yet this is just a brief essay. It is so brief and informal, in fact, that it was published in the form of a letter to the editor, rather than a regular article.

Dijkstra claims to have "discovered why the use of the GOTO statement has such disastrous effects,"[3] but his explanation is nothing more than a reminder of how useful it is to be able to keep track of the program's dynamic behaviour. When carelessly used, he observes, GOTO makes it hard to relate the flow of execution to the nested conditions, iterations, and subroutine calls that make up the program's listing: "The unbridled use of the GOTO statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress."[4]

This is true, of course, but Dijkstra doesn't consider at all the alternative: a disciplined, intelligent programming style, through which we could *benefit* from the power of GOTO. Instead of studying the use of GOTO under this alternative, he simply asserts that "the quality of programmers is a decreasing

[2] E. W. Dijkstra, "Go To Statement Considered Harmful," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE) – paper originally published in *Communications of the ACM* 11, no. 3 (1968): 147–148. An equally informal paper from this period is Harlan D. Mills, "The Case Against GO TO Statements in PL/I," in Harlan D. Mills, *Software Productivity* (New York: Dorset House, 1988) – paper originally published in 1969.

[3] Dijkstra, "Go To Statement," p. 9.

[4] Ibid. The term "unbridled" is used by Dijkstra to describe the *free* use of jumps (as opposed to using jumps only as part of some standard constructs).

function of the density of GOTO statements in the programs they produce,"[5] and concludes that "the GOTO statement should be abolished from all 'higher level' programming languages."[6] His reasoning seems to be as follows: since using GOTO carelessly is harmful, and since good programmers apparently use GOTO less frequently than do bad programmers, then simply by prohibiting everyone from using GOTO we will attain the same results as we would if we turned the bad programmers into good ones.

The logical answer to the careless use of GOTO by bad programmers is not to abolish GOTO, but to encourage those programmers to improve their skills. Yet this possibility is not even mentioned. In the end, in the absence of any real demonstration as to why GOTO is harmful, we must be satisfied with the statement that "it is too much an invitation to make a mess of one's program."[7] What is noteworthy in this paper, therefore, is not just the informal tone, but also the senseless arguments against GOTO.

These were the claims in the late 1960s. Then, the tone changed, and the claims became more ambitious. The software theorists discovered a little paper,[8] written several years earlier and concerned with the logical transformation of flow diagrams, and chose to interpret it as the mathematical proof of their ideas. (We will examine this "proof" later.) Adapted for programming, the ideas presented in this paper became known as the *structure theorem*.

Convinced now that structured programming had a solid mathematical foundation, the theorists started to promote it as the beginning of a new science – the science of programming. Structured programming was no longer seen merely as a body of suggestions for improving programming practices; it was the only correct way to program. And practitioners who did not obey its principles were branded as old-fashioned artisans. After all, rejecting structured programming was now tantamount to rejecting science.

It is *this* theory – the *formal* theory of structured programming – that is important, for it is *this* theory that was promoted as a programming revolution, was refuted in practice, and was then rescued by being turned into a pseudoscience. We could perhaps ignore the informal claims, but it is only by studying the formal theory that we can appreciate why the idea of structured programming was a fraud. For, it was its alleged mathematical foundation that made it respectable. It was thanks to its mathematical promises that it was so widely accepted – precisely those promises that had to be abandoned in order to make it practical.

Thus, a striking characteristic of structured programming is that, even after

[5] Ibid. [6] Ibid. [7] Ibid.

[8] Corrado Böhm and Giuseppe Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," in *Milestones*, eds. Oman and Lewis – paper originally published in *Communications of the ACM* 9, no. 5 (1966): 366–371.

becoming an exact, mechanistic theory, it continued to be defended with *informal* arguments. Its mathematical aspects imparted to it a scientific image, but could not, in fact, support it. So, while advertised as a scientific theory, structured programming was usually presented in the form of a programming methodology, and its benefits could be demonstrated only for simple, carefully selected examples. Moreover, its principles – the GOTO prohibition, in particular – became the subject of endless debates and changes, even among the academics who had invented them.[9]

When a mechanistic theory works, all we need in order to promote it is a mathematical proof. All we need, in other words, is a formal argument; we don't have to resort to persuasion, debates, justifications, case studies, or testimonials. It is only when a theory fails, and its defenders refuse to accept its failure, that we see both formal and informal arguments used side by side (see the discussion in chapter 1, pp. 76–77).

It is impossible to discuss structured programming without stressing this distinction between the formal and the informal concepts. For, by pointing to the *informal* concepts, its advocates can claim to this day that structured programming was successful. And, in a sense, this is true. It is in the nature of informal concepts to be vague and subject to interpretation. Thus, since some of the informal principles are sensible and others silly, one can always praise the former and describe them as "structured programming." The useful principles, as a matter of fact, were known and appreciated by experienced programmers even before being discovered by the academics; and they continue to be appreciated, despite the failure of structured programming. But we must not confuse the small subset of useful principles with the real, mathematical theory of structured programming – the theory that was promoted by the scientists as a revolution.

It is because of their mathematical claims that we accepted structured programming, and the other software theories. Deprived of their formal foundation, these theories are merely collections of programming tips. So the effort to cover up their failure amounts to a fraud: we are being persuaded to depend on the software elites when in reality, since the formal theories are worthless, the elites have nothing to offer us.

[9] Here are two sources for the *formal* theory: Harlan D. Mills, "Mathematical Foundations for Structured Programming," in Harlan D. Mills, *Software Productivity* (New York: Dorset House, 1988) – paper originally published in 1972; Suad Alagić and Michael A. Arbib, *The Design of Well-Structured and Correct Programs* (New York: Springer-Verlag, 1978). And here are two sources for the *informal* theory: Harlan D. Mills, "How to Write Correct Programs and Know It," in Mills, *Software Productivity* – paper originally published in 1975; Edward Yourdon, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice Hall, 1975).

3

The formal theory of structured programming prescribes that software applications, when viewed from the perspective of their flow of execution, be treated as simple hierarchical structures. Applications are to be implemented using only three flow-control constructs: sequential operations, conditions, and iterations. And it is not just the basic elements that must be restricted to these constructs, but the elements at all levels of abstraction. This is accomplished by *nesting* constructs: the complete constructs of one level serve as elements in the constructs of the next higher level, and so on. Thus, although the elements keep growing as we move to higher levels, the constructs remain unchanged.

The sequential construct is shown in figure 7-1 (the arrowheads in flow diagrams indicate the flow of execution). It consists of one operation, *S1*. At the lowest level, the operation is a single statement: assigning a value to a variable, performing a calculation, reading a record from a file, and so on.

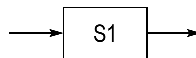

Figure 7-1

The conditional construct, IF, is shown in figure 7-2. This construct consists of a condition, *C1*, and two operations, *S1* and *S2*: if the condition is evaluated as *True*, *S1* is executed; if evaluated as *False*, *S2* is executed. In most high-level languages, the IF statement implements this construct: IF *C1 is True,* THEN *perform S1,* ELSE *perform S2*. Either *S1* or *S2* may be empty (these variants are also shown in figure 7-2). In a program, when *S2* is empty the entire ELSE part is usually omitted.

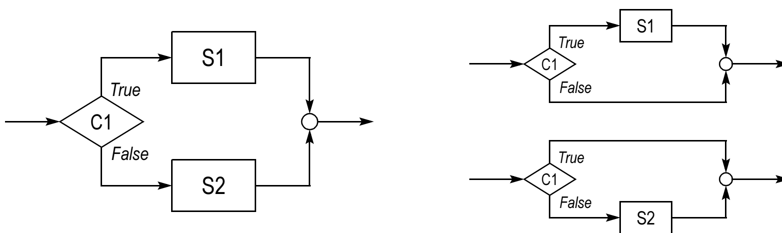

Figure 7-2

The iterative construct, WHILE, is shown in figure 7-3. This construct consists of a condition, *C1*, and one operation, *S1*: if the condition is evaluated as *True*, *S1* is executed and the process is repeated; if evaluated as *False*, the iterations end. In many high-level languages, the WHILE statement implements this construct: WHILE *C1 is True, perform S1*.

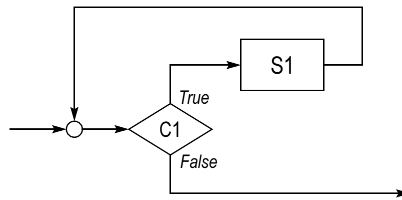

Figure 7-3

To build larger pieces of software, any number of constructs can be connected consecutively: sequential, conditional, and iterative constructs can be combined in any order by connecting the exit of one construct to the entry of the next one. This method of combining constructs is trivial, however. It is only through nesting that we can create the countless combinations of operations, conditions, and iterations required in a serious application.

All three constructs share an important feature: they have one entry and one exit. When viewed from outside, therefore, and disregarding their internal details, the three constructs are identical. It is this feature that makes nesting possible. To nest constructs, we start with one of the three constructs and replace the operation, *S1* or *S2* (or both), with a conditional or iterative construct, or with two consecutive sequential constructs; we then similarly replace *S1* or *S2* in the new constructs, and so on. Thus, the original construct forms the top level of the structure, and each replacement creates an additional, lower level.

This nesting method is known as *top-down design*, and is an important principle in structured programming. To design a new application, we start by depicting the entire project as one sequential construct; going down to the next level of detail, we may note that the application consists in the repetition of a certain operation, so we replace the original operation with an iterative construct; at the next level, we may note that what is repeated is one of two different operations, so we replace the operation in the iterative construct with a conditional construct; then we may note that the operations in this construct are themselves conditions or iterations, so we replace them with further constructs; and so on. (At each step, if the operation cannot be replaced directly with a construct, we replace it first with two simpler, consecutive operations;

then, if necessary, we repeat this for those two operations, and so on.) We continue this process until we reach some low-level constructs, where the operations are so simple that we can replace them directly with the statements of a programming language. If we follow this method, the theorists say, we are bound to end up with a perfect application.

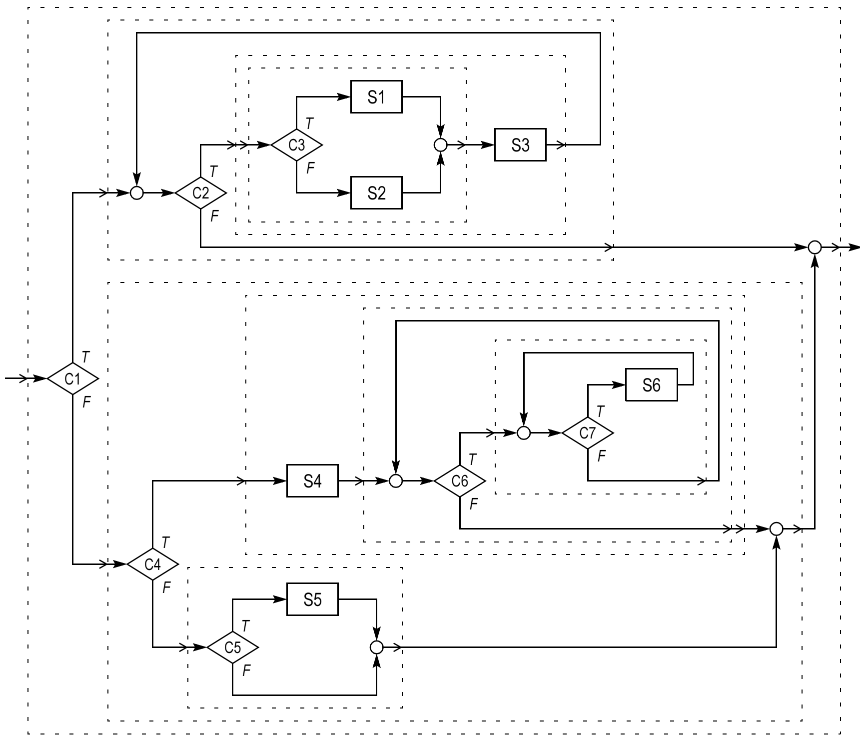

Figure 7-4

The flow diagram in figure 7-4 illustrates this concept. This diagram includes four conditional, three iterative, and two sequential constructs, nested in various ways. Although the format of these constructs is identical to the simple format shown in the previous diagrams, this is obscured by the fact that some of the operations are themselves constructs, rather than simple boxes like *S1*. The dashed boxes depict these constructs, and serve at the same time to indicate pictorially the levels of nesting. (The innermost boxes represent the lowest level, and it is only in these boxes that the constructs' format is immediately recognizable.) Note also the additional arrowheads, drawn to indicate the entry and exit of each dashed box. The arrowheads emphasize

that, regardless of their nesting level, the constructs continue to have only one entry and one exit.

Each dashed box encloses a complete construct – a construct that acts as a single operation in the higher-level construct to which it belongs. When viewed as part of the higher-level construct, then, a dashed box acts just like the box depicting a sequential construct. In other words, the internal details of a given construct, including the lower levels of nesting that make it up, are irrelevant when we study only the constructs at higher levels. (So we could ignore, as it were, the diagram shown inside the dashed box, and replace the entire box with one sequential construct.) An important benefit of this nesting concept is that any construct can be replaced with a functionally equivalent construct, both during development and during maintenance, without affecting the rest of the application. A programmer, thus, can develop or modify a particular construct while knowing nothing about the constructs at higher and lower levels. All he needs to know is the entry and exit characteristics of the constructs at the next lower level.

This is all that practitioners need to learn about the theory of structured programming. Programs developed strictly as nested constructs, and their flow diagrams, are *structured* programs and diagrams. And, the theorists assure us, it has been proved through mathematical logic that any software application can be built in this fashion.

Structured programs can be as large as we want, and can have any number of levels of nesting. It is recommended, nevertheless, for practical reasons, to divide large programs into modules of no more than about a hundred lines, and to have no more than about five levels of nesting in a module. In complicated programs, we can always reduce the number of nesting levels by creating a separate module for the constructs below a given level, and then replacing that whole portion with a statement that invokes the module. Logically, there is no difference between the two alternatives, and smaller modules are easier to understand and to maintain. (From the perspective of the flow of execution, descending the nesting levels formed by the local constructs is the same as invoking a module and then descending the levels of constructs in that module.) Invoking a module is a single operation, so it can be part of any construct; constructs in the invoked module can then invoke other modules in their turn, and so on. Large applications, thus, are generally built by adding levels of modules rather levels of constructs. A module may be invoked from several constructs, of course, if the same operations are required in several places; the module then also functions as subroutine.

Regarding the GOTO issue, it is obvious now why GOTO statements are unnecessary: quite simply, structured programs require no explicit jumps (since all the necessary jumps are implicit, within the standard constructs). The purpose of structured programming is to create structures of nested constructs, so the absence of GOTO is merely a consequence. This is an important point, and in sharp contrast to the original, *informal* claim – the claim that GOTO must be avoided because it tempts us to create messy programs. Now we have the *proof* that GOTO is unnecessary in high-level languages (in fact, in any language that provides the three standard constructs). We have the proof, therefore, that any application can be created without using GOTO statements. GOTO is not bad in itself, but because it indicates that the program is unstructured. Structured programs do not need GOTOs.

To conclude, structured programming is concerned with the flow of execution, and claims that the solution to our programming difficulties lies in designing applications as structures of nested modules, and the modules as structures of nested flow-control constructs. We recognize this as the mechanistic claim that any phenomenon can be represented as a structure of things within things. Structured programming, thus, claims that the flow of execution can be extracted from the rest of the application; that it can be reduced to a simple hierarchical structure, the *flow-control* structure; and that, for all practical purposes, this one structure *is* the application. The logic of nesting and standard constructs is continuous, from the simplest statements to the largest modules. It is this neatness that makes the notion of structured programming so enticing. All we need to know, it seems, is how to create structures of things within things. We are promised, in effect, that by applying a simple method over and over, level after level, we will be able to create perfect applications of any size and complexity.

The Promise

No discussion of the structured programming theory is complete without a review of its promotion and its reception. For, the enthusiasm it generated is as interesting as are its technical aspects.

Harlan Mills, one of the best-known software theorists, compares programming to playing a simple game like tic-tac-toe. The two are similar in that we can account, at each step, for all possible alternatives, and hence discover exact theories. The only difference is that programming gives rise to a greater number of alternatives. Thus, just as a good game theory allows us to play perfect tic-tac-toe, a good programming theory will allow us to write perfect

programs: "Computer programming is a combinatorial activity, like tic-tac-toe.... It does not require perfect resolution in measurement and control; it only requires correct choices out of finite sets of possibilities at every step. The difference between tic-tac-toe and computer programming is complexity. The purpose of structured programming is to control complexity through theory and discipline. And with complexity under better control it now appears that people can write substantial computer programs correctly.... Children, in learning to play tic-tac-toe, soon develop a little theory.... In programming, theory and discipline are critical as well at an adult's level of intellectual activity. Structured programming is such a theory, providing a systematic way of coping with complexity in program design and development. It makes possible a discipline for program design and construction on a level of precision not previously possible."[1]

Structured programming is a fantasy, of course – a mechanistic delusion. As we know, it is impossible to reduce software applications, which are complex phenomena, to simple hierarchical structures; so it is impossible to represent them with exact, mathematical models. Everyone could see that even ordinary requirements cannot be reduced to a neat structure of standard constructs, but it was believed that all we have to do for those requirements is apply certain *transformations*. No one tried to understand the significance of these transformations, or why we need them at all. And when in many situations the transformations turned out to be totally impractical, still no one suspected the theory. These situations were blatant falsifications of the theory; but instead of studying them, the experts chose to interpret the difficulty of creating structured applications as the difficulty of adjusting to the new, disciplined style of programming. No one wondered why, if it has been proved mathematically that any application can be written in a structured fashion, and if everyone is trying to implement this idea, we cannot find a single application that follows strictly the principles of structured programming.

Thus, even though it never worked with serious applications, structured programming was both promoted and received – for twenty years – with the enthusiasm it would have deserved had it been entirely successful.

[1] Harlan D. Mills, "Mathematical Foundations for Structured Programming," in Harlan D. Mills, *Software Productivity* (New York: Dorset House, 1988), pp. 117–118 – paper originally published in 1972. As I have already pointed out (see p. 488), what the software theorists call complexity (i.e., the large number of alternatives) is not the *real* complexity of software (i.e., what makes software applications complex structures, systems of interacting structures). It is impossible to develop applications simply by accounting for the various alternatives, as Mills proposes, because we cannot *identify* all the alternatives.

To appreciate the reaction to the idea of structured programming, we must ignore all we know about complex structures, and imagine ourselves as part of the mechanistic world of programming. Let us think of software applications, thus, as mechanistic phenomena; that is, as phenomena which *can* be represented with simple hierarchical structures. The idea of structured programming is then indeed the answer to our programming difficulties, in the same way that designing physical systems as hierarchical structures of subassemblies is the answer to our manufacturing and construction difficulties.

One promise, we saw, is to reduce programming, from an activity demanding expertise, to the performance of relatively easy and predictable acts: "It is possible for professional programmers, with sufficient care and concentration, to consistently write correct programs by applying the mathematical principles of structured programming."[2] The theorists, thus, are *degrading* the notion of professionalism and expertise to mean *the skills needed to apply a prescribed method*. (I will return to this point in a moment.)

So, to program an application we need to know now only one thing: how to reduce a given problem, expressed as a single operation, to two or three simpler problems; specifically, to two consecutive operations, or a conditional construct (two operations and a condition), or an iterative construct (one operation and a condition). What we do at each level, then, is replace a particular software element with two or three simpler ones. Developing an application consists in repeating this reduction over and over, thereby creating simpler and simpler elements, on lower and lower levels of abstraction. And the skill of programming consists in knowing how to perform the reduction while being certain that, at each level, the new elements are logically equivalent to the original one. But this skill is much easier to acquire than the traditional programming skill, because it is the same types of constructs and reductions that we employ at all levels; besides, each reduction is a small logical step.

Eventually, we reach elements that are simple enough to translate directly into the statements of a programming language. So we must also know how to perform the translation; but this skill is even easier than the reductions – so easy, in fact, that it can be acquired by almost anyone in a few weeks. (This work is often called coding, to distinguish it from programming.)

The key to creating correct applications, then, is the restriction to the standard constructs and the assurance that, at each level, the new elements are logically equivalent to the original one. These conditions are related, since, if we restrict ourselves to these constructs, we can actually prove the equivalence mathematically. Ultimately, structured programming is a matter of discipline:

[2] Richard C. Linger, Harlan D. Mills, and Bernard I. Witt, *Structured Programming: Theory and Practice* (Reading, MA: Addison-Wesley, 1979), p. 3.

we must follow this method *rigorously*, even in situations where a different method is simpler or more efficient. Only if we observe this principle can we be certain that, when the application is finally translated into a programming language, it will be logically equivalent to the original specifications.

This is an important point, as it was discovered that experienced programmers have difficulty adjusting to the discipline of structured programming. Thus, they tend to ignore the aforementioned principle, and enhance their applications with constructs of their own design. They see the restriction to the standard constructs as a handicap, as a dogmatic principle that prevents them from applying their hard-earned talents.

What these programmers fail to see, the theorists explain, is that it is precisely this restriction that allows us to represent software elements mathematically, and hence prove their equivalence from one level to the next. It is precisely because we have so little freedom in our reductions that we can be certain of their correctness. (In fact, the standard constructs are so simple that the correctness of the reductions can usually be confirmed through careful inspection; only in critical situations do we need to resort to a formal, mathematical proof.)

So what appears as a drawback to those accustomed to the old-fashioned, personal style of programming is actually the *strength* of structured programming. Even experienced programmers could benefit from the new programming discipline, if only they learned to resist their creative impulse. But, more importantly, *inexperienced* programmers will now be able to create good applications, simply by applying the principles of top-down design and standard constructs: "Now the new reality is that ordinary programmers, with ordinary care, can learn to write programs which are error free from their inception.... The basis for this new precision in programming is neither human infallibility, nor being more careful, nor trying harder. The basis is understanding programs as mathematical objects that are subject to logic and reason, and rules for orderly combination."[3]

I stated previously that the software theorists are degrading the notions of expertise and professionalism, from their traditional meaning – the utmost that human beings can accomplish – to the trivial knowledge needed to follow methods. This attitude is betrayed by the claim that structured programming will benefit *all* programmers, regardless of skill level. Note the first sentence in the passage just quoted, and compare it with the following sentence: "Now the new reality is that professional programmers, with professional care, can learn to consistently write programs that are error-free from their inception."[4]

[3] Ibid., p. 2.

[4] Harlan D. Mills, "How to Write Correct Programs and Know It," in Mills, *Software Productivity*, p. 194 – paper originally published in 1975.

The two sentences (evidently written by the same author during the same period) are practically identical, but the former says "ordinary" and the latter "professional." For this theorist, then, the ideas of professional programmer, ordinary programmer, and perhaps even novice programmer, are interchangeable. And indeed, there is no difference between an expert and a novice if we reduce programming to the act of following some simple methods.

This attitude is an inevitable consequence of the mechanistic dogma. On the one hand, the software mechanists praise qualities like expertise and professionalism; on the other hand, they promote mechanistic principles and methods. Their mechanistic beliefs prevent them from recognizing that the two views contradict each other. If the benefits of structured programming derive from reducing programming to methods requiring little experience – methods that can be followed by "ordinary" programmers – it is precisely because these methods require only mechanistic knowledge. Expertise, on the contrary, is understood as the highest level that human minds can attain. It entails *complex* knowledge, the kind of knowledge we reach after many years of learning and practice. Following the methods of structured programming, therefore, cannot possibly mean expertise and professionalism in their traditional sense. It is in order to apply these terms to mechanistic knowledge – in order to resolve the contradiction – that the theorists are degrading their meaning.

If the first promise of structured programming is to eliminate the need for programming expertise, the second one is to simplify the development of large applications by breaking them down into small parts. Each reduction from a given element to simpler ones is in effect a separate task, since it can be performed independently of the other reductions. Then, for a particular reduction, we can treat the lower-level reductions as either the same task or as separate, smaller tasks. In this fashion, we can break down the original task – that is, the application – into tasks that are as small as we want. Although the smallest task can be as small as one construct, we rarely need to go that far. For most applications, the smallest tasks are the individual modules; and it is recommended that modules be no larger than one printed page, so that we can conveniently study them.

When each module is a separate task, different programmers can work on different modules of the same application without having to communicate with one another. This has several benefits: if a large application must be finished faster, we can simply employ more programmers; we can replace a programmer at any time without affecting the rest of the project; and later,

during maintenance, a new programmer only needs to understand the logic of individual modules.

With the old style of programming, the complexity of applications, and hence the difficulty of developing and maintaining them, seems to grow exponentially with their size. The time and cost required to develop a new application, or to modify an existing one, can be unpredictable; adding programmers to a project rarely helps; large projects often become unmanageable and must be abandoned. With structured programming, on the other hand, the complexity and the difficulty do not grow with the application's size. No matter how large, an application is no more difficult to develop than is its largest module. The only difference we should see between large and small applications is that large ones take longer, or involve more programmers; but the time and cost are now predictable. What structured programming does, in the final analysis, is replace the challenge of developing a large system of *interrelated* entities, with the easier challenge of developing many small, *separate* entities.

The greatest promise of structured programming, however, and the most fantastic, is the promise of error-free applications; specifically, the claim that structured programming obviates almost entirely the need to test software, since applications will usually run perfectly the first time: "By practicing principles of structured programming and its mathematics you should be able to write correct programs and convince yourself and others that they are correct. Your programs should ordinarily compile and execute properly the first time you try them, and from then on."[5]

Top-down programming, we saw, entails the repeated reduction of elements to simpler ones that are logically equivalent. So, if we perform each reduction correctly, then no matter how many reductions are required, we can be certain that the resulting application will be logically equivalent to the original specifications. (The application may still be faulty, of course, if the *specifications* are faulty; structured programming guarantees only that the application will behave exactly as defined in the specifications.)

Fantastic though it is, this claim is logical – *if* we assume that applications are simple hierarchical structures. Here is how the claim is defended: Since the equivalence of elements in the flow-control structure can be proved mathematically at each level in the top-down process, and since the statements in the resulting application correspond on a one-to-one basis to the lowest-

[5] Ibid.

level elements, the application *must* be correct. In other words, what we create in the end by means of a programming language is in effect the same structure that we created earlier by means of a diagram, and which we could prove to be correct.

We should still test our applications, because we are not infallible; but testing will be a simple, routine task. The only type of errors we should expect to find are those caused by programming slips. And, thanks to the discipline we will observe during development, these errors are bound to be minor bugs, as opposed to the major deficiencies we discover now in our applications (faulty logic, problems necessitating redesign or reprogramming, mysterious bugs for which no one can find the source, defects that give rise to other defects when corrected, and so on). Thus, not only will the errors be few, but they will be trivial: easy to find and easy to correct. This is how Mills puts it: "As technical foundations are developed for programming, its character will undergo radical changes.... We contend here that such a radical change is possible now, that in structured programming the techniques and tools are at hand to permit an entirely new level of precision in programming."[6]

The inevitable conclusion is that, if we adhere to the principles of structured programming, we will write program after program without a single error. This conclusion prompts Mills to make one of those ludicrous predictions that mechanists are notorious for; namely, that programming can become such a precise activity that we will commit just a handful of errors in a lifetime: "The professional programmer of tomorrow will remember, more or less vividly, every error in his career."[7]

It is important to note that these were serious claims, confidently made by the world's greatest software theorists. And, since the theorists never recognized the fallacy of structured programming, since to this day they fail to understand why its mathematical aspects are irrelevant, they are still claiming in effect that it permits us to create directly error-free applications. By implication, then, they are claiming that all software deficiencies and failures since the 1970s, and all the testing we have done, could have been avoided: they were due to our reluctance to observe the principles of structured programming.

The final promise of structured programming is to eliminate programming altogether; that is, to *automate* the creation of software applications. This was not one of the original ideas, but emerged a few years later with the notion of

[6] Mills, "Mathematical Foundations," p. 117.

[7] Mills, "Correct Programs," p. 194.

CASE – software devices that replace the work of programmers. (This promise is perfectly captured in the title of a book written by two well-known experts: *Structured Techniques: The Basis for CASE.*[8])

As with the other claims, if we accept the idea that applications are simple hierarchical structures, the claim of automatic software generation is perfectly logical. Structured programming breaks down the development process into small and simple tasks, most of which can be performed mechanically; and if they can be performed mechanically, they can be replaced with software devices. For example, the translation of the final, low-level constructs into the statements of a programming language can easily be automated. Most reductions, too, are individually simple enough to be automated. The software entities in CASE systems will likely be different from the traditional ones, but the basic principle – depicting an application as a hierarchical structure of constructs within constructs – will be the same.

Application development, thus, will soon require no programmers. An analyst or manager will specify the requirements by interacting with a sophisticated development system, and the computer will do the rest: "There is a major revolution happening in software and system design.… The revolution is the replacement of manual design and coding with automated design and coding."[9] So, while everyone was waiting for the benefits promised by the structured programming revolution, the software theorists were already hailing the *next* revolution – which suffered from the same fallacies.

This, then, is how structured programming was promoted by the software elites. And it is not hard to see how, in a mechanistic culture like ours, such a theory can become fashionable. The enthusiasm of the academics was shared by most managers, who, knowing little about programming, saw in this idea a solution to the lack of competent programmers; and it was also shared by most programmers, who could now, simply by avoiding GOTO, call themselves software engineers. Only the few programmers who were already developing and maintaining applications successfully could recognize the absurdity of structured programming; but their expertise was ridiculed and interpreted as old-fashioned craftsmanship.

The media too joined in the general hysteria, and helped to propagate the structured programming fallacies by repeating uncritically the claims and

[8] James Martin and Carma McClure, *Structured Techniques: The Basis for CASE*, rev. ed. (Englewood Cliffs, NJ: Prentice Hall, 1988). CASE stands for Computer-Aided Software Engineering. [9] Ibid., p. 757.

promises. *Datamation*, for instance, a respected data-processing journal of that period, devoted its December 1973 issue to structured programming, proclaiming it a revolution. The introductory article starts with these words: "Structured programming is a major intellectual invention, one that will come to be ranked with the subroutine concept or even the stored program concept."[10]

The Contradictions

1

Now that we have seen the enthusiasm generated by the idea of structured programming, let us study the contradictions – contradictions which, although well known at the time, did nothing to temper the enthusiasm.

Structured programs, we saw, are pieces of software whose flow of execution can be represented as a structure of standard flow-control constructs. Because these constructs have only one entry and exit, a structured piece of software is a structure of hierarchically nested constructs. The structure can be a part of a module, an entire module, and even the entire application. The flow diagram in figure 7-4 (p. 513) was an example of a structured piece of software.

Our affairs, however, can rarely be represented as neat structures of nested entities, because they consist of *interacting* processes and events. So, if our software applications are to mirror our affairs accurately, they must form systems of *interacting* structures. What this means is that, when designing an application, we will encounter situations that *cannot* be represented with structured flow diagrams.

The theory of structured programming acknowledges this problem, but tells us that the answer is to change the way we view our affairs. The discipline that is the hallmark of structured programming must start with the way we formulate our requirements, and if we cannot depict these requirements with structured diagrams, the reason may be that we are not disciplined in the way we run our affairs. The design of a software application, then, is also a good opportunity to improve the logic of our activities. Much of this improvement will be achieved, in fact, simply by following the top-down method, since this method encourages us to view our activities as levels of abstraction, and hence as nested entities.

[10] Daniel D. McCracken, "Revolution in Programming: An Overview," *Datamation* 19, no. 12 (1973): 50–52.

So far, there is not much to criticize. The benefits of depicting the flow of execution with a simple hierarchical structure are so great that it is indeed a good idea, whenever possible, to design our applications in this manner. But the advocates of structured programming do not stop here. They insist that *every situation* be reduced to a structured diagram, no matter how difficult the changes or how unnatural the results. In other words, even if the use of standard constructs is *more complicated* than the way we normally perform a certain activity, we must resist the temptation to implement the simpler logic of that activity.

And this is not all. The theorists also recognize that, no matter how strictly we follow the top-down design method, some situations will remain that cannot be represented as structured diagrams. (It is possible to prove, in fact, that whole classes of diagrams, including some very common and very simple cases, cannot be reduced to the three standard constructs.) Still, the theorists say, even these situations must be turned into structured software, by applying certain *transformations*. The transformations complicate the application, it is agreed, but complicated software is preferable to unstructured software.

The ultimate purpose of these transformations is to create new relations between software elements as a replacement for the relations formed by explicit jumps, which are prohibited under structured programming. (We will study this idea in greater detail later.) And there are two ways to create the new relations: by sharing operations and by sharing data. I will illustrate the two types of transformations with two examples.

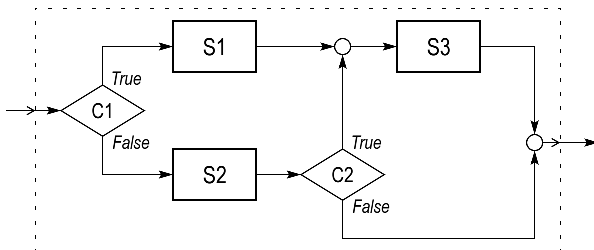

Figure 7-5

Figure 7-5 shows the flow diagram of a requirement that, although very simple, cannot be reduced to standard flow-control constructs. This is a variation of the standard conditional construct: the condition *C1* and the operations *S1* and *S2* form the standard part, but there is an additional operation, *S3*. This operation is always executed after *S1*, but is executed after *S2* only if *C2* is evaluated as *True*. The requirement, in other words, is that an

operation which is part of one branch of a conditional construct be also executed, sometimes, as part of the other branch. And if we study the diagram, we can easily verify that it is unstructured: it is not a structure of nested standard constructs. Standard constructs have only one entry and exit, and here we cannot draw a dashed box with one entry and exit (as we did in figure 7-4) around any part of the diagram larger than a sequential construct.

Note that this is not only a *simple* requirement, but also a very common one. The theory of structured programming is contradicted, therefore, not by an unusual or complicated situation, but by a trivial requirement. There are probably thousands of situations in our affairs where such requirements must become part of an application.

The problem, thus, is not *implementing* the requirement, but implementing it under the restrictions of structured programming. The requirement is readily understood by anyone, and is easily implemented in any programming language by specifying directly the particular combination of operations, conditions, and jumps depicted in the diagram; in other words, by creating our own, non-standard flow-control construct. To implement this requirement, then, we must employ *explicit* jumps – GOTO statements. We need the explicit jumps in order to create our own construct, and we need our own construct because the requirement cannot be expressed as a nested structure of standard constructs.

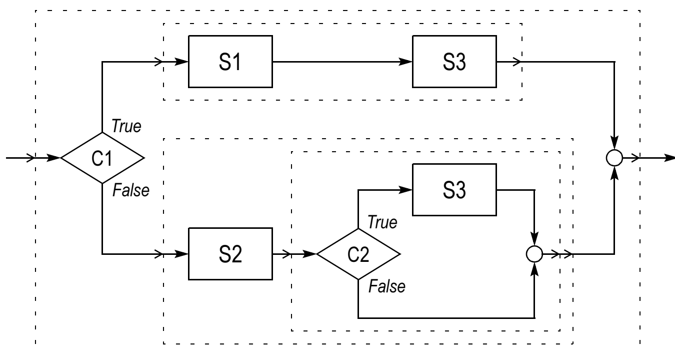

Figure 7-6

But explicit jumps are forbidden under structured programming. So, instead of creating our own construct, we must modify the flow diagram as shown in figure 7-6. If you compare this diagram with the original one, you can see that the transformation consists in duplicating the operation *S3*. As a result, instead of being related through an explicit jump, some elements are related now

through a shared operation. The two diagrams are functionally equivalent, but the new one is properly structured (note the dashed boxes depicting the standard constructs and the nesting levels). In practice, when *S3* is just one or two statements it is usually duplicated in its entirety; when larger, it is turned into a subroutine (i.e., a separate module) and what is duplicated is only the call to the subroutine.

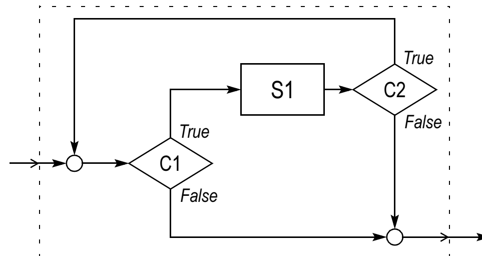

Figure 7-7

Figure 7-7 is the flow diagram of another requirement that cannot be reduced to standard constructs. This is a variation of the standard iterative construct: the condition *C1* and the operation *S1* form the standard part, but the loop is also controlled by a second condition, *C2*. The requirement is to terminate the loop when either *C1* or *C2* is evaluated as *False*; in other words, to test for termination both before and after each iteration. But the diagram that represents this requirement is unstructured: it is not a structure of nested standard constructs. As was the case with the diagram in figure 7-5, we can find no portion (larger than the sequential construct) around which we could draw a dashed box with only one entry and exit.

This is another one of those requirements that are common, simple, and easily implemented by creating our own flow-control construct. One way is to start with the standard iterative construct and modify it by adding the condition *C2* and a GOTO statement (to jump out of the loop); another way is to design the whole loop with explicit jumps.

To implement the requirement under structured programming, however, we must modify the diagram as shown in figure 7-8. This modification illustrates the second type of transformation: creating new relations between elements by sharing data, rather than sharing operations. Although functionally equivalent to the original one, the new diagram is a structure of nested standard constructs. Instead of controlling directly the loop, *C2* controls now the value of *x* (a small piece of storage, even one bit), which serves as switch, or indicator: *x* is cleared before entering the loop, and is set when *C2*

yields *False*. The loop's main condition is now a combination of the original condition and the current value of *x*: the iterations are continued only as long as both conditions, *C1* and *x=0*, are evaluated as *True*.[1]

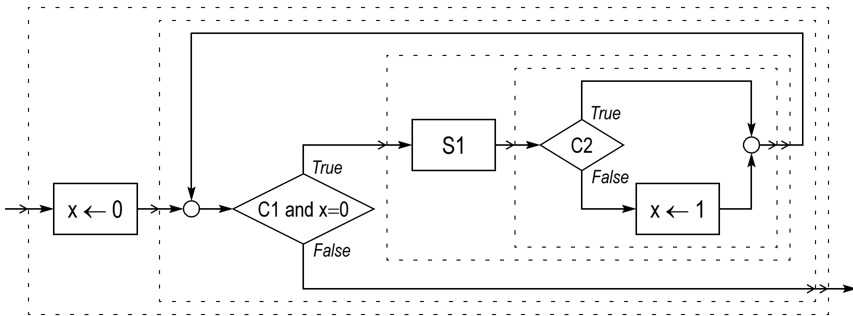

Figure 7-8

It must be noted that only the transformation based on shared data is, in fact, necessary. Structured programming permits any transformations, but the one based on shared operations is not strictly needed; it is merely the simpler alternative in the case of diagrams like that shown in figure 7-5. In principle, we can resort to memory variables to reduce any diagram to a structured format.

These examples demonstrate some basic situations, but we can think of any number of other, similar requirements (to say nothing of more complicated ones) that are easier to implement directly, with non-standard flow-control constructs, than through transformations: a loop nested in a conditional construct and the need to jump from inside the loop to outside the conditional construct; two or more levels of nested conditions and an operation common to more than two elements in this construct; two or more levels of nested iterations and the need to terminate the outermost loop from inside the innermost one; and so on.

Note that the issue is not whether constructs based on transformations are or are not better than constructs based on explicit jumps. Duplicating pieces of software, or using variables as switches, may well be the best alternative in one situation, while creating specialized flow-control constructs may be preferable in another. Ultimately, it is the programmer's task to implement the most effective flow-control structure for a given requirement. The real issue, thus, is

[1] The symbol ← inside the blocks denotes the assignment operation. When using variables as switches, only two values (such as 0 and 1) are needed.

the validity of the claim that a restriction to standard constructs simplifies development, guarantees error-free applications, and so forth. This claim, we will see, is a delusion.

2

Let us review the concept of software structures and attributes (see "Software Structures" in chapter 4). Software applications are complex structures, systems of interacting structures. The elements of these structures are the various entities that make up the application: statements, blocks of statements, larger blocks, modules. The attributes of software entities are those characteristics that can be possessed by more than one entity: accessing a particular file, using a particular memory variable, calling a particular subroutine, being affected by a particular business rule, and so forth. Attributes, therefore, relate the application's elements logically: each attribute creates a different set of relations between the application's elements, thereby giving rise to a different structure. There is a structure for each attribute present in the application – a structure reflecting the manner in which the elements are affected by that attribute. We can also describe these structures as the various *aspects* of the application.

Although an application may have thousands of attributes, any one element has only a few, so each structure involves only *some* of the application's elements. We saw, though, that it is useful to treat *all* the application's elements as elements of *every* structure; specifically, to consider for each element *all* the attributes, those that affect it as well as those that do not, because *not* possessing an attribute can be as significant as possessing it. This is clearly revealed when depicting each attribute with a separate, classification-style diagram: first, we divide the application's elements into those affected and those unaffected by the attribute; then, we divide the former according to the ways they are affected, and the latter according to the reasons they are unaffected.

But even when restricting our structures to those elements that actually possess the attribute, we find that, because they possess *several* attributes, most elements belong in several structures at the same time. And this sharing of elements causes the structures to interact. Thus, a software element can be part of business practices, use memory variables, access files, and call subroutines. Software elements must have several attributes because their function is to represent real entities. Since our affairs comprise entities that are shared by various processes and events, the multiplicity of attributes, and the consequent interaction of structures, is not surprising: it is precisely this interaction that allows software to mirror our affairs. So it is quite silly to attempt to reduce

applications to independent structures, as do structured programming and the other mechanistic theories, and at the same time to hope that these applications will represent our affairs accurately.

Although there is no limit to the *number* of attributes in an application, there are only a few *types* of attributes (or what I called *software principles*). Thus, we may need a large number of attributes to implement all the rules and methods reflected in the application, but all these attributes can be combined under the type *business practices*. Similarly, the use of subroutines in general, as well as the repetition of individual operations, can be combined under the type *shared operations*. And accessing files, as well as using memory variables, can be combined under the type *shared data*.

An important type are the *flow-control* attributes – those attributes that establish the sequence in which the computer executes the application's elements. An element's flow-control attributes determine when that element is to be executed, relative to the other elements. Each flow-control attribute, thus, groups several elements logically, and relates the group as a whole to the rest of the application. The totality of flow-control attributes determines the performance of the application under all possible run-time conditions.

The flow-control attributes are necessary because computers, being sequential machines, normally execute operations in the sequence in which they appear in memory. But, while the operations that make up the application are stored in memory in one particular order (the static sequence), they must be executed in a different order (the dynamic sequence), and also in a different order on different occasions. In a loop, for instance, the repeated block appears only once, but the computer must be instructed to return to its beginning over and over; similarly, in a conditional construct we specify two blocks, and the computer must be instructed to execute one and bypass the other. Any element in the application, in fact, may have to instruct the computer to jump to another operation, forward or backward, instead of executing the one immediately following it. Thus, since it is the elements themselves that control the flow of execution, the flow-control features are attributes of these elements.

The flow-control attributes can also be described as the various means through which programming languages allow us to implement the *jumps* required in the flow of execution; that is, the *exceptions* to the sequential execution. The most versatile operation is the explicit jump – the GOTO statement, in most languages. Each GOTO gives rise to a flow-control attribute, which relates logically several elements: the one from which, and the one to which, the jump occurs, plus any others affected by the jump (those bypassed, for instance).

Most jumps in high-level languages, however, are implicit. The construct known as *block* (a series of consecutive operations, all executed or all bypassed)

defines in effect a jump. Other implicit jumps include exception handling (jumping automatically to a predefined location when a certain run-time error occurs), the conditional construct (jumping over a statement or block), and the iterative construct (jumping back to the beginning of the loop). Additional types of jumps are often provided by language-specific statements and constructs. All jumps, though, whether explicit or implicit, serve in the end the same purpose: they create unique flow-control attributes. With each jump, two or more elements are related logically – as viewed from the perspective of the flow of execution – and this relationship is what we note as a particular flow-control attribute.

As is the case with the other types of attributes, an element can have more than one flow-control attribute. For example, the execution of a certain element may need to be followed by the execution of different elements on different occasions, depending on run-time conditions. Also like the other types of attributes, each flow-control attribute gives rise to a structure – a *flow-control* structure in this case. Although a flow-control structure usually affects a small number of elements, here too it is useful to treat *all* the application's elements as elements of each structure. For, it may be just as important for the application's execution that an element is *not* affected by that particular flow-control attribute, as it is that the element *is* affected. Take, for instance, the case of design faults: the sequence of execution is just as wrong if two elements are *not* connected by a jump when they should be, as it is if two elements *are* connected by a jump when they shouldn't be.

3

Having established the nature of software applications, and of software structures and attributes, we are in a position to understand the delusions of structured programming. I will start with a brief discussion; then, in the following subsections, we will study these delusions in detail.

To begin with, the theorists are only concerned with the *flow-control* structures of the application. These structures are believed to provide a complete representation of the running application, so their correctness is believed to guarantee the correctness of the application. The theorists fail to see that, no matter how important are the flow-control structures, the other structures too influence the application's performance. Once they commit this fallacy, the next step follows logically: they insist that the application be designed in such a way that all flow-control structures are combined into one; and this we can accomplish by restricting each element to *one* flow-control attribute.

Clearly, if each element is related to the rest of the application – from the perspective of the flow of execution – in only one way, the entire application can be designed as one hierarchical structure. This, ultimately, a mechanistic representation of the entire application, is the goal of structured programming. For, once we reduce applications to a mechanistic model, we can design and validate them with the tools of mathematics.

We recognize in this idea the mechanistic fallacy of reification: the theorists assume that one simple structure can provide an accurate representation of the complex phenomenon that is a software application. They extract first *one type* of structures – the *flow-control* structures; then, they go even further and attempt to reduce all structures of this type to *one* structure.

The structure we are left with – the structure believed to represent the application – is the nesting scheme. The neat nesting of constructs and modules we see in the flow diagram constitutes a simple hierarchical structure. Remember that both the nesting and the hierarchy are expected to represent the *execution* of the application's elements, not their static arrangement. The sequence of execution defined through the nesting scheme is as follows: the computer will execute the elements found at a given level of nesting in the order in which they appear; but if one of these elements has others nested within it, they will be executed before continuing at the current level; this rule is applied then to each of the nested elements, and so on. If the nesting scheme is seen as a hierarchical structure, it should be obvious that, by repeating this process recursively, every element in the structure is bound to be executed, executed only once, and executed at a particular time relative to the others.

So the nesting concept is simply a convention: a way to define a precise, unambiguous sequence of execution. By means of a nesting scheme, the programmer specifies the sequence in which he wants the computer to execute the application's elements at run time. The nesting convention is, in effect, an implicit flow-control attribute – an attribute possessed by every element in the application. And when this attribute is the *only* flow-control attribute, the nesting scheme is the only flow-control structure.

Recall the condition that each element have only one entry and exit. This, clearly, is the same as demanding that each element be connected to the rest of the application in only one way, or that each element possess only one flow-control attribute. The hierarchical structure is the answer, since in a hierarchical nesting scheme each element is necessarily connected to the others in only one way. Thus, the principle of nesting, and the restrictions to one entry and exit, one flow-control attribute, and one hierarchical structure, are all related.

It is easy to see that the software nesting scheme is the counterpart of the *physical* hierarchical structure: the mechanistic concept of things within things

that is so useful in manufacturing and construction, and which the software theorists are trying to emulate. The aim of structured programming is to make the flow of execution a perfect structure, a structure of *software* things within things. Just as the nesting scheme of a physical structure determines the *position* of each part and subassembly relative to the others, so the nesting scheme of a software application determines when each element is *executed* relative to the others. While one structure describes space relationships, the other describes time relationships; but both are strict hierarchies.

We can also express this analogy as follows. Physical systems can be studied with the tools of mathematics because their dynamic structure usually mirrors the static one. The sequence of operations of a machine, for instance, closely corresponds to the hierarchical diagram of parts and subassemblies that defines the machine. In software systems, on the other hand, the dynamic structure is very different from the static one: the flow of execution of an application does not correspond closely enough to the flow diagram (the static nesting of constructs and modules).

By forcing the flow of execution to follow the nesting scheme, the advocates of structured programming hope to make the dynamic structure of the application mirror the static one, just as it does in physical systems. It is the discrepancy between the dynamic structure and the static one that makes programming more difficult and less successful than engineering. We know that hierarchical systems can be represented mathematically. Thus, if we ensure that the flow diagram is a hierarchical nesting scheme, the flow of execution will mirror a hierarchical system, and the mathematical model that represents the diagram will represent at the same time the *running* application.

This idea – the dream of structured programming from the beginning – is clearly stated in Dijkstra's notorious paper: "Our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do … our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."[2]

And here is the same idea expressed by other academics *twenty years* later: "Programs are essentially dynamic beings that exhibit a flow of control, while the program listing is a static piece of text. To ease understanding, the problem is to bring the two into harmony – to have the static text closely reflect the

[2] E. W. Dijkstra, "Go To Statement Considered Harmful," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE), p. 9 – paper originally published in *Communications of the ACM* 11, no. 3 (1968): 147–148.

dynamic execution."[3] "The goal of structured programming is to write a program such that its dynamic structure is the same as its static structure. In other words, the program should be written in a manner such that during execution its control flow is linearized and follows the linear organization of the program text."[4]

This wish betrays the naivety of the software theorists: they actually believed that the enormously complex structure that is the flow of execution of an application can mirror the simple diagram that is its static representation. And the persistence of this belief demonstrates the corruptive effect of the mechanistic dogma. There were thousands of opportunities, during those twenty years, for the theorists to observe the complexity of software. Their mechanistic obsession, however, prevented them from recognizing these situations as falsifications of the idea of structured programming.

Now, a running application could, in principle, be a strict nesting scheme – a system of elements whose sequence of execution reflects their position in a hierarchical structure. This is what structured programming appears to promote, but it should be obvious that no serious application can be created in this manner. For, in such an application there would be no way to modify the flow of execution: every element would consist of nothing but one operation or several consecutive operations, would always have to be executed, always executed once, and always in the sequence established by the nesting scheme. The application, in other words, would always do the same thing. This is what we should expect, of course, if we want the execution of a software application – that is, its representation in time – to resemble the nesting scheme of a physical structure. After all, a physical structure like an appliance is always the same thing: its parts and subassemblies always exist, and are always arranged in the same way.

The theorists recognize that software is more versatile than mechanical devices, and that we need more than a nesting scheme if we want to create serious applications. So, while praising the benefits of a single flow-control structure, they give us the means to relate the application's elements in additional ways: the conditional and iterative constructs. The purpose of these constructs is to *override* the nesting scheme: they endow the application's

[3] Doug Bell, Ian Morrey, and John Pugh, *Software Engineering: A Programming Approach* (Hemel Hempstead, UK: Prentice Hall, 1987), p. 17.

[4] Pankaj Jalote, *An Integrated Approach to Software Engineering* (New York: Springer-Verlag, 1991), p. 236.

elements with additional flow-control attributes, thereby creating flow-control structures that are additional to the nesting scheme. Consequently, the application is no longer a strict nesting scheme of sequential constructs. It is a nesting scheme plus other structures – a *system* of flow-control structures. The two constructs, thus, serve to restore the multiplicity of structures and the complexity that had been eliminated when the theorists tried to reduce the application to one structure.

Because the application is still a nesting scheme of constructs with only one entry and exit, the theorists believe that the nesting scheme alone continues to represent the running application. The additional flow-control structures are not reflected in the nesting scheme, so it is easy to ignore them. But, even though they are not as obvious as the nesting scheme, these structures contribute to the complexity of the application – as do the structures created by shared data or operations, and by business or software practices, also ignored by the theorists because they are not obvious.

Finally, the hierarchical nesting scheme with its sequential constructs, and the enhancement provided by the two additional constructs, appear to form a basic set of software operations – basic in that they are the only operations needed, in principle, to implement any application. As a result, the theorists confuse these three types of constructs with the set of operations that forms the *definition* of a hierarchical structure (the operations that combine the elements of one level to create the next one). This leads to the belief that, by restricting ourselves to these constructs, we will realize the original dream: a flow of execution that mirrors the static nesting scheme, and is therefore a simple structure. This also explains why we are asked to convert those flow-control structures that cannot be implemented with these constructs, into other *types* of structures. But if the true purpose of the conditional and iterative constructs is to create additional flow-control structures, this conversion is futile, because the flow of execution is no longer a simple structure in any case.

There are so many fallacies in the theory of structured programming that we must separate it into several stages if we are to study it properly, and to learn from its delusions. These are not chronological stages, though, since they all occurred at about the same time. They are best described as stages in a process of degradation. We can identify four stages, and I will refer to them simply as the first, second, third, and fourth delusions. Bear in mind, however, that these delusions are interrelated, so the distinction may not always be clear-cut.

The first delusion is the belief that one structure alone – a flow-control structure – can accurately represent the performance of the application.

The second delusion is the belief that the standard constructs constitute a basic set of operations, whereas their true role is to restore the multiplicity of flow-control structures lost in the first delusion.

The third delusion is the belief that, if it is possible *in principle* to restrict applications to the standard flow-control constructs, we can develop *actual* applications in this manner. We are to modify our requirements by applying certain transformations, and this effort is believed to be worthwhile because the restriction to standard constructs should reduce the application to one structure. What the transformations do, though, is convert the relations due to flow-control attributes into relations due to other types of attributes, thereby adding to the other types of structures.

The fourth delusion is the notion of inconvenience: if we find the transformations inconvenient, or impractical, we don't have to *actually* implement them; the application will have a single flow-control structure merely because the transformations can be implemented *in principle*. The transformations *are* important, but only when convenient. This belief led to the reinstatement of many non-standard flow-control constructs, while the theorists continued to claim that the flow of execution was being reduced to a simple structure.

Common to all four delusions, thus, is the continuing belief in a mathematical representation of software applications, error-free programming, and the rest, when in fact these qualities had been lost from the start. Even before the detailed analysis, therefore, we can make this observation: When the software experts were promoting structured programming in the 1970s, when they were presenting it as a new science and a revolution in programming, all four delusions had already occurred. Thus, there never existed a useful, serious, scientific theory of structured programming – not even for a day. The movement known as structured programming, propagandized by the software elites and embraced by the software bureaucrats, was a fraud from the very beginning.

The analysis of these delusions also reveals the pseudoscientific nature of structured programming. The theory is falsified again and again, and the experts respond by *expanding* it. They restore, under different names and in complicated ways, the *traditional* programming concepts; so they restore precisely those concepts which they had previously rejected, and which must indeed be rejected, because they *contradict* the principles of structured programming.

The four delusions are, in the end, various stages in the struggle to rescue the theory from refutation by making it cope with those situations that falsify it. Structured programming could be promoted as a practical idea only after most of the original principles had been abandoned, and the complexity of applications again accepted; in other words, at the precise moment when it had

lost the very qualities it was being promoted for. What was left – and what was called structured programming – was not a scientific theory, nor even a methodology, but merely an informal, and largely worthless, collection of programming tips.

The First Delusion

The first delusion is the delusion of the main structure: the belief that one structure alone can represent the application, since the other structures are unimportant, or can be studied separately. In the case of structured programming, the main structure is the nesting scheme: the hierarchical structure of constructs and modules. The static nesting scheme is believed to define completely and precisely the flow of execution, and hence the application's dynamic performance (see pp. 530–532).

If the goal of structured programming is to represent applications mathematically, the theorists are right when attempting to reduce them to a simple structure. As we know, mechanistic systems, as well as mathematical models, are logically equivalent to simple structures. Thus, it is true that only an application that was reduced to a simple structure can have a mathematical model. The fallacy, rather, is in the belief that applications *can* be reduced to a simple structure.

Like all mechanists, the software theorists do not take this possibility as hypothesis but as fact. Naturally, then, they perceive the *flow-control* structure (the sequence in which the computer executes the application's elements) as the structure that determines the application's performance. So, they conclude, we must make *this* structure a strict hierarchy of software entities. And this we can do by making the nesting scheme (which is simply the implementation of the flow-control structure by means of a programming language) a strict hierarchy.

But the flow-control structure is not an independent structure. Its elements are the software entities that make up the application, so they also function as elements in other structures: in the various processes implemented in the application. Every business practice that affects more than one element, every subroutine used by more than one element, every memory variable or database field accessed in more than one element, connects these elements logically, creating relations that are different from the relations defined by the flow of execution. This, obviously, is their purpose. It is precisely because one structure is insufficient that we must relate the application's elements in additional ways. Just like the flow-control structure, *each one* of these

structures could be designed, if we wanted, as a perfect hierarchy. But, while the individual structures can be represented mathematically, the application as a whole cannot. Because they share their elements, the structures interact, and this makes the application a non-mechanistic phenomenon (see p. 528).

No matter how important is the flow-control structure, the other structures too affect the application's performance. Thus, even with a correct flow-control structure, the application will malfunction if a subroutine or variable is misused, or if a business practice is wrongly implemented; in other words, if one of the other structures does not match the requirements.

So, if the other structures affect the application's performance as strongly as does the flow-control structure, if we must ensure that every structure is perfect, how can the theorists claim that a mathematical representation of the flow of execution will guarantee the application's correctness? They are undoubtedly aware that the other structures create additional relations between the same elements, but their mechanistic obsession prevents them from appreciating the significance of these simultaneous relationships.

The theory of structured programming, thus, is refuted by the existence of the other structures. Even if we managed to represent mathematically the flow-control structure of an entire application, this achievement would be worthless, because the application's elements are related at the same time in additional ways. Like all attempts to reduce a complex phenomenon to a simple structure, a mathematical model of the flow-control structure would provide a poor approximation of the running application. What we would note in practice is that the model could not account for all the alternatives that the application is displaying. (Here we are discussing only the complexity created by the other *types* of structures. As we will see under the second delusion, the flow-control structure itself consists of interacting structures.)

All we can say in defence of software mechanism is that each aspect of the application – the flow of execution as well as the various processes – is indeed more easily designed and programmed if we view it as a hierarchical structure. But this well-known quality of the hierarchical concept can hardly form the basis of a formal theory of programming. Only rarely are strict hierarchies the most effective implementation of a requirement, and this is why programming languages permit us to override, when necessary, the neat hierarchical relations. Besides, only rarely is a mathematical representation of even one of these structures, and even a portion of a structure, practical, or useful. And a mathematical representation of the entire application is a fantasy.

Note how similar this delusion is to the linguistic delusions we studied in previous chapters – the attempts to reduce linguistic communication to a mechanistic model. In language, it is usually the syntax or the logic of a sentence that is believed to be the main structure. And the mechanistic theories

of language are failing for the same reason the mechanistic *software* theories are failing: the existence of other structures.

It would have been too much, perhaps, to expect the software theorists to recognize the similarity of software and language, and to learn from the failure of the linguistic theories. But even without this wisdom, it should have been obvious that software entities are related in many ways at the same time; that the flow-control structure is not independent; and that, as a result, applications cannot be represented mathematically. Thus, the theory of structured programming was refuted at this point, and should have been abandoned. Instead, its advocates decided to "improve" it – which they did by reinstating the old concepts, as this is the only way to cope with the complexity of applications. And so they turned structured programming into a pseudoscience.

The Second Delusion

1

The second delusion emerged when the theorists attempted to restore some of the complexity lost through the first delusion. An application implemented as a strict nesting scheme would be trivial, its performance no more complex than what could be represented with a hierarchical structure of sequential constructs. We could perhaps describe mathematically its flow of execution, but it would have no practical value. There are two reasons for this: first, without jumps in the flow of execution – jumps controlled by run-time conditions – the application would always do the same thing; second, without a way to link the flow-control structure to the structures that depict files, subroutines, business practices, and so forth, these processes would remain isolated and would have no bearing on the application's performance.

Real-world applications are complex phenomena, systems of interacting structures. So, to make structured programming practical, the theorists had to abandon the idea of a single structure. In the second delusion, they make the *flow-control* structure (supposed to be just the nesting scheme) a complex structure again, by permitting *multiple* flow-control structures. In the third delusion, we will see later, they make the whole application a complex structure again, by restoring the interactions between the flow-control structures and some of the other *types* of structures. And in the fourth delusion they abandon the last restrictions and permit any flow-control structures that are useful. The pseudoscientific nature of this project is revealed, as I already pointed out, by the reinstatement of concepts that were previously excluded (because they

contradict the principles of structured programming), and by the delusion that the theory can continue to function as originally claimed, despite these reversals.

The second delusion involves the standard conditional and iterative constructs. Under structured programming, you recall, these two constructs, along with the sequential construct, are the only flow-control constructs permitted. Because it is possible – in principle, at least – to implement any application using only these constructs and the nesting scheme, the three constructs are seen as a basic set of software operations.

The theorists look at the application's flow diagram, note that the flow-control constructs create levels of nesting, and conclude that their purpose is to combine software elements into higher-level elements – just as the operations that define a simple hierarchical structure create the elements of each level from those of the lower one. But this conclusion is mistaken: the theorists confuse the hierarchical nesting scheme and the three constructs, with the concept of a hierarchy and its operations.

Now, in the flow diagram the constructs do appear to combine elements on higher and higher levels; but in the running application their role is far more complex. The theorists believe that the restriction to a nesting scheme and standard constructs ensures that the flow-control structure is a *simple* structure, when the real purpose of these constructs is the exact opposite: to make the flow-control structure a *complex* structure.

This is easy to understand if we remember why we need these constructs in the first place. The conditional and iterative constructs provide (implicit) jumps in the flow of execution; and the function of jumps is to override the nesting scheme, by relating software elements in ways *additional* to the way they are related through the nesting scheme. We need the two constructs, thus, when certain requirements cannot be implemented with only one flow-control structure. As we saw earlier, the ability of an element to alter the flow of execution, implicitly or explicitly, is in effect a flow-control attribute (see pp. 529–530). Jumps override the nesting scheme by creating additional flow-control attributes, and hence additional flow-control structures. (We will examine these structures shortly.)

So the whole idea of standard flow-control constructs springs from a misunderstanding: the theorists mistakenly interpret the sequential, conditional, and iterative constructs as the *operations* of a hierarchical structure. To understand this mistake, let us start by recalling what *is* a hierarchical structure.

In a hierarchical structure, we combine a number of relatively simple elements (the *starting* elements) into more and more complex ones, until we reach the top element. The set of starting elements can be described as the basic building blocks of the hierarchy. At each level, the new elements are created by performing certain *operations* with the elements of the lower level. Thus, the elements become more and more complex as we move to higher levels, while the operations themselves may remain quite simple. The *definition* of a hierarchy includes the starting elements, the operations, and some rules describing their permissible uses.

The fallacy committed by the advocates of structured programming is in perceiving the three standard constructs as *operations* in the hierarchical structure that is the nesting scheme. The function of these constructs, in other words, is thought to be simply to combine software elements (the statements of a programming language) into larger and larger elements, one nesting level at a time. In reality, *only the sequential construct* combines elements into higher-level ones; the function of the conditional and iterative constructs is not to combine elements, but to generate multiple flow-control structures.

To appreciate why the conditional and iterative constructs are different, let us look at other kinds of structures and operations. In a physical structure, the starting elements are the basic components, and the operations are the means whereby the components are combined to form the levels of subassemblies. When the physical structure is a device like a machine, its performance too can be represented with a structure; and the operations in this structure are the ways in which the *working* of a subassembly is determined by the working of those at the lower level. In electronic systems, the starting elements are simple parts like resistors, capacitors, and transistors, and the operations are the connections that combine these parts into circuits, circuit boards, and devices. Here, too, in addition to the physical structure there is a structure that represents the performance of the system, and the operations in the latter are the ways in which the electronic functions at one level give rise to the functions we observe at the next higher level.

Turning now to software systems, consider a hypothetical application consisting of only one structure – the flow-control structure, just as structured programming says. The starting elements in this hierarchy are the statements permitted by a particular programming language, and the operations are the various ways in which statements are combined into blocks of statements, and blocks into larger blocks, modules, and so on. (Remember that the operations we are discussing here are the operations that define the hierarchical flow-control structure, which exists in time – *not* the operations we see as statements in a programming language; *those* operations function as the *starting elements* of the flow-control structure.) Clearly, if the flow of execution is to reflect

the flow-control structure, the operations must be determined solely by the nesting scheme. Or, to put it differently, the only operations required in a software structure are the relations that create a nesting scheme of software elements. In particular, we need operations to delimit blocks of statements, and to invoke modules or subroutines. Ultimately, the operations in a software hierarchy must fulfil the same function as those in other types of hierarchies: combining the elements of one level to create the elements of the next higher level.

Note what is common to all these hierarchical systems: they are based on a set of starting elements and a set of operations, which constitute the definition of the hierarchy; and these sets can then generate any number of *actual* structures – different objects, or devices, or circuits, or software applications. Each actual structure is one particular implementation of a certain hierarchical system – a physical system, an electronic system, or a software system; that is, one combination of elements out of the many possible in that system. Note also that the actual structures are fixed: when we create a particular combination of elements, we end up with a *specific* device, circuit, or software application. The same structure – the same combination of elements – cannot represent two devices, circuits, or applications.

Given these common features, the second delusion ought to be obvious: the three standard constructs are *not* the set of operations that make up the definition of hierarchical software systems, as the theorists believe; and consequently, the resulting structures are not simple hierarchical software structures. *That* set of operations is found in the concept of sequential constructs, and in the concepts of blocks, modules, and subroutines. These are the only operations we need in order to generate hierarchical structures of software elements; that is, to generate any nesting scheme. With these operations, we create a higher level by combining several elements into a larger one: *consecutive* elements when using sequential constructs, and *separate*, distant elements when invoking modules and subroutines. (Subroutines, of course, also serve to create other *types* of structures, as we saw under the first delusion. But we must discuss one delusion at a time, so here we assume, with the theorists, that the flow-control structure is an independent structure.)

Of the three standard constructs, then, only the sequential construct performs the kind of operation that defines a hierarchical structure. We do not need conditional or iterative constructs to create software structures, so these two constructs do not perform ordinary operations, and are not part of the definition of a software hierarchy. Hierarchical systems, we just saw, generate actual structures that are *fixed*; and the structures formed with these two constructs are *variable*. While ordinary operations consist in *combining* elements, the operation performed by the conditional and iterative constructs

consists in *selecting* elements. Specifically, instead of combining several elements into a higher-level element, the conditional and iterative constructs select one of two elements: in the conditional construct there is one selection, and one of the two elements may be empty; in the iterative construct the selection is performed in each iteration, and one of the two elements (the one selected when exiting the loop) is always empty.

So these two constructs do not treat software elements in the way a physical system treats the parts of a subassembly, or an electronic system treats the components of a circuit. In the other hierarchies, *all* the lower-level elements become part of the higher level, whereas in software hierarchies *only one* of the two elements that make up these constructs is actually executed. The real function of these constructs, therefore, is not to create higher-level elements within one nesting scheme, but to create multiple nesting schemes. Their function, in other words, is to turn the flow of execution from one structure into a system of structures.

2

If you still can't see how different they are from ordinary operations, note that both the conditional and the iterative constructs employ a *condition*. This is an important clue, and we can now analyze the second delusion with the method of simple and complex structures. Simple structures have no conditions in their operations. Hence, software structures that incorporate these constructs are complex, not simple. The condition, evaluated at run time and variously yielding *True* or *False*, is what generates the multiple structures.

Remember, again, that the structure we are discussing is the application's *flow of execution* (a structure that exists in time), not its *flow diagram* (a structure that exists in space). In flow diagrams these constructs do perhaps combine elements in a simple hierarchical way; but their run-time operation performs a selection, not a combination. And, since the running application includes all possible selections, it embodies all resulting structures.

Let us try, in our imagination, to identify and separate the structures that make up the *complex* flow-control structure – the one depicting the true manner in which the computer executes the application's elements. Let us start with the *static* structure (the flow diagram) and replace all the conditional and iterative constructs with the elements that are *actually* executed at run time. For simplicity, imagine an application that uses only one such construct (see figure 7-9).

Thus, in the case of the conditional construct, instead of a condition and two elements, what we will see in the run-time structure is a sequential construct

with one element – the element actually executed. And in the case of the iterative construct, instead of a condition and an element in a loop, what we will see is a sequential construct made up of several consecutive sequential constructs, their number being the number of times the element is executed at run time. More precisely, each iteration adds a sequential construct containing that element to the previous sequential construct, thereby generating a new, higher-level sequential construct.

In the flow of execution, then, there is only a sequential construct. So, from the perspective of the flow of execution, the original structure can be represented as two or more overlapping structures, which differ only in the sequential construct that replaces the original conditional or iterative one. It is quite easy to visualize the two resulting structures in the case of the conditional construct, where the sequential construct contains one or the other of the two elements that can be selected. In the case of the iterative construct, though, there are *many* structures, as many as the possible number of iterations: the final construct can include none, one, two, three, etc., merged sequential constructs, depending on the condition. Each number gives rise to a slightly different final construct, and hence a different structure. Clearly, since the running application can perform a different number of iterations at different times, it embodies all these structures.

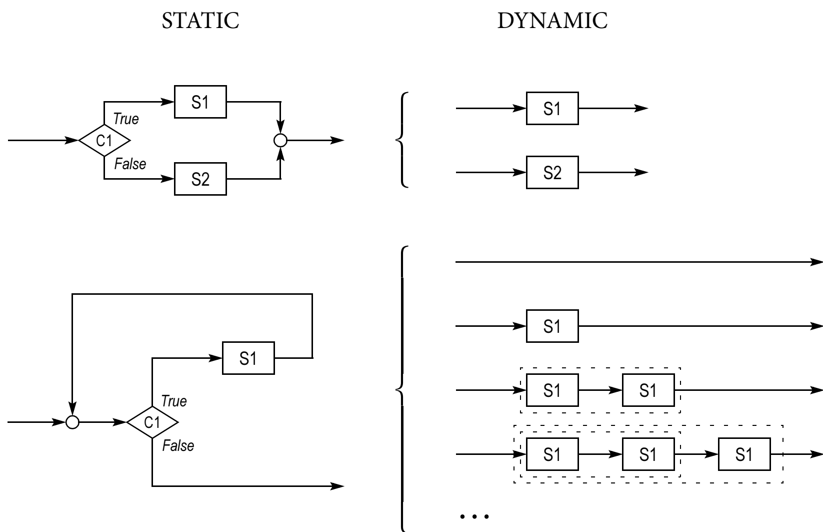

Figure 7-9

We can also explain the additional flow-control structures by counting the number of jumps implicit in a construct. Each jump in execution creates, as we know, a flow-control attribute, and hence a flow-control structure. The number of possible jumps reflects, therefore, the number of different sequences of execution that the application can display – the number of different paths that the execution can follow. In the conditional construct there are two possible paths, but only one needs a jump; the other is, in effect, the flow diagram itself. Let us decide, arbitrarily, that the path selected when the condition is *False* represents the flow diagram; then, the one selected when the condition is *True* represents the jump, and hence the *additional* structure.

In the iterative construct there are many possible paths, because the condition is evaluated in each iteration. For example, if in a particular situation the condition permits five iterations, this means that it is *True* five times, so there are five (backward) jumps. In another situation, the number of iterations, and hence the number of jumps generated by the construct, will be different. The number of possible structures is the largest number of iterations permitted by the condition, which is the same as the number of different paths. It is convenient to interpret these structures as those that are *additional* to the flow diagram; then, the flow diagram itself is represented by the path followed when ending the loop, when the condition is *False*.

To summarize, ordinary operations – the kind we see in other types of hierarchical systems – give rise to *fixed* structures, whereas the conditional and iterative software constructs give rise to *variable* structures. A variable structure is logically equivalent to a family of structures that are almost identical, sharing all their elements except for one sequential construct. And, since these structures exist together in the running application, a variable structure is the same as a complex structure.

We can also describe the flow-control constructs as means of turning a simple static structure (the flow diagram, which reflects the nesting scheme) into a complex dynamic one (the flow-control structure of the running application). Through its condition, each construct creates from one nesting scheme several flow-control structures, and hence several sequences of execution. (We saw earlier how a hierarchical structure defines, through the nesting convention, a specific sequence of execution; see p. 531.) The construct does this by endowing elements with several flow-control attributes, thereby relating them, from the perspective of the flow of execution, in several ways. We can call the individual flow-control structures *dynamic nesting schemes*, since each one is a slightly different version, in the running application, of the

static nesting scheme. The complex flow-control structure that reflects the performance of the application as a whole is then the totality of dynamic nesting schemes.

Complex structures cannot be reduced to simple ones, of course. We can perhaps study the individual structures when we assume one or two conditional or iterative constructs. But in real applications there are thousands of constructs, used at all levels, with elements and modules nested within one another. The links between structures are then too involved to analyze, and even to imagine.

So it is the *static* flow-control structure, not the dynamic one, that is the software equivalent of a physical structure. It is the *dynamic* structure that the theorists attempt to represent mathematically, though. Were their interest limited to flow diagrams, then strictly hierarchical methods like those used to build physical structures would indeed work. They work with the other types of systems because in those systems the dynamic structure usually mirrors the static one.[1]

We saw how each flow-control construct generates, through its condition, a system of flow-control structures. But in addition to the interactions between these structures, the flow-control constructs cause other interactions yet, with other *types* of structures. Here is how: The conditions employed by these constructs perform calculations and comparisons, so they necessarily involve memory variables, database fields, subroutines, or practices. They involve, thus, software processes; and, as we know, each process gives rise to a structure – the structure reflecting how the application's elements are affected by a particular variable, field, subroutine, or practice. Through their conditions, therefore, the flow-control constructs link the complex flow-control structure to some of the other structures that make up the application – structures that were supposedly isolated from the flow-control structure in the first delusion.

[1] Man-made physical systems that change over time (complicated mechanical or electronic systems) may well have a dynamic flow-control structure that is different from their static flow diagram and is, at the same time, complex – just like software systems. Even then, however, they remain relatively simple, so their dynamic behaviour can be usefully approximated with mechanistic means. They are equivalent, thus, to *trivial* software systems. We refrain from creating *physical* systems that we cannot fully understand and control, while being more ambitious with our *software* systems. But then, if we create software systems that are far more involved than our physical ones, we should be prepared to deal with the resulting complexity. It is absurd to attempt to represent them as we do the physical ones, mechanistically. Note that it is quite common for *natural* physical systems to display non-mechanistic behaviour (the three-body system, for instance, see pp. 107–108).

The links to those structures are officially reinstated by the theorists in the third delusion, but they must be mentioned here too, just to demonstrate the great complexity created by the conditional and iterative constructs, even as they are believed to be nothing but ordinary operations.

The complexity created by the conditional and iterative constructs is, in fact, even greater. For, in addition to the links to other *types* of structures, each construct creates links to other *flow-control* structures: to the families of structures generated by other constructs. The nesting process is what causes these links. Because these constructs are used at all levels, the links between the structures generated by a particular construct, at a particular level, also affect the structures generated by the constructs nested within it. So the links at the lower levels are *additional* to the links created by the lower-level constructs themselves.

3

The second delusion, we saw, consists in confusing the standard flow-control constructs with the set of operations that defines a simple hierarchical structure. The theorists are fascinated by the fact that three constructs are all we need, in principle, in order to create software applications; so they conclude, wrongly, that these constructs constitute a *minimal set of software operations*.

Now, a *real* minimal set of operations would indeed be an important discovery. If a set like this existed, then by restricting ourselves to these operations we could perhaps develop our applications mathematically. Even a minimal set defining just the flow-control structure (which is all we can hope for after the first delusion) would still be interesting. But if these constructs are not ordinary operations, the fact that they are a minimal set is irrelevant. If the flow-control structure is not a simple hierarchical structure, we cannot develop applications mathematically no matter what starting elements and operations we use.

The three constructs may well form a minimal set, but all we can say about it is that it is the minimal set of constructs that can generate enough *flow-control* structures to implement any software requirement. Here is why: The nesting scheme, as we know, endows all the elements in the application with one flow-control attribute; but each flow-control construct endows certain elements with *additional* flow-control attributes; finally, each one of these attributes gives rise to a flow-control structure, and this system of flow-control structures constitutes the application's flow of execution. To create serious applications, elements must be related through many different attributes, but only *some* of these attributes need to be of the flow-control type. What has been

proved, thus, is that the three standard constructs – in conjunction with the nesting scheme – provide, in principle, the minimal set of *flow-control* attributes required to create any application. In principle, then, we can replace the extra flow-control attributes present in a given application with other *types* of attributes. It is possible, therefore, to reduce all flow-control structures to structures based on the three standard constructs – if we agree to add other types of structures. (This is the essence of the transformations prescribed in the third delusion.)

So the idea of a minimal set of flow-control constructs may be an interesting subject of research in computer science, and this is how it was perceived by the scientists who first studied it.[2] But it is meaningless as a method of programming. For, if the flow-control structure (to say nothing of the application as a whole) ceases to be a simple hierarchical structure as soon as we add *any* conditional or iterative constructs to the nesting scheme, the dream of mathematical programming is lost, so it doesn't matter whether the minimal set has three constructs or thirty, or whether we restrict ourselves to a minimal set or create our own constructs.

When misinterpreting the function of the flow-control constructs, the software mechanists are committing the same fallacy as all the mechanists before them: attempting to represent a complex phenomenon by means of a simple structure. These constructs are seen as mere operations within the traditional nesting concept, when in reality they constitute a *new* concept – a concept powerful enough to turn simple static structures into complex dynamic ones. The mechanists, though, continue to believe that the flow of execution can be represented with one structure. So the real function of these constructs is to restore some of the complexity that was lost in the first delusion, when the mechanists reduced applications to one structure. (The rest of that complexity is restored in the third and fourth delusions.)

The fallacy, thus, is in the belief that we can discover a simple structure that has the potency of a complex one. The software mechanists note that the hierarchical concept allows us to generate large structures with just a few operations and starting elements, and that this is useful in fields like manufacturing and construction; and they want to have the same qualities in

[2] See, for example, Corrado Böhm and Giuseppe Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE) – paper originally published in *Communications of the ACM* 9, no. 5 (1966): 366–371. We will return to this paper later.

software. They want software systems to be simple hierarchical structures, but to retain their power and versatility; that is, their ability to perform tasks which *cannot* be performed by mechanical or electronic systems. They fail to see that this ability derives precisely from the fact that software allows us to create a kind of structures which the other systems do not – complex structures.

Nothing stops us from restricting software applications to simple hierarchical structures, just like those we create with the other systems. We would be able to develop, however, only trivial applications – only those that could be represented as a nesting scheme of sequential constructs. To create a greater variety of applications, we must enhance the nesting concept with the concept of conditional and iterative constructs; but then the applications are no longer simple structures. In the end, it is only through self-deception that the mechanists manage to have a simple structure with the potency of a complex one: they are creating complex software structures while continuing to believe that they are working with simple ones.

The Third Delusion

1

The first delusion, we recall, was the belief that the flow-control structure can be isolated from the other structures that make up the application, and that it can be reduced to a simple structure. With the second delusion, the flow-control structure became a system of interacting flow-control structures; moreover, it was linked, through the conditions used in the flow-control constructs, to other *types* of structures. Thus, if after the first delusion the expectation of a mechanistic representation of the flow-control structure was still valid, this expectation was illogical after the second delusion, when it became a *complex* structure.

The third delusion is the belief that it is important to reduce the application – through a series of transformations – to the flow-control structure defined in the second delusion. It is important, the theorists insist, because *that* structure can be represented mechanistically. Through this reduction, therefore, we will represent the entire application mechanistically. Just as they succumbed to the second delusion when attempting to suppress the evidence of complexity after the first one, the theorists succumbed to the third delusion because they ignored the evidence of complexity after the second one.

I defined the four delusions as stages in a process of degradation, as distinct opportunities for the theorists and the practitioners to recognize the fallaciousness of structured programming. On this definition, the third

delusion is a new development. The idea of structured programming could have ended with the second delusion, when the conditional and iterative constructs were introduced, since the very need for these constructs proves that the flow-control structure of a serious application is more than a simple hierarchical structure. Having missed the second opportunity to recognize their mistake, the theorists promoted now the idea of transformations: we must modify the application's requirements so as to limit the application to flow-control structures based on the three standard constructs; all other flow-control structures must be replaced with structures based on shared data or shared operations (see pp. 524–527).

Everyone could see that these transformations are artificial, that they complicate the application, that in most situations they are totally impractical, and that even when we manage to implement them we still cannot prove our applications mathematically. Yet no one wondered why, if the principle of hierarchical structures works so well in other fields, if we understand it so readily and implement it so easily with other systems, it is impractical for *software* systems. No one saw this as one more piece of evidence that software applications are not simple hierarchical structures. Thus, the theorists and the practitioners missed the third opportunity to recognize the fallaciousness of structured programming.

2

Let us review the motivation for the transformations. To perform a given task, the application's elements must be *related*; and, usually, they must be related in more than one way. It is by sharing attributes that software elements are related. Each attribute gives rise to a set of relations; namely, the structure representing how the application's elements are affected by that attribute.

There are several types of attributes and relations. A relation is formed, for example, when elements use the same memory variable or database field, when they perform the same operation or call the same subroutine, or when they are part of the same business practice. Elements can also be related through the flow of execution: the relative sequence in which they are executed constitutes a logical relation, so it acts as a shared attribute. And it is only this type of attributes and relations – the *flow-control* type – that structured programming recognizes.

A software element can possess more than one attribute. Thus, an element can use several variables, call several subroutines, and be part of several practices. Each attribute gives rise to a different set of relations between the application's elements, so each element can be related to the others in several

ways at the same time. Since these sets of relations are the structures that make up the application, we can also express this by saying that each element is part of several structures at the same time. The multiplicity of software relations is necessary because this is how the *real* entities – the processes and events that make up our affairs, and which we want to represent in software – are related.

As is the case with the other types of attributes, elements can possess more than one *flow-control* attribute. Elements, therefore, can also be related to one another in more than one way through the flow of execution. Multiple flow-control relations are necessary when the relative position of an element in the flow of execution must change while the application is running.

The nesting scheme (the static arrangement we see in the application's flow diagram) provides *one* of these attributes. The nesting scheme defines a formal, precise set of relations, which constitutes in effect a *default* flow-control attribute – one shared by all the elements in the application. And if this were the *only* flow-control attribute, the application would have only one flow-control structure – a structure mirroring, in the actual flow of execution, the hierarchical structure that is the static nesting scheme.

In serious applications, though, elements must be related through more than one flow-control attribute, so the simple flow of execution established by the nesting scheme is insufficient. The additional flow-control attributes are implemented by performing *jumps* in the flow of execution; that is, by *overriding* the sequence dictated by the nesting scheme. The elements from which and to which a jump occurs, and the elements bypassed by the jump, are then related – when viewed from the perspective of the flow of execution – in two ways: through the nesting scheme, and through the connection created by the jump. Jumps provide an *alternative* to the sequence established by the nesting scheme: whether the flow of execution follows one jump or another, or the nesting scheme, depends on run-time conditions. So the execution of each element in the application reflects, in the end, both the nesting scheme and the various jumps that affect it.

Jumps can be explicit or implicit. Explicit jumps (typically implemented with GOTO statements) permit us to create any flow-control relations we want. Programming languages, though, also provide a number of *built-in* flow-control constructs. These constructs are basic syntactic units designed to create automatically, by means of implicit jumps, some of the more common flow-control relations. The best-known built-in constructs, and the only ones permitted by structured programming, are the elementary conditional and iterative constructs (also known as the standard constructs).

By eliminating the explicit jumps, these constructs simplify programming. But they are not versatile enough to satisfy all likely requirements; in fact, as we

saw earlier, even some very simple requirements cannot be implemented with these constructs alone. The impossibility of implementing a given requirement means that some elements must have more flow-control attributes than what the nesting scheme and the standard constructs provide. Some elements, in other words, must be related to others – when viewed from the perspective of the flow of execution – in more ways than the number of jumps implicit in these constructs. (For the conditional construct, we recall, there is one possible jump, one way to override the nesting scheme; and for the iterative construct, the number of jumps equals the number of possible iterations. The sequential construct is not mentioned in this discussion, since it does not provide a jump that can override the nesting scheme; sequential constructs, in fact, are the entities that form the original nesting scheme, before adding conditional and iterative constructs.)

We shouldn't be surprised that software elements need more flow-control attributes for a difficult requirement than they do for a simple one; after all, we are not surprised that elements need more of the *other* types of attributes for a difficult requirement (they need to use more variables or database fields, for instance, or to call more subroutines).

Now, we could implement the additional flow-control relations by enhancing the standard conditional and iterative constructs, or by creating our own, specialized constructs. In either case, though, we would have to add flow-control attributes in the form of explicit jumps, and this is prohibited under structured programming. The reason it is prohibited, we saw under the second delusion, is the belief that applications restricted to the standard constructs have only one flow-control structure (the nesting scheme). And this, in turn, allows us to represent, develop, and prove them mathematically. Thus, the theorists say, since it is possible, in principle, to transform any requirements into a format programmable with the standard constructs alone, and since the benefits of this concept are so great, any effort invested in realizing it is worthwhile. This is the motivation for the transformations.

The transformations convert those flow-control relations that we need but cannot implement with the standard constructs, into relations based on shared data or shared operations. They convert, thus, some of the flow-control structures into other *types* of structures (so they create more structures of the types that have been ignored since the first delusion). When shared by several elements, data and operations can serve as attributes, since they relate the elements logically. (This, obviously, is why they can be used as substitutes for the *flow-control* attributes.)

Consider a simple example. If we want to override the nesting scheme by jumping across several elements and levels without resorting to GOTO, we can use a memory variable, like this: In the first element, instead of performing a jump, we assign the value 1 to a variable that is normally 0. Then, we enclose each element that would have been bypassed by the jump, inside a conditional construct where the condition is the value of this variable: if 1, the element is bypassed. So the flow of execution can follow the nesting scheme, as before, but those elements controlled by the condition will be bypassed rather than executed. In this way, *one* flow-control relation based on an explicit jump is replaced with *several* flow-control relations based on the standard conditional construct, plus *one* relation based on shared data.

The paper written by Corrado Böhm and Giuseppe Jacopini,[1] regarded by everyone as the mathematical foundation of structured programming, proved that we can always use pieces of storage (in ways similar to the foregoing example) to reduce an arbitrary flow diagram to a diagram based on the sequential, conditional, and iterative constructs. The paper proved, in other words, that any flow-control structure can be transformed into a functionally equivalent structure where the elements possess no more than three types of flow-control attributes: one provided by the nesting concept and by merging consecutive sequential constructs, and the others by the conditional and iterative constructs.[2]

Another way to put this is by stating that any flow of execution can be implemented by using no more than three types of flow-control relations. A simple nesting scheme, made up of sequential constructs alone, is insufficient. We need more than one type of relations between elements if we want the ability to implement any conceivable requirement. But we don't need more than a certain *minimal set* of relations. The minimal set includes the relations created by the nesting scheme, and those created by the standard conditional and iterative constructs. Any other flow-control relations can be replaced with relations based on other types of attributes; specifically, relations based on shared data or shared operations.

It is important to note that the paper only proved these facts *in principle*; that is, from a theoretical perspective. It did not prove that practical applications can actually be programmed in this fashion. This is an important point,

[1] Corrado Böhm and Giuseppe Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE) – paper originally published in *Communications of the ACM* 9, no. 5 (1966): 366–371.

[2] The paper proved, in fact, that conditional constructs can be further transformed into iterative constructs; so, in the end, only sequential and iterative constructs are necessary. I will return to this point later.

because the effort of performing the transformations – the essence of the third delusion – is justified by citing this paper, when in reality the paper is only a study in mathematical logic, unconcerned with the *practicality* of the transformations. (We will analyze this misrepresentation shortly.)

But regardless of their impracticality, the transformations would only make sense if the resulting flow-control structure were indeed a simple structure. The fallacies of the second delusion, thus, beget the fallacies of the third one: because they believe that a flow-control structure restricted to the standard constructs is a simple structure, the advocates of structured programming believe that the effort of performing the transformations is worthwhile.

The difficulty of programming – what demands skills and experience – is largely the need to deal with multiple structures, and hence with simultaneous relations. The theorists acknowledge this when they stress the importance of reducing the application to *one* flow-control structure: if every element in the application is restricted to one flow-control attribute, every element will be related to the others in only one way, and the application – viewed from the perspective of the flow of execution – will have only one structure. We will then be able to represent the application with a mechanistic model, and hence develop and prove it with the tools of mathematics. To put this differently, by eliminating the need to deal with simultaneous relations in our mind we will turn programming into a routine activity, and thereby eliminate the need for personal skills and experience.

This is the idea behind structured programming, but then the theorists contradict themselves and permit *several* flow-control relations per element, not one: the nesting scheme *plus* the relations generated by the standard conditional and iterative constructs. The flow-control structure, as a result, is a system of interacting structures. It was a simple hierarchical structure only when it was a nesting scheme of elements that were all sequential constructs. The *implicit* jumps that are part of the standard constructs create additional flow-control relations between the application's elements in exactly the same way that *explicit* jumps would. It is quite silly to think that, just because there are no explicit jumps – no GOTO statements – we have only one flow-control structure. After all, the very reason we added the conditional and iterative constructs is that the nesting scheme alone (a simple structure) could not provide all the flow-control relations we needed.

The theorists believe that transformations keep the flow-control structure simple because they eliminate the *non-standard* constructs. But if the *standard* constructs already make the flow-control structure complex, the use of non-

standard ones is irrelevant, since we can no longer represent the flow-control structure mechanistically anyway. So, whether easy or difficult to implement, the transformations are futile if their purpose is to turn programming into a routine activity. Both with and without transformations, the flow-control structure is a system of interacting structures, so the most difficult aspect of programming – the need to process multiple structures in the mind – remains unchanged. Thus, because structured programming fails to reduce applications to a simple structure, it also fails to simplify programming.

And we must not forget that the transformations work by replacing flow-control structures with structures of other types, so in the end they add to the complexity of *other* systems of structures. Therefore, in those situations where an explicit jump provides the most effective relation between elements, the transformation will replace one structure with several, making the application as a whole *more* involved. (The impracticality of the transformations is finally acknowledged by the theorists in the fourth delusion.)

It is up to the programmer to select the most effective system of structures for a given requirement, and this system may well include some flow-control structures generated by means of explicit jumps. Discovering the best system and coping with the unavoidable interactions – this, ultimately, is the skill of programming. Since our affairs, and the software that mirrors them, consist of interacting structures, we *must* develop the capacity to deal with these interactions if we want to have useful applications. The aim of structured programming is to obviate the need for this expertise; specifically, to turn programming from an activity demanding skills and experience into one demanding only mechanistic knowledge. But now we see that, in their desire to simplify programming, the theorists *add* to the complexity of software, and end up making programming more difficult.

3

Let us examine next the mechanistic belief that it is possible to *actually* implement an idea that was only shown to be valid *in principle*. We saw that even when we manage to reduce the application to standard constructs, the flow of execution, and the application as a whole, remain complex structures; so the transformations are always futile. Let us ignore this fallacy, though, and assume with the theorists that by applying the transformations we *will* be able to represent the application mathematically, so the effort is worthwhile. But Böhm and Jacopini's paper only shows that applications can be reduced to the standard constructs *in principle*. The theorists, thus, are confidently promoting the idea of transformations when there is nothing – apart from

a blind faith in mechanism – to guarantee that this idea can work with *practical* applications.

It is common for mathematical concepts to be valid in principle but not in practice, and many mechanistic delusions spring from confusing the theoretical with the practical aspects of an idea. The pseudosciences we studied in chapter 3, for instance, are founded upon the idea that it is possible to account for all the alternatives displayed by human minds and human societies. They claim, thus, that it is possible to discover a mechanistic model where the starting elements are some basic physiological entities, and the values of the top element represent every possible mental act, or behaviour pattern, or social phenomenon (see pp. 281–284). Now, it is perhaps true that every alternative of the top element is, ultimately, a combination of some elementary entities or propensities; but it doesn't follow that we can express these combinations in precise, mathematical terms. The mechanists invoke the principles of reductionism and atomism to justify their optimism, but they *cannot* discover a working mechanistic model; that is, a continuous series of reductions down to the basic entities. So, while it may be possible *in principle* to explain human intelligence or behaviour in terms of low-level physiological entities, we cannot *actually* do it.

The most fantastic mechanistic delusion is Laplacean determinism, which makes the following claim: the world is nothing but a system of particles of matter acting upon one another according to the mechanistic theory of gravitation; it should therefore be possible, *in principle*, to explain all current entities and phenomena, and to predict all future ones, simply by expressing the relations between all the particles in the universe in the form of equations and then solving these equations. The mechanists admit that this is only an idea, that we cannot *actually* do it; but this doesn't stop them from concluding that the world is deterministic. (We will discuss this fallacy in greater detail in chapter 8; see pp. 810–812.)

Returning to the domain of computer science, a well-known example of a mechanistic model that is only an idea is the Turing machine.[3] This theoretical device consists of a read-write head and a tape that moves under it in both directions, one position at a time. The device can be in one of a finite number of internal states, and its current state changes at each step. Also at each step, the device reads the symbol found on the tape in the current position, perhaps erases it or writes another one, and then advances the tape one position left or right. The operations performed at each step (erasing, replacing, or leaving the symbol unchanged; advancing the tape left or right; and selecting the

[3] Named after the mathematician and computer pioneer Alan Turing, who invented it while studying the concept of computable and non-computable functions.

next internal state) depend solely on the current state and the symbol found in the current position.

Turing machines can be programmed to execute algorithms. For example, if the permissible symbols are the digits *0* to *9*, a program could read a series of digits written in consecutive positions on the tape, interpret them as a number, calculate its square root by using the tape as working area, and finally erase the temporary symbols and write on the tape the digits that make up the result. (The program for a Turing machine is not a list of instructions, as for a computer, but a table specifying the operations to be performed for every possible combination of machine states and input symbols.)

There are many variations, but the most interesting Turing machines are those that define a *minimal* device: the machine with the smallest number of internal states, or the smallest alphabet of symbols, or the shortest tape, that can still solve any problem from a certain class of problems. It should be obvious, for instance, that we can always restrict Turing machines to two symbols, such as *0* and *1*, since we can reduce any data to this binary representation, just as we do in computers. Compared with devices that use a larger alphabet – the full set of letters and digits, for example – the minimal device would merely need a larger program and a longer tape to execute a given algorithm.

Now, it has been proved that a Turing machine can be programmed to execute, essentially, any algorithm. This simple computational device can represent, therefore, any deterministic phenomenon, any process that can be described precisely and completely. In particular, it can be programmed to execute any task that can be executed by more complicated devices – computers, for instance. Again, the program for the Turing machine would be larger and less efficient, and in most cases totally impractical, but the device is only an idea. We are only interested in the fact that, *in principle*, it can solve any problem. In principle, then, any problem, no matter how complicated, can be reduced to the simple operations possible on a basic Turing machine.

Thus, although the Turing machine is only a theoretical device, it is an interesting subject of study in computer science. Since we know that anything that can be computed can also be computed on a Turing machine, we can determine, say, whether a certain problem can be solved at all mathematically, by determining whether or not it can be programmed on a Turing machine. Often this is easier than actually finding a mathematical solution. The practicality of this program is irrelevant, since we don't have to run it, or even to develop it; all we need is the knowledge that such a program could be developed.

Restricting software applications to the standard flow-control constructs is just like these other ideas: it is only possible *in principle*. Just like the theories that can explain only *in principle* any intelligent act, or those that can predict only *in principle* any future event, or the Turing machine that can execute only *in principle* any algorithm, it is possible only *in principle* to restrict software applications to the three standard constructs. The software theorists, thus, are promoting as a *practical* programming method an idea that is the software counterpart of well-known mechanistic fantasies.

Despite our mechanistic culture, not many scientists seriously claim that those other ideas have immediate, practical applications. But the software experts were enthusiastic about the possibility of mathematical programming. The idea of transformations – and hence the whole idea of structured programming, which ultimately depends on the practicality of these transformations – was taken seriously by every theorist, even though one could see from the start that it is the same type of fantasy as the other ideas.

But it is the Turing machine that is of greatest interest to us, not only because of its connection to programming in general, but also because Böhm and Jacopini actually discuss in their paper the link between Turing machines and standard flow-control constructs. (This link is clearly indicated even in the paper's title: "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.")

Although computers can be reduced to Turing machines, everyone agrees that this is true only in principle, that most tasks would be totally impractical on a Turing machine. Thus, no one has suggested that, given the theoretical benefits of minimal computational devices, we replace our computers with Turing machines. Nor has anyone suggested that, given the theoretical benefits of minimal programming languages, we simulate the operation of a Turing machine on our computers, and then restrict programming languages to the instructions of the Turing machine.

At the same time, the software theorists perceive the transformations as a *practical* programming principle, and insist that we *actually* restrict our applications to the standard constructs. Their naivety is so great that, even in a mechanistic culture like ours, it is hard to find a precedent for such an absurd claim. And we must not forget that the delusion of transformations is *additional* to the two delusions we discussed earlier. This means that, since applications cannot be represented mechanistically in any case, the transformations would be futile even if they were practical.

It is important to emphasize that Böhm and Jacopini discussed the standard constructs and the transformations strictly as a concept in mathematical logic; they say nothing in their paper about grounding a programming theory on this concept. It was only the advocates of structured programming who,

desperate to find a scientific foundation for their mechanistic fantasy, decided to interpret the paper in this manner. Having accepted as established fact what was only a wish – the idea that software applications can be represented mathematically – they saw in this paper something that its authors had not intended: the evidence for the possibility of a *practical* mechanistic programming theory.

The link between flow diagrams and Turing machines discussed by Böhm and Jacopini is this: They demonstrated that there exists a minimal Turing machine which is logically equivalent to a flow diagram restricted to the standard flow-control constructs. More specifically, they showed that a Turing machine restricted to the equivalent of the sequential, conditional, and iterative operations can still execute, essentially, any algorithm. In other words, any Turing machine, no matter how complicated, can be reduced, in principle, to this minimal configuration.

By discussing the link between flow diagrams and Turing machines, then, Böhm and Jacopini asserted in effect that they considered the transformation of flow diagrams to be, like the Turing machine, *a purely theoretical concept*. So it can be said that their study is the exact opposite of what the later theorists claimed it was: it is an *abstract* idea, not the basis of a *practical* programming theory. The study is *misrepresented* when invoked as the foundation of structured programming.

4

We saw that the advocates of structured programming misrepresent Böhm and Jacopini's paper when invoking it as the foundation of a practical programming theory. But this is not all. They also misrepresent the paper when saying that it proved that *only three* flow-control constructs – the sequential, the conditional, and the iterative – are necessary to create software applications. In reality, the paper proved that *only two* constructs are necessary – the sequential and the iterative ones. The conditional construct, it turns out, is merely a special case of the iterative construct. Just as we can reduce through transformations all non-standard constructs to conditional and iterative ones, we can *further* reduce, through similar transformations, all conditional constructs to iterative ones.

Thus, the paper is routinely depicted as the mathematical foundation of structured programming, and we are told that the only way to derive the benefits of mathematics is by restricting our applications to the elementary sequential, conditional, and iterative constructs – while the paper itself shows that the conditional construct is *not* an elementary construct. There are

thousands of references to this paper – in casual as well as formal discussions of structured programming, in popular as well as professional publications – and it is difficult to find a single one stating what Böhm and Jacopini *actually* proved. According to all these references, they proved that applications can be built from three, not two, elementary constructs. We must study now this second aspect of the misrepresentation.

It is true that Böhm and Jacopini started by proving that any flow diagram can be reduced to the three elementary constructs. But they went on and proved that the conditional construct can be reduced to the iterative one. And they also proved that an equivalent reduction is possible for Turing machines (so the minimal Turing machine does not require conditional operations). Like the link to Turing machines, this final reduction is clearly indicated even in the paper's title ("... Languages with Only Two Formation Rules") and in its introduction ("... a language which admits as formation rules only composition [i.e., merging consecutive constructs] and iteration"[4]).

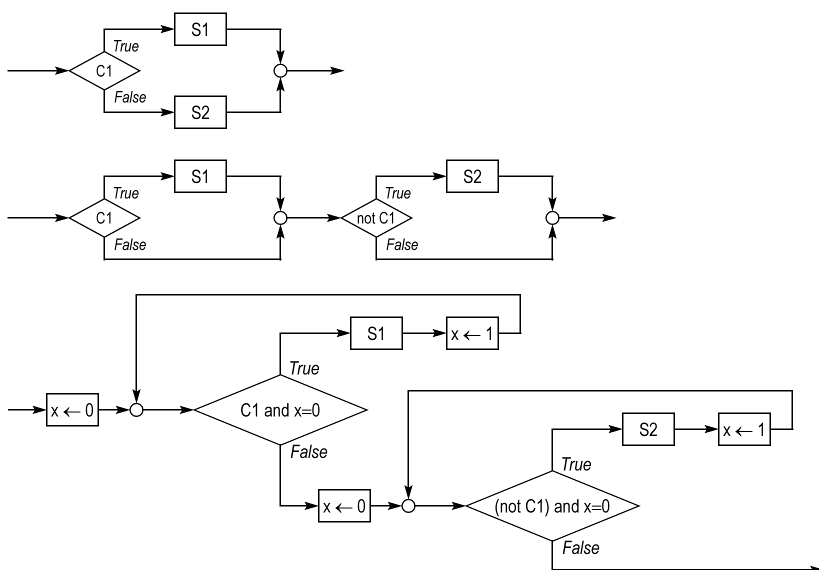

Figure 7-10

Although they proved it through mathematical logic, we can demonstrate this reduction with flow diagrams (see figure 7-10). In the first step, the conditional construct is reduced to two consecutive, simpler conditional

[4] Böhm and Jacopini, "Flow Diagrams," p. 3.

constructs. The new constructs have only one operation each, *S1* and *S2*, and the condition in the second one is the logical negation of the original condition. In the second step, the new constructs are transformed into two consecutive iterative constructs: the variable *x*, cleared before each loop and set in the first iteration, is part of the condition. In the end, either *S1* or *S2* is executed, and only once.[5]

The fallacy of the third delusion, we saw, is the idea of transformations. But now we see that, even *within* this idea, there is an inconsistency. And, even if we ignore the other delusions, and the other fallacies of the third delusion, this inconsistency alone is serious enough to cast doubt on the entire idea of structured programming.

The inconsistency is this: The theorists tell us that we must reduce our applications to the three standard flow-control constructs, because only if created with these elementary entities can applications be developed and proved mathematically. But if the most elementary software entities are *two* constructs, not three, the theorists claim in effect that even with some *unreduced* constructs we can derive the benefits of mathematical programming. It is *unimportant*, they tell us, to reduce our applications from three constructs to two; that is, from the conditional to the iterative construct. This reduction, though, as shown in figure 7-10, is similar to the reduction from any non-standard construct to the standard ones. (We saw examples of these other reductions earlier, from figure 7-5 to 7-6 and from figure 7-7 to 7-8, pp. 524–527.) So, if the mathematical benefits are preserved even without a complete reduction, if it is unimportant to reduce our applications from three constructs to two, why is it important to reduce them to three constructs in the first place?

Imagine an application that also employs a non-standard construct, for a total of *four* types of constructs. If this application can be reduced through similar transformations from four constructs to three and from three to two, and if at the same time it is unimportant to reduce it from three to two, then it must also be unimportant to reduce it from four to three. And, continuing this logic, it must also be unimportant to reduce an application from five constructs to four, from six to five, and so on. In other words, whatever mathematical benefits we are promised to gain from a reduction to the three standard constructs are ours to enjoy with any number of non-standard constructs. The transformations, therefore, and structured programming generally, are unnecessary, and we should be able to develop and prove our applications mathematically no matter how we choose to program them.

[5] The use of a memory variable as switch was explained earlier, for the transformation shown in figure 7-8 (see pp. 526–527).

The theory of structured programming, thus, is inconsistent if its principles prescribe a certain programming method, and the same principles lead to the conclusion that this method is irrelevant. The promoters of structured programming failed to notice what is, in fact, a blatant self-contradiction: claiming, at the same time, that it is important and that it is unimportant to reduce applications to three constructs. Having misrepresented Böhm and Jacopini's paper as the basis of a practical programming theory (as we saw earlier), they were now actually attempting to implement their fantasy. So, in their eagerness, they added to the misrepresentation. Moreover, they added to the theory's fallaciousness, by making it inconsistent.

It is impossible to prove mathematically the correctness of our applications – with or without transformations, with three or with two constructs. Since applications are not simple structures, the idea of mathematical programming is a fantasy, so there are no benefits in reducing them to *any* set of constructs. Let us ignore for a moment, though, this fallacy, and believe with the theorists that the transformations *are* worthwhile. But then, to be consistent – that is, to benefit from these transformations – we would have to seek a complete reduction, to two constructs. This shows that stopping the reduction at three constructs is a *separate* fallacy, *additional* to the fallacy of mathematical programming.

The following quotations are typical of how Böhm and Jacopini's work is misrepresented in programming books (that is, by mentioning the reduction to *three* constructs, not two): "In 1966, Böhm and Jacopini formally proved the basic theory of structured programming, that any program can be written using only three logical constructs."[6] "One of the theoretical milestones of systems science was Böhm and Jacopini's proof that demonstrated it was possible to build a good program using only three logical means of construction: sequences, alternatives, and repetition of instruction."[7] "The first major step toward structured programming was made in a paper published by C. Böhm and G. Jacopini.… They demonstrated that three basic control structures, or constructs, were sufficient for expressing any flowchartable program logic."[8] "According to Böhm and Jacopini, we need three basic building blocks in order to construct a program: 1. A process box. 2. A

[6] Victor Weinberg, *Structured Analysis* (Englewood Cliffs, NJ: Prentice Hall, 1980), p. 27.

[7] Ken Orr, *Structured Requirements Definition* (Topeka, KS: Ken Orr and Associates, 1981), p. 58.

[8] Randall W. Jensen, "Structured Programming," in *Software Engineering*, eds. Randall W. Jensen and Charles C. Tonies (Englewood Cliffs, NJ: Prentice Hall, 1979), p. 228.

generalized loop mechanism. 3. A binary-decision mechanism."[9] "Böhm and Jacopini … first showed that statement sequencing, IF-THEN-ELSE conditional branching, and DO-WHILE conditional iteration would suffice as a set of control structures for expressing *any* flow-chartable program logic."[10] "In a now-classical paper, Böhm and Jacopini proved that any 'proper' program can be solved using only the three *logic structures* … 1. Sequence. 2. Selection. 3. Iteration."[11] "Böhm and Jacopini provided the theoretical framework by showing it possible to write any program using only three logic structures: DOWHILE, IFTHENELSE, and SEQUENCE."[12] "A basic fact about structured programming is that it is known to be possible to duplicate the action of any flowchartable program by a program which uses as few as three basic program figures, namely, a SEQUENCE, an IFTHENELSE, and a WHILEDO.… This fact is due to C. Böhm and G. Jacopini."[13] "Structured programming is a technique of writing programs that is based on the theorem (proved by Böhm and Jacopini) that any program's logic, no matter how complex, can be unambiguously represented as a sequence of operations, using only three basic structures."[14]

Even the *Encyclopedia of Computer Science*, in the article on structured programming, says the same thing: "… a seminal paper by Böhm and Jacopini, who proved that every 'flowchart' (program), however complicated, could be rewritten in an equivalent way using only repeated or nested subunits of no more than three different kinds – a *sequence* of executable statements, a *decision* clause … and an *iteration* construct."[15]

Why did the theorists misrepresent the original study? Why did they not insist on a complete reduction, to two constructs, just as Böhm and Jacopini did in their paper? Why, in other words, do they permit us to use the conditional construct, when the paper proved that it is *not* an elementary construct, and that it can be reduced to the iterative one?

[9] Edward Yourdon, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice Hall, 1975), p. 146.

[10] Clement L. McGowan and John R. Kelly, *Top-Down Structured Programming Techniques* (New York: Petrocelli/Charter, 1975), p. 5.

[11] Robert T. Grauer and Marshal A. Crawford, *The COBOL Environment* (Englewood Cliffs, NJ: Prentice Hall, 1979), p. 4.

[12] Gary L. Richardson, Charles W. Butler, and John D. Tomlinson, *A Primer on Structured Program Design* (New York: Petrocelli Books, 1980), p. 4.

[13] Richard C. Linger and Harlan D. Mills, "On the Development of Large Reliable Programs," in *Current Trends in Programming Methodology*, vol. 1, *Software Specification and Design*, ed. Raymond T. Yeh (Englewood Cliffs, NJ: Prentice Hall, 1977), p. 122.

[14] Donald A. Sordillo, *The Programmer's ANSI COBOL Reference Manual* (Englewood Cliffs, NJ: Prentice Hall, 1978), pp. 296–297.

[15] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1309.

To understand the reason, recall that characteristic feature of structured programming – the continual blending of formal and informal concepts. The theorists like the formal, mechanistic principles, so they invoke them whenever they want to make their claims appear "scientific." But, because software applications are non-mechanistic phenomena, the formal principles are useless; so the theorists are compelled to revert to the informal concepts.

Thus, it would be embarrassing to ask programmers to avoid conditional constructs, just because they are not elementary (that is, to replace them, in the name of science, with their unwieldy transformation into iterative constructs), seeing that programming languages already include the simple IF statement, designed specifically for implementing conditional constructs.

But programming languages also include the simple GOTO statement, designed specifically for implementing jumps, and hence non-elementary flow-control constructs. And yet, while permitting us to use IF, the theorists prohibit us from using GOTO. The explanation for the discrepancy is that asking us to avoid GOTO can be made to look scientific, while asking us to avoid IF can only look silly.

Mathematically, a flow diagram with GOTO statements is no different from one with IF statements, since both can be reduced, through similar transformations, to the same two elementary flow-control constructs. The theorists, though, consider the former to be "unstructured" and the latter "structured." This attitude – invoking the formal, precise principles when practical, and reverting to informal guidelines when the formal principles are inconvenient – is the essence of the fourth delusion, as we will soon see. At that point, many other non-elementary flow-control constructs will be permitted.

The academics and the gurus who routinely cite Böhm and Jacopini's paper probably never set eyes on it. Most likely, only a handful of theorists actually studied it, and, blinded by their mechanistic obsession, saw in it the proof for the possibility of a *science* of programming. The other theorists, and the authors and the teachers, accepted then uncritically this distorted interpretation and helped to spread it further. By the time it reached the books and the periodicals, the programmers and the managers, and the general public, no one was questioning the interpretation, or verifying it against the original ideas. (Few would have understood the original paper anyway, written as it is in a formal, and rather difficult and laconic, language.) Everyone was convinced that structured programming is an important theory, mathematically grounded on Böhm and Jacopini's work, when in reality it is just another mechanistic fantasy, grounded on a *misrepresentation* of that work.

And so it is how Böhm and Jacopini – humble authors of an abstract study of flow diagrams – became unwitting pioneers of the structured programming revolution.

5

For twenty years, in thousands of books, articles, and lectures, the software experts were promoting structured programming. To understand how it is possible to promote an invalid theory for twenty years, it may help to analyze the style of this promotion. Typically, the experts start their discussion by presenting the formal principles and the mathematical foundation; and, often, they mention Böhm and Jacopini's paper explicitly, as in the passages previously quoted. This serves to set a serious, authoritative tone; but then they continue with informal, childish arguments.

For example, the *Encyclopedia of Computer Science*,[16] after citing Böhm and Jacopini's "seminal paper," includes the following "principles" in the definition of structured programming: "judicious use of embedded comments" (notes to explain the program's logic); "a preference for straightforward, easily readable code over slightly more efficient but obtuse code"; modules no larger than about one page, "mostly for the sake of the human reader"; "careful organization of each such page into clearly recognizable paragraphs based on appropriate indentation" of the nested constructs (again, for ease of reading).

Some of these "principles" make good programming sense, but what have they to do with the theory of structured programming? Besides, if the validity of structured programming has been proved mathematically, why are these informal guidelines mentioned here? Or, conversely, if structured programming is no longer a formal theory and "may be defined as a methodological style,"[17] why mention Böhm and Jacopini's mathematical foundation? The formal and the informal arguments overlap continually. They appear to support each other, but in fact the informal ones are needed only because the formal theory does not work.

Incredibly, we also find the following requirement listed as a structured programming principle: "the ability to make assertions about key segments of a structured program so as to 'prove' that the program is correct."[18] The editors enclosed the word "prove" in quotation marks presumably because the principle only stipulates an informal verification, not a real proof. This principle, thus, is quite ludicrous, seeing that structured programming is

[16] The quotations in this paragraph are ibid., pp. 1309–1311.
[17] Ibid., p. 1308.
[18] Ibid., p. 1311.

supposed to guarantee *mathematically* (that is, with no qualifications) the correctness of software; it is supposed to guarantee, moreover, the correctness of the entire program, not just "key segments."

Another absurd principle is the permission to deviate from the standard constructs if this "removes a gross inefficiency."[19] It is illogical to suggest that what is, in fact, the main tenet of this theory – the standard constructs – may cause inefficiency and must be forsaken. This principle is an excellent example of pseudoscientific thinking: every situation where one must deviate from the standard constructs is a *falsification* of structured programming; and the experts suppress these falsifications by turning them into *features* of the theory – non-standard constructs.

Lastly, we are told that "still further evolution of [structured programming] is to be expected."[20] The editors seem to have forgotten that structured programming is a formally defined theory, so it cannot evolve. What can evolve is only the *interpretations* of the theory. Only pseudosciences evolve – expand, that is, and become increasingly vague, as their defenders add more and more "principles" in order to suppress the endless falsifications.

Instead of all these arguments, formal and informal, why don't the theorists simply show us how to develop perfect applications using nothing but neat structures of standard constructs? This, after all, was the promise of structured programming. The theorists promote it as a practical programming concept, but all they can show us is some small, artificial examples (which, presumably, is the only type of software *they* ever wrote). They leave it to us to prove its benefits with real, fifty-thousand-line applications.

It is also worth repeating that, while this discussion is concerned with events that took place in the 1970s and 1980s, the principles of structured programming are being observed today as faithfully as they were then. Current textbooks and courses, for instance, avoid GOTO as carefully as did the earlier ones. In other words, despite their failure, these principles were incorporated into every programming theory and methodology that followed, and are now part of our programming culture. The irresistible appeal that structured programming has to the software bureaucrats, notwithstanding the popularity of more recent theories, can be understood only by recognizing its unique blend of mathematical pretences and trivial principles. Thus, simply by talking about top-down design or about GOTO, ignorant academics, programmers, and managers can feel like scientists.

[19] Ibid., p. 1310. [20] Ibid.

The Fourth Delusion

1

The fourth delusion is the absurd notion of *inconvenience*. The theorists continue to maintain that the principles of structured programming are sound, and the reason it is so difficult to follow them is just the inconvenience of the restriction to standard constructs. They note that structured programming works in simple situations – in their textbook illustrations, for instance. And they also note that the definition of structured programming guarantees its success for programs of any size: all we have to do is combine constructs and modules on higher and higher levels of nesting. So, they conclude, there is nothing wrong with the theory. If we find it increasingly difficult to follow its principles as we move to larger programs – and entirely impractical in serious business applications – we must simply disregard as many of these principles as is necessary to render the theory serviceable.

In particular, the theorists say, we don't have to *actually* restrict ourselves to standard constructs. Their justification for allowing non-standard constructs is this: We know that it is possible, in principle, to develop any application with standard constructs alone. And we know that, in principle, non-standard constructs can be reduced to standard ones through transformations. Why, then, restrict ourselves to the standard constructs? We will enjoy the benefits of structured programming even if we use the more convenient non-standard constructs.

Clearly, the theorists fail to appreciate the absurdity of this line of logic: if structured programming is promoted as a programming theory, the fact that its principles are impractical means that the theory is wrong. As was the case with the previous delusions, the theorists can deny this falsification of structured programming only by concocting an absurd explanation. The benefits of structured programming were only shown to emerge if we *actually* build our applications as hierarchies of standard constructs. If we agree to forgo its principles whenever found to be inconvenient, those benefits will vanish, and we no longer have a theory. What is left then is just some informal guidelines, not very different from what we had *before* structured programming.

The response should have been to determine *why* it is so difficult to apply these principles. We don't find it difficult to apply mechanistic principles in those fields where the mechanistic model is indeed practical – in engineering, for instance. We don't find these principles inconvenient with physical systems, or with electronic systems, so why are they inconvenient with *software* systems?

For mechanistic phenomena, the simple hierarchical structure works well

no matter how large is the system. In fact, the larger the system, the more important it is to have a mechanistic model. When building a *toy* airplane, for example, we may well find it inconvenient to follow strictly the hierarchical principle of subassemblies, and impractical to adhere strictly to the mathematical principles of aerodynamics; but we couldn't build a jumbo jet without these principles. With software systems the problem is reversed: the mechanistic principles of structured programming seem to work in simple cases, but break down in large, serious applications.

The inconvenience is due, as we know, to the non-mechanistic nature of software applications. While the hierarchical structure is a good model for mechanistic phenomena, for non-mechanistic ones it is not: the approximation it provides is rarely close enough to be useful. What we notice with poor approximations is that the model works only in simple cases, or works in some cases but not in others. Thus, the fact that structured programming fails in serious applications while appearing to work in simple situations indicates that software systems cannot be usefully represented with a simple hierarchical structure.

Structured programming fails because it attempts to reduce software applications, which consist of many interacting structures, to *one* structure. It starts by taking into account only the flow-control structures and ignoring the others. And then it goes even further and recognizes only one flow-control structure – the nesting scheme. But this reified model cannot represent accurately enough the complex structure that is the actual application.

2

Let us examine some of the non-standard constructs that were incorporated into structured programming, and their justification. The simplest example is a loop where the terminating condition is tested *after* the operation – rather than before it, as in the standard iterative construct (see figure 7-11). Although this construct can be reduced to the standard one by means of a transformation,[1] most programming languages provide statements for both. And since these statements are equally simple, and the two types of loops are equally common in applications, the theorists could hardly ask us to use one construct and avoid the other. The justification for permitting the non-standard one, thus, is the inconvenience of the transformation: "Do we need both iteration

[1] There is one transformation based on memory variables, and another based on duplicating operations. The latter, for instance, is as follows: convert the non-standard construct into a standard one that has the same operation and condition, *S1* and *C1*, and add in front of it an operation identical to *S1*.

variants? The Böhm-Jacopini theorem says 'no,' but that theorem addresses only constructibility and not convenience. For this reason, programmers like to have both variants."[2]

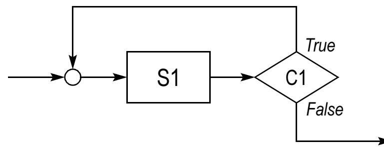

Figure 7-11

Another example is the conditional construct CASE, shown in figure 7-12. A variable, or the result of an expression, is compared with several values; a certain sequential construct (a statement, or a block of statements) is specified for each value, and only the one for which the comparison is successful is actually executed.[3]

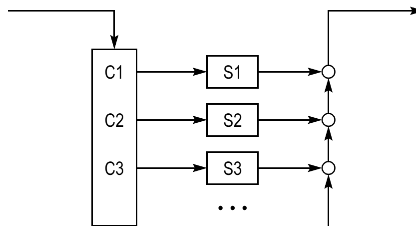

Figure 7-12

Again, we could use the standard conditional construct instead: we would specify a series of consecutive IF statements and perform the comparison with each value in turn. When only a few values are involved, this solution (or the alternative solution of nesting a second IF in the ELSE part of the previous one, and so on) is quite effective. But there are situations where we must specify many values, sometimes more than a hundred; the CASE construct is then more convenient (and also more efficient, because the compiler can optimize the comparisons).

[2] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1310.

[3] The term "CASE" refers to the different cases, or alternatives, handled by this construct. An additional alternative (the default) is usually included to allow for the case when none of the comparisons is successful.

The most embarrassing problem for structured programming, however, is the ordinary loop, one of the most common software constructs. Practically every statement in a typical application is part of a loop of some sort, its execution repeated under the control of various conditions. And it is not just inconvenient, but totally impractical, to reduce all forms of repetitive execution to the standard iterative construct. We already saw how the need to specify the terminating condition at the end of the loop led to the acceptance of a new construct. But this is only *one* of the situations that cannot be managed with the standard construct. Just as common is the situation where the terminating condition is in the middle of the loop, or where there are conditions throughout the loop, or where a condition terminates the current iteration but not the loop. Since the standard construct cannot handle these situations, we must either perform some complicated transformations, or add to programming languages a new construct for each situation, or resort to explicit jumps (GOTO statements) and create our own, specialized constructs.

The problem is even more serious in the case of *nested* loops. A loop nested within another is nearly as frequent as a single loop, and even three and four levels of nesting are common. Thus, since the situations previously mentioned can occur at all levels, without explicit jumps even more complicated transformations would be required, or dozens of constructs would have to be added to cover all possible combinations.

In the end, the software theorists adopted all three methods: they incorporated into structured programming a small number of built-in constructs (typically, a statement that lets us terminate the iterations, and a statement that lets us terminate just the current iteration, from anywhere in the loop); they recommend transformations in the other situations; and they permit the use of GOTO when the first two methods are impractical.

Nearly as difficult as the combination of loops and conditions is the combination of conditions alone. Although we could, in principle, express all combinations by nesting IF-THEN-ELSE statements, this often leads to unwieldy transformations, or too many nesting levels, or huge blocks of statements in the THEN or ELSE part. A common requirement, for instance, is to terminate prematurely the current block or the current module. As in the case of loops, we can implement this requirement through transformations, or by adding to the language new constructs, or with explicit jumps. The theorists, in the end, incorporated into structured programming such constructs as EXIT and RETURN, which terminate a module; but we must still use transformations to terminate a block, unless the transformations are especially awkward, in which case we are permitted to use GOTO.

The following quotations illustrate how the advocates of structured programming justify the adoption of non-standard constructs. The fact that we need these constructs at all proves that the theory of structured programming has failed. The constructs, though, are presented as "extensions" of the theory. They are substantially "in the spirit" of structured programming, we are told, and the only reason we need them is to make structured programming easier, or more practical, or more convenient. This explanation is illogical, of course: one cannot claim that non-standard constructs make structured programming easier, when the very essence of structured programming is the *absence* of non-standard constructs. What these experts are doing, in effect, is promoting the principles of structured programming, praising their benefits, and then showing us how to override them.

Edward Yourdon, one of the best-known experts, has this to say: "While the [three standard constructs] are sufficient to write any computer program, a number of organizations have found it practical to add some 'extensions.'"[4] And after describing some of these "extensions," Yourdon concludes: "A number of other modifications or compromises of the basic structured programming theory could be suggested and probably *will* be suggested as more programming organizations gain familiarity with the concept. As indicated, many of the compromises do not violate the black-box principle behind the original Böhm and Jacopini structures; other compromises *do* represent a violation and should be allowed only under extenuating circumstances."[5]

Note again the misrepresentation of Böhm and Jacopini's paper. What Yourdon calls the black-box principle – namely, the restriction to constructs with one entry and exit, which allows us to ignore their internal details and nest them hierarchically – is not a principle but a *consequence* of Böhm and Jacopini's theorem. (I will return to this point later.) Yourdon cites their work, but ignores the *real* principle – the restriction to nested *standard* constructs. Böhm and Jacopini did not say that we can use any construct that has one entry and exit. Yourdon invokes an exact theorem, but feels free to treat it as an informal rule: we can add to the theorem any "extensions," "modifications," and "compromises" (and, "under extenuating circumstances," violate even what is left of it), and the result continues to be called structured programming.

Here is another author who expresses the same view: "Although it is theoretically possible to write all well-formed programs using nothing more than the three basic logic structures shown here, we will find that programming is easier if we expand our repertoire a little. Extensions to the three basic logic structures are permitted as long as they retain the one-entry, one-exit

[4] Edward Yourdon, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice Hall, 1975), p. 149. [5] Ibid., p. 152.

property."[6] And here is another misrepresentation of Böhm and Jacopini's work: "The legitimate code blocks using structured programming theory are as follows: 1. SEQUENCE .... 2. IFTHENELSE .... 3. DOWHILE .... 4. DOUNTIL .... 5. CASE .... This basic set of logic structures is a practical extension of Böhm and Jacopini's original form, which proved theoretically that any problem can be broken down into small subproblems whose equivalent form can be expressed with only the first three logic types described above. However, from a practical coding viewpoint, all five logic types outlined above facilitate the process without destroying its basic intent."[7] So the authors of this book have decided that, in order to make structured programming practical, the original theorem should be interpreted as the combination of a "basic intent," which must be respected, and some other parts, which may be ignored.

Some additional examples of the same justifications: "Usually the restriction to allow only these three control constructs in a structured program is relaxed to include extensions such as the nested IF, the CASE statement, and the escape. Allowing these extensions makes the program easier to code and to maintain."[8] "To the three basic figures SEQUENCE, IF-THEN-ELSE, and DO-WHILE we have added for programming convenience the ITERATIVE-DO ... and the REPEAT-UNTIL, LOOP-EXITIF-ENDLOOP, and the SELECT-CASE figures."[9] "One should always try to solve the problem using the basic composition rules (sequencing, conditionals, repetition and recursion). If this does not give a good solution, then use of some of the special types of jumps is justified."[10] "In general, the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required."[11] The best solution, the author explains, is to redesign the module so as to avoid this requirement; alternatively, though, the structured programming restrictions may be "violated in a controlled manner,"[12] because such a violation "can be accommodated without violating the spirit of structured programming."[13]

[6] Dennie Van Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, 2nd ed. (Englewood Cliffs, NJ: Prentice Hall, 1978), p. 76.

[7] Gary L. Richardson, Charles W. Butler, and John D. Tomlinson, *A Primer on Structured Program Design* (New York: Petrocelli Books, 1980), pp. 46–47.

[8] James Martin and Carma McClure, *Structured Techniques: The Basis for CASE*, rev. ed. (Englewood Cliffs, NJ: Prentice Hall, 1988), p. 46. (Regarding "nested IF," the authors are wrong, of course: this is the nesting of standard conditional constructs, and hence not an extension but *within* the concept of structured programming.)

[9] Clement L. McGowan and John R. Kelly, *Top-Down Structured Programming Techniques* (New York: Petrocelli/Charter, 1975), p. 76.

[10] Suad Alagić and Michael A. Arbib, *The Design of Well-Structured and Correct Programs* (New York: Springer-Verlag, 1978), p. 226.

[11] Roger S. Pressman, *Software Engineering: A Practitioner's Approach* (New York: McGraw-Hill, 1982), p. 246. [12] Ibid. [13] Ibid., p. 247.

So, with all useful constructs again permitted, what was left of the theory of structured programming was just some informal bits of advice on disciplined programming, supplemented with the exhortation to use standard constructs and avoid GOTO "as much as possible."

In general, whenever a programming language included some new, useful constructs, these constructs were enthusiastically adopted, simply because they obviated the need for GOTO. The jumps implicit in these constructs could be easily implemented with GOTO; but this alternative was considered bad programming, even as the constructs themselves were praised as good programming. Many theorists went so far as to describe these constructs as modern language enhancements that help us adhere to the principles of structured programming, when their role, clearly, is to help us *override* those principles. (The only jumps allowed in structured programming, we recall, are those implicit in the standard conditional and iterative constructs.) The absence of the phrase "go to" was enough to turn their jumps from bad to good programming, and the resulting programs from unstructured to structured.

Thus, despite the insistence that structured programming is more than just GOTO-less programming, this concern – contriving transformations or new constructs in order to avoid GOTO, or debating whether GOTO is permissible in a particular situation – became in fact the main preoccupation of both the academics and the practitioners.

Their reaction to what is in reality a blatant falsification of structured programming – the need for explicit jumps – clearly reveals the theorists' ignorance and dishonesty. Not only were they naive enough to believe that we can program without jumps, but they refused to accept the evidence when this was tried in actual applications. Even more than the difficulties encountered under the first three delusions, the apparent inconvenience of the standard constructs should have motivated them to question the validity of structured programming. Instead, they suppressed this falsification by reinstating the very feature that the original theory had excluded (the use of non-standard constructs), and on the exclusion of which its promises were based. The original dream, thus, was now impossible even if we forget that the previous delusions had already negated it. The theorists, nevertheless, continued to advertise structured programming with the same promises.

3

Returning to the previous quotations, what is striking is the lack of explanation. The theorists mention rather casually that the reason we are permitted to violate the principles of structured programming is the inconvenience of the

standard constructs. They are oblivious to the absurdity of this justification; it doesn't occur to them that this inconvenience is an important clue, that we ought to study it rather than avoid it. Incredibly, they are convinced that if the theory of structured programming does not work, we can make it work simply by disregarding some of its principles. Specifically, those important transformations we discussed earlier need to be performed now only when convenient. We will derive the same benefits, the theorists say, whether we *actually* reduce the application to standard constructs, or simply know that *in principle* we could do it.

When we do find an explanation, it is the black-box principle that is invoked. This principle, we are told now, is the only important one: since any flow-control construct with one entry and one exit will function, for all practical purposes, just like a standard construct, there is no need to restrict ourselves to the standard constructs. We will enjoy the benefits of structured programming with *any* constructs that have one entry and one exit.[14]

We saw the meaning of a software "black box" in figures 7-4 to 7-8 (pp. 513, 524–527). A program is deemed structured if its flow diagram can be represented with nested boxes. Each box encloses a standard construct and can be treated as a separate software element. And, since the standard constructs have only one entry and exit, from the perspective of the flow of execution each box is in effect a sequential construct. Moreover, when studying a certain box from higher nesting levels, its internal details are immaterial; each element, therefore, can be programmed independently of the others.

This principle applies at all nesting levels, so the entire application can be developed simply by creating standard constructs: one construct at a time, one level at a time. The restriction to software elements with one entry and exit guarantees that, regardless of the number of nesting levels, the application can be represented as a perfect hierarchical structure. The ultimate benefit of this restriction, then, is that we can develop and prove our applications with the formal methods of mathematics: if each element is correct, and if the relations between levels (reflected in the single entries and exits) are correct, the application as a whole is bound to be correct.

[14] The term *black box* refers to a device, real or theoretical, that consists of an input, an output, and an internal process that is unknown, or is immaterial in the current context. All we can see, and all we need to know, is how the output changes as a function of the input. Strictly speaking, then, since software flow-control constructs do not have input and output, the theorists are wrong to describe them as black boxes: the entry and exit points seen in the diagram do not depict an input value converted by a process into an output value, but rather a construct's place relative to the other constructs in the sequence of execution. Only if taken in its most general and informal sense, simply as a device whose internal operation is immaterial, can the concept of a black box be used at all with software flow-control constructs.

Suddenly, then, it seems that the principles of structured programming can be relaxed: from a restriction to the standard constructs, to a restriction to any constructs that have one entry and exit. Incredibly, structured programming can be expanded from three to an infinity of constructs without having to give up any of its original benefits. Still, as we just saw, there is no obvious fallacy in this expansion; those benefits appear indeed attainable through any constructs that have one entry and exit. But this sudden freedom, this ease of expanding the theory, is precisely what should have *worried* the advocates of structured programming, what should have prompted them to doubt its soundness. Instead, they interpreted this apparent freedom as a *good* thing, as evidence of its *power*: we can now combine in one theory, they concluded, the strictness of a mathematical concept and the convenience needed in a practical programming methodology.

This freedom is an illusion, of course. It appears logical only if we study the new claim in isolation, only if we forget that the principles of structured programming had been refuted *before* the theorists discovered the inconvenience of the standard constructs. Thus, to recognize the fallacies inherent in the new delusion we must bear in mind the previous ones.

We note, first, that what the theorists call the black-box principle is not a principle at all; it is a corollary, a *consequence* of the principles of structured programming. Since the standard constructs have only one entry and exit, if we restrict ourselves to standard constructs the flow diagram will display this characteristic at every nesting level. The main principle is the restriction to the standard constructs. The theorists take what is one of the *results* of this principle (constructs with only one entry and exit) and make *it* the main principle. Then, they substitute for what is the *actual* restriction (the three standard constructs) a new, vague restriction.

The new restriction merely states that we should use non-standard constructs "as little as possible." The number of constructs varies from expert to expert: some permit only three or four (the minimum needed to alleviate the inconvenience of the standard ones), others permit a dozen, and some go so far as permitting *any* constructs with one entry and exit. Whether permitting few or many, though, this restriction is specious; it is not an exact principle, as was the restriction to standard constructs. In reality, if permitted to use a construct just because it has one entry and exit, it matters little whether we use one or a hundred: the issue now is, at best, one of programming style. But by counting, studying, and debating the new constructs, and by describing them as extensions of structured programming, the experts can delude themselves that they still have a theory.

So the experts embraced the black-box principle because it allowed them to bypass the rigours of structured programming. For, once we annul the *real* principle (the restriction to standard constructs), *any* flow-control construct can be said to have only one entry and exit. Take the CASE construct, for instance – one of the first to be permitted in structured programming (see figure 7-12, p. 568). Its flow diagram contains one component with several exits; but, if we draw a rectangular box around the whole diagram, *that box* will have only one entry and exit.

The same trick, obviously, can be performed with any piece of software: first, we create the most effective or convenient construct, which will likely violate the principles of structured programming by containing parts with more than one entry or exit; then, we draw a box around the whole thing and declare it an extension of structured programming, because now it has only one entry and exit. It is entirely up to us to decide at what point this structuring method becomes silly.

All non-standard constructs are based, ultimately, on this trick. And every expert was convinced that structured programming could be saved by extending it in this fashion. To pick just one case, Jensen allows six non-standard constructs in *his* definition of structured programming.[15] One of these constructs, for example, is called POSIT, and its purpose is to replace a particular combination of conditional statements, which involves an unusual jump.[16] Jensen shows us how much simpler his POSIT is than using standard constructs and transformations, and he considers this to be sufficient justification for including it in structured programming. (The jump, of course, is even easier to implement with GOTO; the sole reason for his new construct is to avoid the GOTO.) But Jensen may well be the only expert who deems this particular instance of explicit jumps important enough to become an official construct. Some experts would recommend transformations, others would permit the use of GOTO, and others yet would suggest more than six non-standard constructs. Still, no one saw how absurd these ideas were. Clearly, if each expert is free to interpret the theory of structured programming in his own way, there is no limit to the number of variants that can be invented. Is structured programming, then, this open-ended collection of variants?

Significantly, the experts did not replace the original principle with a more flexible, but equally precise, one. The original principle was strict and simple: in only two situations – *within* the standard conditional and iterative constructs – can the flow diagram include a component with more than one entry or exit.

[15] Randall W. Jensen, "Structured Programming," in *Software Engineering*, eds. Randall W. Jensen and Charles C. Tonies (Englewood Cliffs, NJ: Prentice Hall, 1979), p. 238.

[16] Ibid., p. 250.

By adopting the black-box principle, the experts increased the number of permitted situations from two to an infinity.

And with this permission, structured programming finally became a practical concept: whenever we want to avoid an awkward transformation, we can simply use a language-specific construct, or create a specialized flow-control construct, and justify this by claiming that it is a logical extension of structured programming.

We recognize in the black-box principle the pseudoscientific stratagem of turning falsifications into features (see "Popper's Principles of Demarcation" in chapter 3): the theory is expanded by permitting those situations that would otherwise falsify it, and calling them new features. Thus, the black-box principle permits almost any constructs, while the principle of standard constructs permitted only three. As a result, constructs whose usefulness originally *falsified* the theory are now *features* of the theory. This saves the idea of structured programming from refutation, but at the price of making it unfalsifiable, and hence worthless.

4

It is even easier to understand the fourth delusion when we represent applications as systems of interacting software structures. Recall our discussion under the second delusion. The purpose of the standard conditional and iterative constructs is to provide alternatives to the flow of execution defined by the nesting scheme. Each alternative endows the application with a unique flow-control structure. And, since the actual flow of execution is affected by all the flow-control constructs in the application, it is in reality a system comprising all the individual flow-control structures (see pp. 541–545).

So the two standard constructs already make the flow of execution a complex structure. When we study the application from the perspective of the nesting scheme alone – when we study the flow diagram, for instance – what we see is elements neatly related through one hierarchical structure. But if some of these elements are conditional or iterative constructs, the actual flow of execution will comprise *many* structures. Each one of these structures differs from the nesting scheme only slightly, in one element of one particular flow-control construct.

As far as their effect on the flow of execution is concerned, then, there is indeed no difference between the standard and the non-standard flow-control constructs. Just like the standard ones, any flow-control construct provides, by means of jumps, alternatives to the flow of execution defined by the nesting scheme: each possible jump creates a different flow-control attribute, and

hence a flow-control structure. In the standard constructs, the jumps are implicit. In non-standard constructs, the jumps can be both implicit (when we use built-in, language-specific constructs) and explicit (when we create our own constructs with GOTO).

Thus, in a certain sense, the software theorists are right to claim that any construct with one entry and one exit is a valid structured programming extension. Since constructs possessing this quality can be elements in a hierarchical structure, a nesting scheme that includes non-standard constructs remains a correct hierarchy. There is no fallacy in the claim that, within the nesting scheme, non-standard constructs function just like the standard ones. The fallacy, rather, is in the belief that the nesting scheme alone represents the flow of execution.

The reason it seems that we can add an infinity of extensions and still enjoy the benefits promised by structured programming is that those benefits were *already* lost, since the first delusion. If the application consists of interacting flow-control structures even when restricted to the standard constructs, this means that it cannot be represented mathematically in any case. So it is true that there is nothing to lose by allowing non-standard constructs. The extensions are logical, just as the theorists say, but for a different reason: they are logical, not because structured programming is valid both with and without them, but because it is *invalid* both with and without them.

The dream of structured programming was always to establish a direct, one-to-one correspondence between the static flow diagram and the actual flow of execution (see pp. 532–533). Since flow diagrams can be drawn as hierarchical structures, and hence represented with the exact tools of mathematics, such a correspondence means that the same model that describes mathematically the flow diagram would also describe the flow of execution, and therefore the behaviour of the *running* application.

So the restriction to one entry and exit is important to the theorists because it guarantees that all the elements in the application are related through a simple hierarchical structure. And indeed, this restriction makes the *flow diagram* a hierarchical structure. The theorists then mistakenly conclude that the flow of execution, formed as it is from the same elements and relations, will mirror at run time the flow diagram; so it too will be a hierarchical structure. The flow of execution, though, is the combination of all the flow-control structures in the application. We could perhaps represent mathematically *each one* of those structures. But even if we accomplished this, we still could not represent mathematically the complex structure that is their totality, and which

is the only true model of the application's flow of execution. And we must not forget the other *types* of structures – structures based on shared data or operations, and on business or software practices – all interacting with one another and with the flow-control structures, and therefore affecting the application's performance.

It is only when we recognize the great complexity of software that we can appreciate how ignorant the software experts are, and how naive is their belief that the nesting scheme represents the flow of execution. As I pointed out earlier, the *theory* of structured programming was refuted in the first delusion, when this belief was born. The *movement* known as structured programming was merely the pursuit of the various delusions that followed. It was, thus, a fraud: a series of dishonest and futile attempts to defend an invalid mechanistic theory.

5

Our concept of software structures can also help us to understand why the restriction to standard constructs is, indeed, inconvenient. The theorists, we saw, make no attempt to explain the reason for this inconvenience. They correctly note that non-standard constructs are more convenient, but they don't feel there is a need to understand this phenomenon. They invoke their convenience to justify their use, but, ultimately, it is precisely this phenomenon – the difference in convenience between the standard and the non-standard constructs – that must be explained.

The few theorists who actually attempt to explain this phenomenon seem to conclude that, since non-standard constructs can be reduced to standard ones, they function as software subassemblies: non-standard constructs consist of combinations of standard ones in the same way that subassemblies consist of simpler parts in a *physical* structure. So they are more convenient in building software structures for the same reason it is more convenient to start with subassemblies than with individual parts when building *physical* structures. In short, non-standard constructs are believed to be at a higher level of abstraction than the standard ones.[17] Let us analyze this fallacy.

We saw, under the second delusion, that the theorists confuse the operations performed by the three standard constructs with the operations that define a

[17] Knuth, for example, expresses this mistaken view when he says that the various flow-control constructs represent in effect different levels of abstraction in a given programming language, and that we can resolve the inconvenience of the standard constructs simply by inventing some new, higher-level constructs. Donald E. Knuth, "Structured Programming with *go to* Statements," in *Computing Surveys* 6, no. 4 (©1974 ACM, Inc.): 295–296.

hierarchical structure; they confuse these constructs, thus, with the operations that create the elements of one level from those of the lower level. Now it seems that this confusion extends to the non-standard constructs.

The only operations that define software structures, we saw, are those performed by the *sequential* constructs, and those that invoke modules and subroutines; in other words, the kind of operations that combine software elements into larger elements, one level at a time (see pp. 541–542). This is how the application's nesting scheme is formed, and we can create hierarchical software structures of any size with sequential constructs alone. The conditional and iterative constructs do not perform the same kind of operation; they do not combine elements into larger, higher-level elements. Their role, in fact, is to *override* the operations performed by the sequential constructs, by providing alternatives to the nesting scheme. And they do it by endowing the software elements with flow-control attributes (in the form of jumps): each attribute gives rise to an additional flow-control structure.

The non-standard constructs, too, endow elements with flow-control attributes; so their role, too, is to create additional flow-control structures. The two kinds of constructs fulfil a similar function, therefore, and their relationship is not one of high to low levels.

So the theorists are correct when noting that the non-standard constructs are more convenient, but they are wrong about the reason: the convenience is *not* due to starting from higher levels of abstraction. Let us try to find the real explanation.

Whether we employ non-standard constructs or restrict ourselves to the standard ones, the application will have multiple, interacting flow-control structures. In either case, then, it is our mind that must solve the most difficult programming problem – dealing with the interactions between structures. Thus, even when following the principles of structured programming, our success depends largely on our skills and experience, not on the soundness of these principles.

The defenders of structured programming delude themselves when maintaining that a perfectly structured program can be represented with an exact, mechanistic model. The static flow-control structure – the nesting scheme depicted by the flow diagram – has perhaps a mathematical representation. But this cannot help us, since we must ensure that the dynamic, complex flow-control structure is correct; we must ensure, in other words, that all the individual flow-control structures, *and* their interactions, are correct. So the most difficult aspect of programming is precisely that aspect which *cannot* be

represented mathematically, and which lies therefore beyond the scope of structured programming (or any other mechanistic theory).

The goal of all mechanistic software theories is to eliminate our dependence on the non-mechanistic capabilities of the mind, on such imprecise qualities as talent, skill, and experience. And the only way to eliminate this dependence is by treating software applications as simple structures, or as systems of separable structures. This is an illogical quest, however, because software structures *must* interact. Our affairs consist of interacting processes and events; so, if we want our software to mirror our affairs accurately, the software entities that make up applications must be related in several ways at the same time. The only way for software entities to have multiple relations is by sharing more than one attribute. And, since each attribute gives rise to a different structure, these entities will belong to several structures at the same time, causing them to interact.

Even within that one aspect of the application that is the flow of execution, we find the need for multiple, interacting structures: to represent our affairs, the application's elements must possess and share several *flow-control* attributes. Each flow-control attribute serves to relate a number of elements, from the perspective of the flow of execution, in a unique way. The nesting scheme is, in effect, a flow-control attribute shared by all the elements in the application. And we create the other flow-control attributes by introducing jumps in the flow of execution: each possible jump, whether explicit or implicit, gives rise to a unique flow-control attribute, and hence a different flow-control structure.

The standard conditional and iterative constructs, useful as they are, can provide only two types of jumps; so they can create only *some* of the flow-control relations between the application's elements, only *some* of the attributes. In order to mirror in software our affairs, we need more types of relations, and hence more types of flow-control attributes. We need, in other words, more types of jumps than those provided by the standard constructs. We can provide the additional relations with our own, explicit jumps, or with the implicit jumps found in some language-specific constructs. Or, if we want to avoid jumps altogether, as structured programming recommends, we can resort to transformations: we provide the additional relations then, not through flow-control attributes, but through attributes based on shared data or shared operations.

And herein lies the explanation for the inconvenience of the standard constructs and the transformations. We usually need *several* relations of other types to replace *one* flow-control relation. That is, instead of one flow-control attribute, our elements must have several attributes of other types in order to implement a given requirement. More precisely, to replace *one* flow-

control attribute deriving from non-standard constructs, we need *one or more* flow-control attributes deriving from standard constructs, plus *one or more* attributes deriving from shared data or shared operations. Thus, since each attribute gives rise to a structure, we end up with more structures, and more interactions. While the additional complexity may be insignificant with only a few elements and attributes, as in a small piece of software, it becomes prohibitive in a serious application, because the number of interactions grows exponentially relative to the number of structures.

To make matters worse, the substitute relations are *less intuitive* than the flow-control ones. They do not reflect the actual relations – those relations we observe in our activities, and which we wanted to implement in software. The substitute relations are unnatural, in that they exist only between software elements, and are necessary only in order to satisfy an illogical principle.

Our programming languages, as a matter of fact, do permit us to implement the actual relations simply and naturally, but only if we use both standard and non-standard constructs. It is the restriction to standard constructs that creates artificial relations, and makes the application larger and more complicated.

Let us analyze a specific case: the requirement to exit an iterative construct depending on a condition encountered in the middle of the loop. The simplest way to implement this requirement is by jumping out of the loop with a GOTO. This jump, moreover, simulates naturally in software what we do in our everyday activities when we want to end a repetitive act. If, however, we want to avoid the explicit jump, we must use a memory variable as switch (this transformation is similar to the one shown in figure 7-8, p. 527). Instead of simply terminating the loop, the condition only sets the switch; the operations to the end of the loop are placed in the other branch of this condition, so they are bypassed; then, the switch is checked in the main condition, so the loop will end before the next iteration. This method is more complicated than a GOTO, but it is the one recommended by the advocates of structured programming.

In our everyday activities, we terminate a repetitive act simply by ending the repetition; we don't make a note about ending the repetition, go back to the beginning of the act as if we intended to repeat it, pretend to discover the note we made a moment earlier, and only then decide to end the repetition. A person who regularly behaved in this manner would be considered stupid. Yet, we are asked to display this behaviour in our *programming* activities. No wonder we find the transformations inconvenient – unnatural and impractical.

We need *thousands* of such transformations in a serious application. Still, the real difficulty is not the large number of *individual* transformations, but their *interactions*. We saw that the application remains a system of interacting structures, and that the transformations add even more structures. Thus, in

addition to the original interactions, we must now deal with the interactions between the new structures, and between these and the original ones. So, when multiplying, transformations that individually are merely inconvenient become a major part of the application's logic. In the end, it is the transformations, rather than the actual requirements, that govern the application's design.

Since it is quite easy to implement *isolated* transformations, we can justify the additional effort by calling this activity "software engineering." Software engineering, though, becomes increasingly awkward as our applications grow in size and detail. So what we perceive then as a new problem – the impracticality of the transformations – is in reality the same phenomenon as in simple situations, where structured programming appears to work. The only difference is that we can disregard the inconvenience when slight, but must face it when it becomes a handicap.

The inconvenience of the restriction to standard constructs indicates that our mental effort, even when developing "structured" software, entails more than just following mechanistic principles. It indicates that we are also using our *non-mechanistic* capabilities. It indicates, therefore, that we are dealing with systems of interacting structures; for, were applications mechanistic in nature, the restriction to standard constructs would be *increasingly helpful* as they grew in size.

The phenomenon of inconvenience proves, then, that it is not the mechanistic principles of structured programming but *our mind* – our skills and experience – that we are relying on. This phenomenon proves, in other words, that the theory of structured programming is invalid. So, by misinterpreting the inconvenience, the theorists missed the fourth opportunity to recognize the fallaciousness of structured programming.

In conclusion, non-standard constructs are more convenient because they result in fewer structures and interactions for implementing the same requirements. We need a certain number of flow-control attributes in order to mirror in software a given combination of processes and events; and we end up with more attributes, and hence more structures, when replacing the flow-control attributes with attributes of other types. There is a limit to our capacity to process interacting structures in our mind, and we reach this limit much sooner when following the principles of structured programming.

The best programming method, needless to say, is the one that results in the fewest interactions. The promise of structured programming is to eliminate the interactions altogether, and thereby obviate the need for non-mechanistic thinking. But now we see that the opposite is taking place: programmers need

even greater non-mechanistic capabilities – an even greater capacity to process complex structures – with structured programming than without it.

A simple, ten-line piece of software will be changed by structured programming into a slightly more involved piece of software. Thus, if we believe in some ultimate benefits, we will gladly accept the small increase in complexity. But this self-deception cannot help us in real-world situations. Because the complexity induced by structured programming grows exponentially, a serious application will become, not *slightly* more, but *much* more, involved. Creating and maintaining such an application is not just inconvenient but totally impractical. Moreover, because it is still a system of interacting structures, the application will still be impossible to represent mathematically. There are no ultimate benefits, and no one ever developed a serious application while rigorously adhering to the principles of structured programming.

In the end, structured programming turned the activities of programmers into a kind of game: searching for ways to avoid GOTO. The responsibility of programmers shifted, from creating useful applications and improving their skills, to merely conforming to a certain dogma. They were pleased with their success in performing transformations on isolated pieces of software, while reverting to non-standard constructs whenever the transformations were inconvenient. And they believed that this senseless programming style was structured programming; after all, even the experts were recommending it. Thus, just as the experts were deluding themselves that structured programming was a valid theory, the programmers could delude themselves that what they were practising was structured programming.

6

Let us examine, lastly, that aspect of the fourth delusion that is the continued belief in an exact, mathematical representation of software applications, when in fact no one ever managed to represent mathematically anything but small and isolated pieces of software. When a phenomenon is mechanistic, mathematics works just as well for large systems as it does for small ones. Thus, the fact that a mathematical theory that works for small pieces of software becomes increasingly impractical as software grows in size should have convinced the theorists that software systems give rise to *non-mechanistic* phenomena.

Take a trivial case: an IF statement where, for instance, a memory variable is either incremented or decremented depending on a condition. Even this simple construct, with just one condition and two elements, is related to the rest of the application through more than one structure; so it is part of a complex system. In its static representation, there are at least two logical

connections between the two elements, and between these elements and the rest of the application: the flow-control structure, the structure based on the memory variable, and further structures if the condition itself entails variables, subroutines, etc. And in the *dynamic* representation there are at least three logical connections, because the condition's branches generate an additional structure (we studied these structures under the second delusion). We are dealing with a complex system; but because it is such a small system, we can identify all the structures and even some of the interactions. In real applications, however, we must deal with *thousands* of structures, most of them interacting with one another; and, while *in principle* we can still study these systems, we cannot *actually* do it. Many mechanistic delusions, we saw earlier, spring from the failure to appreciate this difference between simple and real-world situations (see p. 555).

Thus, even at this advanced stage, even after all the falsifications, many theorists remained convinced that structured programming allows us to develop and prove applications mathematically. The success of this idea in simple situations gave them hope that, with further work, we would be able to represent mathematically increasingly large pieces of software. Entire books have been written with nothing more solid than this belief as their foundation. In one example after another, we are shown how to prove the validity of a piece of software by reducing it to simpler entities, just as we do with mathematical problems. But these demonstrations are worthless, because the theorists recognize only one structure – the static nesting scheme, typically – and ignore the other relations that exist between the same software elements; they only prove, therefore, the validity of *one aspect* of that piece of software. So, even when correct, these demonstrations remain abstract studies and have no practical value. Whether empirical (using software transformations) or analytical (using mathematical logic), they rely on the fact that, in simple situations, those structures and interactions that we can identify constitute a major portion of the system. Thus, although their study only *approximates* the complex software phenomenon, for small pieces of software the approximation may well be close enough to be useful.

The theorists take the success of these demonstrations as evidence that it is possible to represent software mathematically: if the method works in simple cases, they say, the principles of reductionism and atomism guarantee its success for larger and larger pieces of software, and eventually entire applications. But the number of structures and interactions in real-world situations grows very quickly, and any method that relies on identifying and studying them separately is bound to fail. No one ever managed to prove, either empirically or analytically, the validity of a significant piece of software, let alone a whole application. The software mechanists remain convinced

that mathematical programming is a practical concept, when, like the other mechanistic delusions, it is only valid in principle.[18]

In principle, then, it is indeed possible to develop and prove applications mathematically – just as it is possible, in principle, to predict future events through Laplacean determinism, or to explain human acts with the theories of behaviourism, or to depict social phenomena with the theories of structuralism, or to represent languages with Chomskyan linguistics. But *actually* using a mathematical programming theory – just like using those other theories – is *inconvenient*.

The mathematical representation of software, thus, is treated by the theorists just like the restriction to standard constructs: they show that it works in simple, isolated cases; they believe that the principles of structured programming assure its success in *actual* applications; and they refuse to see its failure in actual applications as evidence that structured programming does *not* work.

So the conclusion we must draw from the fourth delusion is that structured programming *never* works, not even in those situations where we do *not* find it inconvenient. Even requirements simple enough to program with standard constructs alone, and simple enough to represent mathematically, give rise to multiple, interacting structures. But because in these situations we can identify the structures and the interactions, we can delude ourselves that we are dealing with a mechanistic phenomenon. The fourth delusion, then, can also be described as the belief that structured programming is inconvenient only in certain situations, while in reality the inconvenience is *always* present. We just don't notice it, or don't mind it, for simple requirements.

Simple requirements, in fact, can be programmed with mechanistic knowledge alone, if we follow a method that takes into account the most important structures and interactions. Thus, we often hear the remark that inexperienced programmers find it easier than experienced ones to adapt to the rigours of structured programming. As usual, the theorists misinterpret this fact. Experienced programmers dislike structured programming, the theorists say, because they are accustomed to the old-fashioned, undisciplined style of programming. Actually, experienced programmers dislike structured programming because they already possess superior, *non-mechanistic* knowledge, which

[18] It must be noted that this fallacy affected, not just the specific theory known as structured programming, but *all* theories based on structures of nested constructs. As example, here is a methodology that claims to validate mathematically entire applications, and an actual development system based on it: James Martin, *System Design from Provably Correct Constructs* (Englewood Cliffs, NJ: Prentice Hall, 1985).

exceeds the benefits of a mechanistic theory. Inexperienced programmers possess no knowledge at all, or a modicum of *mechanistic* knowledge; so they like structured programming because they indeed accomplish more with it than without it. But substituting rules and methods for skills and experience can benefit them only in *simple* situations. Ultimately, with serious applications, programmers possessing non-mechanistic knowledge easily outperform those who attempt to practise strict structured programming.

This, incidentally, explains also why CASE (the promise of automatic software generation, see pp. 465–469) works in simple situations while failing for real-world applications. Only by following precise rules and methods – that is, by treating software as a mechanistic phenomenon – can a device convert requirements into applications. (Software devices, thus, display the same type of behaviour as inexperienced programmers.) In simple situations, the device can account for most interactions; but this method of programming breaks down when tried with serious applications, where the number of interactions is practically infinite.

Mathematics can represent large systems as easily as it can small ones. This is why phenomena that are truly mechanistic can be represented mathematically no matter how many elements, levels, and relations are involved. But, because software phenomena are *not* mechanistic, mechanistic theories only *appear* to represent software systems mathematically. When practical at all, they work merely by accounting for the individual interactions – not through general principles, like the truly useful mathematical theories.[19]

The *GOTO* Delusion

1

There is no better way to conclude our discussion of the structured programming delusions than with an analysis of the GOTO delusion – the prohibition and the debate.

We have already encountered the GOTO delusion: under the third delusion, we saw that the reason for transformations was simply to avoid GOTOs; and under the fourth delusion, we saw that the reason for introducing nonstandard constructs into structured programming was, again, to avoid GOTOs.

[19] A related fallacy is the idea of *software metrics* – the attempt to measure the complexity of an application by counting and weighing in various ways the conditions, iterations, subroutines, etc., that make it up. Like the mathematical fallacy, these measurements reflect individual aspects of the application, not their interactions; so the result is a poor approximation of the *actual* complexity.

The GOTO delusion, however, deserves a closer analysis. The most famous problem in the history of programming, and unresolved to this day, this delusion provides a vivid demonstration of the ignorance and dishonesty of the software theorists. They turned what is the most blatant falsification of structured programming – the need for explicit jumps in the flow of execution – into its most important feature: new flow-control constructs that hide the jumps within them. The sole purpose of these constructs is to perform jumps without using GOTO statements. Thus, while purposely designed to help programmers *override* the principles of structured programming, these constructs were described as language enhancements that *facilitate* structured programming.

Turning falsifications into features is how fallacious theories are saved from refutation (see "Popper's Principles of Demarcation" in chapter 3). The GOTO delusion alone, therefore, ignoring all the others, is enough to characterize structured programming as a pseudoscience.

Clearly, if it was proved mathematically that structured programming needs no GOTOs, the very fact that a debate is taking place indicates that structured programming has failed as a practical programming concept. In the end, the GOTO delusion is nothing but the denial of this reality, a way for the theorists and the practitioners to cling to the idea of structured programming years and decades after its failure.

It is difficult for a lay person to appreciate the morbid obsession that was structured programming, and its impact on our programming practices. Consider, first, the direct consequence: programmers were more preoccupied with the "principles" of structured programming – with trivial concepts like top-down design and avoiding GOTO – than with the actual applications they were supposed to develop, and with improving their skills. A true mass madness possessed the programming community in the 1970s – a madness which the rest of society was unaware of. We can recall this madness today by studying the thousands of books and papers published during that period, something well worth doing if we want to understand the origins of our software bureaucracy. All universities, all software experts, all computer publications, all institutes and associations, and management in all major corporations were praising and promoting structured programming – even as its claims and promises were being falsified in a million instances every day, and the only evidence of usefulness consisted of a few anecdotal and distorted "success stories."

The worst consequence of structured programming, though, is not what happened in the 1970s, but what has happened *since* then. For, the incompetence and irresponsibility engendered by this worthless theory have remained the distinguishing characteristic of our software culture. As programmers and

managers learned nothing from the failure of structured programming, they accepted with the same enthusiasm the following theories, which suffer in fact from the same fallacies.

2

Recall what is the GOTO problem. We need GOTO statements in order to implement explicit jumps in the flow of execution, and we need explicit jumps in order to create non-standard flow-control constructs. But explicit jumps and non-standard constructs are forbidden under structured programming. If we restrict ourselves to the three standard constructs, the theorists said at first, we will need no explicit jumps, and hence no GOTOs. We may have to subject our requirements to some awkward transformations, but the benefits of this restriction are so great that the effort is worthwhile.

The theorists started, thus, by attempting to replace the application's flow-control structures with structures based on shared data or shared operations; in other words, to replace the unwanted flow-control relations between elements with relations of other types. Then, they admitted that it is impractical to develop applications in this fashion, and rescued the idea of structured programming by permitting the use of *built-in* non-standard constructs; that is, constructs already present in a particular programming language. These constructs, specifically prohibited previously, were described now as *extensions* of the original theory, as *features* of structured programming. Only the use of GOTO – that is, creating our own constructs – continued to be prohibited.

The original goal of structured programming had been to eliminate *all* jumps, and thereby restrict the flow-control relations between elements to those defined by a single hierarchical structure. This is what the restriction to a nesting scheme of standard flow-control constructs was thought to accomplish – mistakenly, as we saw under the second delusion, because the *implicit* jumps present in these constructs already create multiple flow-control structures. Apart from this fallacy, though, it is illogical to permit *built-in* non-standard constructs while prohibiting *our own* constructs. For, just as there is no real difference between standard constructs and non-standard ones, there is no real difference between built-in non-standard constructs and those we create ourselves. All these constructs fulfil, in the end, the same function: they create additional flow-control structures in order to provide alternatives to the flow of execution established by the nesting scheme. Thus, all that the built-in constructs accomplish is to relate elements through implicit rather than explicit jumps. So they render the GOTOs unnecessary, not by *eliminating* the unwanted jumps, but by turning the *explicit* unwanted jumps into *implicit*

unwanted ones. The unwanted relations between elements, therefore, and the multiple flow-control structures, remain.

The goal of structured programming, thus, was now reversed: from the restriction to standard constructs – the absence of GOTO being then merely a consequence – to searching for ways to replace GOTOs with implicit jumps; in other words, from *avoiding* non-standard constructs, to seeking and praising them. More and more constructs were introduced, but everyone agreed in the end that it is impractical to provide GOTO substitutes for all conceivable situations. So GOTO itself was eventually reinstated, with the severe admonition to use it "only when absolutely necessary." The theory of structured programming was now, in effect, defunct. Incredibly, though, it was precisely at this point that it generated the greatest enthusiasm and was seen as a programming revolution. The reason, obviously, is that it was only at this point – only after its fundamental principles were annulled – that it could be used at all in practical situations.

The GOTO delusion, thus, is the belief that the preoccupation with GOTO is an essential part of a structured programming project. In reality, the idea of structured programming had been refuted, and the use or avoidance of GOTO is just a matter of programming style. What had started as a precise, mathematical theory was now an endless series of arguments on whether GOTO or a transformation or a built-in construct is the best method in one situation or another. And, while engaged in these childish arguments, the theorists and the practitioners called their preoccupation structured programming, and defended it on the strength of the original, mathematical theory.[1]

Let us see first some examples of the GOTO prohibition – that part of the debate which claims, without any reservation, that GOTO leads to bad programming, and that structured programming means avoiding GOTO: "The primary *technique* of structured programming is the elimination of the GOTO statement

[1] For example, as late as 1986, and despite the blatant falsifications, the theorists were discussing structured programming just as they had been discussing it in the early 1970s: it allows us to prove mathematically the correctness of applications, write programs that work perfectly the first time, and so on. Then, as evidence, they mention a couple of "success stories" (using, thus, the type of argument used to advertise weight-loss gadgets on television). See Harlan D. Mills, "Structured Programming: Retrospect and Prospect," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE), pp. 286–287 – paper originally published in *IEEE Software* 3, no. 6 (1986): 58–66. See also Harlan D. Mills, Michael Dyer, and Richard C. Linger, "Cleanroom Software Engineering," in *Milestones*, eds. Oman and Lewis, pp. 217–218 – paper originally published in *IEEE Software* 4, no. 5 (1987): 19–24.

and its replacement with a number of other, well-structured branching and control statements."[2] "The freedom offered by the GOTO statement has been recognized as not in keeping with the idea of structures in control flow. For this reason we will *never* use it."[3] "If a programmer actively endeavours to program without the use of GOTO statements, he or she is less likely to make programming errors."[4] "By eliminating *all* GOTO statements, we can do even better, as we shall see."[5] "In order to obtain a simple structure for each segment of the program, GOTO statements should be avoided."[6] "Using the techniques of structured programming, the GOTO or branch statement is avoided entirely."[7]

And the *Encyclopedia of Computer Science* offers us the following (wrong and silly) analogy as an explanation for the reason why we must avoid GOTO: it makes programs hard to read, just like those articles on the front page of a newspaper that are continued (with a sort of "go to") to another page. Then the editors conclude: "At least some magazines are more considerate, however, and always finish one thought (article) before beginning another. Why can't programmers? Their ability to do so is at the heart of structured programming."[8]

It is not difficult to understand why the subject of GOTO became such an important part of the structured programming movement. After all the falsifications, what was left of structured programming was just a handful of trivial concepts: top-down design, hierarchical structures of software elements, constructs with only one entry and exit, etc. These concepts were then supplemented with a few other, even less important ones: indenting the nested elements in the program's listing, inserting comments to explain the program's logic, restricting modules to a hundred lines, etc. The theorists call these concepts "principles," but these simple ideas are hardly the basis of a programming theory. Some are perhaps a *consequence* of the original structured programming principles, but they are not principles themselves.

[2] Edward Yourdon, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice Hall, 1975), p. 145.

[3] J. N. P. Hume and R. C. Holt, *Structured Programming Using PL/1*, 2nd ed. (Reston, VA: Reston, 1982), p. 82.

[4] Ian Sommerville, *Software Engineering*, 3rd ed. (Reading, MA: Addison-Wesley, 1989), p. 32.

[5] Gerald M. Weinberg et al., *High Level COBOL Programming* (Cambridge, MA: Winthrop, 1977), p. 43.

[6] Dennie Van Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, 2nd ed. (Englewood Cliffs, NJ: Prentice Hall, 1978), p. 78.

[7] Nancy Stern and Robert A. Stern, *Structured COBOL Programming*, 7th ed. (New York: John Wiley and Sons, 1994), p. 13.

[8] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1308.

To appreciate this, imagine that the only structured programming concepts we ever knew were top-down design, hierarchical structures, indenting statements, etc. Clearly, no one would call it a programming revolution on the strength of these concepts. It was the promise of precision and rigour that made it famous – the promise of developing and proving software applications mathematically.

So, now that what was left of structured programming was only the trivial concepts, the preoccupation with GOTO provided a critical substitute for the original, strict principles: it allowed both the theorists and the practitioners to delude themselves that they were still pursuing a serious idea. GOTO-less programming is the only remnant of the formal theory, so it serves as a link to the original claims, to the promise of mathematical programming.

The formal theory, however, was about structures of standard constructs, not about avoiding GOTO. All the theory says is that, if we adhere to these principles, we will end up with GOTO-less programs. The defenders of structured programming violate the strict principles (because impractical), and direct their efforts instead to what was meant to be merely a *consequence* of those principles. By restricting and debating the use of GOTO, and by contriving substitutes, they hope now to attain the same benefits as those promised by the formal theory.

Here are some examples of the attempt to ground the GOTO prohibition on the original, mathematical principles: "A theorem proved by Böhm and Jacopini tells us that any program written using GOTO statements can be transformed into an equivalent program that uses only the [three] structured constructs."[9] "Böhm and Jacopini showed that essentially any control flow can be achieved without the GOTO by using appropriately chosen sequential, selection, and repetition control structures."[10] "Dijkstra's [structured programming] proposal could, indeed, be shown to be theoretically sound by previous results from [Böhm and Jacopini,] who had showed that the control logic of any flowchartable program … could be expressed without GOTOs, using sequence, selection, and iteration statements."[11]

We saw under the third delusion that the theorists *misrepresent* Böhm and Jacopini's work (see pp. 557–561). Thus, invoking their work to support the GOTO prohibition is part of the misrepresentation.

[9] Doug Bell, Ian Morrey, and John Pugh, *Software Engineering: A Programming Approach* (Hemel Hempstead, UK: Prentice Hall, 1987), p. 14.

[10] Ralston and Reilly, *Encyclopedia*, p. 361.

[11] Harlan D. Mills, "Structured Programming: Retrospect and Prospect," in *Milestones in Software Evolution*, eds. Paul W. Oman and Ted G. Lewis (Los Alamitos, CA: IEEE Computer Society Press, ©1990 IEEE), p. 286 – paper originally published in *IEEE Software* 3, no. 6 (1986): 58–66.

The GOTO preoccupation, then, was the answer to the failure of the formal theory. By degrading the definition of structured programming from exact principles to a preoccupation with GOTO, everyone appeared to be practising scientific programming while pursuing in reality some trivial and largely irrelevant ideas.

It is important to note that the absurdity of the GOTO delusion is not so much in the idea of avoiding GOTO, as in the never-ending debates and arguments *about* avoiding it: in which situations should it be permitted, and in which ones forbidden. Had the GOTO avoidance been a strict prohibition, it could have been considered perhaps a serious principle. In that case, we could have agreed perhaps to redefine structured programming as programming without the use of explicit jumps. But, since a strict GOTO prohibition is impractical, what started as a principle became an informal rule: the exhortation to avoid it "as much as possible." The prohibition, in other words, was to be enforced only when the GOTO alternatives were not too inconvenient.

An even more absurd manifestation of the GOTO delusion was the attempt to avoid GOTO by replacing it with certain built-in, language-specific constructs, which perform in fact the same jumps as GOTO. The purpose of avoiding GOTO had been to avoid all jumps in the flow of execution, not to replace explicit jumps with implicit ones. Thus, in their struggle to save structured programming, the theorists ended up interpreting the idea of avoiding GOTO as a requirement to avoid the *phrase* "go to," not the jumps. I will return to this point later.

Recognizing perhaps the shallowness of the GOTO preoccupation, some theorists were defending structured programming by insisting that the GOTO prohibition is only *one* of its principles. Thus, the statement we see repeated again and again is that structured programming is "more" than just GOTO-less programming: "The objective of structured programming is much more far reaching than the creation of programs without GOTO statements."[12] "There is, however, much more to structured programming than modularity and the elimination of GOTO statements."[13] "Indeed, there *is* more to structured programming than eliminating the GOTO statement."[14]

These statements, though, are specious. They sound as if "more" meant the original, mathematical principles. But, as we saw, those principles were falsified. So "more" can only mean the *trivial* principles – top-down design

[12] James Martin and Carma McClure, *Structured Techniques: The Basis for CASE*, rev. ed. (Englewood Cliffs, NJ: Prentice Hall, 1988), p. 39.

[13] L. Wayne Horn and Gary M. Gleason, *Advanced Structured COBOL: Batch and Interactive* (Boston: Boyd and Fraser, 1985), p. 1.

[14] Yourdon, *Techniques*, p. 140.

and nested constructs, writing and documenting programs clearly, etc. – which had replaced the original ones.

The degradation from a formal theory to trivial principles is also seen in the fact that the term "structured" was commonly applied now, not just to programs restricted to certain flow-control constructs, but to almost any software-related activity. Thus, in addition to structured programming, we had structured coding, structured techniques, structured analysis, structured design, structured development, structured documentation, structured flow-charts, structured requirements, structured specifications, structured English (for writing the specifications), structured walkthrough (visual inspection of the program's listing), structured testing, structured maintenance, and structured meetings.

3

To summarize, there are three aspects to the GOTO delusion. The first one is the reversal in logic: from the original principle that applications be developed as structures of standard constructs, to the stipulation that applications be developed without GOTO. The GOTO statement is not even mentioned in the original theory; its absence is merely a consequence of the restriction to standard constructs. Thus, the first aspect of the GOTO delusion is the belief that a preoccupation with ways to avoid GOTO can be a substitute for an adherence to the original principle.

The second aspect is the belief that avoiding GOTO need not be a strict, formal principle: we should strive to avoid it, but we may use it when its elimination is inconvenient. So, if the first belief is that we can derive the same benefits by avoiding GOTO as we could by restricting applications to standard constructs, the second belief is that we can derive the same benefits if we avoid GOTO only when it is convenient to do so. The second aspect of the GOTO delusion can also be described as the fallacy of making two contradictory claims: the claim that GOTO is harmful and must be banned (which sounds scientific and evokes the original theory), and the claim that GOTO is sometimes acceptable (which turns the GOTO prohibition from a fantasy into a practical method). Although in reality the two claims cancel each other, they appear to express important programming concepts.

Lastly, the third aspect of the GOTO delusion is the attempt to avoid GOTO, not by *eliminating* those programming situations that require jumps in the flow of execution, but by replacing GOTO with some new constructs, specifically designed to perform those jumps in its stead. The third aspect, thus, is the belief that we can derive the same benefits by converting explicit jumps into

implicit ones, as we could with no jumps at all; in other words, the belief that it is not the jumps, but just the GOTO statement, that must be avoided.

We already saw examples of the first aspect of the GOTO delusion – those statements simply asserting that structured programming means programming without GOTO (see pp. 589–590). Let us see now some examples of the second aspect; namely, claiming at the same time that GOTO must be avoided and that it may be used.

The best-known case is probably that of E. W. Dijkstra himself. One of the earliest advocates of structured programming, Dijkstra is the author of the famous paper "Go To Statement Considered Harmful." We have already discussed this paper (see pp. 508–509), so I will only repeat his remark that he was "convinced that the GOTO statement should be abolished from all 'higher level' programming languages"[15] (in order to make it *impossible* for programmers to use it, in *any* situation). He reasserted this on every opportunity, so much so that his "memorable indictment of the GOTO statement" is specifically mentioned in the citation for the Turing award he received in 1972.[16]

Curiously, though, *after* structured programming became a formal theory – that is, when it was claimed that Böhm and Jacopini's paper vindicated mathematically the abolition of GOTO – Dijkstra makes the following remark: "Please don't fall into the trap of believing that I am terribly dogmatical about [the GOTO statement]."[17]

Now, anyone can change his mind. Dijkstra, however, did not change his mind about the validity of structured programming, but only about the strictness of the GOTO prohibition. Evidently, faced with the impossibility of programming without explicit jumps, he now believes that we can enjoy the benefits of structured programming whether or not we restrict ourselves to the standard constructs. Thus, the popularity of structured programming was unaffected by his change of mind. Those who held that GOTO must be banned could continue to cite his former statement, while those who accepted GOTO could cite the latter. Whether against or in favour of GOTO, everyone could base his interpretation of structured programming on a statement made by the famous theorist Dijkstra.

[15] E. W. Dijkstra, "Go To Statement Considered Harmful," in *Milestones*, eds. Oman and Lewis, p. 9. [16] Ralston and Reilly, *Encyclopedia*, p. 1396.

[17] E. W. Dijkstra, quoted as personal communication in Donald E. Knuth, "Structured Programming with *go to* Statements," in *Computing Surveys* 6, no. 4 (©1974 ACM, Inc.): 262 (brackets in the original).

One of those who chose Dijkstra's latter statement, and a famous theorist and Turing award recipient himself, is Donald Knuth: "I believe that by presenting such a view I am not in fact disagreeing sharply with Dijkstra's ideas"[18] (meaning his *new* idea, that GOTO is *not* harmful). Knuth makes this statement in the introduction to a paper that bears the striking title "Structured Programming with *go to* Statements" – a forty-page study whose goal is "to lay [the GOTO] controversy to rest."[19] It is not clear how Knuth hoped to accomplish this, seeing that the paper is largely an analysis of various programming examples, some with and others without GOTO, some where GOTO is said to be bad and others where it is said to be good; in other words, exactly what was being done by every other expert, in hundreds of other studies. The examples, needless to say, are typical textbook cases: trivial, isolated pieces of software (the largest has sixteen statements), where GOTO is harmless even if misused, and which have little to do, therefore, with the real reasons why jumps are good or bad in actual applications. One would think that if the GOTO controversy were simple enough to be resolved by such examples, it would have ended long before, through the previous studies. Knuth, evidently, is convinced that his discussion is better.

From the paper's title, and from some of his arguments, it appears at first that Knuth intends to "lay to rest" the controversy by boldly stating that the use of GOTO is merely a matter of programming style, or simplicity, or efficiency. But he only says this in certain parts of the paper. In other parts he tells us that it is important to avoid GOTO, shows us how to eliminate it in various situations, and suggests changes to our programming languages to help us program without GOTO.[20]

By the time he reaches the end of the paper, Knuth seems to have forgotten its title, and concludes that GOTO is not really necessary: "I guess the big question, although it really shouldn't be so big, is whether or not the ultimate language will have GOTO statements in its higher levels, or whether GOTO will be confined to lower levels. I personally wouldn't mind having GOTO in the highest level, just in case I really need it; but I probably would never use it, if the general iteration and situation constructs suggested in this paper were present."[21]

[18] Donald E. Knuth, "Structured Programming with *go to* Statements," in *Computing Surveys* 6, no. 4 (©1974 ACM, Inc.): 262. [19] Ibid., p. 291.

[20] Knuth admits proudly that he deliberately chose "to present the material in this apparently vacillating manner" (ibid., p. 264). This approach, he explains, "worked beautifully" in lectures: "Nearly everybody in the audience had the illusion that I was largely supporting his or her views, regardless of what those views were!" (ibid.). What is the point of this approach, and this confession? Knuth and his audiences are evidently having fun debating GOTO, but are they also interested in solving this problem? [21] Ibid., p. 295.

Note how absurd this passage is: "wouldn't mind … just in case I really need it; but I probably would never use it … ." This is as confused and equivocal as a statement can get. Knuth is trying to say that it is possible to program without GOTO, but he is afraid to commit himself. So what was the point of this lengthy paper? Why doesn't he state, unambiguously, either that the ideal high-level programming language must include certain constructs but not GOTO, or, conversely, that it must include GOTO, because we will always encounter situations where it is the best alternative?

Knuth also says, at the end of the paper, that "it's certainly possible to write well-structured programs with GOTO statements,"[22] and points to a certain program that "used three GOTO statements, all of which were perfectly easy to understand." But then he adds that some of these GOTOs "would have disappeared" if that particular language "had had a WHILE statement." Again, he is unable to make up his mind. He notes that the GOTOs are harmless when used correctly, then he contradicts himself: he carefully counts them, and is pleased that more recent languages permit us to reduce their number.

One more example: In their classic book, *The C Programming Language*, Brian Kernighan and Dennis Ritchie seem unsure whether to reject or accept GOTO.[23] It was included in C, and it appears to be useful, but they feel they must conform to the current ideology and criticize it. First they reject it: "Formally, the GOTO is never necessary, and in practice it is almost always easy to write code without it. We have not used GOTO in this book."[24] We are not told how many situations are left outside the "almost always" category, but their two GOTO examples represent in fact a very common situation (the requirement to exit from a loop that is nested two or more levels within the current one).

At this point, then, the authors are demonstrating the *benefits* of GOTO. They even point out (and illustrate with actual C code) that any attempt to eliminate the GOTO in these situations results in an unnatural and complicated piece of software. The logical conclusion, thus, ought to be that GOTO *is* necessary in C. Nevertheless, they end their argument with this vague and ambiguous remark: "Although we are not dogmatic about the matter, it does seem that GOTO statements should be used sparingly, if at all."[25]

[22] The quotations in this paragraph are ibid., p. 294.

[23] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, NJ: Prentice Hall, 1978), pp. 62–63.

[24] Ibid., p. 62. Incidentally, they managed to avoid GOTO in all their examples simply because, as in any book of this kind, the examples are limited to small, isolated, artificial bits of logic. But the very fact that the avoidance of GOTO in examples was a priority demonstrates the morbidity of this preoccupation.

[25] Ibid., p. 63.

It is the third aspect of the GOTO delusion, however, that is the most absurd: eliminating the GOTO statements by replacing them with new constructs that are designed to perform exactly the same jumps. At this point, it is no longer the *jumps* that we are asked to avoid, but just the *phrase* "go to."

At first, we saw under the fourth delusion, the idea of structured programming was modified to include a number of non-standard constructs – constructs already found in the existing programming languages. Originally, these constructs had been invented simply as language enhancements, as alternatives to the most common jumps. (They simplify the jumps, typically, by obviating the need for a destination label.) But, as they allowed practitioners to bypass the restriction to standard constructs, they were enthusiastically incorporated into structured programming and described as "extensions" of the theory.

Although the inclusion of language-specific constructs appeared to rescue the idea of structured programming, there remained many situations where GOTO could only be eliminated through some unwieldy transformations, and still others where GOTO-based constructs were the only practical alternative. So the concept of language-specific constructs – what had been originally intended merely as a way to improve programming languages – was expanded and turned by the theorists into a means to eliminate GOTO. Situations easily implemented with GOTO in any language became the subject of research, debate, and new constructs. More and more constructs were suggested as GOTO replacements, although, in the end, few were actually added to the existing languages.

The theorists hoped to discover a set of constructs that would eliminate forever the need for GOTO by providing built-in jumps for all conceivable programming situations. They hoped, in other words, to redeem the idea of structured programming by finding an alternative to the contrived and impractical transformations. No such set was ever found, but this failure was not recognized as the answer to the GOTO delusion, and the controversy continued.

The theorists justified their attempts to replace GOTO with language-specific constructs by saying that these constructs facilitate structured programming. But this explanation is illogical. If we interpret structured programming as the original theory, with its restriction to standard constructs, the role of the non-standard constructs is not to facilitate but to *override* structured programming. And if we interpret structured programming as the extended theory, which allows any construct with one entry and exit, we can implement all the constructs we need by combining standard constructs and GOTO statements; in this case, then, the role of the non-standard constructs is not to facilitate structured programming but to facilitate GOTO-less programming.

The theorists, therefore, were not inventing built-in constructs out of a concern for structured programming – no matter how we interpret this theory – but only in order to eliminate GOTO.

As an example of the attempts to define a set of flow-control constructs that would make GOTO unnecessary, consider Jensen's study.[26] Jensen starts by defining three "atomic" components: "We use the word *atomic* to characterize the lowest level constituents to which we can reduce the structure of a program."[27] The three atomic components are called process node, predicate node, and collector node, and represent lower software levels than do the three standard constructs of structured programming. Then, Jensen defines nine flow-control constructs based on these components (the three standard constructs plus six non-standard ones), proclaims structured programming to mean the restriction, not to the three standard constructs but to his nine constructs, and asserts that any application can be developed in this manner: "By establishing program structure building blocks (akin to molecules made from our three types of atoms) and a structuring methodology, we can scientifically implement structured programs."[28] But, even though Jensen discusses the practical implementation of this concept with actual programming languages and illustrates it with a small program, the concept remains a theoretical study, and we don't know how successful it would be with real-world applications.

An example of a set of constructs that was actually put into effect is found in a language called Bliss. One of its designers makes the following statement in a paper presented at an important conference: "The inescapable conclusion from the Bliss experience is that the purported inconvenience of programming without a GOTO is a myth."[29]

It doesn't seem possible that the GOTO delusion could reach such levels, but it did. That statement is ludicrous even if we overlook the fact that Bliss was just a special-purpose language (designed for systems software, so the conclusion about the need for GOTO is not at all inescapable in the case of other types of programs). The academics who created Bliss invented a number of constructs purposely in order to replace, one by one, various uses of GOTO. The constructs, thus, were specifically designed to perform *exactly* the same jumps as GOTO. To claim, then, that using these constructs instead of GOTO proves that it is possible to program without GOTO, and to have such claims published and debated, demonstrates the utter madness that had possessed the academic and the programming communities.

[26] Randall W. Jensen, "Structured Programming," in *Software Engineering*, eds. Randall W. Jensen and Charles C. Tonies (Englewood Cliffs, NJ: Prentice Hall, 1979).

[27] Ibid., p. 238. [28] Ibid., p. 241.

[29] William A. Wulf, "A Case against the GOTO," *Proceedings of the ACM Annual Conference*, vol. 2 (1972), p. 795.

Here is how Knuth, in the aforementioned paper, describes this madness: "During the last few years several languages have appeared in which the designers proudly announced that they have abolished the GOTO statement. Perhaps the most prominent of these is Bliss, which originally replaced GOTO's by eight so-called 'escape' statements. And the eight weren't even enough.... Other GOTO-less languages for systems programming have similarly introduced other statements which provide 'equally powerful' alternative ways to jump.... In other words, it seems that there is widespread agreement that GOTO statements are harmful, yet programmers and language designers still feel the need for some euphemism that 'goes to' without saying GOTO."[30]

Unfortunately, Knuth ends his paper contradicting himself; now he *praises* the idea of replacing GOTO with new constructs designed to perform the same operation: "But GOTO is hardly ever the best alternative now, since better language features are appearing. If the invariant for a label is closely related to another invariant, we can usually save complexity by combining those two into one abstraction, using something other than GOTO for the combination."[31] What Knuth suggests is that we improve our programming languages by creating higher levels of abstraction: built-in flow-control constructs that combine several operations, including all necessary jumps. Explicit jumps, and hence GOTO, will then become unnecessary: "As soon as people learn to apply principles of abstraction consciously, they won't see the need for GOTO."[32]

Knuth's mistake here is the fallacy we discussed under the second and fourth delusions (see pp. 539–542, 578–579): he confuses the flow-control constructs with the *operations* of a hierarchical structure. In the static flow diagram – that is, in the nesting scheme – these constructs do indeed combine elements to form higher levels of abstraction. But because they employ conditions, their task in the flow of execution is not to create higher levels, but to create multiple, interacting nesting schemes.

The idea of replacing GOTO with higher-level constructs is, therefore, fallacious. Only an application restricted to a nesting scheme of sequential constructs has a flow of execution that is a simple hierarchical structure, allowing us to substitute one construct for several lower-level ones. And no serious application can be restricted to such a nesting scheme. This is why no one could invent a general-purpose language that eliminates the need for jumps. In the end, all flow-control constructs added to programming languages over the years are doing exactly what GOTO-based constructs are doing, but without using the *phrase* "go to."

[30] Knuth, "Structured Programming," pp. 265–266.
[31] Ibid., p. 294.
[32] Ibid., pp. 295–296.

4

Because of its irrationality, the GOTO prohibition acquired in the end the character of a superstition: despite the attempt to ground the debate on programming principles, avoiding GOTO became a preoccupation similar in nature to avoiding black cats, or avoiding the number 13.

People who cling to an unproven idea develop various attitudes to rationalize their belief. For example, since it is difficult to follow strictly the precepts of any superstition, we must find ways to make the pursuit of superstitions practical. Thus, even if convinced that certain events bring misfortune, we will tolerate them when avoiding them is inconvenient – and we will contrive an explanation to justify our inconsistency. Similarly, we saw, while GOTO is believed to bring software misfortune, most theorists agree that there is no need to be dogmatic: GOTO is tolerable when avoiding it is inconvenient.

Humour is an especially effective way to mask the irrationality of our acts. Thus, it is common to see people joke about their superstitions – about their habit of touching wood, for instance – even as they continue to practise them. So we shouldn't be surprised to find humorous remarks accompanying the most serious GOTO discussions. Let us study a few examples.

In his assessment of the benefits of structured programming, Yourdon makes the following comment: "Many programmers feel that programming without the GOTO statement would be awkward, tedious, and cumbersome. For the most part, this complaint is due to force of habit. . . . The only response that can be given to this complaint comes from a popular television commercial that made the rounds recently: 'Try it – you'll like it!'"[33] This is funny, perhaps, but what is the point of this quip? After explaining and praising GOTO-less programming, Yourdon admits that the only way to demonstrate its benefits is with the techniques of television advertising.

Another example of humour is the statement COME FROM, introduced as an alternative to GOTO. Although meant as a joke, this statement was actually implemented in several programming languages, and its merits are being discussed to this day in certain circles. Its operation is, in a sense, the reverse of GOTO; for instance, COME FROM L1 tells the computer to jump to the statement following it when the flow of execution encounters the label L1 somewhere in the program. (The joke is that, apart from being quite useless, COME FROM is even more difficult than GOTO to understand and to manage.) It is notable that the official introduction of this idea was in *Datamation*'s issue that proclaimed

[33] Yourdon, *Techniques*, p. 178.

structured programming a revolution (see p. 523). Thus, out of the five articles devoted to this revolution, one was meant in its entirety as a joke.[34]

One expert claims that the GOTO prohibition does not go far enough: the next step must be to abolish the ELSE in IF statements.[35] Since an IF-THEN-ELSE statement can be expressed as two consecutive IF-THEN statements where the second condition is the logical negation of the first, ELSE is unnecessary and complicates the program. The expert discusses in some detail the benefits of ELSE-less programming. The article, which apparently was *not* meant as a joke, ends with this sentence: "Structured programming, with elimination of the GOTO, is claimed to be a step toward changing programming from an art to a cost-effective science, but the ELSE will have to go, too, before the promise is a reality"[36] (note the pun, "go, too").

Knuth likes to head his writings with epigraphs, but from the quotations he chose for his aforementioned paper on GOTO, it is impossible to tell whether this is a serious study or a piece of entertainment. Two quotations, from a poem and from a song, were chosen, it seems, only because they include the word "go"; the third one is from an advertisement offering a remedy for "painful elimination." Also, we find the following remark in the paper: "The use of four-letter words like GOTO can occasionally be justified even in the best of company."[37]

The most puzzling part of Knuth's humour, however, is his allusion to Orwell's *Nineteen Eighty-Four*. He dubs the ideal programming language Utopia 84, as his "dream is that by 1984 we will see a consensus developing.... At present we are far from that goal, yet there are indications that such a language is very slowly taking shape.... Will Utopia 84, or perhaps we should call it Newspeak, contain GOTO statements?"[38]

Is this a joke or a serious remark? Does Knuth imply that the role of programming languages should be the same as the role of Newspeak in Orwell's totalitarian society – that is, to degrade knowledge and minds? (See "Orwell's Newspeak" in chapter 5.) Perhaps this *is* Knuth's dream, unless the following statement, too, is only a joke: "The question is whether we should ban [GOTO], or educate against it; should we attempt to legislate program morality? In this case I vote for legislation, with appropriate legal substitutes in place of the former overwhelming temptations."[39]

As the theorists and the practitioners recognized the shallowness of their preoccupation with GOTO, humour was the device through which they could

[34] R. Lawrence Clark, "A Linguistic Contribution to GOTO-less Programming," *Datamation* 19, no. 12 (1973): 62–63.

[35] Allan M. Bloom, "The 'ELSE' Must Go, Too," *Datamation* 21, no. 5 (1975): 123–128.

[36] Ibid., p. 128.

[37] Knuth, "Structured Programming," p. 282.

[38] Ibid., pp. 263–264.

[39] Ibid., p. 296.

pursue two contradictory ideas: that the issue is important, and that it is irrelevant. Humour, generally, is a good way to deal with the emotional conflict arising when we must believe in two contradictory concepts at the same time. Thus, like people joking about their superstitions, the advocates of structured programming discovered that humour allowed them to denounce the irrational preoccupation with GOTO even while continuing to foster it.

5

The foregoing analysis has demonstrated that the GOTO prohibition had no logical foundation. It has little to do with the original structured programming idea, and can even be seen as a new theory: the theory of structured programming failed, and the GOTO preoccupation took its place. The theorists and the practitioners kept saying that structured programming is more than just GOTO-less programming, but in reality the elimination of GOTO was now the most important aspect of their work. What was left of structured programming was only some trivial concepts: top-down design, constructs with one entry and exit, indenting the levels of nesting in the program's listing, and the like.

To appreciate this, consider the following argument. First, within the original, *formal* theory of structured programming, we cannot even discuss GOTO; for, if we adhere to the formal principles we will never encounter situations requiring GOTO. So, if we have to debate the use of GOTO, it means that we are not practising structured programming.

It is only within the modified, *informal* theory that we can discuss GOTO at all. And here, too, the GOTO debate is absurd, because this degraded variant of structured programming can be practised both with and without GOTO. We can have structured programs either without GOTO (if we use only built-in constructs) or with GOTO (if we also design our own constructs). The only difference between the two alternatives is the presence of explicit jumps in some of the constructs, and explicit jumps are compatible with the informal principles. With both methods we can practise top-down design, create constructs with one entry and exit, restrict modules to a hundred lines, indent the levels of nesting in the program's listing, and so forth. *Every principle stipulated by the informal theory of structured programming can be rigorously followed whether or not we use* GOTO.

The use of GOTO, thus, is simply a matter of programming style, or programming standards, which can vary from person to person and from place to place. Since it doesn't depend on a particular set of built-in constructs, the informal style of structured programming can be practised with any programming

language (even with low-level, assembly languages): we use built-in constructs when available and when effective, and create our own with explicit jumps when this alternative is better. (So we will have more GOTOs in COBOL, for example, than in C.)

Then, if GOTO does not stop us from practising the new, informal structured programming, why was its prohibition so important? As I stated earlier (see pp. 590–591), the GOTO preoccupation served as a substitute for the original theory: that theory restricted us to the three standard flow-control constructs (a rigorous principle that is all but impossible to follow), while the new theory permits us to use an arbitrary, larger set of constructs (in fact, any *built-in* constructs). Thus, the only restriction now is to use built-in constructs rather than create our own with GOTO. This principle is more practical than the original one, while still appearing precise. By describing this easier principle as an *extension* of structured programming, the theorists could delude themselves that they had a serious theory even after the actual theory had been refuted.

The same experts who had promised us the means to develop and prove applications mathematically were engaged now in the childish task of studying the use of GOTO in small, artificial pieces of software. And yet, no one saw this as evidence that the theory of structured programming had failed. While still talking about scientific programming, the experts were debating whether one trivial construct is easier or harder to understand than some other trivial construct. Is this the role of software theorists, to decide for us which style of programming is clearer? Surely, practitioners can deal with such matters on their own. We listened to the theorists because of their claim that software development can be a formal and precise activity. And if this idea turned out to be mistaken, they should have studied the reasons, admitted that they could not help us, and tried perhaps to discover what is the *true* nature of programming. Instead, they shifted their preoccupation to the GOTO issue, and continued to claim that programming would one day become a formal and precise activity.

The theorists knew, probably, that the small bits of software they were studying were just as easy to understand with GOTO as they were without it. But they remained convinced that this was a critical issue: it was important to find a set of ideal constructs because a flow-control structure free of GOTOs would eventually render the same benefits as a structure restricted to the three standard constructs. The dream of rigorous, scientific programming was still within reach.

The theorists fancied themselves as the counterpart of the old thinkers, who, while studying what looked like minute philosophical problems, were laying in fact the foundation of modern knowledge. Similarly, the theorists say, subjects

like GOTO may seem trivial, but when studying the appearance of small bits of software with and without GOTO they are determining in fact some important software principles, and laying the foundation of the new science of programming.

The GOTO issue was important to the theorists, thus, as a substitute for the formal principles of structured programming. But there was a second, even more important motivation for the GOTO prohibition.

Earlier in this chapter we saw that the chief purpose of structured programming, and of software engineering generally, was to get inexperienced programmers to perform tasks that require in fact great skills. The software theorists planned to solve the software crisis, not by promoting programming expertise, but, on the contrary, by eliminating the *need* for expertise: by turning programming from a difficult profession, demanding knowledge, experience, and responsibility, into a routine activity, which could be performed by almost anyone. And they hoped to accomplish this by discovering some exact, mechanistic programming principles – principles that could be incorporated in methodologies and development tools. The difficult skills needed to create software applications would then be reduced to the easier skills needed to follow methods and to operate software devices. Ultimately, programmers would only need to know how to use the tools provided by the software elite.

The GOTO prohibition was part of this ideology. Structured programs, we saw, can be written both with and without GOTO: we use only built-in flow-control constructs, or also create our own with GOTO statements. The difference is a matter of style and efficiency. So, if structured programming is what matters, all that the theorists had to do was to explain the principle of nested flow-control constructs. Responsible practitioners would appreciate its benefits, but the principle would not prevent them from developing an individual programming style. They would use custom constructs when better than the built-in ones, and the GOTOs would make their programs easier, not harder, to understand.

Thus, it was pointed out more than once that good programmers were practising structured programming even before the theorists were promoting it. And this is true: a programmer who develops and maintains large and complex applications inevitably discovers the benefits of hierarchical flow-control structures, indenting the levels of nesting in the program's listing, and other such practices; and he doesn't have to avoid GOTO in order to enjoy these benefits.

But the theorists had decided that programmers should not be expected to advance beyond the level attained by an average person after a few months of practice – beyond what is, in effect, the level of novices. The possibility of educating and training programmers as we do individuals in other professions – that is, giving them the time and opportunity to develop all the knowledge that human minds are capable of – was not even considered. It was simply assumed that if programmers with a few months of experience write bad software, the only way to improve their performance is by preventing them from dealing with the more difficult aspects of programming.

And, since the theorists believed that the flow-control structure is the most important aspect of the application, the conclusion was obvious: programmers must be forced to use built-in flow-control constructs, and prohibited from creating their own. In this way, even inexperienced programmers will create perfect flow-control structures, and hence perfect applications. Restricting programmers to built-in constructs, the theorists believed, is like starting with subassemblies rather than basic parts when building appliances: programming is easier and faster, and one needs lower skills and less experience to create the same applications. (We examined this fallacy earlier; see pp. 578–579.) Thus, simply by prohibiting mediocre programmers from creating their own flow-control constructs, we will attain about the same results as we would by employing expert programmers.

It is clear, then, why the theorists could not just *advise* programmers to follow the principles of structured programming. Since their goal was to control programming practices, it was inconceivable to allow the *programmers* to decide whether to use a built-in construct or a non-standard one, much less to allow them to *design* a construct. With its restriction to the three standard constructs, the original theory had the same goal, but it was impractical. So the theorists looked for a substitute, a different way to control the work of programmers. With its restriction to built-in constructs – constructs sanctioned by the theorists and incorporated into programming languages – the GOTO prohibition was the answer.

We find evidence that this ideology was the chief motivation for the GOTO prohibition in the reasons typically adduced for avoiding GOTO. The theorists remind us that its use gives rise to constructs with more than one entry or exit, thereby destroying the hierarchical nature of the flow-control structure; and they point out that it has been proved mathematically that GOTO is unnecessary. But despite the power of these formal explanations, they ground the prohibition, ultimately, on the idea that GOTO tempts programmers to

write "messy" programs. It is significant, thus, that the theorists consider the informal observation that GOTO allows programmers to create bad software more convincing than the formal demonstration that GOTO is unnecessary.

Here are some examples: "The GOTO statement should be abolished" because "it is too much an invitation to make a mess of one's program."[40] "GOTO instructions in programs can go to *anywhere*, permitting the programmer to weave a tangled mess."[41] "It would be wise to avoid the GOTO statement altogether. Unconditional branching encourages a patchwork (spaghetti code) style of programming that leads to messy code and unreliable performance."[42] "The GOTO can be used to produce 'bowl-of-spaghetti' programs – ones in which the flow of control is involuted in arbitrarily complex ways."[43] "Unrestricted use of the GOTO encourages jumping around within programs, making them difficult to read and difficult to follow."[44] "One of the most confusing styles in computer programs involves overuse of the GOTO statement."[45] "GOTO statements make large programs very difficult to read."[46]

What these authors are saying is true. What they are describing, though, is not programming with GOTO, but simply *bad* programming. They believe that there are only two alternatives to software development: bad programmers allowed to use GOTO and writing therefore bad programs, and bad programmers prevented from using GOTO. The possibility of having *good* programmers, who write good programs with or without GOTO, is not considered at all.

The argument about messy programs is ludicrous. It is true that, if used incorrectly, GOTO can cause execution to "go to anywhere," can create an "arbitrarily complex" flow of control, and can make the program "difficult to follow." But the GOTO problem is no different from any other aspect of programming: bad programmers do *everything* badly, so the messiness of their flow-control constructs is not surprising. Had these authors studied other aspects of those programs, they would have discovered that the file operations, or the definition of memory variables, or the use of subroutines, or the calculations, were also messy. The solution, however, is not to prohibit bad programmers from using certain features of a programming language, but to teach them how to program; in particular, how to create simple and consistent

[40] Dijkstra, "Go To Statement," p. 9.

[41] Martin and McClure, *Structured Techniques*, p. 133.

[42] David M. Collopy, *Introduction to C Programming: A Modular Approach* (Upper Saddle River, NJ: Prentice Hall, 1997), p. 142.

[43] William A. Wulf, "Languages and Structured Programs," in *Current Trends in Programming Methodology*, vol. 1, *Software Specification and Design*, ed. Raymond T. Yeh (Englewood Cliffs, NJ: Prentice Hall, 1977), p. 37.

[44] Clement L. McGowan and John R. Kelly, *Top-Down Structured Programming Techniques* (New York: Petrocelli/Charter, 1975), p. 43.

[45] Weinberg et al., *High Level COBOL*, p. 39.

[46] Van Tassel, *Program Style*, p. 78.

flow-control constructs. And if they are incapable or unwilling to improve their work, they should be replaced with better programmers.

The very use of terms like "messy" to describe the work of programmers betrays the distorted attitude that the software elite has toward this profession. Programmers whose work is messy should not even be employed, of course. Incredibly, the fact that individuals considered professional programmers create messy software is perceived as a normal state of affairs. Theorists, employers, and society accept the incompetence of programmers as a necessary and irremediable situation. And we accept not only their incompetence, but also the fact that they are irresponsible and incapable of improving their skills. Thus, everyone agrees that it is futile to teach them how to use GOTO correctly; they cannot understand, or don't care, so it is best simply to prohibit them from using it.

To be considered a professional programmer, an individual ought to display the highest skill level attainable in the domain of programming. This is how we define professionalism in other domains, so why do we accept a different definition for programmers? The software theorists claim that programmers are, or are becoming, "software engineers." At the same time, they are redefining the notions of expertise and responsibility to mean something entirely different from what they mean for engineers and for other professionals. In the case of programmers, expertise means acquaintance with the latest theories and standards, and responsibility means following them blindly. And what do these theories and standards try to accomplish? To obviate the need for *true* expertise and responsibility. No one seems to note the absurdity of this ideology.

6

We must take a moment here to discuss some of the programming aspects of the GOTO problem; namely, what programming style creates excellent, rather than messy, GOTO-based constructs. Had the correct use of GOTO demanded great expertise – outstanding knowledge of computers or mathematics, for instance – the effort to prevent programmers from creating their own constructs might have been justified. I want to show, however, that the correct use of GOTO is a trivial issue: from the many kinds of knowledge involved in programming, this is one of the simplest.

The following discussion, thus, is not intended to promote a particular programming style, but to demonstrate the triviality of the GOTO problem, and hence the absurdity of its prohibition. This will serve as additional evidence for my argument that the GOTO prohibition was motivated, not by some valid

software concerns, but by the corrupt ideology held by the software theorists. They had already decided that programmers must remain incompetent, and that it is they, the elite, who will control programming practices.

The first step is to establish, within the application, the boundaries for each set of jumps: the whole program in the case of a small application, but usually a module, a subroutine, or some other section that is logically distinct. Thus, even when the programming language allows jumps to go anywhere in the program, we will restrict each set of jumps to the section that constitutes a particular procedure, report, data entry function, file updating operation, and the like.

The second step is to decide what types of jumps we want to implement with GOTO. The number of reasons for having jumps in the flow of execution is surprisingly small, so we can easily account for all the possibilities. We can agree, for example, to restrict the *forward* jumps to the following situations: bypassing blocks of statements (in order to create conditional constructs); jumping to the point past the end of a block that is at a lower nesting level than the current one (in order to exit from any combination of nested conditions and iterations); jumping to any common point (in order to terminate one logical process and start another). And we can agree to restrict the *backward* jumps to the following situations: jumping to the beginning of a block (in order to create iterative constructs, and also to end prematurely a particular iteration); jumping to any common point (in order to repeat the current process starting from a particular operation).

We need, thus, less than ten types of jumps; and by *combining* jumps we can create any flow-control constructs we like. We will continue to use whatever built-in constructs are available in a particular language, but we will not *depend* on them; we will simply use them when more effective than our own. Recall the failed attempts to replace all possible uses of GOTO with built-in constructs. Now we see that this idea is impractical, not because of the large number of *types* of jumps, but because of the large number of *combinations* of jumps. And the problem disappears if we can design our own constructs, because now we don't have to plan in advance all conceivable combinations; we simply create them as needed.

Lastly, we must agree on a good naming system for labels. Labels are those flow-control variables that identify the statement where execution is to continue after a jump. And, since each GOTO statement specifies a label, we can choose names that link logically the jump's origin, its destination, and the purpose of the jump. This simple fact is overlooked by those who claim that

jumps unavoidably make programs hard to follow. If we adopt an intelligent naming system, the jumps, instead of confusing us, will *explain* the program's logic. (The compiler, of course, will accept any combination of characters as label names; it is the human readers that will benefit from a good naming convention.)

Here is one system: the first character or two of the name are letters identifying that section of the program where a particular set of jumps and labels are in effect; the next character is a letter identifying the type of jump; and these letters are followed by a number identifying the relative position of the label within the current set of jumps. In the name RKL3, for example, RK is the section, L identifies the start of a loop, and 3 means that the label is found after labels with numbers like 1 or 25, but before labels with numbers like 31 or 6. Similarly, T could identify the point past the end of a loop, S the point past a block bypassed by a condition, E the common point for dealing with an error, and so on.[47]

Note that the label numbers identify their order hierarchically, not through their values. For example, in a section called EM, the sequence of labels might be as follows: EMS2, EML3, EMS32, EMS326, EML35, EMT36, EMT4, EME82. The advantage of hierarchical numbering is that we can add new labels later without having to modify the existing ones. Note also that, while the numbers can be assigned at will, we can also use them to convey some additional information. For example, labels with one- or two-digit numbers could signify points in the program that are more important than those employing labels with three- or four-digit numbers (say, the main loop versus an ordinary condition).

Another detail worth mentioning is that we will sometimes end up with two or more consecutive labels. For example, a jump that terminates a loop and one that bypasses the block in which the loop is nested will go to the same point in the program, but for different reasons. Therefore, even though the compiler allows us to use one label for both jumps, each operation should have its own label. Also, while the *order* of consecutive labels has no effect on the program's execution, here it should match the nesting levels (for the benefit of the human readers); thus, the label that terminates the loop should come before the one that bypasses the whole block (EMT62, EMS64).

Simple as it is, this system is actually too elaborate for most applications. First, since the jump boundaries usually parallel syntactic units like subroutines, in many languages the label names need to be unique only within each

[47] In COBOL, labels are known as paragraph names, and paragraphs function also as procedures, or subroutines; but the method described here works the same way. (It is poor practice to use the same paragraph both as a GOTO destination and as a procedure, except for jumps within the procedure.)

section; so we can often dispose of the section identifier and start all label names in the program with the same letter. Second, in well-designed programs the purpose of most jumps is self-evident, so we can usually dispose of the type identifier too. (It is clear, for instance, whether a forward jump is to a common error exit or is part of a conditional construct.) The method I have followed for many years in my applications is to use even-numbered labels for forward jumps (EM4, EM56, EM836, etc.) and odd-numbered ones for backward jumps (EM3, EM43, EM627, etc.). I find this simplified identification of jump types adequate even in the most intricate situations.[48]

It is obvious that many other systems of jump types and label names are possible. It is also obvious that the consistent use of a particular system is more important than its level of sophistication. Thus, if we can be sure that every jump and label in a given application obeys a particular convention, we will have no difficulty following the flow of execution.

So the solution to the famous GOTO problem is something as simple as a consistent system of jump types and label names. All the problems that the software theorists attribute to GOTO have now disappeared. We can enjoy the benefits of a hierarchical flow-control structure and the versatility of explicit jumps at the same time.

The maintenance problem – the difficulty of understanding software created by others – has also disappeared: no matter how many GOTOs are present in the program, we know now for each jump where execution is going, and for each label where execution is coming from. We know, moreover, the *purpose* of each jump and label. Designing an effective flow-control structure, or following the logic of an existing one, may still pose a challenge; but, unlike the challenge of dealing with a messy structure, this is now a genuine programming problem. The challenge, in fact, is easier than it is with *built-in* constructs, because we have the actual, self-documented jumps and labels, rather than just the implicit ones. So, even when a built-in construct is available, the GOTO-based one is often a better alternative.

Now, it is hard to believe that any programmer can fail to understand a system of jumps and labels; and it is also hard to believe that no theorist ever thought of such a system. Thus, since a system of jumps and labels answers all the objections the theorists have to using GOTO, why were they trying to *eliminate* it rather than simply suggesting such a system? They describe the

[48] Figures 7-13 to 7-16 (pp. 680, 683–685) exemplify this style. Note that this method also makes the levels of nesting self-evident, obviating the need to indent the loops.

harmful effects of GOTO as if the only way to use it were with arbitrary jumps and arbitrary label names. They say nothing about the possibility of an intelligent and consistent system of jumps, or meaningful label names. They describe the use of GOTO, in other words, as if the only alternative were to have incompetent and irresponsible programmers. They appear to be describing a programming problem, but what they are describing is their distorted view of the programming profession: by stating that the best solution to the GOTO problem is avoidance, they are saying in effect that programmers will forever be too stupid even to follow a simple convention.

Structured programming, and the GOTO prohibition, did not make programming an exact activity and did not solve the software crisis. Programmers who had been writing messy programs before were now writing messy GOTO-less programs: they were messy in the way they were *avoiding* GOTO, and also in the way they were implementing subroutines, calculations, file operations, and everything else. Clearly, programmers who must be prohibited from using GOTO (because they cannot follow a simple system of jumps and labels) are unlikely to perform correctly any other programming task.

Recall what was the purpose of this discussion. I wanted to show that the GOTO prohibition, while being part of the structured programming movement, has little to do with its principles, or with any other programming principles. It is just another aspect of a corrupt ideology. The software elites claim that their theories are turning programming into a scientific activity, and programmers into engineers. In reality, the goal of these theories is to turn programmers into bureaucrats. The programming profession, according to the elites, is a large body of mediocre workers trained to follow certain methods and to use certain tools. Structured programming was the first attempt to implement this ideology, and the GOTO prohibition in particular is a blatant demonstration of it.

The Legacy

Because the theorists thought that the flow-control structure is the most important part of an application, they noticed at first only the GOTO messiness, and concluded that a restriction to built-in flow-control constructs would solve the problem of bad programming. Then, when this restriction was found to make no difference, they started to notice the other types of messiness. But the

solution was thought to be, again, not helping programmers to improve their skills, but preventing them from dealing on their own with various aspects of programming. Thus, structured programming was followed by many other theories, languages, methodologies, and database systems, all having the same goal: to degrade the work of programmers by shifting it to higher and higher levels of abstraction; to replace programming skills with a dependence on development systems; and to reduce the contribution of programmers to simple acts that require practically no knowledge or experience.

Had the theorists tried to understand *why* structured programming failed, perhaps they would have discovered the true nature of software and programming. They would have realized then that no mechanistic theory can help us, because software applications consist of interacting structures. The mechanistic software delusions, thus, could have ended with structured programming. But because they denied its failure, and because they continued to claim that formal programming methods are possible, the theorists established a mechanistic software culture. After structured programming, the traditional idea of expertise – skills that are mainly the result of personal knowledge and experience – was no longer accepted in the field of programming.

Unlike structured programming, today's theories are embodied in *development environments* – large and complicated systems known as object-oriented, fourth-generation, database management, CASE, and so on. Consequently, it is mainly the software companies behind these systems, rather than the theorists, that form now the software elite. No matter how popular they are, though, the development environments are ultimately grounded on mechanistic principles. So, if the mechanistic programming theories cannot help us, these systems cannot help us either. The reason they appear to work is that their promoters continually "enhance" them: while praising their high-level features, they reinstate – *within* these systems, and under new names – the low-level, versatile capabilities of the traditional programming languages. In other words, instead of correctly interpreting a particular inadequacy as a falsification of the original principles, they eliminate the inadequacy by annulling those principles. Thus, the same stratagem that made structured programming appear successful – modifying the theory by reinstating the very features it was supposed to replace – also serves to cover up the failure of development environments. (See "The Delusion of High Levels" in chapter 6; see also "The Quest for Higher Levels" in the next section.)

Turning falsifications into features, we recall, is how pseudoscientists manage to rescue their theories from refutation. High-level programming aids, thus, are fraudulent: after all the "enhancements," using a development environment is merely a more complicated form of the same programming work that we had been performing all along, with the traditional languages.

We must also recall the other method employed by pseudoscientists to defend their theories: looking for confirmations instead of falsifications; that is, studying the few cases where the theory appears to work, and ignoring the many cases where it fails. All software theories are promoted with this simple trick, whether or not they also benefit from the more sophisticated stratagem of turning falsifications into features.

Thus, it is common to see a particular theory or development system praised in books and periodicals on the basis of just one or two "success stories." Structured programming, for example, was tried with thousands of applications, but the only evidence of usefulness comes from a handful of cases: we see the same stories repeated over and over everywhere structured programming is promoted. (And there is not a single case where a serious application was implemented by following the original, formal principles.)

The study of structured programming is more than the study of a chapter in the history of programming. If all mechanistic theories suffer from the same fallacy – the belief that software applications can be separated into independent structures – then what we learned in our analysis of structured programming can help us to recognize the fallaciousness of any other programming theory. All we need to do is identify the structures that each theory attempts to extract from the complex whole.

The failure of structured programming is the failure of all mechanistic programming theories, and hence the failure of the whole idea of software engineering. This is true because software engineering is, in the final analysis, the ideology of software mechanism; so one cannot say that the idea of software engineering is sound if the individual theories are failing. The dream of structured programming was to represent software applications mathematically, and to turn programming into a precise, predictable activity. And it is the same dream that we find in the other theories, and in the general idea of software engineering. Individual theories may come and go, but if they are all based on mechanistic principles, they are in effect different manifestations of the same delusion.

If the individual theories are failing, the whole project of software engineering – replacing personal skills with formal methods, developing software the way we build appliances, designing and proving applications mathematically – is failing. Each theory displays the characteristics of a pseudoscience; but, in addition, the failure of each theory constitutes a falsification of the very idea of software engineering. Thus, by denying the failure of the individual theories, software engineering as a whole has been turned into a pseudoscience.

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.

The Relational Database Model

The relational database model is the theoretical concept upon which the relational database systems are founded. Database systems are environments for data management, and among them the relational kind are the most popular. In this section we will try to determine how much of this popularity is due to their data management capabilities, and how much to our mechanistic delusions. What we will find is that, like the other mechanistic software theories, the relational database model is a pseudoscience; that it is worthless as a programming concept; and that the relational systems became practical only after *annulling* their relational features, and after *reinstating* – in a more complicated form, and under new names – the *traditional* data management principles.

The relational model belongs to the class of theories that promise us higher levels of abstraction than those offered by the traditional programming

languages. Based on these theories, elaborate development systems are created and promoted. But instead of being abandoned when the idea of higher levels proves to be a fantasy, these systems undergo a series of "improvements." And the improvements, it turns out, consist in the addition of *low-level* capabilities; that is, precisely those features we had in the traditional languages, and which these systems were meant to supersede. So, in the end, all we accomplish is to replace efficient and straightforward languages with slow, complicated, and expensive development systems. (See "The Delusion of High Levels" in chapter 6; see also "The Quest for Higher Levels" in the previous section.)

The Promise

1

The idea of a database management system emerged in the late 1960s, when it was noticed that programmers had difficulty designing correct file relationships. Individually, the file operations are quite simple: reading a particular data record, writing a new record or modifying an existing one, and the like. The difficulty, rather, lies in creating correct *combinations* of operations. Applications need many files, and a file may have many records. Moreover, through the data present in their fields, the records form intricate relationships, and the file operations must exactly match these relationships if the application is to run correctly. It is the programmer's task to specify the iterations and conditions through which the application will create and use the various records at run time; and even a small error can have such consequences as reading or deleting the wrong record, corrupting the data stored in a record, or slowing down the application by performing unnecessary file operations.

The challenges that programmers face with file operations, thus, are similar to those they face with any other aspect of the application. So, as is the case with the other challenges, what they need is expertise: the knowledge and skills one develops over the years by programming increasingly complex applications. Programmers have difficulty designing correct and efficient file operations because they lack this expertise, because our programming culture prevents them from advancing past the level of novices.

The theories of structured programming and object-oriented programming, we saw, were invented by the software elite in an effort to obviate the need for programming expertise. Since programmers had difficulty creating correct flow-control constructs, restricting them to a few, standard constructs was seen as the answer; and since they had difficulty creating useful and modifiable applications, restricting them to ready-made modules was seen as the answer.

The solution to programming incompetence, in other words, was always thought to lie, not in encouraging programmers to *improve* their skills, but in discovering methods that would eliminate the *need* for skills; specifically, methods that would permit inexperienced programmers to accomplish tasks demanding expertise.

And it was the same ideology that prompted the invention of the relational database model. If programmers have difficulty designing combinations of file operations, let us provide these combinations in the form of built-in, high-level operations. For example, simply by specifying a few parameters, programmers would be able to read a set of logically related records. In contrast, to read the same records with the basic file operations, programmers must scan the file one record at a time, and control this process using iterative and conditional constructs.

Historically, the first database systems (which were based on the so-called hierarchical and network database models) were seen largely as management tools: means to take away from programmers the responsibility of designing and maintaining the application's database. The software experts hoped that, by keeping the most important database functions outside the application, database systems would eliminate our dependence on the skills of programmers: the database would become the responsibility of managers or analysts, and the programmers would simply be told, for each requirement, what database operations to invoke.

Now, the general trend was already to break down the application into smaller and smaller parts, in order to match the capabilities of inexperienced programmers. The trend, in other words, was to prevent programmers from *designing* software, and to reduce their work to little more than translating into a programming language the instructions received from a superior. So what the first database systems were promising was to reduce database programming to the same type of work. For all practical purposes, programmers would no longer need to know anything about the files used by the application, or about the relations between files. All they would have to do is translate some simple instructions into the equivalent database operations.

As was the case with the relational model later, attempting to simplify programming by raising the level of abstraction only made it more complicated. New software concepts, design methodologies, and languages (known as data definition and data manipulation languages) had to be introduced to support the hierarchical and network models; and in the end, the high-level operations turned out to be more difficult than the traditional ones. Database

systems, thus, were a fraud from the start: they complicated programming instead of simplifying it, and did not provide any functions that could not be implemented through the traditional file operations. (In fact, certain file relationships, easily programmed using file operations, cannot be implemented at all with the hierarchical and network models.)

When judged from within our corrupt software culture, however, the appeal of the original promise is understandable. Once we accept the idea that the highest programming skills we can expect are those attained by an average person after a few months of practice, replacing programmers with software devices seems logical. The complexity created by database systems is a small price to pay, the software experts tell us, for what we gain: successful database management regardless of the skills of the available personnel. Surely, we cannot trust a *programmer* with the task of designing the complex relationships that make up a database. Besides, with so many programmers working on the same application, it is impractical to allow each one to modify the file definitions. A specialist should design the database, and the best way to separate this task from the programming tasks is with a database management system.

In reality, the programmer is the best person to design the application's database, just as he is the best person to deal with every other aspect of the application. However, this is true only of experienced, professional programmers. Everyone could see that the existing programmers were novices, not professionals. But instead of giving them the time and opportunity to develop their skills, the preferred solution was to employ hordes of these novices, and to create several levels of management – bureaucrats with titles like systems analyst and project manager – to supervise them.[1]

As I have already remarked, this ideology – the belief that programming skills can be replaced with management skills – was already accepted when the first database systems were being introduced. So the idea of transferring the responsibility for the application's *database* from programmers to managers, although just as absurd as the attempt to replace the other programming skills,

[1] Most business applications can be developed and supported by *one* person. Thus, working alone and only part-time, I designed, programmed, and maintained several business systems over the years – the kind of systems for which companies normally employ teams of programmers and analysts. Few people are aware of the immense inefficiency created when a number of inexperienced individuals work together on one software project. The resulting application can become, quite literally, hundreds of times larger than necessary, and also far more involved. The combination of large teams, incompetence, and the dependence on development environments and ready-made pieces of software gives rise to an inefficiency that feeds on itself. In the end, these bureaucrats spend most of their time solving specious problems, which they themselves keep creating, instead of genuine software and business problems.

appeared quite logical. (As it turned out, though, the complexity of the database systems exceeded the capacity of existing management, and a new type of software bureaucrat had to be invented – the database administrator.)

And so it is how, from a totally unnecessary tool, database systems became one of the main preoccupations, and a major contribution to the astronomic cost of data processing, in most computer installations.

2

Although all database models suffer from the same fallacies, it is the relational model that concerns us, because it was beginning with this model that the software experts presented the concept of database systems as a *scientific* theory. If the earlier systems were promoted as management tools, the relational systems were also seen as a step in the formalization of application development. Because the relational model is based on certain mathematical concepts, the experts were now convinced that the benefits of database systems had been proved. Accordingly, a manager who refused to adopt this model was guilty of more than just resisting software progress; he was guilty of rejecting science.

The relational model is much more ambitious than the earlier ones. What we are promised is that, if we keep the data in a particular format, and if we restrict ourselves to a particular type of operations, we will never again have to deal with *low-level* entities (indexes, individual records and fields, and the related iterative and conditional constructs). In addition, the relational model will eliminate all data inconsistencies. The files are treated now simply as tables with rows and columns, and all we have to do is select and combine logical portions of these tables. Any database requirement, we are told, can be implemented in this fashion – in the same way that any mathematical expression is, ultimately, a combination of some basic operations.

As they did for the other mechanistic ideas, the software experts failed to understand why the mathematical background of a theory does not guarantee its usefulness for application development. Its exact nature only means that a mechanistic model has been found for *one* aspect of the application. And this is a trivial accomplishment: We know that complex phenomena can be represented as systems of simple hierarchical structures; and we also know that it is possible to extract any one of these structures and to represent it with a mechanistic model. Software applications comprise many structures, so it is not difficult to find exact models if all we want is to represent these structures *individually*.

Thus, the theory of structured programming asked us to extract that

aspect of the application that is its static flow diagram. Since it is possible to reduce this one aspect to a perfect hierarchy, and hence to represent it mathematically, the experts believed that the application as a whole can be represented mathematically. The theory of object-oriented programming asked us to identify the various aspects of our affairs, and to depict each one with a hierarchical classification of software entities. The experts believed that if each aspect is represented with a perfect hierarchy, whole applications can be developed simply by combining these hierarchies. Finally, the relational database theory asks us to extract that aspect of the application that is its database, and to reduce to perfect hierarchies the file relationships. This will permit us to represent mathematically the database structures, and hence the database operations.

What a mechanistic software theory does, in the final analysis, is model structures of a particular type, after separating them from the other structures that make up the application. So, no matter how successful these theories are in representing the individual structures, they are worthless as *programming* theories, because we cannot develop the application by dealing with each structure separately. The elements of these structures are the software entities that make up the application. Thus, since they share their elements, the structures interact, and we must deal with all of them at the same time. No theory that represents individual structures can model the whole application closely enough to be useful as a programming theory.

Like all mechanistic delusions, these theories are very naive. Since we already know that simple hierarchical structures can be represented mathematically, what these experts perceive as an important discovery is in reality a predictable achievement. All they are doing is breaking down software phenomena into smaller and smaller aspects, until they reach aspects that are simple enough to model with a hierarchical structure; and at that point they discover a mathematical theory for one of those aspects. But this is not surprising. The mathematical nature of the theory is a quality possessed by every hierarchical structure. We knew all along that they would find mathematical theories for the individual aspects. The very reason they separated the original, complex phenomenon into simpler phenomena is that simple structures can be represented with mathematical models while complex ones cannot.

Practitioners, though, must deal with whole applications, not isolated software phenomena. So, for these theories to have any value, we also need an exact theory for *combining* the various aspects – those neat hierarchical structures – into actual applications. And no such theory exists. The structures that represent the various aspects of an application are not related mechanistically, as one within another. They form complex structures.

The only way to create an application, therefore, is by relying on the non-mechanistic capabilities of our mind. But then, if we must have the expertise to deal with complex structures, why do we need theories that break down applications into simple structures in the first place? The software theorists are naive, thus, because they underestimate the difficulty of combining the isolated software structures into actual applications: they believe that we can combine them mechanistically, when the very reason for separating them was the impossibility of representing their totality mechanistically.

3

Like the other software theories, the delusion of the relational database model stems from the mechanistic fallacies of reification and abstraction: the belief that we can extract one aspect (the database structures, in this case) from the complex phenomenon that is a software application, and the belief that we can accomplish the same tasks by starting from high levels of abstraction as we can when starting from the lower levels.

So, as was the case with the ideas of structured programming and object-oriented programming, two great benefits are believed to emerge from the idea of a relational database. First, by reducing the application to strict hierarchical structures we will be able to represent software mathematically. Whether it is the flow of execution, the representation of a business process, or the database structures, we will deal with that aspect of the application formally, and thereby attain perfect, error-free software. Second, when we have strict hierarchical structures we can treat applications as systems of things within things. As we do in manufacturing, then, we will be able to use prefabricated software subassemblies, rather than depend on basic components. Application development will be easier and faster, since we will start our software projects from parts of a higher level of complexity – parts that already include other parts within them.

In the case of the relational model, this is accomplished by moving the low-level definitions and operations into the database system. All we need to do in the application is specify a relational operation, and the system will generate a database structure for us. In other words, not only do we have now a method that guarantees the correctness of the database, but this method consists of just a few simple, high-level operations. Instead of having to work with indexes and individual records, and with iterative and conditional constructs, we only need to understand now the concept of tables, of rows and columns: by extracting and combining portions of tables, we can generate all the database structures that we are likely to need in our applications.

Like the other software theories, the relational database model was seen as a critical step in the automation of programming. We only need to follow certain methods, and to use certain software systems; and because these methods and systems are based on mathematical concepts, we will end up with *provably* correct applications. It is already possible, we are told, to turn programming into a formal, routine activity. The concept of software engineering, if rigorously applied, already offers us the means to create perfect applications without depending on the skills or experience of individual programmers. And soon our systems will be powerful enough to eliminate the need for programmers altogether. Application development will then be completely automated: by means of sophisticated, interactive environments, managers and analysts will generate the application directly from the requirements.

We discussed the fallacy of high-level starting elements in chapter 6. We saw, in particular, that even for a simple requirement like file maintenance we must link the file operations to the other types of operations – display and calculations, for instance. And consequently, if we want to be able to implement *any* file maintenance functions, we must start with the basic file operations and with the statements of a traditional programming language (see pp. 435–438).

Similarly, we can perhaps replace with a high-level operation the file operations and the logic needed to read a set of related records, but only for common requirements: comparing or totaling the values present in certain fields, displaying or printing these values, specifying some criteria for record selection or grouping, and the like. High-level operations are useless if what we need is a combination of file operations and logic peculiar to a particular requirement: displaying one field when a certain condition occurs and another field otherwise, performing one calculation for some records and another for other records, and so forth. Clearly, there is no limit to the number of situations that may require a particular combination of file operations and business requirements, so the idea of replacing the basic operations with high-level ones is absurd.

High-level database operations are useful, thus, if provided *in addition* to the basic file operations; that is, if we retain the means to create our own combinations of file operations, and use the high-level operations only when they are indeed more effective. But this is not how the relational database systems are presented. The concept of a database environment is promoted as a *replacement* of the basic file operations. We are told that these operations are no longer necessary, and that we must depend exclusively on the high-level relational ones.

It is easy to tell when starting from higher levels is indeed a practical alternative: the resulting operations are simple and beneficial. A good example is the idea of a mathematical function library: a collection of subroutines that evaluate for us, through built-in algorithms, various mathematical functions. Thus, simply by calling one of these subroutines, we can determine in the application such values as the logarithm or the sine of a variable. Because we seldom need to link the operations that make up these algorithms with the operations performed in the application, the calculations can be extracted from the rest of the application and replaced with independent modules – modules that interact with the application only through their input and output. And we notice the success of this idea in that the concept of a mathematical function library has remained practically unchanged over the years. One of the oldest programming concepts, the mathematical library is as simple and effective today as it was when first implemented. We didn't have to continually "improve" and "enhance" this concept, as we do our database systems.

But the best example of a practical move to higher levels is provided by the file operations themselves. The basic file operations are usually described as "low-level," but they are low-level only relative to the operations promised by database systems. The basic operations are executed by a file management system, and, relative to the operations performed internally by that system, their level is quite high. (The terms "basic file operations" and "file management system" will be discussed in greater detail in the next subsection; see pp. 672–673.) Thus, a simple statement that reads, writes, or deletes a record in the application becomes, when executed by the file management system, a complex set of operations involving indexes, buffers, search algorithms, and disk accesses. But because there are no links between these operations and the various operations performed by the application, they can be separated from the application and invoked by means of simple statements.

So, relative to the operations performed internally by the file management system, the basic file operations constitute in effect a library of file management functions, as do the mathematical functions relative to *their* internal operations. And, like the mathematical functions, the success of this concept is seen in the fact that it has remained practically unchanged since the 1960s, when it was introduced. But, just because we moved successfully from the low level of buffers and direct disk access to a level where all we need to do is read, write, and delete data records, it doesn't follow that we can move to an even higher level.

It is precisely because no higher levels are possible that the relational database systems evolved into such complicated environments. Were high-level database operations a practical idea, their use would be as straightforward as is the use of mathematical functions or basic file operations. The reason

these systems became increasingly large and complicated is that, in order to make them practical, their designers had to add more and more "features." These features are perceived as *enhancements* of the relational concept, but their real function is to counteract the *falsifications* of this concept. They may have new and fancy names, but these are features we always had – in our programming languages.

Thus, in order to save the relational theory from refutation, the software pseudoscientists had to incorporate into database systems *programming* features: means to define integrity and security checks, to access individual records, to deal with run-time errors, and so forth. These, obviously, are the low-level, application-related processes which they had originally hoped to eliminate through high-level database operations.

So the relational database systems became in the end a fraud: instead of admitting that the idea of high-level database operations had failed, the theorists reinstated the low-level capabilities of the traditional programming languages while making them look like features of a database system. Entire new languages had to be invented, in order to let programmers perform *within the database system* those operations they had been performing all along, through traditional programming languages, *within the application*.

The reason for the complexity of the relational systems, thus, is that they ended up incorporating concepts which belong in the application. Programming problems that are quite easy to solve as part of the application become awkward and complicated when separated from the application and moved into a database environment. Besides, the new languages are not as versatile as the traditional, general-purpose ones: they provide only *some* of the low-level elements we need, and only as artificial extensions to the high-level features. So our work is complicated also by having to solve low-level programming problems in a high-level environment. Like other development environments, the relational systems have reversed a basic programming principle: instead of freely creating high-level elements by combining low-level ones, we are forced to start with high-level elements, and to treat the low-level ones as extensions.

The idea of a database system emerged, we recall, not because of any requirements that could not be implemented with the basic file operations, but as an answer to the lack of programmers who could use these operations correctly. So, if what programmers must do now is even more difficult than is the use of file operations, how can the database systems help them? High-level database operations cannot replace programming expertise any more than could the idea of structured programming, or the idea of object-oriented programming.

The Basic File Operations

1

To appreciate the inanity of the relational model, we must start by examining the basic file operations; that is, those operations which the relational systems are attempting to supplant. What I want to show is that these operations are *both necessary and sufficient* for implementing database management requirements, particularly in business applications. Thus, once we recognize the importance of the basic file operations, we will be in a better position to understand why the relational systems are fraudulent. For, as we will see, the only way to make them useful was by enhancing them with precisely those capabilities provided by the basic file operations; in other words, by restoring the very features that the database experts had claimed to be unnecessary.

Also, it is important to remember that the basic file operations have been available to programmers from the start, ever since mass storage devices with random access became popular. (They are sometimes called ISAM, Indexed Sequential Access Method.) For example, they have been available through COBOL (a language specifically designed for business applications) since about 1970. So these operations have always been well known: COBOL was always a public language, was implemented on all major computers, and was adopted by most companies. Thus, in addition to being an introduction to the basic file operations, this discussion serves to support my claim that the only motivation for database systems in general, and for the relational systems in particular, was to find a substitute for the knowledge required of programmers to use these operations correctly.

Before examining the basic file operations, we must take a moment to clarify this term and the related terms "file operations" and "database operations." The basic file operations are a basic set of file management functions. They formed in the past an integral part of every major operating system, and were accessible through programming languages. These operations deal with *indexed data files* – the most versatile form of data storage; and, in conjunction with the features provided by the languages themselves, they allow us to use and to relate these files in any way we like.

"File operations" is a more general term. It refers to the basic file operations, but also to the various ways in which we combine them, using the flow-control constructs of a programming language, in order to implement file management requirements. "Database operations" is an even more general

term. It refers to the file operations, but in the context of the whole application, so it usually means *combinations* of file operations; in particular, combinations involving several files. The terms "traditional file operations" and "low-level file operations" refer to any one of the operations defined above.

The term "database" refers to a set of related files; typically, the files used by a particular application. Hence, the term "database system" ought to mean any software system that helps us to manage a database.[1] Through their propaganda, though, the software elites have created in our minds a strong association between terms like "database," "database system," and "database management system" (or DBMS) and *high-level* database operations. And as a result, most people believe that the only way to manage a database is through high-level operations; that the current database systems provide indispensable features; and that it is impossible to implement a serious application without depending on such a system.

But we must not allow the software charlatans to control our language and our minds. Since we can implement any database functions through the basic file operations and a programming language, systems that provide high-level operations are not at all essential for database management. So we can continue to use the terms "database" and "database operations" even while rejecting the notion of a system that restricts us to high-level operations.

Strictly speaking, since the basic file operations permit us to manage a database, they too form a database system. But it would be confusing to use this term for the basic operations, now that it is associated with the high-level operations. Thus, I call the systems that provide basic file operations "*file* management systems," or "*file* systems" for short. This term is quite appropriate, in fact, seeing that these systems are limited to operations involving single files; it is *we* who implement the actual database management, by combining the operations provided by the file system with those provided by a programming language.

So I use the term "database," and terms like "database operations" and "database management," to refer to *any* set of related files – regardless of whether the files and relations are managed through the high-level operations of a *database* system, or through the basic operations of a *file* system.

The term "database structures" refers to the various hierarchical structures created by the files that make up the database: related files can be seen as the levels of a structure, and their records as the elements that make up these levels (see p. 688). In most applications, the totality of database structures is a complex structure.

[1] The term "database system" is used by everyone as an abbreviation of "database management system." It is somewhat misleading, though, since it sounds as if it refers to the database itself.

2

Two types of files make up the database structures of an application: *data* files and *index* files. The data files contain the actual data, organized as *records*; the index files (or indexes, for short) contain the pointers that permit us to access these records.

The record is the unit that the application typically reads from the file, or writes to the file. But within each record the data is broken down into *fields*, and it is the values present in the individual fields that we normally use in the application. For example, if each record in the file has 100 bytes, the first field may take the first 6 bytes, the second one the next 24 bytes, and so on. This is how the fields reside on disk, and in memory when the record is read from disk, but in most cases their relative order within the record is immaterial. For, in the application we assign names to these fields, and we refer to them simply by their names. Thus, once a record is read into memory, we treat database fields, for all practical purposes, as we do memory variables.

The records and fields of a data file reflect the structure and type of the information stored in the file. In an employee file, for example, there is a record for each employee, and each record contains such fields as employee number, name, salary, and year-to-date earnings and deductions; in a sales history file there is a record for each line in a sales order, with such fields as the customer and order numbers, date, price, and quantity sold. While in simple cases the required fields are self-evident, generally it takes some experience to design the most effective database for a given set of requirements. We must decide what information should be processed by the application, how to represent this information, how to distribute it among files, how to index the files, and how to relate them. Needless to say, it is impossible to predict all future requirements, so we must be prepared to alter the application's database structure later: we may need to add or delete fields, move fields from one file to another, and create new files or indexes.

We don't normally access data records directly, but through an index. Indexes, thus, are service files, means to access the data files. Indexes fulfil two essential functions: they allow us to identify a specific record, and to scan a series of records in a specific sequence. It is through *keys* that indexes perform these tasks. The key is one of the fields that make up the record, or a set of several fields. Clearly, if the combination of values present in these fields is different for each record in the file, each record can be uniquely identified. In addition, key uniqueness allows us to scan the records in a particular sequence – the sequence that reflects the current key values – regardless of their actual,

physical sequence on disk. When the key is one field, the value present in the field is the value of the key. When the key consists of several fields, the value of the key is the combination of the field values, in the order in which they make up the key. The records are scanned, in effect, in a sorted sequence. For example, if the key is defined as the set of three fields, *A*, *B*, and *C*, the sorting sequence can be expressed as either "by *A* by *B* by *C*" or "by *C* within *B* within *A*."

Note that if we permit *duplicate* keys – if, that is, some combinations of values in the key fields are not unique – we will be unable to identify the individual records within a set of duplicates. Such an index is still useful, however, if all we need is to *scan* those records. The scanning sequence within a set of duplicate records is usually the order in which they were added to the file. Thus, for scanning too, if we want better control we must ensure key uniqueness.

An especially useful feature is the capability to create several indexes for the same data file. This permits us to access the same records in different ways – scan the file in one sequence or another, or read a record through one key or another. For example, we may scan a sales history file either by order number or by product number; or, we may search for a particular sales record through a key consisting of the customer number and order number, or through a key consisting of the product number and order date.

Another useful indexing feature is the option of *descending* keys. The normal scanning sequence is *ascending*, from low to high key values; but some file systems also allow indexes that scan records from high to low key values. Any one field, or all the fields in the key, can then be either ascending or descending. Simply by scanning the data file through such an index we can list, for instance, orders in ascending sequence by customer number, but within each customer those orders with a higher amount first; or we can list the sales history by ascending product number, but within each product by descending date (so those sold most recently come first), and within each date by ascending customer number. A related indexing feature, useful in its own right but also as an alternative to descending keys, is the capability to scan records backward.

In addition to indexed data files, most file management systems support two other types of files, *relative* and *sequential*. These files provide simpler record access, and are useful for data that does not require an elaborate indexing scheme. In relative data files, we access a record by specifying its relative position in the file (first, second, third, etc.). These files are useful, therefore, in situations where the individual records cannot, or need not, be identified by the values present in their fields (to store the entries of a large table, for instance). Sequential data files are organized as a series of consecutive

records, which can only be accessed sequentially, starting from the beginning. These files are useful in situations where we don't need to access individual records directly, and where we normally read the whole file anyway (to store data that has no specific structure, for instance). Text data, too, is usually stored in sequential files. I will not discuss further the relative and sequential files. It is the indexed data files that interest us, because it is only *their* operations that the relational database systems are attempting to replace with high-level operations.

File systems provide at least two types of fields, *alphanumeric* (or *alpha*, for short) and *numeric*. And, since these types are the same as the memory variables supported by most high-level languages (COBOL, in particular), database fields and memory variables can be used together, and in the same manner, in the application. In alphanumeric fields, data is stored as character symbols, so these fields are useful for names, addresses, descriptions, notes, identifiers, and the like. When these fields are part of an indexing key, the scanning sequence is alphabetical. In numeric fields, the data is stored as numeric values, so these fields can be used directly in calculations. Numeric fields are useful for any data that can be expressed as a numeric value: quantities, dollar amounts, codes, and the like. When these fields are part of an indexing key, the scanning sequence is determined by the numeric value.

Some file systems provide additional field types. *Date* fields, for instance, are useful for storing dates. In the absence of date fields, we must store dates in numeric fields, as six- or eight-digit values representing the combination of the month, day, and year; alternatively, we can store dates as values representing the number of days elapsed since some arbitrary, distant date in the past. (The latter method is preferable, as it simplifies date calculations, comparisons, and indexing.) Another field type is the *binary* field, used to store such data as text, graphics, and sound; that is, data which can be in any format whatsoever (hence "binary," or raw), and which may require many thousands of bytes. (Because of its large size, this data is stored in separate files, and only pointers to it are kept in the field itself.)

3

Now that we have examined the structure of indexed data files, let us review the basic file operations. Six operations, combined with the iterative and conditional constructs of high-level languages, are all we need in order to use

indexed data files. I will first describe these operations, and then show how they are combined with language features to implement various requirements. The names I use for the basic operations are taken from COBOL. (There may be some small variations in the way these operations are implemented in a particular file system, or in a particular version of COBOL; for example, in the way multiple indexes or duplicate keys are supported.)

The following terms are used in the description of the file operations: The *current index* is the index file specified in the operation. *File* is a data file; although the file actually specified in the operation is an index file, the record read or written belongs to the data file (we always access a data file through one of its indexes). *Record area* is a storage area – the portion of memory where the fields that make up the record are specified; each file has its own record area, and this area is accessed by both the file system and the application (the application treats the fields as ordinary memory variables). *Key* is the field or set of fields, within the record area, that was defined as the key of a particular index; the *current key* is the key that was defined for the current index. The record *pointer* is an indicator maintained by the file system to identify the next record in the scanning sequence established by a particular index; each index has its own pointer, and the *current pointer* is the pointer corresponding to the current index.

WRITE: A new record is added to the file. Typically, the data in this record consists of the values previously placed by the application into the fields that make up the file's record area. The values present in the fields that make up the current key will become the new record's key in the current index. If the file has additional indexes, the values in their respective key fields will become the keys in those indexes. All indexes are updated together: following this operation, the new record can be accessed either through the current index or through another index. If one of the file's indexes does not permit duplicate keys and the new record would cause such a condition, the operation is aborted and the system returns an error code (so that the application can take appropriate action).

REWRITE: The data in the record area replaces the data in the record currently in the file. Typically, the application read previously the record into the record area through the current index, and modified some of the fields. The record is identified by the current key, so the fields that make up this key should not be modified. If there are other indexes, the fields that make up their keys may be modified, and REWRITE will update those indexes to reflect the change. REWRITE, however, can also be used without first reading the existing record: the application must place some values in all the fields, and REWRITE functions then like WRITE, except that it replaces an existing record. In either case, if no record is found with the current key, or if one of the file's indexes

does not permit duplicate keys and the modified record would cause such a condition, the operation is aborted and the system returns an error code.

DELETE: The record identified by the current key is removed from the file. Only the values present in the current key fields are important for the operation; the rest of the record area is ignored. The application, therefore, can delete a record either by reading it first into the record area (through any one of its indexes) or just by placing the appropriate values into the current key fields. If no record is found with the current key, the system returns an error code.

READ: The record identified by the current key is read into the record area. The current index can be any one of the file's indexes, and only the values present in the current key fields are important for the operation. Following this operation, the fields in the record area contain the values present in that record in the file. If no record is found with the current key, the system returns an error code.

START: The current pointer is positioned at the record identified by the current key. The current index can be any one of the file's indexes, and only the values present in the current key fields are important for the operation. The specification for the operation includes a relation like *equal*, *greater*, or *greater or equal*, so the application need not indicate a valid key; the record identified is simply the first one, in the scanning sequence of the current index, whose key satisfies the condition specified (for example, the first one whose key is *greater* than the values present in the current key fields). If no record in the file satisfies that condition, the system returns an error code.

READ NEXT: The record identified by the current pointer is read into the record area. This operation, in conjunction with START, makes the file scanning feature available to the application. The application must first perform a START for the current index, in order to set the current pointer at the first record in the series of records to be scanned. (To indicate the first record in the file, null values are typically placed in the key fields, and the condition *greater* is specified.) READ NEXT will then read that record and advance the pointer to the next record in the scanning sequence of the current index. The subsequent READ NEXT will read the record indicated by the pointer's new position and advance the pointer to the next record, and so on. Through this process, then, the application can read a series of consecutive records without having to know their keys.[2] Typically, READ NEXT is part of a loop, and the application knows when the last record in the series is reached by checking a certain condition (for example, whether the key exceeds a particular value). If the pointer was already positioned past the last record in the file (the *end-of-file* condition), the

[2] Since no search is involved, it is not only simpler but also faster to read a record in this fashion, than by specifying its key. Thus, even when the keys are known, it is more efficient to read consecutive records with READ NEXT than with READ.

system returns an error code. (Simply checking for this code after each READ NEXT is how applications typically handle the situation where the last record in the series is also the last one in the file.)

These six operations form the minimal practical set of file operations: the set of operations that are both necessary and sufficient for using indexed data files in serious applications.[3] I will demonstrate now, with a few examples, how the basic file operations are used in conjunction with other types of operations to implement typical requirements. Again, I am describing COBOL constructs and statements, but the implementation would be very similar in other high-level languages.

A common requirement involves the *display* of data from a particular record: the user identifies the record by entering the value of its key (customer number, part number, invoice number, and the like), and the application responds by retrieving that record and displaying some of its fields. When the key consists of several fields, the user must enter several values. To implement this operation in the application, all we need is a READ: we place the values entered by the user into the current key fields, perform the READ, and then display for the user various fields from the record area. If, however, the system returns an error code, we display a message such as "record not found."

If the user wants to *modify* some of the fields in a particular record, we start by performing a READ and displaying the current values, as before; but then we allow the user to enter the new values, place them in the appropriate fields in the record area, and perform a REWRITE. And if what the user wants is to *delete* a particular record, we usually start with a READ, display some of the fields to allow the user to confirm it is the right record, and then perform a DELETE.

Lastly, to *add* a record, we display blank fields and allow the user to enter their actual values. (In a new record, some fields may have null values, or some default values; so these fields may be left out, or just displayed, or displayed with the option to modify them.) The user must also enter the value of the key fields, to identify the new record. We then perform a WRITE, and the system will add this record to the file. If, however, it returns an error code, we display a message such as "duplicate key" to tell the user why the record could not be added.

[3] I will not discuss here the various *support* operations – opening and closing files, locking and unlocking records in multiuser applications, and the like. Since there is little difference between these operations in file systems and in database systems, they have no bearing on my argument. Many of these operations can be performed automatically, in fact, in both types of systems.

Examples of this type of record access are found in the *file maintenance* operations – those operations that permit the user to add, delete, and modify records in the database. And, clearly, any maintenance requirement can be implemented through the basic file operations: any file, record, and field in the database can be read, displayed, or modified. If we must restrict this freedom (permit only a range of values for a certain field, permit the addition or deletion of a record only under certain conditions, etc.), all we have to do is add appropriate checks; then, if the checks fail, we bypass the file operation and display a message.

So far I have discussed the *interactive* access of individual records, but the basic file operations are used in the same way when the user is not directly involved. Thus, if we need to know at some point in the application the quantity on hand for a certain part, we place the part number in the key field, perform a READ, and then get the value from the quantity field; if we want to add a new transaction to the sales history file, we place the appropriate values in the key fields (customer number, invoice number, etc.) and in the non-key fields (date, price, quantity, etc.), and perform a WRITE; if we want to update a customer's balance, we place the customer number in the key field, perform a READ, calculate the new value, place it in the balance field, and then perform a REWRITE. Again, any conceivable requirement can be implemented through the basic file operations.

Accessing *individual* records, as described above, is one way of using indexed data files. The other way is by *scanning* records, an operation accomplished with an iterative construct based on START and READ NEXT. This construct, which may be called the basic file scanning loop, is used every time we read a series of records sequentially through an index. The best way to illustrate this loop is with a simple example (see figure 7-13). The loop here is designed to read the PART file in ascending part number sequence. The indexing key, P-KEY, consists of one field, P-NUM (part number). START positions the record pointer so that the first record read has a part number no less than P1, and the

```
  MOVE P1 TO P-NUM START PART KEY>=P-KEY INVALID GO TO L4.
L3. READ PART NEXT END GO TO L4. IF P-NUM>P2 GO TO L4.
  IF P-QTY<Q1 GO TO L3.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-13

condition >P2 terminates the loop at the first record with a part number greater than P2. The loop will read, therefore, only the *range* of records, P1 through P2, inclusive.[4] In addition, within this range, the loop selects only those records where the quantity field, P-QTY, is no less than a certain value, Q1. The operations following the selection conditions will be performed for every record that satisfies these conditions. The labels L3 and L4 delimit the loop.[5]

We rarely perform the same operations with all the records in a file, so the selection of records is a common requirement in file scanning. The previous example illustrates the two selection methods – based on key fields, and on non-key fields. The method based on key fields is preferable when what we select is a range of records, as the records left out don't even have to be read. This can greatly reduce the processing time, especially if the file is large and the range selected is relatively small. In contrast, when the selection is based on non-key fields, each record in the file must be read. This is true because the value of non-key fields is unrelated to the record's position in the scanning sequence, so the only way to know what the values are is by reading the record. The two methods are often combined in the same loop, as illustrated in the example.

It should be obvious that these two selection methods are completely general, and can satisfy any requirement. For example, if the range must include all the records in the file, we specify null values for the key fields in START and omit the test for the end of the range. The loop also deals correctly with the case where no records should be selected (because there are none in the specified range, or because the selection based on non-key fields excludes all those in the range). It must be noted that the selection conditions can be as complex as we need: they can involve several fields, or fields from other files (by reading in the loop records from those files), or a combination of fields, memory variables, and constants. A complex condition can be formulated either as one complex IF statement or as several consecutive IF statements. And,

[4] Note the END clause in READ NEXT, specifying the action to take if the end of the file is reached before P2. (INVALID and END are the abbreviated forms of the COBOL keywords INVALID KEY and AT END. Similarly, GOTO can be abbreviated in COBOL as GO.)

[5] It is evident from this example that the most effective way to implement the basic file scanning loop in COBOL is with GOTO jumps. This demonstrates again the absurdity of the claim that GOTO is harmful and must be avoided (the delusion we discussed under structured programming). Modifying this loop to avoid the GOTOs renders the simple operations of file scanning and record selection complicated and abstruse; yet this is exactly what the experts have been advocating since 1970. It is quite likely that the complexity engendered by the delusions of structured programming contributed to the difficulty programmers had in using file operations, and was a factor in the evolution of database systems: because they tried to avoid the complications created by one pseudoscience, programmers must now deal with the greater complications created by another.

in addition to the conditions that affect all the operations in the loop, we can have conditions *within* the loop; any portion of the loop, therefore, can be restricted to certain records.

Let us see now how the basic file scanning loop is used to implement various file operations. In a typical file listing, or query, or report, the scanning sequence and the record selection criteria specified by the user become the index and the selection conditions for the scanning loop. And within the loop, for each record selected, we show certain fields and perhaps accumulate their values. Typically, one line is printed or displayed for each record, and the totals are shown at the end. When the indexing key consists of several fields, their value will change hierarchically, one within another, in the sorting sequence of the index; thus, we can have various levels of subtotals by noting within the loop when the value of these fields changes. In an orders file, for instance, if the key consists of order number within customer number, and if we need the quantity and amount subtotals for the orders belonging to each customer, we must show and then clear these subtotals every time the customer number changes.

Another use of the scanning loop is for *modifying* records. The reading and selection are performed as before, but here we modify the value stored in certain fields; then we perform a REWRITE (at the end of the loop, typically). This is useful when we must modify a series of records according to some common logic. Not all the selected records need to be modified, of course; we can perform some calculations and display the results for all the records in a given range, for instance, but modify only those where the fields satisfy a certain condition. Rather than modify records, we can use the scanning loop to *delete* certain records; in this case we perform a DELETE at the end of the loop.

An interesting use of indexed data files is for sorting. If, for instance, we need a listing of certain values in a particular scanning sequence (values derived from files or from calculations), we create a temporary data file where the indexing key is the combination of fields for that scanning sequence, while the non-key fields are the other values to be listed. All we have to do then is perform a WRITE to add a record to the temporary file for each entry required in the listing. The system will build for us the appropriate index, and, once complete, we can scan the temporary file in the usual manner. Similarly, if we need to scan a portion of a data file in a certain sequence, but only occasionally, then instead of having a permanent index for that sequence we create a temporary data file that is a subset of the main data file: we read the main data file in a loop through one of its indexes, and for each selected record we copy the required fields to the record of the temporary file and perform a WRITE.

If we want to analyze certain fields in a data file according to the value present in some other fields (total the quantity by territory, total various

amounts by the combination of territory and category, etc.), we must create a temporary data file where the indexing key is the field or combination of fields by which we want to group the records (the *analysis* fields in the main data file), while the non-key fields are the values to be totaled (the *analyzed* fields). We read the main file in a loop, and, for each record, we copy the analysis values and the analyzed values to the respective fields in the record of the temporary file. We then perform a WRITE for this file and check the return code. If the system indicates that the record already exists, it means this is not the first time that combination of key values was encountered; the response then is to perform a READ, *add* the analyzed values to the respective fields, and perform a REWRITE. In other words, we create a new record in the temporary file only the first time a particular combination of analysis values is encountered, and *update* that record on subsequent occasions. At the end, the temporary file will contain one record for each unique combination of analysis values. This concept is illustrated in figure 7-14.

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  MOVE C-TER TO SR-TER MOVE C-QTY TO SR-QTY.
  WRITE SR-RECORD INVALID READ SORTFL
    ADD C-QTY TO SR-QTY REWRITE SR-RECORD.
  GO TO L3.
L4.
```

Figure 7-14

In this example, a certain quantity in the CUSTOMER file is analyzed by territory for the customers in the range C1 through C2. SORTFL is the temporary file, and SR-RECORD is its record area. The simplicity of this operation is due to the fact that much of the logic is implicit in the READ, WRITE, and REWRITE.

4

One of the most important uses of the file scanning loop is to *relate* files. If we nest the scanning loop of one file within that of another, a logical relationship is created between the two files. From a programming standpoint, the nesting of file scanning loops is no different from the nesting of any iterative constructs: the whole series of iterations through the inner loop is repeated for every iteration through the outer loop. In the inner loop we can use fields from both files; any operation, therefore, including the record selection conditions, can depend on the record currently read in the outer loop.

Figure 7-15 illustrates this concept. The outer loop scans the CUSTOMER file and selects the range of customer numbers C1 through C2. The indexing key, C-KEY, consists of one field, C-NUM (customer number). Within this loop, in addition to any other operations performed for each customer record, we include a loop that scans the ORDERS file. The indexing key here, O-KEY, consists of two fields, O-CUS (customer number) and O-ORD (order number), in this sorting sequence. Thus, to restrict the inner loop to the orders belonging to one customer, we select only the range of records where the customer number equals the one currently read in the outer loop, while allowing the order number to be any value. (Note that the terminating condition, "IF O-CUS NOT=C-NUM," could be replaced with "IF O-CUS>C-NUM," since the first O-CUS read that is not equal to C-NUM is necessarily greater than it.) The inner loop here selects *all* the orders for the customer read in the outer loop; but we could have additional selection conditions, based on non-key fields, as in figure 7-13 (for example, to select only orders in a certain date range, or over a certain amount).

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  [various operations]
  MOVE C-NUM TO O-CUS MOVE 0 TO O-ORD.
  START ORDERS KEY>O-KEY INVALID GO TO L34.
L33. READ ORDERS NEXT END GO TO L34. IF O-CUS NOT=C-NUM GO TO L34.
  [various operations]
  GO TO L33.
L34.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-15

Although most file relations involve only two files, the idea of loop nesting can be used to relate hierarchically any number of files, simply by increasing the number of nesting levels. Thus, by nesting a third loop within the second one and using the same logic, the third file will be related to the second in the same way that the second is related to the first. With two files, we saw, the second file's key consists of two fields, and the range selected includes the records where the first field equals the first file's key. With three files, the third file's key must have three fields, and the range will include the records where the first two fields equal the second file's key. (The keys may have additional fields; two and three are the minimum needed to implement this logic.)

To illustrate this concept, figure 7-16 adds to the previous example a loop to scan the LINES file (the individual item lines associated with each order).

If ORDERS has fields like customer number, order number, date, and total amount, which apply to the whole order, LINES has fields like item number, quantity, and price, which are different for each line. Its indexing key consists of customer number, order number, and line number, in this sorting sequence. And the third loop isolates the lines belonging to a particular order by selecting the range of records where the customer and order numbers equal those of the order currently read in the second loop, while the line number is any value. Another example of a third nesting level is a transaction file, where each record is an invoice, payment, or adjustment pertaining to an order, and the indexing key consists of customer number, order number, and transaction number.[6]

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  [various operations]
  MOVE C-NUM TO O-CUS MOVE 0 TO O-ORD.
  START ORDERS KEY>O-KEY INVALID GO TO L34.
L33. READ ORDERS NEXT END GO TO L34. IF O-CUS NOT=C-NUM GO TO L34.
  [various operations]
  MOVE O-CUS TO L-CUS MOVE O-ORD TO L-ORD MOVE 0 TO L-LINE.
  START LINES KEY>L-KEY INVALID GO TO L334.
L333. READ LINES NEXT END GO TO L334.
  IF NOT(L-CUS=O-CUS AND L-ORD=O-ORD) GO TO L334.
  [various operations]
  GO TO L333.
L334.
  [various operations]
  GO TO L33.
L34.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-16

Note that in the sections marked "various operations" we can access fields from all the currently read records: in the outer loop, fields from the current CUSTOMER record; in the second loop, fields from the current CUSTOMER and ORDERS records; and in the inner loop, fields from the current CUSTOMER, ORDERS, and LINES records.

Note also that the sections marked "various operations" may contain additional file scanning loops; in other words, we can have more than one

[6] Note, in figures 7-13 to 7-16, the numbering system used for labels in order to make the jumps self-explanatory (as discussed under the GOTO delusion, pp. 621–624).

scanning loop at a given nesting level. For instance, by creating two consecutive third-level loops, we can scan first the lines and then the transactions of the order read in the second-level loop.

The arrangement where the key used in the outer loop is part of the key used in the inner loop, as in these examples, is the most common and the most effective way to relate files, because it permits us to select records through their key fields (and to read therefore only a *range* of records). We can also relate files, though, by using non-key fields to select records (when it is practical to read the entire file in the inner loop).

Lastly, another way to relate files is by reading within the loop of one file just one record of another file, with no inner loop at all (or, as a special case, reading just one record in both files, with no outer loop either). Imagine that we are scanning an invoice file where the key is the invoice number and one of the key or non-key fields is the customer number, and that we need some data from the customer record – the name and address fields, for instance. (This kind of data is normally stored only in the customer record because, even though required in many operations, it is the same for all the transactions pertaining to a particular customer.) So, to get this data, we place the customer number from the currently read invoice record into the customer key field, and perform a READ. All the customer fields are then available within the loop, along with the current invoice fields.

The relationship just described, where several records from one file point to the same record in another file, is called *many-to-one* relationship. And the relationship we discussed previously, where one record from the first file points to several records in the second file (because several records are read in the inner loop for each record read in the outer loop) is called *one-to-many* relationship. These two types of file relationships are the most common, but the other two, *one-to-one* and *many-to-many*, are also important.

We have a one-to-one relationship when the same field is used as a key in two files. For example, if in addition to the customer file we create a second file where the indexing key is the customer number (in order to store some of the customer data separately), then each record in one file corresponds to one record in the other. And we have a many-to-many relationship when one record in the first file points to several records in the second one, and at the same time one record in the second file points to several records in the first one. (We will study the four types of file relationships in greater detail later; see pp. 738–741.)

To understand the many-to-many relationship, imagine a factory where a

number of different products are being built by assembling various parts from a common inventory. Thus, each product is made from a number of different parts, and at the same time a part may be used in different products. The product file has one record for each product, and the key is the product number. And the part file has one record for each part, and the key is the part number. We can use these files separately in the usual manner, but to implement the many-to-many relationship between products and parts we need an additional file – a service file for storing the cross-references. This file is a dummy data file that consists of key fields only. It has two indexes: in the first one the key is the product number and the part number, and in the second one it is the part number and the product number, in these sorting sequences. In the service file, therefore, there will be one record for each pair of product and part that are related in the manufacturing process (far more records, probably, than there are either products or parts). Now we can scan the product file in the outer loop, and the service file, through its first index, in the inner loop; or, we can scan the part file in the outer loop, and the service file, through its second index, in the inner loop. Then, by selecting in the inner loop a range of records in the usual manner, we will read in the first case the parts used by a particular product, and in the second case the products that use a particular part. What is left is to perform a READ in the inner loop using the part or product number, respectively, in order to read the actual records.

The Lost Integration

The preceding discussion was not meant to be an exhaustive study of indexed data files. My main intent was to show that any conceivable database requirement can be implemented with file operations, and that this is a fairly easy programming challenge: every one of the examples we examined takes just a few statements in COBOL. We only need to understand the two ways of using indexes (reading individual records or scanning a range of records) and the two ways of selecting records (through key fields or non-key fields). Then, simply by combining the basic file operations with the other operations available in a programming language, we can access and relate the files in the database in any way we like.

So the difficulties encountered by programmers are not caused by the basic file operations, nor by the selection of records, nor by the file scanning loops. The difficulties emerge, rather, when we combine file operations, and when we combine them with the other types of operations required by the application. The difficulties, in other words, are due to the need to deal with

interacting software structures. Two kinds of structures, and hence two kinds of interactions, are generated: one through the file relationships we discussed earlier (one-to-many, many-to-many, etc.), the other through the links created between the application's elements by the file operations.

Regarding the first kind of structures, the file relationships are easy to understand individually, because we can view them as simple hierarchical structures. If we depict the nesting of files as a structure, each file can be seen as a different level of the structure, and its records as the various elements which make up that level. The relationship between files is then the relationship between the elements of one level and the next. But, even though each relationship is hierarchical, most files take part in several relationships, through different fields. In other words, a record in a certain file can be an element in several structures at the same time, so these structures interact. The totality of file relationships in the database is a complex structure.

As for the second kind of structures, we already know that the file operations give rise to processes based on shared data (see pp. 349–351). So they link the application's elements through many structures – one structure for each field, record, or file that is accessed by several elements. Thus, in addition to the interactions due to the file relationships, we must cope with the interactions between the structures generated by file operations. And we must also cope with the interactions between these structures and the structures formed by the other types of processes – practices, subroutines, memory variables, etc. To implement database requirements we must deal with complex software structures.

When replacing the basic file operations with higher-level operations, what are the database experts trying to accomplish? All that a database system can do is replace with a built-in process the two or three statements that constitute the use of a basic file operation. The experts misinterpret the difficulty that programmers have in implementing file operations as the problem of dealing with the relatively low levels. But, as we saw, the difficulty is not due to the individual file operations, nor to the individual relationships. The difficulty emerges when we deal with *interacting* operations and relationships, and with their interaction with the rest of the application. And these interactions cannot be eliminated; we must have them in a database system too, if the application is to do what we want it to do. Even with a database system, then, the difficult part of database programming remains. The database systems can perhaps replace the easy challenges – the individual operations; but they cannot eliminate the difficult part – the need to deal with interacting structures.

What is worse, database systems make the interactions even more complex, because some of the operations are now in the application while others are in the database system. The original idea was to have database functions akin to

the functions provided by a mathematical library; that is, entities of a high level of abstraction, which interact with the application only through their input and output. But this is impossible, because database operations must interact with the rest of the application at a lower level – at the level of fields, variables, and conditions. Thus, the level of abstraction that a database system can provide while remaining a practical system is not as high as the one provided by a mathematical library. We cannot extract, for example, a complete file scanning loop, with all the operations in the loop, and turn it into a high-level database function – not if we want to retain the freedom of implementing *any* scanning loops and operations.

All we needed before was the six basic file operations. The database operations, and their interaction with the rest of the application, could then be implemented with the same programming languages, and with the same methods and principles, that we use for the other operations in the application. With a database system, on the other hand, we need new and complicated principles, languages, rules, and methods; we must deal with a new kind of operations in the database system, plus a new kind of operations in the application, the latter necessary in order to link the application to the database system. So, in the end, the difficulties faced by programmers in implementing database operations are even greater than before.

It is easy to see why the basic file operations are both necessary and sufficient for implementing database operations: for most applications – business applications, in particular – they are just the right level of abstraction. The demands imposed by our applications rarely permit us to move to higher levels, and we rarely need lower ones. An example of lower-level file operations is the requirement for a kind of fields, indexes, or records that is different from the one provided by the standard data files. And, in the rare situations where such a requirement is important, we can implement it in a language like C. Similarly, in those situations where we can indeed benefit from higher-level operations, we can create them by means of subroutines in the same language as the application itself: we design the appropriate combination of basic file operations and flow-control constructs, store it as a separate module, and invoke it whenever we need that particular combination.

For the vast majority of applications, however, we need neither lower nor higher levels, since the level provided by the basic file operations is just right. This level is similar to the level provided, for general programming requirements, by our high-level languages. With the features found in a language like COBOL, for instance, we can implement any business application. Thus, it

is no coincidence that, in conjunction with the operations provided by a programming language, the basic file operations can be used quite naturally to implement practically all database operations, and also to link these operations to the other types of operations: iterative constructs are just right for scanning a data file sequentially through one of its indexes; nested iterations are just right for relating files hierarchically; conditional constructs are just right for selecting records; and assignment constructs are just right for moving data between fields, and between fields and memory variables. It is difficult to find a single database operation that cannot be easily and naturally implemented with the constructs found in the traditional languages.

This flexibility is due to the correct level of abstraction of both the basic file operations and the traditional languages. This level is sufficiently low to make all conceivable database operations possible, and at the same time sufficiently high to make them simple and convenient – for an experienced programmer, at least. We can so easily implement any database requirement using ordinary features, available in most languages, that it is silly to search for higher-level operations.

High-level database operations offer no benefits, therefore, for two reasons: first, because we can so easily implement database requirements using the basic file operations, and second, because it is impossible to have built-in operations for all conceivable situations. No matter how many high-level operations we are offered, and no matter how useful they are, we will always encounter requirements that cannot be implemented with high-level operations alone. We cannot give up the lower levels, thus, because we need them to implement details, and because the links between database operations, and also between database operations and the other types of operations, occur at the low level of these details.

So the idea of higher levels is fallacious for database operations in the same way it is fallacious for the other types of operations. This was also the idea behind the so-called fourth-generation languages (see pp. 452–453). And, like the 4GL systems, the relational systems became in the end a fraud.

The theorists start by promising us higher levels. Then, when it becomes clear that the restriction to high levels is impractical, they restore – in the guise of enhancements – the low levels. Thus, with 4GL systems we still use such concepts as conditions, iterations, and assigning values to variables; in other words, concepts of the same level of abstraction as those found in a traditional language. It is true that these systems provide *some* higher-level operations (in user interface, for instance), but they do not eliminate the lower levels. In any case, even in those situations where operations of a higher level are indeed useful, we don't need these systems; for, we can always provide the higher levels ourselves, in any language, through subroutines. Similarly, we will see in the

present section, the relational database systems became practical only after restoring the low levels; that is, the traditional file management concepts.

In conclusion, the software elites promote ideas like 4GL and relational databases, not on the basis of any real benefits, but in order to deprive us of the programming freedom conferred by the traditional languages. Their real motive is to force us to depend on expensive and complicated development systems, which they control.

I want to stress again that remarkable quality found in the basic file operations, the fact that they are at the same level of abstraction as the operations provided by the traditional programming languages. This is why we can so easily link these operations and implement database requirements. One of the most successful of all software concepts, this simple feature greatly simplifies both programming and the resulting applications.

There is a *seamless integration* of the database and the rest of the application, for both data and operations. The fields, the record area, and the record keys function as both database entities and memory variables at the same time. Database fields can be mixed freely with memory variables in assignments, calculations, or comparisons. Transferring data between disk and memory is a logical extension of the data transfers performed in memory. Most statements, constructs, and methods we use in programming have the same form and meaning for file operations as they have for the other types of operations; iterative and conditional constructs, for example, are used in the same way to scan and select records from a file as they are to scan and select items from an array or table stored in memory.

Just by learning to use the six basic file operations, then, a programmer gains the means to design and control databases of any size and complexity. The most difficult part of this work is handled by the file management system, and what is left to the programmer is not very different from the challenges he faces when dealing with any other aspect of the application.

The seamless integration of the database and the application is such an important feature that, had we not already had it in the traditional file operations, we could have rightly called its introduction today a breakthrough in programming techniques. The ignorance of the academics and the practitioners is betrayed, thus, by their lack of appreciation of a feature that has been widely available (through COBOL, for instance) since the 1960s. Instead of studying it and learning how to make the most of it, the software experts have been promoting the relational model, whose express purpose is to *eliminate* the integration. In their attempt to simplify programming, they restrict the links

between files, and between files and the rest of the application, to high levels of abstraction. But this is an absurd idea, as we saw, because serious applications require low-level links too.

Then, instead of admitting that the relational model had failed, the experts proceeded to *reestablish* the low-level links. For, in order to make the relational model practical, they had to restore the integration – the very quality that the relational model had tried to eliminate. But the only way to provide the low levels and the integration now, as part of a database system, is through a series of artificial enhancements. When examined, the new features turn out to be nothing but particular instances of the important quality of integration: they are means to link the database to the rest of the application in specific situations. What is the very nature of the traditional file operations, and in effect just one simple feature, is now being restored by annulling the relational principles and replacing them with a multitude of complicated features. Each new feature is, in reality, a substitute for a particular high-level software element (a particular database function) that can no longer be implemented naturally, by combining lower-level elements.

Like all development systems that promise a higher level of abstraction, the relational systems became increasingly large and complicated because they attempted to replace with built-in operations the infinity of alternatives that we need at high levels but can no longer create by starting from low levels. Recall the analogy of software with language: If we had to express ourselves through ready-made sentences, instead of creating our own starting with words, we would end up depending on systems that become increasingly large and complicated as they attempt to provide all necessary sentences. But even with thousands of sentences, we would be unable to express all possible ideas. So we would spend more and more time trying to communicate through these systems, even while being restricted to a fraction of the ideas that can be expressed by combining words.

Thus, the endless problems engendered by relational database systems, and the astronomic cost of using them, are due to the ongoing effort to overcome the restrictions imposed by the relational model. They are due, in the end, to the software experts, who not only failed to understand why this model is worthless, but continued to promote it while its claims were being falsified.

The relational model became a pseudoscience when the experts decided to "enhance" it, which they did by turning its falsifications into features (see "Popper's Principles of Demarcation" in chapter 3); specifically, by restoring the traditional data management concepts. It is impossible, however, to restore the seamless integration we had before. So all we have in the end is some complicated and inefficient database systems that are struggling to emulate the simple, straightforward file systems.

The Theory

1

To understand the relational delusions, we must start with a brief review of *formal logic* – that branch of mathematics upon which the relational model is said to be founded.

Formal logic is treated as a branch of mathematics because its exact principles and its deductive methods are similar to those of traditional mathematics. For this reason, it is also called *mathematical logic*. But, whereas algebra and calculus deal with numerical values, and geometry with lines and planes, logic deals with *truth values*: assertions that can be either *True* or *False*. As in other branches of mathematics, the elements and formulas of logic are expressed as variables – abstract entities that stand for a large number of particular instances. Thus, we normally use symbols like *x* and *y*, rather than actual assertions. This is why formal logic is also known as *symbolic logic*.

The oldest system of formal logic is *syllogistics*. Created by Aristotle in the fourth century BC, and further developed over time, syllogistic logic is based on propositions of the form "all *S* are *P*," "no *S* is *P*," "some *S* are *P*," and "some *S* are not *P*." (Examples: all fishes are swimmers, some buildings are tall, some people are not nice.) These propositions consist of two terms (the subject *S* and the predicate *P*) and a quantifier (all, some, none). The propositions assert, therefore, that a certain thing, or a class of things, possess a certain attribute. Additional flexibility is attained by permitting negative terms: "all *S* are *not-P*," "some *not-S* are *P*," and so on. A syllogism consists of three such propositions: two are premises, and the third one is the conclusion. The premises are related through one of their terms, and the conclusion uses the other two terms. (Example: Some *A* are *B*, all *A* are *C*, therefore some *C* are *B*.)

Clearly, many combinations of propositions are possible, but not all constitute valid syllogisms. A syllogism is valid if the conclusion follows by logical necessity from the two premises, as in the classic inference "All men are mortal, Socrates is a man, therefore Socrates is mortal." An example of invalid syllogisms is "Some dogs are vicious, this animal is not a dog, therefore this animal is not vicious" (even if the two premises are true, the conclusion can be either true or false).[1]

The study of syllogisms involves the classification of the various combinations of propositions, their logical relationships and transformations, and the methods for determining their validity. It should be obvious that, if we can

[1] In syllogisms, a reference to an individual entity is interpreted as a class of things that comprises only one element, and therefore implies the quantifier *all*.

reduce an argument to a structure of propositions consisting of subjects and predicates, syllogistic logic allows us to determine *formally* whether a particular statement can or cannot be inferred from certain premises. In other words, if we know that the two premises are true, we can determine whether the conclusion is true or false strictly from the *structure* of the three propositions; we don't have to concern ourselves with the *meaning* of the terms that make them up.

Syllogistic logic is seen today as only one of the many systems comprising the field of formal logic. Modern logic, born in the nineteenth century, attempts to extend beyond the capabilities of syllogistics the range of discourse and the types of phenomena that can be represented formally. The benefits of a formal representation are well known: as with traditional mathematics, it allows us to build increasingly large and complex entities that are guaranteed to be valid – simply by combining hierarchically, level after level, entities whose validity is already established. Conversely, if confronted with an expression too complex to understand directly, we can determine its validity by reducing it to simpler entities, one level at a time, until we reach entities known to be valid. Formal logic, thus, permits us to apply the deductive methods of mathematics to any type of phenomena.

We also know what are the *limitations* of formal logic. We can reduce a phenomenon to an exact representation only when its links to other phenomena are weak enough to be ignored. If we recall the concept of simple and complex structures, logic systems allow us to create only *simple* structures; so they are useful only for phenomena that can be studied in isolation. While common in the natural sciences, this is rare for phenomena involving human minds and societies. In chapter 4, for example, we saw the attempts made by scientists to represent *knowledge* by means of logic systems. These attempts fail because the entities that make up knowledge are connected in many ways, not just through the hierarchical relations recognized by a particular logic system. These entities can only be represented, therefore, with a *complex* structure. To this day, few scientists are ready to admit that most human phenomena *cannot* be reduced to an exact, formal model. So they keep inventing one mechanistic theory after another, hoping to represent mathematically such phenomena as intelligence, language, and software.

Although differing in complexity and versatility, the modern systems of logic have a lot in common. To create a system of logic, we start by defining its basic entities – those entities that act as starting elements in the hierarchical structures created with that system: objects, propositions, etc. Logical *variables*

(single letters, usually) are used to represent these entities in definitions and expressions. Next, we define a set of logical *operations* – the means of creating the elements of one level by combining those from the lower level. We also need some *rules of inference* – principles that justify the various transformations performed when moving from one level to the next. These rules serve, in effect, to restrict the use of operations to those cases where the new element can be derived from the others only through logical deduction. (For example, the rule known as *modus ponens* states that, if we know that whenever p is true q is also true, then if p is found to be true we must conclude that q is true.) Lastly, we agree on a number of *axioms*. Axioms are assumptions taken to be valid by convention, and which can be employed therefore in logical expressions just as we do premises. (A common axiom, for example, is the assertion that any entity is identical to the negation of its negation.) The theorems of a logic system are the various assertions that can be proved deductively within the system by manipulating expressions. Clearly, increasingly complex expressions and theorems can be constructed by combining elements hierarchically, on higher and higher levels of abstraction.[2]

Despite their formality, there is considerable freedom in designing a logic system. For example, what is a rule in one system may be an axiom in another, and what is a theorem may be a rule. What matters is only that the system be *consistent*. A system is consistent when no contradictions can arise between the expressions derived by means of its operations, rules, and axioms. That is, if we can show that a certain expression or theorem is true, we should not be able to show in the same system, through a different deduction, that it is false. Another quality found in a correct logic system is that of *independence*: every one of its axioms and rules is necessary, and none can be derived from the others. To put this differently, if any one of them were omitted, we would no longer be able to determine the truth or falsity of some expressions or theorems.

The chief difference between logic systems, then, is in their basic entities, and in the way these entities are combined to create correct expressions (what is known as *well-formed formulas*). And, once we reach a hierarchical level where expressions can only yield truth values, *True* or *False*, the same operations can be used to manipulate them in any logic system. The starting elements themselves may be entities restricted to truth values, but many systems have starting elements of other types. In syllogistic logic, we saw, the

[2] Note that, when used with the simple structures created with logic systems, the term “complexity” is employed here (as it always is when discussing simple structures) to indicate the shift to a higher level within a structure, or to a structure with more levels. (The levels of complexity in a simple structure are its levels of abstraction.) So don’t confuse this complexity with the complexity of complex structures, which is due to the interaction of structures.

starting elements are subjects and predicates (things and attributes), and only their combinations are propositions that can be true or false.

The most common logical operations, thus, are those that manipulate truth values. And among them, the best known are conjunction, disjunction, and negation (AND, OR, and NOT). Only conjunction and negation are usually defined as basic operations, though, since disjunction can be expressed in terms of them: *A* OR *B* is equivalent to NOT(NOT *A* AND NOT *B*). Additional operations (equivalence, implication, etc.) can be similarly defined in terms of conjunction and negation, or by combining previously defined operations. A *truth function* is an expression involving operands that have truth values, so its result is also a truth value. This result can then act as operand in other expressions.

One way of determining the result of a truth function is with *truth tables*. A truth table has a column for each operand used by the function, and a row for each possible combination of truth values (hence, two rows for one operand, four for two operands, eight for three operands, etc.). A final column depicts the truth value of the result, and there may be additional columns for intermediate results. (Figure 4-2, p. 330, illustrates the concept of truth tables.)

Another characteristic common to all logic systems is that the validity of their low-level elements, and of their axioms and premises, cannot be determined from *within* the system. A logic system only guarantees that, if certain expressions are known to be true or false, then the truth or falsity of other expressions – derived from the original ones strictly through the rules and operations permitted by the system – can be determined with certainty. It cannot verify for us whether the expressions we *start* with are true or not.

For example, a premise like "*A* is larger than *B*" could be used with numbers in one application and with animals in another. In either case, it would be true in some instances and false in others. But within the logic system, this statement appears simply as a symbol, say, *S*; and it is handled the same way regardless of what *A* and *B* stand for, or whether the statement is false while believed to be true. It is our responsibility to ensure that it is true – by means *external* to the system – before using it as premise in a particular application.

Logic systems, then, are only concerned with the *form* and *structure* of elements and expressions, not with their *interpretation*. Needless to say, though, both aspects are important in actual applications. If all we want is that the conclusion be sound logically, its correct deduction from premises is indeed sufficient. But for the system to be of practical value, the deduction *and* the premises must be correct.

Thus, along with their limitation to simple, isolated phenomena, their dependence on what is usually just an *informal* verification of premises and starting elements reduces considerably the usefulness of formal logic systems

in real-world applications. The delusions of the relational database model, for instance, stem from overlooking the severity of these limitations, as we will soon see.

The simplest system of logic is the one known as *propositional calculus*. The basic elements in this system are whole propositions, and expressions are formed by combining propositions through logical operations, as described earlier. Although expressions of any complexity can be formed in this manner, this system is handicapped by its inability to analyze the individual propositions. For example, if two propositions comprise subject and predicate, as in syllogisms, the system cannot distinguish between the case where the propositions share their subject or predicate, and the case where they are unrelated. The chief quality of propositional calculus is its simplicity, so it is the system of choice in applications where the elements can be treated as either atomic entities or logical expressions built from these entities. The Boolean logic system, upon which digital circuits and many software concepts are founded, is a type of propositional calculus.

A more versatile system of logic, and the one that served as inspiration for the relational database model, is *predicate calculus*. The basic elements in this system are subjects and predicates, as in syllogistic logic, but a predicate can be shared by several subjects in one proposition. A predicate, in other words, is seen as an attribute that can be possessed by one, two, three, or generally *n* different elements. And when possessed by more than one, it serves not only as attribute, but also to relate them. Each set of *n* elements related through a predicate is known as an *n-tuple* (or *tuple*, for short).

An expression like *P(x,y,z)* – which says that the elements *x*, *y*, and *z* are related and form a 3-tuple through the predicate *P* – is a basic proposition in predicate calculus. Since the elements are represented with variables, the expression stands for any number of such tuples. Each element has its own domain of permissible values, and when we substitute actual values for the three variables, the relationship will generally hold for some combinations of values but not for others. So the expression will be true for some tuples and false for others. The totality of tuples that share a particular predicate (or, usually, just those for which the expression is true) is called a *relation*.

An example of a relation is the sets of three integers, *a*, *b*, and *c*, each one selected perhaps from a different range of values, and fulfilling the condition that *a* is greater than *b* and *b* is greater than *c*. An expression like *G(a,b,c)*, representing this relationship, is then true for some sets of values and false for others. Another example is the sets of five men, *p*, *c*, *b*, *n*, and *g*, who could

have been selected from various domains to act as crews in WWII B-25 bombers: pilot, co-pilot, bombardier, navigator, and gunner. An expression like $B(p,c,b,n,g)$, representing this relationship, is true only for those sets of men that formed actual crews.

Basic propositions can be combined by means of logical operations, in the usual manner, to form more complex propositions. Thus, we can form relations that are a logical function of other relations. Take, for example, these two relations: $P(x,y)$ as the sets of two elements, x and y, related through P; and $Q(y,z)$ as the sets of two elements, y and z, related through Q. The expression $P(x,y)$ AND $Q(y,z)$ may then be defined to mean, depending on the application, either the sets of two elements common to P and Q, or the sets of three elements, x, y, and z, for which both relations hold. Similarly, the expression $P(x,y)$ OR $Q(y,z)$ may be defined to mean either the sets of two elements that exist in either relation (excluding duplicates), or the sets of three elements for which either relation holds.

Additional flexibility can be achieved in expressions by binding each variable with the *universal* quantifier $\forall$ (which says that the relation holds for *all* instances of that variable) or with the *existential* quantifier $\exists$ (which says that the relation holds at least for *some* instances of that variable). These quantifiers become then part of the expression, and participate in operations and transformations, much like operators. Thus, if $\forall$ is applied to both x and y in the expression $R(x,y)$, the expression is true only if the relation R holds for all possible pairs of values of x and y; but if $\exists$ is applied to x and y, the expression is true even if the relation holds for just one pair of values.

This brief review will suffice for our purpose, to assess the mathematical merits of the relational database model. It is worth mentioning, though, that many other systems of logic have been designed. The system we have just examined, for example, is called *first-order* predicate calculus, and is only the simplest of the predicate calculi. (In higher-order systems, the quantified variables and the predicates can themselves be logical expressions.) Some logic systems include special axioms, rules, and operations to deal with such imprecise concepts as necessity, possibility, and contingency, which lie outside the scope of propositional and predicate calculi. Other systems attempt to deal with propositions whose truth value changes over time, and some systems even attempt to reduce to logic such moral issues as belief, obligation, and responsibility.

As I have already stated, the motivation for these systems is to bring phenomena involving minds and societies into the range of phenomena that

can be represented with formal, mechanistic methods. And they have had very little success, because few human phenomena are simple enough to be reduced to a mechanistic representation.

Programming phenomena are largely human phenomena. So the relational model is, ultimately, an example of the attempts to find a mechanistic model for phenomena that are, in fact, too complex to represent mechanistically. Thus, apart from our interest in the theories of software engineering as pseudosciences in their own right, their analysis complements our study of mechanistic delusions, and serves to remind us of the degradation of the idea of research. We saw in chapters 3 and 4 the childish attempts made by some of our most famous scientists to represent behaviour, intelligence, and language with diagrams, or formulas, or logic. And the same fallacy is committed with *software* theories: the mechanists discover a model that explains *isolated aspects* of a complex phenomenon, and they interpret this trivial success as evidence that their theory is valid, and hence worth pursuing.

So, by invoking the official definition of science – which is simply the pursuit of mechanistic ideas, whether useful or not – academics can now spend their entire career developing worthless theories. Merely because mechanism works in fields like physics or chemistry, they feel justified to seek mechanistic explanations in psychology, or sociology, or linguistics, or economics, or programming. Then, because of our mechanistic culture, we admire and respect them, and regard their activities as serious research – even as we see that their theories never work, and that they resort to deception in order to defend them.

2

Let us see now how predicate calculus was adapted for database work. The inventor of the relational model is E. F. Codd, who presented his ideas in a series of papers starting in 1969.[3] We are not concerned here with the evolution of the model in the first few years, or with the specific contributions made by individual researchers, but only with the relational database ideas in general. And, in fact, apart from a few refinements, the theory presented by Codd in his original papers depicts quite accurately what became in the end the formal relational model. In 1981, Codd received the prestigious Turing award for his invention.

[3] The first paper was published in 1969 (as an IBM document), but it was only the second one, published the following year, that was widely read: E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387.

Recall the organization of data files as records, and fields within records. To represent a file by means of predicate calculus, the fields are seen as the basic elements of a logic system, and the records as tuples – sets of *n* elements, where *n* is the number of fields in a record. Each field can possess a value from a domain of permissible values. Thus, if the field is a part number, the domain is all valid part numbers; if a vendor name, the names of all possible vendors; if a quantity, all numeric values that are valid quantities; and so on. Generally, some combinations of values exist as actual records in the file, and others do not; and, *by convention*, the relationship that links these fields holds only for those combinations that exist. In other words, an expression representing this relationship is deemed to yield the value *True* when the tuple actually exists in the file, and the value *False* otherwise. As in logic, the totality of tuples (i.e., records) in the file is called a relation.

To define an employee file with four fields, for instance, we would use a logical expression like *E(a,b,c,d)*, where *a* is the employee number, *b* the name, *c* the hourly rate, and *d* the number of hours worked. *E* stands then for the predicate that relates the four fields. *E* says, in effect, that each set of four values, taken from the respective domains of permissible values (all possible employee numbers, names, rates, and hours), are related in such a way that they represent a potential employee. The expression is true for the sets that actually exist in the file, and false for the others.

To this basic system, which matches the system of predicate calculus, a number of features were added in order to make the relational model suitable for database work. One feature is the idea of *field names*. In predicate calculus, the elements are identified by their relative position within the tuple, but this is impractical for database fields. Fields, therefore, are given names (QUANTITY, VENDOR-NO, INVOICE-DATE, etc.); we can then refer to them by their name, so their relative position within the physical record (as they are stored on disk, for example) is immaterial. These names are sometimes described as a special tuple that exists in every file but does not take part in operations; its function is similar to the top row in a typical table – the row that contains the column headers.

Another feature is the idea of a *key*: one field in the record is designated as key, and its value in each record must be unique within the file; alternatively, the key can consist of several fields, and then their combined values must be unique. The key, therefore, can be used to identify a specific record within the file, or to order the records in a logical sequence (so the actual sequence of records, as they are stored on disk or as they were added to the file, is irrelevant to the application). It is often useful to have several keys for the same record; in this case, one is designated as the *primary* key, and the others are called *candidate* keys. Lastly, in order to relate files, a field (or a group of fields) can

be designated as a *foreign* key. This type of key is used to identify the records of another file, where that field usually functions as the primary key. The customer number in an invoice file, for example, is a foreign key that relates the invoice file to the customer file, where the customer number is the primary key. The values stored in a foreign key need not be unique in each record; thus, we can have several invoices with the same customer number.

It should be obvious, if you recall our discussion of indexed data files and the basic file operations, that the relational concepts we have examined so far are *identical* to the traditional data file concepts. The only difference is in the use of terms like "relation" and "tuple" instead of the terms traditionally associated with data files. The relational theory is rich in new terminology. Thus, in addition to the concepts and terms taken from logic, we are told that files are best perceived as tables: the rows of these tables are then the records, and the columns are the fields. Also, the term *attribute* is often used for columns. So the accepted relational terms are *tables* and *relations*, *rows* and *tuples*, *columns* and *attributes*.[4]

While tables still resemble the traditional data files, the way we access them is entirely different. The traditional file operations are based on indexes, and are used through the flow-control constructs of a programming language. The relational operations, on the other hand, are defined in the manner of logical operations. In predicate calculus, we saw, operations like AND and OR take relations as operands and produce a new relation; similarly, the relational operations take tables as operands and produce a new table.

In predicate calculus, the tuples in the resulting relation consist of variables that were elements in the tuples of the original relations. When the same values that made up the tuples of the original relations are substituted for the variables of the new tuples, the expression that represents the new relation may be true for some combinations and false for others; and the new relation is defined as those tuples for which the expression is true.

Similarly, the operations in the relational model are defined in such a manner that the columns of the resulting table are selected from among the columns of the original tables. Then, depending on the operation and the values present in the rows of the original tables, only *some* of the rows are retained in the new table. In other words, each operation has its own definition

[4] Generally, *tables*, *rows*, and *columns* are considered informal terms, while *relations*, *tuples*, and *attributes* are the formal ones. Since the entities described by these terms are identical to the traditional *files*, *records*, and *fields*, I am using both the new and the traditional terms in the discussion of relational databases.

of truth and falsity, and if we represent the rows with a logical expression, the new table is defined as those rows containing combinations of values for which the expression is true. For example, if we represent a customer table as *C(a,b,c,d)* and an orders table as *O(a,e,f)* (where the lower-case letters stand for columns, and *a* is the customer number), a particular operation could be defined as follows: create a new table *R(a,b,e)*, whose rows are those pairs of rows from the customer and orders tables where *a* has the same value in both. The expression *R(a,b,e)* is said in this case to be true for these rows (i.e., where the customer matches the invoice) and false for the others. This expression – that is, the new table – can then be combined with others in further operations.

There are five basic operations: The UNION of tables *A* and *B* is a table containing the rows present in either *A* or *B* or both (*A* and *B* must have the same number of columns, and rows common to *A* and *B* appear only once in the new table). The DIFFERENCE of tables *A* and *B* is a table containing those rows present in *A* but not in *B* (*A* and *B* must have the same number of columns). SELECTION takes one table, *A*, and produces a new table containing only those rows from *A* for which an expression involving one of the columns is evaluated as true (for example, only those rows where the value in a given column is greater than zero). PROJECTION takes one table, *A*, and produces a new table containing all the rows from *A*, but only some of its columns. The PRODUCT of tables *A* and *B* is a table whose columns are the columns of *A* plus those of *B*, and whose rows are every combination of rows from *A* and *B*; each row, thus, is built by taking a row from *A* and extending it with a row from *B* (so, for example, if *A* has 10 rows and *B* has 20 rows, the new table will have 200 rows).

Additional operations may be necessary in practice, but they can always be expressed as a combination of the five basic ones. For example, to reduce a table to only some of its rows and columns, we perform first a SELECTION to retain the specified rows, and then a PROJECTION on the resulting table to retain the specified columns. (Note that the order in which we perform these two operations is immaterial.) Most database systems, in line with their promise to give us higher levels of abstraction, provide some of the most common combinations in the form of built-in operations.

PRODUCT, in particular, is rarely useful on its own, and is normally employed as just the first step in a series of operations. JOIN, for instance, consists of PRODUCT followed by SELECTION and then by PROJECTION. JOIN selects from all the combinations of rows in tables *A* and *B* those rows where a particular column in *A* stands in a certain relationship to a particular column in *B*. Most often, JOIN is used to combine two tables on the basis of *equality* of values. For example, the JOIN of a customer table and an invoice table based on the customer number (present in both) will result in a table that has the combined

customer and invoice columns, but (through SELECTION) only the rows where the customer number was the same in both tables. JOIN, thus, will match invoices and customers: it will have one row for each invoice, and each row will include the customer columns in addition to the invoice columns. (The PROJECTION in the last step serves to eliminate one of the two columns containing the customer number, since they are identical.) JOIN can be performed on key columns as well as non-key columns, and its chief use is to relate files.

For most operations and combinations of operations, we can understand intuitively how the resulting table is derived from the original ones. It is possible, though, to define these operations formally, as transformations based on the operations of formal logic. There are several ways to do it: the relational *algebra* describes them as operations on tables, as we just saw; the relational *calculus* – of which there are two versions, *tuple* calculus and *domain* calculus – describes them with logical expressions similar to those used in predicate calculus. The operations are the same; only the way they are described differs.

The value of the relational model is due to expressing the data in the resulting table as a logical expression of the data in the original tables. Thus, if the original data is correct, the final data will also be correct. When permitted to combine records and fields at will – with the traditional file operations, for instance – a programmer may make mistakes and generate files that do not reflect correctly the original data, or generate files containing inconsistent data. This cannot happen with a relational database. Because we are restricted to operations on whole tables, and because the relational operations introduce no spurious dependencies between fields, we can be certain that the final table will express the same data and relationships as the tables we start with. In a database query, for instance, no matter how many operations and combinations of tables are involved, the final entities are guaranteed to be the same as the original ones – only arranged differently. It is impossible, in fact, to generate wrong or inconsistent data if we restrict ourselves to the relational operations.

The relational model permits us to implement any database requirements that we are likely to encounter in applications. The restriction to whole tables is not really a handicap, because any portion of a table – any subset of rows and columns – is itself, in effect, a table. We can even isolate a single row (with appropriate SELECTIONs), and that row is treated as a table and can be used in further operations. Even one column of that row can be isolated (with a PROJECTION), and that single element still is, as far as the relational operations are concerned, a table.

Note that the resulting table need not be a real entity. When using SELECTION to answer a query, for example, the database system may simply *display* the

selected rows, without actually creating a table. Generally, to perform a series of operations, the system may create some intermediate tables or use only the original ones, and may employ indexes or other expedients. But we don't have to concern ourselves with these details, because a good system will automatically discover the most effective alternative. All we need to do is specify, through the relational algebra or calculus, the original tables and the desired operations.

What we gain with the restriction to tables, then, is simplicity and accuracy: all we need now is a few operations, which are founded on formal logic and can be safely combined into more complex ones. Just as importantly, these operations permit us to view the data from a higher level of abstraction: we no longer need to access individual records, as in the traditional file systems; nor do we need file scanning loops, or intricate conditions to select records and to relate files. Whether our requirements involve single tables, or combinations of tables, or portions of tables, or just one row or column, all we need now is the high-level relational operations. Thus, a relational database is said to be "tables and nothing but tables."

3

It is not enough for the database *operations* to conform to a logic system. The relational theory also requires that the data be *stored* in a logical format. Specifically, the fields that make up the individual tuples must be simple, indivisible entities, with no unnecessary dependency between them. Tables that adhere to this format are said to be *normalized*, and the process of bringing them to this format is called *normalization*. There are several levels of normalization, each one a more stringent enforcement of these principles. The levels are known as first normal form, second normal form, third normal form, etc., and are abbreviated as 1NF, 2NF, 3NF, etc.

The fundamental requirement is 1NF. For a table to be in first normal form, each field must be a simple entity – a single, atomic value. In traditional data files, a field may consist of a series of values, or a multidimensional array of values, or a hierarchical structure of values. A twelve-month transaction history, for example, can be stored in one field of a customer record as an array of twelve rows by three columns – month, quantity, and amount. The relational model prohibits this format: data that comprises a set of related values must be stored in a separate table, where each value has its own column. Thus, to reduce the customer table just described to 1NF, we must create a separate table for the transaction history. The columns in this table will be the customer number, month, quantity, and amount, and the key will be the combination of customer

number and month. For each row in the customer table there will be twelve rows in the transaction history table.

It should be obvious why the first normal form is so important. The relational operations expect to find tuples, and cannot process multiple values – data stored, in effect, as tuples within tuples. To deal with this format we need a more complex database model, and operations that can process more than just rows and columns.

The other normal forms deal with the problem of *field dependency*; specifically, the dependency of one field on another within a tuple. Since such a relationship is likely to cause data *redundancy* and *inconsistencies*, the only type of relationship permitted between fields within a tuple is the obvious dependency of the tuple's fields on the tuple's unique key. All other field relationships must be implemented by moving the fields to other tables and linking the tables logically.

An example of misplaced dependency is a customer orders table where the key is the combination of customer number and order number, and the other fields are the customer name and address, and the order date, quantity, and amount. All these fields depend on the customer number; but, whereas order-specific data like quantity and amount must indeed be included in each order, fixed customer data like name and address must not. This design is wrong because, while a customer's name and address are the same for all his orders, we *repeat* them in every order. The faulty design, thus, will cause data redundancy. Worse still, it will cause various inconsistencies ("anomalies," in relational terminology) when we run the application: first, if a customer's name or address changes, we may have to update not one but several rows – all his outstanding orders; second, we can store a customer's name and address in the database only if that customer has at least one outstanding order.

The solution, of course, is to store the name and address in a separate table, where the key is the customer number and there is only one row per customer. From the order rows we can then access the appropriate name and address by using the customer number as link. The process of normalization, thus, consists in creating two tables from one. In general, we eliminate a misplaced dependency by increasing the number of tables: we extract the fields with repeated values and place them in a separate table, where we discard the duplicate rows; then we choose a field (or a combination of fields) with unique values to act as a key for the new table and as a link to the original one.

The redundancy and inconsistencies were caused, obviously, by an incorrect design – a design that did not match the application's requirements: the name and address are the same for all orders, and yet we repeated them in every order. With the fields in a separate table, the design matches the requirements, and consequently there is no redundancy or inconsistency.

The normalization theory, however, describes the problem of incorrect design as a problem of misplaced dependency: the name and address depended on only a portion of the key (the customer number), instead of depending on the whole key (the combination of customer and order numbers), as do the order date, quantity, and amount. And we correct this dependency by placing the name and address in a separate table – a table where the customer number is the whole key. Clearly, what we do is the same as before, match the design to the requirements; but the normalization theory describes this process as the elimination of misplaced dependencies.

Few people would design a database with the kind of redundancy we have just examined. The theorists, nevertheless, treat the subject of normalization very seriously. Various types of field dependencies are defined and studied in great detail, along with the steps required to eliminate them. Thus, five types were discovered, each one more rare and more subtle. Tables, we saw, are already in first normal form when their fields are single elements. Then, after eliminating one type of dependency, they are in second normal form (2NF). After eliminating a second type, they are in third normal form (3NF). This is followed by a level known as Boyce/Codd normal form (BCNF), and then by the fourth and fifth normal forms (4NF and 5NF). When in fifth normal form, tables are in the ultimate relational format, devoid of any misplaced dependencies. Very few databases, however, require all five levels of normalization. If an application is not complicated, tables will likely be in their highest possible normal form after just one or two levels, simply because there are no other dependencies. In any case, many experts consider 3NF or BCNF adequate, and don't even mention 4NF and 5NF.

The second and higher normal forms are in reality very similar, and their differences need not concern us here. The reason for having several types of normalization and a numbering system is largely historical: while the first normal form was described by Codd in his original papers, the others were incorporated into the relational theory later – as they were discovered, one by one. (More specifically, the higher normal forms became necessary when the relational model was expanded to include *updating* operations.) Thus, it is worth noting that the theorists needed several years, and innumerable papers and conferences, to discover what an experienced programmer could have told them from the beginning. For, the problems caused by misplaced dependencies, as well as their solutions, are *identical* in relational databases and in databases created with traditional data files; only the attempt to treat these problems formally is new. We will return to this subject later, in our discussion of the relational delusions.

The Contradictions

1

To summarize, the relational model is an attempt to turn database programming, as well as database use, into an exact, formal activity. Since data records resemble the so-called tuples of predicate calculus, and since they can be manipulated with operations resembling logical operations, the theorists concluded that the rigour and exactness of mathematical logic can now be attained in database work. All we have to do is restrict the files and records to a certain format, and restrict the operations to a high level of abstraction; we have then a mathematical guarantee that the answers to queries will reflect accurately the data and relationships present in the database.

From the start, then, the relational theory was grounded on the curious principle that only *some* aspects of the database need to be covered by the formal, mathematical model; the others can remain informal. This principle is sometimes expressed with the statement that certain aspects lie *within* the scope of the formal model, while others are *outside* its scope. Thus, if we call "relational model" the whole body of relational principles and features, the "formal relational model" constitutes only a small part of it.

Within the scope of the formal model lie, as we just saw, the format of files and records, the concept of queries, and the use of high-level query operations. The theorists recognize, of course, that there is a lot more to databases and applications. So, while asking us to treat these aspects formally, they expect us to deal with the other aspects of the database *informally*: by relying on traditional programming methods and on personal skills.

In particular, operations that *update* the database – adding and deleting records, modifying the data in fields, creating and deleting files – cannot be treated formally, and therefore lie outside the scope of the formal model. Note that this is a necessary consequence of the model's mathematical foundation: predicate calculus is concerned with the logical expressions that *use* the tuples of a given relation, not with the way the tuples became part of that relation, or with the way the elements of these tuples acquired their current value. Thus, if the updating of tuples and relations lies outside the scope of predicate calculus, it must also be left out of the formal relational model.

Data normalization, too, is largely informal. Only the first normal form, which deals with the record format, is part of the formal model. The second and higher normal forms are only needed in order to prevent redundancy and inconsistencies when *updating* the files; thus, if the updating operations are informal, so must be the normalization. In any case, the process of normalization entails an *interpretation* of the application's requirements:

whether or not a certain field depends on another can be determined only from the way we intend to use them in the application, something that no formal system can know.

Another aspect of the database that cannot be formalized concerns *data integrity* – the countless rules that ensure the validity of the updating operations within the context of a particular application. Again, what is valid in one case may be invalid in another, and only *we* can decide how to interpret the result of a certain operation.

Lastly, the formal model does not include the means we use to *specify* the query and updating operations. These means – a set of commands, or a database language – can only be used informally. As is the case with any programming language, we can define with precision the commands or statements themselves, but not their effect when combined to perform a particular task in a given application.

In conclusion, the updating operations, the normalization process, the integrity rules, and the database language, even though needed in any application that uses relational databases, lie outside the scope of the formal relational model. So they can be no more formal or exact than they are in applications using traditional data files.

What, then, is the meaning of the relational model? What is the point, for instance, of including in the formal model the query operations while excluding the updating operations? Clearly, the two types of operations are equally important in an application. What is the value of a model that guarantees correct answers to queries while being unable to guarantee the correctness of the data upon which the queries are based?

The theorists acknowledge that the formal model is insufficient, that we must depend on some informal operations too, but they fail to appreciate the implication: if we must deal with certain aspects of the database by relying largely on personal knowledge, the inexactness of this method will annul the exactness of those aspects treated formally. The result of a process cannot be more exact than the least exact of its parts. The answer to a query may well be mathematically derivable from the original data, but this quality of the relational model has little value if we cannot prove that the original data is correct to begin with.

We just saw how, in order to attain a practical relational model, the theorists were compelled to separate it into a formal and an informal part. But this is not all. There is one aspect of the database that is considered to lie, not only outside the scope of the *formal* model, but outside the scope of the relational

model altogether: the actual, physical implementation of the database and operations.

Like the logic system that inspired it, the relational model is a mathematical, and hence abstract, concept. This limitation, however, is interpreted by the theorists as a *quality*: thanks to its abstract nature, they say, we no longer need to be concerned with such issues as the system's *performance* (the time required to execute the database operations). In general, the independence of the logical database structures from their physical implementation permits us to access the data from a higher level of abstraction. Here are some statements expressing this view: "The ideas of the relational model apply at the external and conceptual levels of the system, not the internal level. To put this another way, the relational model represents a database system at a level of abstraction that is somewhat removed from the details of the underlying machine."[1] "The eight relational operators express functionality without concern for (or knowledge of) technical implementation. An obvious benefit is that relational users apply relational operators without concern for storage and access techniques."[2] "The aim of the relational model is to represent logically all relationships, and hence alleviate the user from physical implementation details."[3] "The relational data model removes the details of storage structure and access strategy from the user interface."[4]

The operations of a mathematical system are assumed to occur instantaneously; we don't think of addition or multiplication, for instance, as physical processes that may take some time. Similarly, the high-level operations of the relational model – selection, projection, join, and the rest – are assumed to be executed instantaneously by the database system. Incredibly, while presenting the relational model as the foundation of practical database systems, the theorists insisted that the subject of performance lies outside the scope of the model. Everyone knew, of course, that the application's performance is limited by the speed of the computer's processor, and that databases rely on physical devices like disk drives, which impose additional speed limits on data access. Nevertheless, the claim that it is possible to design real databases without having to concern ourselves with their performance was received with enthusiasm. All we need to do, promised the theorists, is specify the relational

[1] C. J. Date, *An Introduction to Database Systems*, 6th ed. (Reading, MA: Addison-Wesley, 1995), p. 98.

[2] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design* (Reading, MA: Addison-Wesley, 1989), p. 38.

[3] M. Papazoglou and W. Valder, *Relational Database Management: A Systems Programming Approach* (Hemel Hempstead, UK: Prentice Hall, 1989), p. 30.

[4] Ken S. Brathwaite, *Relational Databases: Concepts, Design, and Administration* (New York: McGraw-Hill, 1991), p. 26.

operations, just as we do in mathematics. The database system will analyze the request, determine the most efficient implementation, and then execute the necessary low-level operations.

Separating the performance issue from the relational model is just as illogical as separating the updating operations from the query operations. It shouldn't come as a surprise, therefore, that the relational database systems have proved to be incurably slow, and that, in addition, their users have remained as preoccupied with the performance issue as those who use the traditional file operations.

It is absurd to expect the database system itself to know what is the most efficient implementation of a high-level request. It is absurd because most requests do not depend on database structures alone, but also on such other structures (i.e., aspects) of the application as its various processes (see pp. 345–346). To discover the most efficient implementation we must link, therefore, the database structures with the other structures that make up the application. These links, moreover, occur usually at the low level of database fields, memory variables, and individual statements; so the only way to implement them is through traditional programming means.

The inefficiency caused by the lack of low-level links, then, was the main reason for the continued preoccupation with the performance issue. And this inefficiency was also the reason for annulling, in the end, two fundamental relational principles: the restriction to normalized files, and the restriction to high-level operations.

We know, of course, why the theorists separated the relational model into formal and informal aspects: because this is the only way to attain a precise, mechanistic representation of the database and the database operations. If we want to represent an indeterministic phenomenon with a deterministic theory, we must exclude from the phenomenon those aspects that prevent such a representation.

Thus, we can start with any phenomenon, no matter how complex, and invent an exact theory – a mathematical model – that depicts what we *wish* the phenomenon to be. Then, we match the phenomenon to the theory by eliminating, one by one, those aspects that contradict the theory – by branding them as "informal" parts of the phenomenon. If we eliminate enough aspects, we are certain to reduce the phenomenon eventually to a version that is simple enough to match the theory.

But this is a trivial accomplishment – we knew all along that it could be done. It is impossible, in fact, to *fail* in this project, if we place no limit on the

number of aspects that we are willing to eliminate. Mechanistic projects of this nature are, therefore, intrinsically pseudoscientific. This is true because the concept of separating the phenomenon into aspects that are, and aspects that are not, within the scope of the model is unfalsifiable: since we are free at any moment to exclude any number of additional aspects in order to make the model work, there is no condition under which we can say that a mechanistic model *cannot* be found.

The issue, then, is not whether we can find a mathematical model for the phenomenon of a database, as this is always possible by simplifying the phenomenon. Rather, the issue is whether, by the time we simplify the phenomenon sufficiently to have an exact model, such a model is still meaningful. It is quite easy to discover mathematical models for *individual aspects* of software phenomena. The theorists happened to discover a database model grounded on predicate calculus, but with a little imagination we could find any number of other models. The real challenge, again, is not to discover a mathematical model by simplifying the phenomenon, but to discover a useful model for the original, complex phenomenon.

So, like all mechanistic delusions, the relational model failed because its mathematical foundation is insignificant: we can represent mathematically only a small fraction of the concepts involved in programming and using a database. If we were to rely on the original relational model, we would perhaps enjoy the promised benefits, but only with small and simple databases; we would be unable to develop the kind of databases we need in real applications.

Having failed as a practical concept, the relational model was rescued by expanding its *informal* aspects – precisely those aspects that had been excluded from the formal, mathematical model. The early works discuss in detail the formal model, including the various types of query operations, but mention only briefly the informal aspects – the updating operations, the integrity and performance problems, and the database language.[5] These aspects are presented merely as miscellaneous features needed to support the formal model in actual applications. In the end, however, it is precisely these features (the database language, in particular) that became the main concern of relational database systems, while the formal model declined in importance and became practically irrelevant.

Specifically, the restriction to normalized files and the restriction to high-level operations were both lifted; and no one, of course, is using databases through mathematical logic. Today's relational systems are promoted by praising the power of their programming language (usually SQL), the power of

[5] See, for example, Codd's original paper, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387. The other informal aspect (the second and higher normal forms) is not mentioned at all.

certain features described as integrity functions (but whose true role is to bypass the limitations of the high-level operations), and the power of a variety of new data formats and *low-level* file operations. In other words, while the power of the original model was said to derive from its formal, mathematical foundation, the power of what is seen today as the relational model derives entirely from *informal* concepts – concepts that are practically identical to the traditional ones. We will analyze this degradation in "The Third Delusion."

2

When studying the relational model and its evolution from a mechanistic fantasy to a pseudoscience, we can distinguish three major delusions. These delusions are summarized below; then, in the following subsections, we will study them in detail.

The first delusion is the belief that the relational model's mathematical background is an important quality. It is true that the model is grounded upon certain mathematical principles, and that these principles guarantee the soundness of certain database operations. But this quality constitutes an insignificant part of the phenomenon of a database: we can ground a database on mathematics only after limiting the data to a certain format, after separating the database from the rest of the application, and after restricting its use to queries expressed through high-level operations. Important aspects – the operations that modify the database, and the links to the rest of the application's logic – are not included in the mathematical model. Thus, we must deal with the most difficult aspects of database programming *informally*, just as we do when using the traditional file operations.

The second delusion is the belief that the principles of normalization are an essential part of the relational theory. In reality, data normalization is a totally useless concept, even *within* the relational model. It is a contrived theory that attempts to eliminate data redundancy and inconsistencies by identifying misplaced field dependencies. But misplaced dependencies occur only in an incorrectly designed database. So, in order to justify the need for normalization, the theorists ignore the application's requirements and deliberately create an incorrect database; then, they use normalization to convert it into a correct one. The theorists also delude themselves when claiming that they have turned database design into a formal, exact procedure. All they do, in fact, is discuss formally their invention, the various types of field dependencies; the design problem itself has remained as informal as before.

The third delusion emerged when the relational model was found to be impractical. In order to reduce them to a mathematical representation,

the database format, relationships, and operations were simplified so much that very few actual requirements could be implemented. Consequently, the theorists were compelled to "enhance" the model. And this consisted in restoring, one by one, those features that had been eliminated in the first two delusions in order to attain the exact, formal representation. By the time the model was versatile enough to be practical, there was nothing left of the preciseness of the original theory, not even in that narrow domain where the database operations had indeed been mathematical. The third delusion, thus, is in the belief that the original restrictions are not really necessary; in other words, the belief that we can enjoy the benefits of an exact theory without having to adhere to its principles.

Historically, the first two delusions can be said to make up the original idea, while the third one emerged when trying to implement that idea. The first two are a manifestation of the mechanistic fallacies; that is, attempting to represent a complex phenomenon with simple structures. And the third one is the consequence of this attempt. Since the only way to save a fallacious theory from refutation is by making it a pseudoscience, the software experts rescued the relational model by turning its falsifications into what they describe as new relational features. But what they are doing is merely to restore those features which they had eliminated previously in order to make the theory mechanistic. So, if the first two delusions demonstrate the naivety of the software experts, the third one demonstrates their dishonesty: they continue to praise the benefits of the relational model even while annulling the relational concepts and replacing them with the traditional ones.

The relational database theory is an excellent example of what I have called *the new pseudosciences*. Even better than structured programming or object-oriented programming, it can serve as a model of the modern mechanistic delusions.

Recall how these delusions evolve. The scientists start by noticing *one* aspect of a complex phenomenon; so they extract, from the *system* of structures that make up the phenomenon, the structure depicting that one aspect. Then, they enthusiastically announce a formal, mathematical theory based on this structure alone – claiming, in effect, that a complex structure can be reduced to a simple one. A further benefit of having only one structure, they say, is that we can choose our starting elements from higher levels of abstraction – an expedient that makes it even easier to represent the phenomenon.

But the theory does not represent the phenomenon accurately enough to be useful. So, instead of trying to understand the reason for its failure, the

scientists decide to "improve" it: they suppress the falsifications by reinstating, in the guise of new features, the very features they had previously excluded – features that must indeed be excluded if we seek a mechanistic theory. The purpose of the new features, thus, is to restore some of the original structures, and the links between them. In the end, the theory becomes useful only when enough of the old features are reinstated to allow us to represent the entire complex phenomenon again; that is, when we are allowed to represent it *informally*, the way we always did. The scientists, though, continue to praise the exact, mechanistic qualities of their theory – even as everyone can see that what made the theory useful is the *annulment* of these qualities, and their replacement with complex, indeterministic ones.

The First Delusion

1

The first delusion is the belief in the mathematical merits of the relational model. For more than thirty years, we have been hearing the claim that the relational model is based on mathematical logic, and therefore relational databases benefit from the rigour and precision of mathematics. Although few people actually understand the connection between databases and mathematics, no one doubts this claim. After all, the mathematical benefits are being praised, not just by the vendors of relational systems, but also by university professors, database experts, and professional computer associations. In this subsection I want to show, however, that the claim is a fraud: relational databases do not benefit at all from mathematical logic.

Mathematical systems, which include logic systems, are artificial models invented by us in order to represent with precision various aspects of the world. It is not difficult to invent a mathematical system (see pp. 694–695). Essentially, we define its basic elements, the operations that combine elements from one level of complexity to the next, the rules that control the use of these operations, and the axioms (those assumptions taken by convention to be valid assertions). We can then build increasingly complex expressions and theorems by combining elements on higher and higher levels. For the system to be useful mathematically, it must be consistent: no contradictions should be possible between the expressions or the theorems derived within the system. The basic elements vary with the system: numerical values for the

classical mathematical systems, or subjects, predicates, and propositions for the logic systems.

The more elaborate the system, the more complex its elements and operations. Differential calculus, for example, is more complex than arithmetic or algebra. Since mathematical systems are simple hierarchical structures, a higher complexity means only that the system can have more levels, and more intricate elements at the higher levels, within *one* structure. While still a simple structure, though, a more elaborate system allows us to represent more difficult phenomena.

To use a mathematical system as model, we start by translating the entities and processes that constitute the phenomenon into the entities and operations permitted by the system. Once this is accomplished, we can study the phenomenon by working strictly with the mathematical concepts. We create expressions and higher-level elements, manipulate them in various ways, and finally translate the results back into real entities and processes. With this method, we can explain and predict events that may be difficult or impossible to study directly.

A classic example of a mathematical model is Newton's theory of gravitation: if we represent with mathematical entities and operations the bodies that make up the solar system, their state at a given instant, and the natural laws that govern their motion, we can determine with accuracy their position at any other instant in the past or in the future. Clearly, it would be impossible to accomplish this without a mathematical model.

Imagine now a trivial system, a small subset of traditional mathematics: the basic elements in this system are integers, and the only operations are addition and subtraction. Thus, since expressions are limited to these two operations, the most complex elements possible are still integers. And, even though the system permits any number of levels and hence increasingly complex expressions, because of its simplicity it is unlikely that we will ever need more than a few levels. Nevertheless, while simple, this system is not without practical applications; we can employ it, for example, to create accounting models (if we agree to use only whole dollars). The chief difference between it and the mathematical systems of science and engineering is that the latter reach much higher levels, and much more complex elements and operations.[1]

Turning now to the relational model, we find a modification of the logic system known as predicate calculus. To this system, features like record keys and field names were added in order to adapt it for database work. The simplest

[1] It is worth stressing again that the term "complexity," when used with the simple structures of mathematical and logic systems, refers to *levels of complexity* (also known as levels of abstraction), and it must not be confused with the complexity of complex structures (which is due to the interaction of several simple structures).

elements in the relational model are the fields – called now columns, or attributes. Fields are combined to form records – called now rows, or tuples; and records are combined to form files – called now tables, or relations. Relations can then be combined into expressions by means of standard logical operations (AND, OR, and NOT) and some new, more complex operations (UNION, DIFFERENCE, SELECTION, PROJECTION, and PRODUCT). Relations can be combined in this manner to form increasingly high levels, but the result is still a relation. Relations, thus, are the most complex elements in a relational system. Although more intricate than our system of integers and two operations, it is still very simple – far simpler than the mathematical systems employed in science and engineering.

And herein lies the explanation for the first delusion, why the mathematical background of the relational model is irrelevant. It is true that the relational entities and operations can be defined rigorously, with the same methods and notation we use in mathematics. But this preciseness is specious. The relational definitions resemble perhaps the definitions found in the traditional mathematical systems, but, because the relational model is such a simple system, its formality is superfluous, even silly.

The operations of a mathematical system, and the rules that govern the use of these operations, determine how the elements that make up one level are combined to form the elements of the next level. These combinations become the theorems and expressions possible in the system, and, ultimately, the mathematical representation of a phenomenon when the system is used as model. A formal definition of entities, operations, and rules is important in the *traditional* systems, therefore, because this formality is our only guarantee that the theorems and expressions remain valid as we move to higher levels of complexity. But if in a relational system all we have is some simple elements and operations – some simple transformations of one file into another, or of two files into one – and if we rarely need more than a few levels, the formality is hardly necessary. The concept of files, records, and fields is so simple that we can accomplish the same tasks using nothing more than personal skills.

Let us divide the use of a mathematical system into two parts, *translation* and *manipulation*. The translation is the work required to convert into a mathematical representation the entities and processes that make up the phenomenon, and to convert the mathematical entities back into real entities and processes. The manipulation is the work performed *within* the system, with the mathematical entities alone.

When praising the power of mathematics, it is the manipulation that we

have in mind, not the translation. The translation – an effort to represent a complex world with a neat, artificial system – is necessarily informal, and cannot benefit from the exactness of the mathematical system itself. Thus, there is no way to guarantee that we selected the right system and operations to model a particular phenomenon, or represented the phenomenon accurately, or interpreted the results correctly. All we have to guide us in the translation is our skills.

To take a simple example, we can use mathematics to model the relationship between the speed of a car and the distance traveled in a period of time. All we need to do is represent these entities with appropriate values, perform the operation of multiplication or division, and then translate the result back into an actual entity. So, if we know the car's speed, the mathematical model lets us predict the distance it will travel in a given period of time, or the time required to travel a given distance. But mathematics cannot verify for us that we employ the formulas correctly. It cannot stop us, for instance, from using an incorrect speed, or from measuring the speed in miles per hour and the distance in kilometers. We praise the power of mathematics to predict the distance or time, but, in reality, mathematics only guarantees that the higher-level element (the result) is indeed the product or quotient of the two elements we started with.

No mathematical system can also be a substitute for the expertise required to use it. Although no less important than the system itself, the work involved in using it – particularly the translation from real entities into mathematical ones and back – is largely informal, and hence open to errors despite the exactness of the system.

In relational systems, we saw earlier, this problem led to the separation between the formal and the informal aspects – those aspects deemed to be within, and those deemed to be outside, the scope of the formal model. The formal aspects, we see now, correspond to the manipulation, while the informal ones form the translation. The manipulation includes the definition of fields and tuples, and the query operations. And the translation includes everything else: the updating operations, the normalization, the integrity rules, and the database language. This separation is artificial, of course, since all aspects of the database are equally important. But it is inevitable if we want to have a mathematical model: if we must exclude from the model any database aspect that is too complex to treat formally, that aspect is bound to end up as part of the translation, where we can deal with it informally.

This limitation – the need to treat the translation informally – is inherent in all mathematical systems. No matter how rigorous and exact is the manipulation, we depend largely on personal skills when selecting a particular system for a given phenomenon, and when translating the real entities into

mathematical ones. And the relational model is no different. What makes it silly, then, is not this limitation, which is universal, but the fact that it consists almost entirely of the informal translation. In the end, what is for the traditional mathematical systems the ultimate purpose – the formal, exact manipulation – plays in relational systems an insignificant part.

2

Recall the predicate calculus system, the logical foundation of the relational model. A logical expression like *P(x,y,z)* describes the tuples of elements *x*, *y*, and *z* related through predicate *P*. When we substitute actual values for the three elements, the expression will be evaluated as *True* for some tuples and *False* for others. We retain then, usually, the set of tuples for which the expression is true (a relation); and, using logical operations, we combine it with other sets, which are based on different predicates and elements. Such a combination is an expression that describes a new set of tuples – a set that relates in a different and perhaps more complex way the elements of the original tuples. The new set may then be combined with others, and so on, to create higher levels of complexity.

Like all logic systems, predicate calculus is concerned with the *structure* of variables and expressions, not their meaning. All it can guarantee is that, if we restrict ourselves to combinations based on the operations permitted by the system, the result of each combination will be correct within the definition of the system. Thus, if we know that the tuples in the original sets are true, we can determine with certainty, level after level, whether those in the resulting sets are true or false.[2] The system guarantees the validity of the manipulation, but the translation remains our responsibility: it is up to us to determine – by means external to the system – whether the tuples we start with are true or false, and whether the expressions that define the tuples, along with the operations that combine them from one level to the next, match the relations between the entities we want to model in that system. A logic system, in the end, is only a tool. It is up to us to judge whether it is the right tool in a given situation, and to use it correctly.

The fallacy, thus, lies in the belief that if the database model resembles a logic system, database work will become an exact, mathematical activity. In reality, this new tool is inappropriate for database programming, because the

[2] Strictly speaking, it is not the tuples that are true or false, but the result of the logical expression that defines the tuples; the more accurate description, though, would make these sentences too complicated.

database structures are closely linked to the other structures that make up the application. Mathematical and logic systems can only model phenomena that can be represented with a simple structure. So their ability to model database structures has little value if these structures must interact with others.

Like the predicate calculus system, a relational system guarantees only that the sets of tuples generated from the previous ones, as we move from one level to the next, are correct within the system's definition. The elements are now fields, the tuples are records, the sets of tuples are files, and the logical expressions depict combinations of files or portions of files. The database and the relational operations can be represented, therefore, with the same formality and preciseness we enjoy in predicate calculus. An expression like *F(a,b,c)* defines a file by stating that the fields *a*, *b*, and *c* are related through the predicate *F.* When actual values are stored in these fields, the expression will be true for some tuples (i.e., records) and false for others; and the actual file consists of those tuples that are true.

In other words, we use simply the *existence* of the tuple to determine the truth or falsity of the expression that defines the tuple: a tuple is deemed "true" if it currently exists as a record in the file, and "false" if it does not. Thus, the definition of truth and falsity in a relational database is merely a *convention*. The convention states in effect that *the data in the original files is valid by default*, simply because the records present in the file are deemed to be "true." The absurdity of this convention is the root of the relational model's mathematical delusion, as we will see in a moment.

The relational operations are designed to create a new set of tuples from one or two existing sets; that is, to create a new file by selecting and combining records (or portions of records) from one or two existing files. So, if we restrict ourselves to the relational operations, the validity of the existing data guarantees the validity of the combinations: since the original records are true by default, the records in the new file will also be true. The new file can then be combined with others to create a higher level of complexity, and so on. No matter how we use the relational operations, we can be sure that the final result will reflect the data we started with. There can be no false records in the resulting files, because there were no false records in the original ones.

It should now be obvious why logic systems are inappropriate as database models. A logic system like predicate calculus cannot control the addition and deletion of tuples in the original sets, nor the modification of their elements. All it can do is create new sets of tuples from existing ones; that is, *read* the original data. And if predicate calculus is limited to reading its data, so must

be the relational model. With a database, the limitation to reading is, of course, the limitation to queries. So the fact that the relational model is limited to queries follows *necessarily* from its logical grounding. In general-purpose applications, though, adding, deleting, and modifying records are as important as queries. It is absurd, therefore, to ground a database system on predicate calculus.

The great weakness of the relational model, then, is the need to ensure by *informal* means that the original tuples are true. Since it is the truth or falsity of each tuple that determines whether it can be a record in the file, what is merely the truth value of a logical expression in predicate calculus becomes the critical issue of data integrity in a relational system. This means that, before we add a new record to a file, we must ensure somehow that the record is true; similarly, when we modify the data in an existing record, we must ensure that it continues to be true; and before we delete a record, we must ensure that it is indeed false.

But the only way to perform these validity checks is by accessing the tuples from *outside* the relational model, with *traditional* programming methods. The software theorists underrate the significance of this weakness: they casually say that the database modifications and the associated integrity issue lie outside the scope of the formal model, as if this limitation were just a minor implementation detail.

Clearly, what the model offers us – the assurance that the files resulting from relational operations are correct if the original ones are – is meaningful only if we can be sure that the data in the database is correct at all times. But the values stored in database fields are not right or wrong in an absolute sense. Their validity can only be assessed within the context of the running application; that is, by performing certain operations that link the database structures with some of the other structures that make up the application. The database is one aspect of a complex structure, and the validation process cannot be represented with a formal, mechanistic model.

So, if the validation is an informal, error-prone process, how can anyone claim that the relational model guarantees the correctness of the resulting files? All it can guarantee is that the data in the resulting files reflects accurately the data in the original ones. Consequently, the validity of the resulting data can be no more certain than the validity of the original data. And ascertaining *that* validity is no different in a relational system than it is for traditional data files. Thus, since the ultimate precision of a system is limited by its least precise part, the belief that a relational database is more precise than a traditional one is a delusion.

Here are some of the mistakes that can be committed in applications based on a relational system – mistakes that would not be detected by the system:

adding a new record that has invalid values in fields like address, phone number, price, or part description; placing invalid values in the fields of an existing record; retrieving a part record using the vendor number as part number (record keys for parts, vendors, employees, customers, etc., may well share a common range of values, so this mistake would not result in an invalid key, and the wrong record would indeed be retrieved); adding the quantity purchased to the quantity in stock of a part, instead of subtracting it; deleting a record that must not, in fact, be deleted – and, generally, omitting a record that *should* be in the file (the convention that the records present in the file are true does not imply that all those not in the file are false, so the model cannot determine which records are *missing*).

The reason a relational system does not prevent us from performing such operations is that, within the relational model, these operations are perfectly correct. The only way to discover the mistakes is by performing these operations together with some *other* operations, which take into account both the database structures and the other aspects of the application; in particular, the business rules implemented in the application. In other words, we can only discover the mistakes by checking the data from *outside* the model.

It is not surprising, therefore, that the relational model had to be "enhanced" with informal means that allow us to discover such mistakes. We will study these enhancements under the third delusion, but it is worth noting at this point that, despite some new and impressive terminology, what we are doing with the new features – linking the database structures with the other structures of the application – is exactly what we had been doing all along, in a much simpler way, with ordinary programming languages.

3

Let us return to the formal model. To explain the relational theory, textbooks give us page after page of definitions and expressions in mathematical logic. Yet, as we just saw, this formality and preciseness cannot stop us from committing outrageous mistakes. No matter how rigorous the relational model is from a mathematical perspective, the only part that is formal and precise is the definition of database entities and operations; specifically, how we combine tuples into files, and files into other files, as we move from one level of complexity to the next. And these entities and operations are so simple that we can use them just as effectively without the formal definitions.

Recall the simple system that can handle only integers and two operations, addition and subtraction. In this system, all that mathematics does is ensure that the integer of the next level is indeed the sum or difference of the integers

of the current level. Thus, the formal definitions in such a system offer very little beyond what we can accomplish using just common sense. For, we can also add and subtract integers correctly by replacing the formal definitions with an informal method, and carefully following that method. Because this system is so simple, the formal and the informal alternatives are equally practical. Even more importantly, the formal system cannot prevent us from committing such mistakes as using wrong values when translating an actual phenomenon into integers, or adding two integers when in fact we ought to subtract them. Whether we choose the formal system or an informal method, we must deal with these problems informally.

And the same is true of the relational model. All that mathematics does is assure us that each operation combines elements just as its definition says. It assures us, for example, that the selection operation indeed selects the specified records. Thus, an expression like $G(a,b,c) = F(a,b,c)$ AND $a>k$ defines a selection operation by saying that the new file G includes those tuples (a,b,c) which satisfy two conditions: they are true (i.e., are actual records) in file F, and the element a is greater than a certain value k. This formal definition is very impressive, but it is also very silly. Because record selection is such a simple concept, we can easily perform this operation by relying on common sense alone: we describe informally what we mean by record selection, and then carefully implement this concept using the basic file operations and a programming language (see figure 7-13, p. 680).

All mathematical systems appear silly, of course, if we study only the low levels. The low-level elements and operations are usually simple enough to understand intuitively, so the rigour and preciseness of their definition appear superfluous. But there is a reason for this formality. In serious mathematical systems there are *many* levels of complexity. We always start with simple elements, but we combine them so many times that we end up with very intricate ones at the higher levels. Since the elements and operations at these levels can no longer be understood intuitively, the formal definitions are our only assurance that the system functions correctly.

In a simple system, on the other hand, the elements do not increase in complexity as we move to higher levels, so there is no benefit in combining them more than a few times. With a simple system, therefore, we rarely create more than a few levels, and we can only model simple phenomena. In the system of integers and two operations, for instance, the sum or difference of two integers is still an integer. And there aren't many applications where all we need is to add or subtract integers while repeating these operations endlessly, level after level. Applications that require many levels also require an increase in the complexity of elements and operations.

In the relational model, too, the same type of elements (tuples and files) and

the same type of operations (SELECTION, UNION, PRODUCT, etc.) are found at both the lowest and the highest levels. And as a result, there is no benefit in combining elements more than a few times. Thus, few requirements involve more than three or four files, or more than three or four operations, in the creation of a new file. It is difficult to imagine a situation where we have to perform a series of a dozen selections, products, and projections, each operation starting with the result of the previous ones.

The software theorists claim that the relational model offers benefits similar to those of the traditional mathematical systems, but this is not true. In science and engineering we start with simple elements like integers, and simple operations like addition, but after many levels we end up with such concepts as calculus and analytic geometry. The complexity of the elements, as well as the complexity of the operations, keeps increasing as we move to higher levels. With the traditional mathematical systems, therefore, we derive important benefits when adding levels; in particular, the higher complexity permits us to model more complex phenomena. Evidence of these benefits is also found in that the formality and preciseness are now critical: unlike the selection of records in a database, we can hardly replace concepts like differential equations with methods based on common sense alone.

In the relational model, it is the restriction to high-level operations that prevents us from using more than a few levels. Software applications do have many levels of complexity, starting with simple entities like statements and database fields, and ending with the logic of a whole business system. But these are not the levels of a simple structure. Unlike mathematical systems, which can be represented with one structure, software applications comprise *many* structures – structures that must share their elements if they are to model our affairs accurately. And it is often low-level elements like statements and database fields that must be shared.

Among these structures are also the database structures, but with a relational system the lowest-level elements that can be shared are the files. If we take the fields, records, and files to be the three lowest levels of a database structure, the relational operations only permit us to access files. Starting with files we can then create even higher levels – that is, further files. But the interactions with the other structures are not as versatile as those we could create by starting with records and fields. Most interactions, in fact, are now too awkward or inefficient to be practical.

Thus, we see no benefits in creating more than a few levels of relational operations, not because we do not *need* higher levels, but because the restriction to operations on whole files prevents us from creating the combinations of software entities needed to attain those levels. This is why the original model was useless, and why the features added later serve mainly to bypass the

restriction to whole files: they restore the means to link the database structures to the rest of the application through lower-level elements (through individual records and fields), thus permitting more alternatives at the high levels.

4

With any mathematical system, we must perform the translation in order to attain the precise format required for the manipulation. But in itself the translation is a detriment: not only does it constitute additional work, but its informality detracts from the exactness of the manipulation. We can justify the use of a mathematical system, therefore, only if the manipulation confers significant benefits; that is, if it permits us to perform some important and difficult tasks. And this is indeed the case for the mathematical systems used in science and engineering: the manipulation in these systems is very elaborate, with many levels of complexity, while the translation may be as simple as converting things like weight, voltage, or time into numerical values.

In a relational system, the opposite is true: the manipulation is trivial, and it is the translation that ends up very elaborate. In order to have a mathematical database model, the part that is the manipulation had to be restricted so much that it involves in the end only trivial mathematics. The most difficult aspects of database programming – updating operations, integrity rules, the second and higher normal forms, the database language – were left out of the model and became part of the translation. They were left out, not because they do not entail manipulation, but because *that* manipulation cannot be represented mathematically.

So the manipulation includes only queries, and the queries permit only high-level operations on whole files. In any case, these queries are so simple that they can be implemented without mathematics. The basic file operations, we saw earlier, allow us to scan and relate files, and to select records and fields. Thus, the operations permitted by the relational model – selection, union, product, and the rest – can be easily programmed with ordinary iterative and conditional constructs.

To deal with those aspects of the database that make up the translation, and which were left out of the formal model, we need programming skills. So in the end we use the power of mathematics for the relatively simple manipulation, which hardly requires a formal system, while depending on informal programming methods for the difficult tasks.

Unlike the mathematical systems used in science and engineering, then, the relational model confers no benefits. In the traditional fields, mathematics permits us to accomplish tasks that are impossible without a formal system; so

the translation, with its drawbacks, is worthwhile. Relational mathematics, on the other hand, is so simple that it can be replaced with a few lines of programming; so the drawbacks of the translation exceed the benefits of the manipulation. The idea behind the relational model is, therefore, senseless. What is the point in seeking a formal system for the query operations, if all the work required to prepare the database for these queries must remain informal? Since we must continue to depend on programming for the difficult translation, we may as well use programming also for the relatively simple manipulation.

The theorists promote the relational model by pointing to its mathematics, and implying that it provides the same benefits as the models of science and engineering. But if the relational model uses only trivial mathematics, the claim is a fraud. In reality, very little of the phenomenon of a database is amenable to an exact, mechanistic representation. Mathematics is useful for phenomena where changes are rare. Take the bodies in the solar system, for instance: we can represent their motion mathematically because their properties are fixed; so, with only a small investment in the translation, we gain the great benefits of the manipulation. In the database phenomenon, however, changes are very common. These changes include adding or deleting records, and modifying the data stored in fields. Each change produces a slightly different database – different data, and hence different relationships. No mathematical system can accurately represent such a changeable phenomenon, and it is for this reason that the theorists exclude the updating operations from the formal model.

The relational idea is worthless because we have to leave too much out of the manipulation in order to represent the database functions mathematically. What we leave out is far more than what we leave out in the traditional uses of mathematics. We must leave out of the formal model the database changes and all the related issues; in particular, the integrity rules and most of the normalization. These features are in reality as much part of the application as are the queries. So, for the model to be truly useful, they would have to be included in the manipulation. Only if we decide that databases are mainly query systems can we treat issues like updating, integrity, and normalization as part of the informal translation, rather than the formal manipulation. But then we must no longer claim that the relational model is useful for general applications.

The fact that so much had to be left out of their formal model ought to have worried the theorists. This was an opportunity to realize that the relational

concept is fallacious, that databases cannot be usefully represented with a mathematical model. Instead, fascinated by the little that *could* be represented mathematically, they saw in the relational concept the beginning of a new science.

But an even greater deficiency than the separation of query operations from updating operations is the separation of the query operations from the other operations performed by the application. As we saw, the relational operations are restricted to manipulating whole files, rather than individual records and fields, like the traditional file operations. And, while in principle individual records and fields can be treated as tiny files and accessed with the relational operations, this method is far too awkward and inefficient to be practical. In effect, then, the relational model does not permit us to manipulate freely the low-level database entities. The immediate consequence of this limitation is that it is impossible to link, at the level of records and fields, the structures formed by database entities and operations with the structures formed by the other aspects of the application. And no serious application can be developed without these links.

Even for query systems, therefore, the relational model cannot be said to work if the only queries that can be implemented are those possible through operations on whole files. To be truly useful, a database system must allow us to manipulate database entities in any conceivable way.

In conclusion, the relational model is indeed a revolution in database concepts, in that it imparts to database programming the rigour and exactness of mathematics; but only *if* we restrict ourselves to queries; and *if* we restrict ourselves to queries that can be expressed through certain parameters (so that the database can be separated from the application and accessed only through operations performed on whole files); and *if* we restrict ourselves to normalized files (although it is possible, in principle, to implement any queries using normalized files, this is usually too complicated or too slow to be practical); and *if* we can ensure that all the operations that modify the database are performed correctly (so that the data upon which the queries are based is valid at all times).

Note that these restrictions describe the *original* model; so this model, absurd as it is, is in fact optimistic. As no practical uses were found for it, means had to be provided eventually to link the database structures with the other structures of the application. And the only way to do this was by permitting *low-level* database operations, which bypass the original restrictions. But if those restrictions are essential in order to attain an exact model, if we bypass them we will no longer enjoy the benefits of mathematics, not even in a narrow range of applications. So those benefits, which were insignificant in the original model already, were reduced in the end to zero.

5

There is no better way to conclude our discussion of the first delusion than by showing how the relational model is presented to the public. We just saw that there are no mathematical benefits in using a relational database. The database experts, however, promote the relational systems by praising *precisely* their mathematical background. Here are some examples: "The mathematical concept underlying the relational model is the set-theoretic *relation*."[3] "The relational model is founded on the mathematical disciplines of predicate calculus and set theory."[4] "The relational data model is based on the well developed mathematical theory of relations. The rigorous method of designing a data base (using normalization …) gives this model a solid foundation. This kind of foundation does not exist for the other data models."[5] "The relational approach is based on the mathematical theory of relations.… The results of relational mathematics can be applied directly to relational data bases, and hence operations on data can be described with precision."[6] "The relational model is based on the mathematical notion of a relation. Codd and others have extended the notion to apply to database design."[7] "The solid theoretical foundation guarantees that results of relational requests are well defined and, therefore, predictable."[8] "One of the benefits of working with the relational approach to databases is that it can be couched within the formalism of first-order predicate logic. As a result a mathematical foundation is available for dealing with database issues when databases are all relational."[9] "The reason we could define rigorous approaches to relational database design is that the relational data model rests on a firm mathematical foundation."[10] "In the

[3] Jeffrey D. Ullman, *Principles of Database Systems* (Potomac, MD: Computer Science Press, 1980), p. 73.

[4] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1161.

[5] Shaku Atre, *Data Base: Structured Techniques for Design, Performance, and Management*, 2nd ed. (New York: John Wiley and Sons, 1988), p. 90.

[6] James Martin, *Computer Data-Base Organization*, 2nd ed. (Englewood Cliffs, NJ: Prentice Hall, 1977), p. 204.

[7] Catherine M. Ricardo, *Database Systems: Principles, Design, and Implementation* (New York: Macmillan, 1990), p. 177.

[8] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design* (Reading, MA: Addison-Wesley, 1989), p. 32.

[9] Barry E. Jacobs, *Applied Database Logic*, vol. 1, *Fundamental Database Issues* (Englewood Cliffs, NJ: Prentice Hall, 1985), p. 9.

[10] Henry F. Korth and Abraham Silberschatz, *Database System Concepts*, 2nd ed. (New York: McGraw-Hill, 1991), p. 209.

formulation of relational data models, the mathematical theory of relations is extended logically where required to meet data management objectives. The mathematical foundation of relational data models permits elegant and concise definition and deduction of their properties."[11]

As we saw, it is not difficult to show that the model's mathematical foundation is irrelevant. Yet no one in the academic world – not the mathematicians, not the philosophers, not the engineers – ever challenged these claims. Nor did anyone challenge the other software theories. The computer scientists can invent any theories, thus, no matter how absurd, confident that the academic community, the software practitioners, and the rest of society will accept them unquestioningly.

Most people trust and respect the universities, without realizing that what the academics are promoting is not ideas that are useful, but ideas that help them maintain their privileged position even if useless; in particular, the idea that science means simply the pursuit of mechanistic theories – whether sound or not, whether useful or not.

Moreover, by fostering the mechanistic ideology, universities make it possible for the software companies to promote fraudulent concepts. The mechanistic ideology benefits incompetents and charlatans, therefore, by making their activities look like serious research, or like legitimate business.

The Second Delusion

1

The second delusion is the idea of normalization: the belief that, within the relational model, the problem of database design has been turned into a formal theory. In reality, the principles of normalization do not constitute an exact procedure, but one that can only be implemented informally. (The concept of normalization was introduced earlier; see pp. 704–706.)

The delusion of normalization can be summarized by saying that it is an attempt to replace the simple process of *avoiding* incorrect file relationships, with the complicated process of *eliminating* them after allowing them into the database. To justify the need for normalization, the theorists misrepresent the design problem. Traditionally, we used methods that helped us to create a correct database, and thereby avoided data inconsistencies. Now we are expected to ignore those methods, deliberately create an incorrect database,

[11] Dionysios C. Tsichritzis and Frederick H. Lochovsky, *Data Models* (Englewood Cliffs, NJ: Prentice Hall, 1982), p. 93.

discover the consequent problems, and then use normalization to convert the incorrect database into a correct one.

In the end, it is only this contrived, absurd procedure that the theorists managed to formalize, not the *actual* problem of database design. As we will see presently, even under normalization the correct database structures can only be discovered *informally*, by studying the application's requirements – just as we do when following the traditional design methods. The concept of normalization, thus, is a fraud. By inventing pompous terms to describe what are in fact senseless principles, and by discussing these principles with great seriousness, the relational experts delude themselves that they have turned database design into an exact theory.

Note that here, in the discussion of the second delusion, I am using the term "normalization" to refer only to the second and higher normal forms; that is, to the problem of field dependency. The *first* normal form (which restricts fields to single values) is unrelated to the higher ones; it is part of the formal relational model, and hence part of the first delusion.

Note also that, although the mathematical pretences of the second and higher normal forms resemble the first delusion, these transformations are not required at all by the formal model. A database, in other words, does not have to be normalized in order to satisfy the mathematical restrictions of the formal model (I will return to this point later). The higher normal forms are only needed in order to prevent problems that arise when *updating* the database; and the updating operations lie outside the scope of the formal model.

Thus, it would be wrong to treat the higher normal forms as part of the first delusion. Their *annulment* (the process known as *denormalization*) does constitute, however, the same kind of delusion as the annulment of the first normal form, or the annulment of the other aspects of the relational model. All annulments, therefore, are discussed under the third delusion.

Let us review the concept of normalization – how the relational theorists present the problem of data inconsistencies, and its solution.

Each piece of information in the database should exist in only one place, because data that is duplicated may cause various inconsistencies when records are added, deleted, or modified. The relational theorists call these inconsistencies "update anomalies." The unnecessary repetition of data also wastes storage space, but it is the anomalies that are the main reason for

normalization. In fact, depending on the size of the duplicated fields and the number of records involved, normalization sometimes *increases* storage requirements. Still, the theorists say, the benefits are so important that we should normalize our files even at the cost of increased storage space.

It must also be noted that there is always an alternative to normalization: the inconsistencies can be avoided by performing additional operations in the application (additional checks and, when required, additional updating). Normalizing the files is generally a simpler and more efficient solution, but sometimes those operations are the better alternative. In the relational model, though, this too is unacceptable: we must *always* normalize our files.

Duplication, and hence redundancy, occurs when we store some data in several records in a certain file while that data could be stored in only one record in another file: repeating customer data like name or address in every order belonging to that customer, repeating product data like description or price in every order line with that product, and so on. Clearly, fixed data should be stored in a separate file – a customer file or a product file, in this case. Only data specific to an order should be stored in the orders file, and only data specific to an order line in the order lines file. A field in the orders file will contain the customer number, and a field in the order lines file will contain the product number. These fields will serve as links to the customer and product files. Thus, when processing an order, we can access the customer data by reading the customer record; and when processing an order line, we can access the product data by reading the product record. The same is true of an invoice file, a transaction file, a sales history file, and any other file that needs customer or product data.

With data separated in this manner, when there is a change in a customer's name or address, or in a product's description or price, we only need to modify the customer or product record, rather than all the orders for that customer, or all the order lines with that product. If there were no customer and product files, an anomaly would occur if we modified the customer data in an order, or the product data in an order line: any other orders for that customer, or any other lines with that product, would continue to have the old, and hence wrong, values. Another anomaly would occur if we had to store data for a customer that has no outstanding orders, or for a product that is not currently on order: we would have to create a dummy order just so that we had a place to store customer or product data.

Data redundancy can also be viewed as the result of a mistaken relationship between two fields in the same record; specifically, a *misplaced dependency* of one field on another. A field should depend only on the field or fields that make up the record's key. There is no need for other relationships *within* a record; and if such a relationship exists, some data will be redundant. This is

true because, if one field can be determined from another, its value will be repeated unnecessarily in all the records where the other field has a particular value. We only need to specify the dependency between the two fields in one place. So the correct way to store this information is as a single record, in a separate file.

Generally, to eliminate the redundancy associated with one misplaced dependency (the dependency of one or several fields on a given field), we must create one extra file in the database.[1] Each level of normalization – the levels known as second, third, Boyce/Codd, fourth, and fifth normal forms – is a more stringent implementation of this principle. Each level, that is, will eliminate a more subtle type of dependency. These types – known as functional dependency, transitive dependency, multivalued dependency, and join dependency – differ in the types and combinations of fields that form the misplaced dependency: a non-key field depending on only some of the fields that make up the key (instead of depending on the whole key), or a non-key field depending on another non-key field, or a key field depending on another key field, or more than two fields depending on one another. The classification of the normal forms is such that, in addition to being more subtle and more rare, each level represents a broader category of misplaced dependencies – a category that includes as a special case the one at the next lower level.

2

One aspect of the second delusion is the belief that, because the ideals of normalization are discussed only with the relational model, they are exclusive to relational databases. The theorists present the concept of normalization as if no one had been aware of the problem of data redundancy and inconsistencies before we had relational databases, and as if the relational model and the normalization principles were the only way to deal with this problem. They never mention the fact that this problem and its solution are *identical* to their counterparts in databases created with the traditional file operations. And they are identical because they are concerned with files, records, fields, and keys – elements that are identical (despite the new terminology) in relational and in traditional databases. With one type of database or the other, redundancy and inconsistencies indicate a faulty design, a database that does not match the application's requirements. And the solution is to modify the design so as to satisfy the requirements.

[1] Several extra files are required when the relationship involves three or more interrelated fields (the kind of dependency resolved by the fifth normal form).

The issue of normalization, then, is perceived as an important part of the relational model while being, for all practical purposes, a separate theory. It was tacked on to the relational model because it was invented by the same theorists, but it could be applied to any database model that uses files, records, fields, and keys. For, what we are asked is simply to replace the traditional principle of designing a database so as to *avoid* redundancy and inconsistencies, with the absurd principle that we must start with a faulty design and then modify it so as to *eliminate* the redundancy and inconsistencies. Thus, nothing stops us from employing this absurd principle with a *traditional* database. All we have to do is deliberately create an incorrect database, and then normalize it in order to eliminate the consequent problems. The final, correct database would be identical to the one we create now simply by following the traditional design principle.

It is also worth noting that we can attain the ideals sought by normalization more effectively with traditional databases than we can with relational ones. Ironically, while the relational theory makes the problem of redundancy and inconsistencies look like a new discovery, insists on strict normalization, and overwhelms us with formality and new terminology, its restriction to high-level operations often *prevents* us from solving this problem. And it is with the traditional file operations, where we don't even use terms like "normal form," that we can more easily deal with it. This is true because those operations are more versatile and more efficient than JOIN, the operation that combines files in the relational model. Since the process of normalization separates fields by creating additional files, we must read and combine more files and more records later, when *accessing* the database. And it is when the performance degradation caused by these additional operations becomes unacceptable that we must leave some data unnormalized. Thus, since the traditional file operations permit us to combine files and records more efficiently than does JOIN, we can afford to separate more fields – and hence attain a higher level of normalization – in a traditional database than in a relational one.

We can appreciate even better why the problem of redundancy and inconsistencies is not part of the relational theory by recalling the mathematical foundation of the relational model, predicate calculus (see pp. 697–698). The relation described by the logical expression $P(x,y,z)$, for instance, consists of those tuples of elements x, y, and z that are related through the predicate P. Specifically, we substitute for the three elements certain values selected from their respective domains of permissible values, and we retain those combinations of values for which the expression yields *True*.

Now, there is nothing in this definition of a relation to prevent two elements in a tuple from forming an additional relationship. For example, if y depends on x in such a way that we can always derive its value from that of x, y is in effect redundant. But this redundancy is harmless; that is, if we combine this expression with other expressions, the redundancy will be reflected perhaps in the final result, but it will not cause a logical inconsistency.

The original, formal relational model is similar: even if mistaken, the dependency of one field on another in a given file does not cause an inconsistency when that file is combined with others through relational operations. The formal model is concerned only with the *structure* and *combination* of files. Thus, even if there is a misplaced dependency in that file, the consequent redundancy is harmless. All that will happen is that some fields in the resulting file will also be related through a misplaced dependency.

The reason we can have misplaced dependencies in predicate calculus and in the formal relational model is that these systems do not include *updating* operations. Predicate calculus is not concerned with the way tuples ended up in the relation, or the way elements acquired their current value, but only with the operations that *use* the tuples. And the formal relational model is not concerned with the way records are created, deleted, or modified, but only with the operations that *read* the records. Thus, since only updating operations can cause inconsistencies, we need not worry about misplaced dependencies when we restrict ourselves to the formal model. (The relational theorists acknowledge this fact by calling the inconsistencies *update* anomalies.) To put it differently, if we restrict ourselves to queries, and particularly to queries expressed through the relational operations, we need not worry about misplaced field dependencies.

So the theory of normalization is irrelevant for applications restricted to the formal model. It is only for the broader model, which includes various informal aspects, that it has any significance. The original papers mentioned only briefly the updating operations that would be required in an application, and the language through which they would be specified.[2] It was assumed that these operations, along with the problems they might cause and the checks needed to avoid these problems, would be similar to those used in other database systems. No one tried to extend the formal model by including, say, a formal set of updating operations. It was assumed, in other words, that the exact, formal model would provide all the important database operations. The updating operations, as well as the operations needed to protect the database from redundancy and inconsistencies, were seen as a minor issue; so the plan

[2] See, for example, E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387.

was to implement them informally, just as they were implemented in other systems. We find evidence for this interpretation in that the second and higher normal forms are not mentioned at all in the original papers. The terms “normal form” and “normalization” in these papers refer only to what is called now the *first* normal form.

It was when the theorists turned the formal model – originally meant only for queries – into the basis of general database systems, that the idea of normalization had to be extended. By calling the new normal forms “second,” “third,” etc., the theorists made them look like a natural extension of the first one, although they are unrelated. While the first one is concerned with eliminating data structures *within* fields, the higher ones are concerned with eliminating misplaced dependencies *between* fields. The first one is needed in order to base the formal model on predicate calculus, but the higher ones are needed only if we perform updating operations. By making the latter look like an extension of the first one, though, the theorists managed to mask the fact that the relational model was changing from an exact theory into a collection of informal concepts. While everyone thought that the precision of the formal model was being extended to cover all aspects of database work, in reality the exact opposite was taking place: what had been originally the *informal* aspects of the model – the updating operations, the higher normalization, the integrity rules, the database language – was becoming the actual model, and the *formal* part was becoming irrelevant.

One wonders, if the updating operations constitute an informal aspect of the relational model, why is it so important to formalize the normalization process? Why do the theorists attempt to reduce the problem of redundancy and inconsistencies to an exact model, if the problem only concerns the updating operations, which are informal in any case? The answer is that the theorists saw in the normalization principles an extension of the original, formal model. The precision which that model offered for queries, they thought, can now be extended to the design phase, and to the updating operations; so we will soon have a mathematical model for the whole database concept.

As we will see later, the formality of the normalization process is specious. The theorists are indeed discussing the subject of dependencies in a formal manner, but, ultimately, we can determine the relationship between two given fields only by studying and interpreting the application’s requirements; that is, informally. As is the case with all mechanistic pseudosciences, the relational theorists noticed a few patterns and regularities (the normal forms and the field dependencies), and jumped to the conclusion that an exact theory is possible for the design of file relationships. The same naivety that led earlier to the belief that the resemblance of records to the tuples of predicate calculus can

be the basis of a practical database model, led now to the belief that a neat classification of field dependencies can be the basis of a formal model for database design.

3

When studying the problem of data redundancy and inconsistencies, we notice a marked discrepancy between the way it is presented by the relational theorists and its *actual* difficulty. The theorists discuss this subject with great seriousness, the way one would discuss the most difficult problems a programmer can encounter. In reality, this is one of the simplest programming problems. And it is a problem that does not lend itself to formal treatment, so an exact theory has no practical value in any case.

It is hard to think of anyone designing a database where the "anomalies" so seriously discussed by the theorists could occur at all. Even a novice can recognize the absurdity of storing fixed customer information only in the invoice records, or repeating fixed product information in every order line with that product. And if mistakes like these go undetected and end up in the working application, it is hard to imagine a place where the programmers or the users fail to understand why customer data is lost when an invoice is paid, or why two order lines with the same product show different descriptions. Then, once they understand the problem, it is hard to imagine them failing to discover the solution; that is, keeping the fixed data in a separate file. To put this differently, a person incapable of dealing with this simple problem would be unable to deal with any other programming problem. His applications wouldn't work, and those anomalies would be the least of his worries. Thus, it is highly unlikely that a place can exist at all where the theory of normalization can confer any benefits.

And indeed, before it was brought into the limelight by the relational experts, we treated the problem of redundancy and inconsistencies as we did every other programming problem: we recognized its importance, but we never tried to explain it with an exact theory, or to solve it with a formal method. As evidence of its simplicity, we didn't even think that the process of solving it needed a special name; it is only for the relational theory that terms like "normalization" and "normal form" had to be introduced. And for those of us who have continued to use the traditional database design method, the attempt to turn this subject into an exact theory has had no significance whatever. We are treating the problem of redundancy and inconsistencies exactly as we did thirty or forty years ago, simply because the theory of normalization is irrelevant.

Relational database books devote at least one chapter to the subject of normalization. And the more thorough among them intimidate us with their formal tone and lengthy explanations, the countless definitions and theorems, and the new terms and symbols. Thus, if we ignore its content and judge it solely by its style, the discussion of normalization in a database book resembles the kind of discussions found in engineering books. Readers new to the relational theory are impressed by this formality and expect to learn some important facts. Invariably, though, they find the discussion hard to follow. Then, when the book illustrates the theory with actual examples of unnormalized files and their conversion to normalized ones, these readers react by exclaiming, "But this is how I would have designed the database in the first place!" So it is only through actual examples that we can comprehend the theory of normalization at all, and at that point the reason for the earlier difficulty becomes clear: since we would intuitively create the correct, normalized files to begin with, we struggle to understand what is the problem that the theory is trying to solve. To most of us it doesn't even occur – until we read a relational book – that anyone would design a database by repeating, say, the customer address fields for each order, or the product description field for each order line.

The difficulty, then, is not in understanding the principles of database design, but in understanding the theory of normalization: the attempt to reduce database design to formal and exact methods. We must make an effort to understand the normalization problems and their solutions because we normally don't think in a way that can create these problems. The problems are contrived, unreal. They were *invented* by the theorists, in order to have a reason for seeking a formal solution.

Typically, the books start by showing us an incorrect design and its drawbacks. They continue then by showing us how to convert it into a correct design. But what is the point of this discussion if hardly anyone would even *consider* the incorrect alternative? The theorists are defining, classifying, and explaining in the style of mathematical analysis some implausible situations – situations we never encounter in real life. With just common sense and a little practice, we already know how to create correct databases. The normalization theory, on the other hand, asks us to study some strange problems (the difference between the second and third normal forms, why we have the so-called Boyce/Codd normal form between the third and fourth, how to convert a file from first to second or from second to third, etc.) and to assimilate an endless list of strange concepts (superkey, dependency preservation, nonloss decomposition, left-irreducible functional dependency, etc.).

It is the attempt to formalize the problem of field dependency, data redundancy, and data inconsistencies, and the need to fit the incorrect designs into

the classification of normal forms, that we find hard to understand – not the actual principles of database design. And when we finally understand the new concepts, we realize that in practice we never encounter these problems. When we learn to program we don't learn two things – how to design incorrect databases, and how to convert incorrect databases into correct ones; we simply learn how to design correct ones. Thus, since the problems studied by the theory of normalization concern mostly the transition from bad to good design, it is not surprising that the theory is, for all practical purposes, irrelevant.

4

We saw earlier, in "The Basic File Operations," that the concept of records, fields, and keys allows us to implement any file relationships we need in our applications. And we also saw that this concept is identical in traditional and in relational databases. The difference lies mainly in the new terminology and in the way the files are used. In traditional databases, we use the basic file operations through the flow-control constructs of a programming language; and we specify, through indexes, the individual records. In relational databases, we use only the high-level relational operations; and, rather than individual records, we specify whole files or logical portions of files. But in both cases we must create the same files and fields, and the same relationships, in order to implement a particular set of requirements.

Thus, although our earlier discussion concerned traditional databases, the same design principles apply to relational ones. With one type of database or the other, the traditional principles permit us to create correct – that is, normalized – databases directly from the application's requirements. To appreciate the absurdity of the normalization theory, then, let us review the traditional design concepts.

The decision we must make when designing a database is what files, fields, and keys are needed; that is, what data to store in the database, how to distribute it among files, and how to relate the files, in order to satisfy the application's requirements. Thus, since the files depend on the application's logic, we usually implement them together with the various parts of the application. It is the file *relationships* that pose the greatest challenge. For, if all we needed were isolated files (a customer file, a product file, a history file, etc., with no links between them), designing the database would be trivial, little more than creating the respective fields.

Files are related through the values present in their fields. Typically, identifiers and codes are used to relate files (product number, invoice number,

category, etc.). A relationship is established when two files use such a field, and some records in both files contain the same value in this field.[3] Depending on how the relationship is used in the application, we may use either key fields or non-key fields. Often, a combination of several fields, rather than a single field, is needed to relate files.

And it is the correct choice of relationships that ultimately determines whether or not there will be redundancy or inconsistencies in the database – what the theory of normalization is concerned with. The various normal forms, as we will see shortly, are nothing but a complicated way of expressing these relationships. In reality, all we have to do is create a database that correctly represents the application's requirements; and if we do this, there will be no redundancy or inconsistencies. In other words, if we understand the application's requirements, and if we implement them correctly, we don't need a theory of normalization (because we create "normalized" files from the start); and if we don't understand the requirements, or fail to implement them correctly, no theory can help us.

Four types of file relationships are possible between two files: one-to-one, one-to-many, many-to-one, and many-to-many. The terms "one" and "many" refer to the number of records in the first and second file that are logically linked.

Two files are in a *one-to-one* relationship when one record in the first file is related to no more than one record in the second file. Thus, the two files will have the same number of records when each record in the first one has a corresponding record in the second one, and they will have a different number of records when some records in either file have no corresponding records in the other. Files that are in a one-to-one relationship can always be combined into a single file, where each record comprises the two corresponding records: we simply merge their fields, and when there is no corresponding record we assign null values or default values to the respective fields. For practical reasons, though, it is sometimes preferable to have two files rather than one. For example, if a file has many fields but some operations involve only a few, we may decide to keep these fields in a separate, smaller record, in order to improve the application's performance.

An example of one-to-one relationship is an employee file and a special functions file, with the condition that a function may be performed by only one

[3] Relations based on field equality are the most common, but, strictly speaking, any values can be used to relate files. With a date field, for example, we can create a relationship where a record containing a certain date in the first file is logically linked to those records in the second file where a date is up to one year earlier.

employee, and an employee may select no more than one function. At some point in time we may have, say, 80 records in the employee file and 30 records in the functions file, but only 20 functions actually selected; thus, 60 employees will have no corresponding function, and 10 functions no corresponding employee. The two files are linked by adding a function number field to the employee record, or an employee number field to the function record (or both, if we need two-way links).

The most common relationship is *one-to-many*. Two files are in a one-to-many relationship when one record in the first file (the "one" file) is related to one, several, or no records in the second file (the "many" file), while each record in the second file is related to one or no records in the first file. Here are some examples: customer file and customer orders file (one customer may have one, several, or no outstanding orders, but each order belongs to one customer); orders file and order lines file (one order may include one or several order lines, but each line belongs to one order); employee file and payment history file (each employee has one record in the history file for each pay period, but each pay period record belongs to one employee). A one-to-many relationship is also a *many-to-one* relationship, when seen from the perspective of the second file: several orders are related to the same customer, several order lines to the same order, several pay periods to the same employee.

The one-to-many relationship is implemented by making the "many" file's key a combination of both files' identifying fields. For example, if we make the key in the orders file the combination of customer number and order number, we will be able to select for a given customer any one of the corresponding records in the orders file, and for a given order the single, corresponding record in the customer file. However, when the relationship is seen as many-to-one and a direct link is not required from the "one" file to the "many" file, the "one" file's identifying field can be just a non-key field in the "many" file. Thus, the key in the orders file would be just the order number, and we would access the customer records by including the customer number as a non-key field.

Two files are in a *many-to-many* relationship when one record in the first file is related to one, several, or no records in the second file, and at the same time one record in the second file is related to one, several, or no records in the first file. To implement such a relationship, we create a service file to act as a link between the main files. The service file has only key fields, and its key is simply the combination of the two main keys. For example, if some vendors supply several products, and certain products are supplied by several vendors, the vendor and product files form a many-to-many relationship. The key in the service file is the combination of vendor and product numbers, and we implement the two-way links between files (vendor to product, and product to vendor) by providing *both* sorting sequences: products within vendors, and

vendors within products. (If using traditional file operations, we accomplish this by creating two indexes for the service file.) We can then select for a given vendor the corresponding records in the product file, and for a given product the corresponding records in the vendor file.

The four types of relationships can be combined to link more than two files. Thus, a set of *several* files can form a one-to-one relationship, when any two files in the set are in a one-to-one relationship. Also, a file can be in two many-to-many relationships at the same time: with one file through one field, and with another file through another field.

The most versatile relationship, however, is one-to-many. One way to combine one-to-many relationships is by having several "many" files share the "one" file, through the same field or through different fields. The customer file, for example, can be related through the customer number to both the orders and the sales history files. One-to-many relationships can also be combined to form hierarchies of more than two levels, by using the "many" file of one relationship as the "one" file of another. For example, for each order in the orders file we can have several lines. We store then the line-related data in an order lines file, and use the combination of customer, order, and line numbers as the key. The order records will function, at the same time, as "many" in their relationship with the customer records, and as "one" in their relationship with the order lines records. Most applications require a mixture of combinations: several levels, and several files on each level. Thus, several "many" files may share the "one" file while acting at the same time as "one" files in other relationships.

It is also possible for two "one" files to share the "many" file. For example, if a customer purchases several products and a product is purchased by several customers, there will be a set of records in the sales history file for each customer record, and another set for each product record. But these sets will overlap: each history record will be related at the same time to a certain customer and to a certain product. Thus, in addition to being the "many" file for both the customer and the product files, the history file serves to create a many-to-many relationship between them. (The many-to-many relationship, we see now, is merely a special case of two one-to-many relationships that share the "many" file – the case where this file's sole purpose is to link the "one" files.)

Although the four types of relationships are usually described as *file* relationships, they are also *field* relationships. When two files are related as one-to-one, or one-to-many, or many-to-many, it is through their records that the

relationship exists: one or several records in one file correspond to one or several records in the other. But records are made up of fields, so the same correspondence exists between fields: the relationship between files is reflected in each pair of fields. Thus, when two files are related as one-to-one, each field in the first file is in a one-to-one relationship with each field in the second file; in addition, fields that belong to the same file are in effect in a one-to-one relationship with one another. When two files are related as one-to-many, each field in the first file is in a one-to-many relationship with each field in the second file. And when two files are related as many-to-many, each field in the first file is in a many-to-many relationship with each field in the second file.

For example, if the product and the orders files are related as one-to-many, a field like the product description or price in the former will be related as one-to-many to fields like the order date or quantity in the latter. What this means in practice is that the same product description or price may be associated with several order dates and quantities.

We can regard the four types of relationships, therefore, as either file or field relationships. So, rather than saying that two files are related and the field relationships reflect the file relationship, we can say that it is the fields that must be related, and the file relationship will reflect the field relationships. We design a database by creating relationships that match the requirements. In some situations we think in terms of *file* relationships; and once we create the files, it is obvious to which file each field must be assigned. In other situations it is better to think in terms of *field* relationships; and we implement the files and file links that will allow us to relate those fields as required.

Consider this example. We want to store some information about our products, so we start with a file that contains just the key field, the product number. If the requirements say that there may be several orders for each product, the product number is related as one-to-many to the order number. The order number must be, therefore, in a separate file, so we create an orders file with two fields: the order number as the key, and the product number as the link to the product file. Next, we need a product description field, which is always the same for a given product; it is related as one-to-one, therefore, to the product number, so we assign it to the product file. We then need an order date field, which is always the same for a given order; it is related as one-to-one to the order number, so we assign it to the orders file. (This also relates it as many-to-one to the product number and description, which is what we want.) Next, we need an order quantity field; like the date, it is related as one-to-one to the order number, so we assign it to the orders file.

This process, clearly, can be continued for each new field. And, since most requirements reflect common needs, an experienced programmer will easily design a correct database. Only in unusual situations do we have to analyze carefully the requirements to determine how to treat a new field.

The foregoing example, while very simple, already demonstrates that it is the application's requirements, not some database principles, that determine what is a correct database. Thus, if the requirements changed and the product description were permitted to differ from one order to the next, the description field would have to be in the orders file rather than the product file (because it would now be related as one-to-one to the order number, date, and quantity, and as many-to-one to the product number). Similarly, if the requirements permitted several lines in an order, the product number and quantity would be related as many-to-one to the order number. So they would be assigned to a separate file, order lines, where the key is the combination of order number and line number, and several records correspond to one order record. The order date, though, would stay in the orders file, because it continues to be related as one-to-one to the order number.

We know that relationships can be one-to-one, one-to-many (or many-to-one, if seen in reverse), many-to-many, and combinations of these. So, if we understand the role that a new field must play in the application, we already know what relationship to create, and hence to which file to assign it. (Key fields duplicated in another file in order to relate the two files are treated differently, of course.) All we need in order to design a correct database is to study the application's requirements. Then, we use an appropriate combination of relationships to represent these requirements. In other words, we create the database that matches the requirements – one field at a time. Ultimately, if we understand the requirements, we are bound to create a correct database.

And when we create a correct database, the problem of redundancy and inconsistencies does not arise. (The only time we must deal with this problem is when we *deliberately* introduce redundancy into the database; that is, when avoiding it would make the application too slow.) This is true because in a correct database all field relationships reflect actual requirements. Thus, in the foregoing example we assigned the product description to the product file because the requirements stated that it was the same for all orders. If we assigned it to the orders file instead, we would end up with unwanted duplication: a product's description would be repeated unnecessarily in each order that includes the product. The duplication can be explained by noting that this relationship does not reflect the requirements: the description field would be related as many-to-one to the fields in the product record, while the requirements called for a one-to-one relationship. (Alternatively, the error can be described as a one-to-one relationship with the fields in the

orders record, while the requirements called for a one-to-many relationship with these fields.)

The most important lesson from this analysis is that data redundancy and inconsistencies can only be defined within the context of a particular set of requirements. So this is not a problem that can be solved by means of a formal database theory. This is a *programming* problem, one that can be solved only by taking into account both the database structures and the other structures that make up the application. It is the way we plan to use the files that determines what are the correct relationships. And with correct relationships, there will be no redundancy or inconsistencies.

A database, then, can be correct only for a specific set of requirements. With just a small change in requirements, the same database would no longer be correct. The incorrectness may manifest itself in the form of wrong values or unnecessarily duplicated values. In the earlier example, storing the description in the product record is correct if it must be the same in all orders, and wrong if it must change; conversely, storing it with each order is correct if it must change, and wrong if it must be the same in all orders. The presence of redundancy and inconsistencies, therefore, when unintended, is similar to any other programming error: we neglected the requirements, and consequently the application malfunctions. The error, in this case, is a discrepancy between the required file relationships and the actual ones.

It is worth repeating: the concept of file and field relationships applies to relational databases *exactly* as it does to traditional ones, because both types are based on files, records, fields, and keys. Thus, even those programmers who prefer the relational model can benefit from the traditional design methods. They too can avoid data redundancy and inconsistencies by creating a correct database directly from requirements. Even with a relational database system, therefore, there is no need for a theory of normalization – because, if we create correct relationships, there is no redundancy or inconsistency to eliminate. As is the case with the traditional databases, we simply need to understand the application's requirements and the four types of relationships.

We can appreciate even better the connection between file relationships and the application's requirements if we remember that requirements are in effect rules, or restrictions. Specifically, from all the operations that the application can perform, and from all possible values that memory variables and database fields can take, only a few must be permitted if the application is to run correctly. One type of restrictions concerns the *combinations* of values that the database fields will display at run time: how the value of one field depends on

the value of another. And it is through the four types of file relationships that we implement these restrictions.

Two fields are related as one-to-one when they can have any combination of values; that is, when neither field depends on the other. Two fields are related as one-to-many when one field is restricted to a specific value by a series of values in the other. (Many-to-one is the same relationship seen in reverse.) And two fields are related as many-to-many when there are two simultaneous one-to-many restrictions: one field is restricted by the values of the other, and at the same time possesses values that restrict the other.

By interpreting the requirements as restrictions, we can explain the problem of redundancy as follows: we provide for all possible combinations of values in a situation where only a few can actually occur. If the requirement is for one-to-many and we place the two fields by mistake in the same file, they will be related as one-to-one. We provide for *any* combination of values when, in fact, the first field will have the *same* value for a series of values in the second. So that one value will be repeated unnecessarily every time the second field's value is in that series. We only need to specify their relationship once, and yet we do it several times.

There is an obvious correspondence between the various file relationships and the normal forms of the normalization theory: the relationship that is correct for a given requirement corresponds to the highest normal form attainable for that requirement (the one for which the files are deemed to be fully normalized). The relational theorists avoid the subject of file relationships – perhaps because this would reveal the shallowness of the normalization theory. Let us take a moment, though, to study this correspondence.

The first normal form is the highest one attainable when the application's requirements place no restriction on the combinations of values that two fields can take. From the perspective of the normalization theory, this means that there is no dependency between the two fields; so they can be assigned to the same file (or to separate files if those files are in a one-to-one relationship).

The second and higher normal forms can be attained when the application's requirements place some restrictions on the combinations of values. Because of these restrictions, the correct relationship is now one-to-many; and if we create one-to-one instead (by placing the fields in the same file), we will have a relationship that permits *any* combinations, while the actual data includes in fact only *some* combinations. The normalization theory describes this problem as a misplaced dependency: the only dependency permitted within a tuple is that of a non-key field on the field or fields that make up the key. We also note

the mistake in that the file is only in first normal form, while a higher normal form is now attainable. The solution is to place its fields in separate files, thereby creating files that are in second, third, or Boyce/Codd normal form. (Which form is actually attainable depends on the combination of field types, key or non-key, that constitutes the misplaced dependency.) In traditional terms, what we do when using two files instead of one is replace the incorrect one-to-one relationship with a one-to-many relationship, which is what the requirements had called for to begin with.

Combinations of these three normal forms correspond to combinations of one-to-many relationships: two "many" files sharing the same "one" file, or two "one" files sharing the same "many" file. They also correspond, therefore, to a many-to-many relationship between two files. The more complicated fourth and fifth normal forms correspond to various many-to-many relationships involving three or more files, when some of the two-file relationships are restricted.

But this correspondence, while perhaps interesting, is irrelevant; for, in practice we don't need to know anything about field dependencies, or about the notion of normal forms. We can create the correct relationships directly from requirements, as we saw earlier. We don't have to start with an incorrect, one-to-one relationship (as the normalization theory says), note the redundancy and inconsistencies, and then try to attain the correct relationship by discovering misplaced dependencies.

5

We are now in a position to explain the fallacies behind the delusion of normalization. We saw that all we need in order to create correct file relationships is to understand the application's requirements. We can avoid data redundancy and inconsistencies, therefore, simply by implementing relationships that match the requirements. But, while not especially difficult, this task demands skills that most programmers lack.

Without exception, the mechanistic software theories attempt to solve the problem of programming incompetence, not by encouraging programmers to improve their skills, but by providing *substitutes* for skills. The relational theory, in particular, was meant to obviate the need for database programming skills. Instead of the traditional file operations, which must be used through a programming language, programmers will only need to understand the high-level relational operations. Moreover, the mathematical foundation of the theory will guarantee data correctness: since the relational operations are as exact as mathematical functions, and since any database requirement

can be expressed as a combination of these operations, even inexperienced programmers will create correct database structures.

But, as we saw under the first delusion, the mathematical database model is a fantasy. To attain such a model, we must restrict it so much that it loses all practical value. If we divide the use of a mathematical system into translation (the conversion of the actual phenomenon into its mathematical representation) and manipulation (the work performed with the mathematical entities within the system), only the manipulation can be formal and exact. The translation entails an *interpretation* of the phenomenon, so it is necessarily informal. The relational model is senseless because it consists almost entirely of the translation. The manipulation, while indeed exact, forms a very small part of the model; and, besides, it is so simple that we can implement the same operations by relying on common sense alone. The theorists praise the mathematical benefits of the model, but these benefits can only help us to deal with a few, simple aspects of database work. Most work, including the most difficult aspects, lie outside the scope of the formal model. So, in the end, we need the same programming skills as before.

If the manipulation includes only the little that can be reduced to an exact representation, every other aspect of database work must become part of the translation. This includes the *design* of the database; that is, discovering the combinations of files and fields that correctly represent the real entities and the relationships between them. With a traditional database or a relational one, this is an informal activity: using our knowledge and experience, we study the application's requirements and ensure that the database entities and relationships match the real ones. And if we accomplish this, there will be no redundancy or inconsistencies. The relational theory never promised to replace this activity with an exact method; it simply left the issue out of the formal model (along with such other issues as integrity rules, updating operations, database language, and database performance).

The relational theory, thus, failed to eliminate the need for programming skills. Programmers continued to create incorrect database structures, but the theorists did not recognize this problem – the fact that so much had to be left out of the formal model – as a falsification of the relational concept. So, instead of studying the problem, they introduced an additional concept – the normalization theory. Their attitude, in other words, did not change: confronted with the evidence that mechanistic theories cannot be a substitute for expertise, they hoped to contend with the persisting incompetence by inventing yet another substitute. The second relational delusion (the delusion of normalization) emerged, therefore, because the theorists refused to face the first one (the delusion of a formal database model).

The normalization theory differs from the original relational theory in that

it promises us exact methods for identifying the incorrect file relationships, not *before*, but *after* they are implemented. Rather than invoking the power of mathematics to *prevent* a bad design (something that everyone now agrees is impossible), we are told that the same power can be invoked to *correct* a bad design. Clearly, the theorists do not see the absurdity of this idea. For, were it possible to discover formally the incorrect relationships in an *existing* database, we could also discover them formally *while designing* the database. The phenomenon is the same in both cases: file relationships that do not match the application's requirements.

So the theorists still fail to understand why the original model could not help us to design correct file relationships. This is not a technical problem that might be solved with an additional theory, but a fundamental limitation: it is only through an informal interpretation of the requirements that we can determine what *are* the correct relationships. Thus, there is no difference between determining this *while* designing the database or *after*. In both cases, we must process the database structures together with the other structures that make up the application; in particular, the business practices reflected in the application. In both cases, then, we must deal with the complex structure that is the whole application, and this is something that only minds can do.

The normalization theory claims to eliminate the need for expertise by eliminating the need to design correct databases. Unlike the traditional design methods, which expect us to create file relationships that match the requirements, the new method permits us to create relationships that are as incorrect as we like. To take an extreme case, we can ignore the need for file relationships altogether: we create a database that consists of just one file, and assign *all* the fields to this file, regardless of their actual relationships. We can do this because the database we create now is only a starting point. By applying the principles of normalization, we will be able to transform the incorrect database, step by step, into a correct one.

As we know, files created within the formal model are already in first normal form. To attain the higher normal forms, we must modify the database by discovering and eliminating the misplaced field dependencies. And this can be accomplished, we are told, through the formal methods provided by the normalization theory. Through one procedure we eliminate one type of dependency, and thereby convert the files from first to second normal form; then, through another procedure we eliminate a different type of dependency, and convert them from second to third normal form; and so on. We continue this process until we find at a certain level – a level that varies from one

database to another – that there are no misplaced dependencies left. At that point, the database is fully normalized. By eliminating all misplaced dependencies, we eliminated the possibility for any data redundancy or inconsistencies to emerge later, when the database is used.

The normalization theory, thus, claims to have solved the problem of programming incompetence by replacing the challenge of designing a correct database, with an easier challenge: eliminating the errors found in an existing, incorrect database. This shift, the theorists believe, reduces database design to a series of simple, mechanical activities. Their naivety is so great that, although the logic is the same (matching the file relationships to the application's requirements), and although the ultimate database is the same, they believe that the new principles are formal and exact while the old ones are not.

In the end, the problem of design became the problem of dependency: an elaborate system for defining, analyzing, and classifying the field dependencies found in a database. Date describes this shift perfectly: "The fact is, the theory of normalization and related topics – now usually known as *dependency theory* – has grown into a very considerable field in its own right, with several distinct (though of course interrelated) aspects and with a very extensive literature. Research in the area is continuing, and indeed flourishing."[4] But this research is a fraud: the theorists are distorting and complicating the problem of database design in order to have a reason for seeking an alternative. The delusion is not so much in the shift from design to dependency, as in the belief that this shift has turned the problem into a formal theory; specifically, the belief that we have now exact methods to prevent redundancy and inconsistencies.

In reality, redundancy, inconsistencies, and misplaced dependencies are different aspects of the same phenomenon: a discrepancy between the file relationships and the application's requirements. Thus, whether we wish to avoid redundancy and inconsistencies, or to eliminate misplaced dependencies, the only way to do it is by interpreting the requirements correctly; and this task cannot be formalized. What the theorists did is *add* to this task a complicated system of principles and procedures – the theory of normalization. And it is only this theory that is formal and exact. Their "research," then, is merely a preoccupation with this theory, with the problems they invented themselves. The real problem – creating a correct database – is as informal as before. So, if the normalization principles did not replace the original problem, if we continue to assess dependencies informally, the normalization theory is fraudulent.

[4] C. J. Date, *An Introduction to Database Systems*, 6th ed. (Reading, MA: Addison-Wesley, 1995), p. 337.

To repeat, dependency is indeed part of the same phenomenon that causes redundancy and inconsistencies. So the shift from design principles to dependency principles is wrong only because it is unnecessary, because it complicates the problem without providing any benefits in return. Recalling the earlier examples, repeating the unchangeable product description in every order entails redundancy. We can describe this redundancy as the result of an incorrect relationship: we created a one-to-one relationship between the description field and fields like order number, when their required relationship is one-to-many. But we can also describe the redundancy as the result of a misplaced dependency: the description depends on the part number, which is not the key in the orders file. Regardless of how we describe the redundancy, though, it is the incorrect relationship between the product description and the other fields that is the root of the problem. And in both cases it is this relationship that must be modified in order to solve the problem.

Generally, with the traditional design concept we create the correct relationships from the start. With the normalization theory, we start by creating one-to-one relationships – which are usually wrong, because most relationships are one-to-many or many-to-many; we then search for misplaced dependencies, which direct us to the incorrect relationships; and finally, we modify the relationships in order to eliminate those dependencies, and with them the redundancy and inconsistencies.

But with both the traditional method and the new one, we always reach the point where we must decide, for a given field, whether it must be in the same file as some other fields, or in another file. With the traditional method, this decision is also the design. With the normalization theory, this decision is only a small part in a long and complicated process. For, now we must also identify the current normal form, determine the type of dependency between fields and the higher normal form that would eliminate it, and convert the files to that normal form.

The decision itself, however, entails the same challenge: interpreting the application's requirements correctly. Thus, what is the critical step with both design methods – discovering the correct relationship between two fields – is necessarily an *informal* process. So the formality of the normalization theory is silly if normalization depends ultimately on an informal process, just like the traditional method. Before, we made that decision in order to create a correct file relationship. Now we make it in order to correct an incorrect one. But, if in the end it is only through our interpretation of the requirements that we can determine what *is* the correct relationship, we may as well use the traditional method, which is so much simpler.

To conclude, there are two stages to the delusion of normalization. The first stage is the belief that we need a theory of normalization at all; namely, that preventing redundancy and inconsistencies is a special problem, which demands a formal theory. This problem, though, is no different from all the other problems that make up the challenge of programming. Regardless of which aspect of the application we are dealing with, we must create structures of software entities that correctly represent the structures of real entities. And to accomplish this task we must understand the application's requirements and the means of implementing them. Moreover, a given requirement usually affects *several* aspects of the application, and we cannot deal with them separately. The database structures, in particular, are always linked to the other structures that make up the application. Searching for a formal, mechanistic theory of database design is an absurd and futile quest.

The theorists assume that it is impossible, or very difficult, to design a correct database directly from requirements; that programmers cannot attain the necessary expertise, so this task must be replaced with a method which they can follow mechanically; and that it is possible to discover such a method. But, quite apart from the fact that no formal method can exist, the traditional design principles already provide a fairly simple method for creating correct databases. All we need to do is determine, for each new field, the appropriate relationship with the existing fields (one-to-one, one-to-many, many-to-one, or many-to-many). If we do this, we will end up with a correct database – a database that matches the requirements. And, among the many benefits of a correct design, there will be no redundancy or inconsistencies.

The second stage in the delusion of normalization is the belief that the body of principles that make up this theory constitutes indeed a formal solution to the problem of database design. In reality, the database structures are still based on the relationships between fields, and we can only determine the correct relationships by interpreting the requirements; in other words, informally, just as before. The theorists think that studying field dependencies rather than field relationships has resulted in a method that is formal and exact, but what is formal and exact is only the new principles. These principles did not replace the informal task of understanding the requirements; so that task – upon which the correctness of the database ultimately depends – has remained unchanged.

If we divide the design process into two parts, formal and informal, the traditional method is almost entirely informal, while the new one is almost entirely formal. But this improvement is an illusion. What confuses the theorists is that the part which they invented, and which is indeed formal, keeps growing, while the traditional part (understanding the requirements) remains the same. Recalling an earlier quotation, research in this area is flourishing. Thus, the more preoccupied they are with the dependency theory,

the smaller the informal part appears to be. The informal part, after all, consists simply in determining, for a given field, its relationship with the other fields. In the end, though, this decision is the only thing that matters – what will make the database correct or incorrect – with both the traditional method and the new one. But, while this decision is practically the whole design process with the traditional method, with the new method it is such a small part that it goes unnoticed. So the theorists delude themselves that the new method is entirely formal.

The formal part, thus, did not eliminate the informal one in the new method; it is *additional* to it. The formal part, while impressive, is absurd if the correctness of the database depends ultimately on the small part that is informal – on the part that, with the traditional method, is the only thing we need.

So the conclusion must be that the concept of normalization is worthless. It is an artificial, unnecessary theory. The critical part is still the informal task of determining what field relationships match the application's requirements. But by spending most of their time with formal and complicated procedures, and only moments with that informal task, the relational enthusiasts can claim that database design is now an exact science.

We examined earlier the first stage of the delusion of normalization: the belief that we need some new, formal principles, when in fact the traditional concepts provide an excellent and relatively simple design method. In the following pages we examine the second stage: the belief that the principles of normalization provide indeed a formal design method, when in fact the critical part is as informal as before.

6

Like predicate calculus, which inspired it, the formal relational model is a true mathematical system, complete with operations and formulas. Its weakness, we saw under the first delusion, is only that it is irrelevant to database work: when we depict the use of a relational system as the *translation* of database entities into mathematical ones and their *manipulation* within the system, we find that the manipulation – the most important aspect in other mathematical systems, and the reason for performing the translation – plays an insignificant part.

The normalization theory, on the other hand, is not a mathematical system at all. The theorists discuss it as seriously as they do the formal relational model, but on closer analysis we discover that all they do is *present* it formally. There are no true operations or formulas in this theory, as there are in the formal model; all we have is a study of field dependencies, expressed through

formal notation. The theory of normalization, in other words, consists *entirely* of a process of translation: from the real entities into the relational ones. There is no manipulation at all. The only operations available are those we had under the formal relational model.

An example of the specious mathematics of the normalization theory is found in a long paper written by E. F. Codd – a paper generally regarded as the most rigorous treatment of the second and third normal forms.[5] The paper provides an exhaustive analysis of field dependencies and their elimination, but despite the formal tone and terminology, this is not a mathematical theory. The paper describes various combinations of data elements, and represents their relationships and dependencies by means of a formal system of notation. The resulting expressions *look* perhaps like mathematical formulas, but they serve no purpose beyond this representation. Page after page of expressions are, in reality, only the *translation* of files and fields into the new notation. Once the translation is complete, we have no way to *manipulate* the expressions. All the system does, then, is represent field dependencies formally. Were this a true mathematical system, we would have some new relational operations, to replace the original ones.

We find the same style in thousands of other writings. What is described as mathematics is merely a system of definitions and theorems expressing in formal notation various issues pertaining to the subject of field dependency. Typically, the papers introduce new terms and define them through references to other terms, show how to derive certain parts of the system from other parts, prove that if certain conditions hold then other conditions will also hold, and so forth. And this is where the mathematics ends.

It is the introduction of new terms that the authors are especially fond of. The relational theory in general overwhelms us with new terminology, but the principles of normalization in particular seem to require some new terms at every step. Thus, along with the formal tone, the rich terminology helps to make the normalization theory appear important, no matter how shallow it actually is. But, while the mathematical style of these writings impresses naive readers, an intelligent person merely finds the writings incomprehensible. The reason is that, since we know that the whole theory is unnecessary, we have little motivation to assimilate the countless terms and definitions; and without understanding the new concepts it is impossible to follow the author's discussion.

To convey the flavour of this style, I will quote a few lines from Date's book (out of the seventy pages devoted to the subject of normalization). After

[5] E. F. Codd, "Further Normalization of the Data Base Relational Model," in *Data Base Systems*, ed. Randall Rustin (Englewood Cliffs, NJ: Prentice Hall, 1972), pp. 33–64.

presenting several related theorems, Date defines the fourth normal form as follows: "Relation *R* is in 4NF if and only if, whenever there exist subsets *A* and *B* of the attributes of *R* such that the (nontrivial) MVD $A \twoheadrightarrow B$ is satisfied, then all attributes of *R* are also *functionally* dependent on *A*."[6] Concepts like "nontrivial," "MVD," and "functionally dependent," used in this definition, are explained on previous pages. For example, MVD (multivalued dependency) is defined as follows: "Let *R* be a relation, and let *A*, *B*, and *C* be arbitrary subsets of the set of attributes of *R*. Then we say that *B* is *multidependent* on *A* – in symbols, $A \twoheadrightarrow B$ (read '*A* multidetermines *B*,' or simply '*A* double-arrow *B*') – if and only if the set of *B*-values matching a given (*A*-value, *C*-value) pair in *R* depends only on the *A*-value and is independent of the *C*-value."[7]

It is also worth mentioning the following warning: "We stress the point that the discussions that follow are intended to explain a *formal theory*, albeit in a fairly informal manner."[8] In other words, definitions and explanations like those quoted above, and the endless formulas and diagrams, are not the actual theory but a *simplified* version. For the *really* formal discussion we must consult the original papers, in academic journals.

To summarize, all that the normalization theory does is represent formally the relationships between fields. A true mathematical system would provide operations that combine entities to create increasingly high levels, as do the systems used in engineering. There are no such operations here, so the normalization theory does not describe a mathematical system. What it describes is *a formal system of representation*. This system may have its uses, but not in the way a mathematical system has. In the end, the only mathematical manipulation remains the one provided by the original relational model. The normalization theory is not a true enhancement of that model.

So what the relational theorists invented is akin to a game. The normalization work is *additional* to the task of studying and implementing the application's requirements. That task has remained as important – and as informal – as before. It is only the game that is formal and exact. This is a sophisticated and difficult game, demanding a special kind of knowledge. It is not surprising, therefore, that the academics who invented it, and the practitioners who learn it, feel that their normalization work is a sign of expertise. This is expertise in playing a game, though, not in designing databases.

[6] Date, *Database Systems*, p. 329. [7] Ibid., p. 328. [8] Ibid., p. 327.

7

The reason we cannot have a formal *and* useful theory of normalization is that the dependency of one field on another is not a *database* problem, but part of the application's logic. Formal normalization principles can only deal with the *database* structures. They cannot take into account the other structures that make up the application – the business practices, for instance. And it is these other structures that determine, ultimately, the relationships between database fields. A formal theory, thus, can deal with such issues as the definition and classification of dependencies, or the conversion from one normal form to another; but it cannot tell us whether the relationships are correct. In particular, no formal theory can tell us to which file to assign a given field. Only our knowledge of the application can do this.

Recalling the earlier examples, assigning the product description to the same file as the product number is not right or wrong in an absolute sense, but only relative to the requirements being implemented: if the description is fixed, it should be in the same file; if changeable, in the other file. We *must* understand the requirements. And when we do, we already know how to implement them: as a one-to-one or as a one-to-many relationship. Thus, a formal theory cannot replace the need to study the requirements, and is unnecessary once we understand them. It is, in other words, useless.

Let us take another example. An employee file usually includes such fields as department, position, salary, seniority code, and vacation code. Now, these fields may be related in one company, and unrelated in another. The salary, for instance, may be independent, or the same for all the employees with a particular position; the vacation code may be independent, or the same for all the employees with a particular seniority; the position and salary may be independent, or the same for all the employees in a particular department. Some of these fields, therefore, may be dependent on others, in which case they should be moved into separate files: a salary file where the key is the position, a vacation code file where the key is the seniority code, and so on. But only *we* can know whether a given field is or is not independent; and we would know this in the same way we know the other requirements that define the payroll application. The same application, in fact, may be used by two companies while a certain field is independent in one but not in the other. So, just like the business practices that make up an application, the normalization requirements may be different in each case; and as a result, a database that is deemed to be normalized for one company may not be for the other. Again, since it is only *we* that can discover the field relationships, a formal theory is useless.

Another situation where the need for normalization is determined largely by our knowledge of the application occurs when files are updated only under certain conditions. Thus, some files may be used by the application in such a way that a relationship of dependency between two fields in the same record would be harmless. For example, records may be added but not modified or deleted; or those fields alone may never be modified. Also, there are situations where it may be simpler or more efficient to deal with the problem of dependency through the application's logic, rather than through database restrictions. In all these situations, what we do is simplify the application or improve its performance by noting that not all conceivable database operations will *actually* be performed. Clearly, no formal theory can include such knowledge.

The only formal theory of normalization possible is one that assumes the worst case; namely, the case where *every* field may depend on another field. As we saw, we eliminate each dependency by separating the two fields: we place one field in a new file, where the records are linked through their key to the field left in the first file. Thus, if we want to be absolutely certain that there are no dependencies, and if we don't want to rely on an interpretation of the requirements, we must separate in this manner every field, in every file. In the end, every file in the database will have only one non-key field. This is an exact, formal procedure – a procedure that can even be automated. However, because many of the separated fields must be put back together in the running application, this overnormalization would make the application too complicated and too slow; so no one seriously suggests that we follow it. (In fact, as we will see under the third delusion, even minimal normalization – separating just a few fields – is often impractical and must be forsaken.)

A database where the smallest necessary number of fields (rather than an arbitrarily large number) were separated in an attempt to eliminate all dependencies is said to be in optimal second normal form. This sounds like a precise definition, but in reality it is only informally, through our knowledge of the application, that we can determine whether or not the normalization of a given database is "optimal." Again, the only way to have a formal theory is by separating *every* field in the database, regardless of how it is used in the application.

But even if we succeeded somehow in developing an exact and complete theory of normalization, it would still be inadequate. This is true because normalization deals only with dependencies that can be eliminated by separating fields. There are many other types of field dependencies in an application, all a natural

part of the application's logic. Every application includes operations that relate fields in the same record, or fields in separate files. Some of these fields, therefore, depend on others; so they are, strictly speaking, unnecessary. But we cannot eliminate these dependencies through normalization, by separating fields.

Let us examine a simple example of the type of dependency that cannot be eliminated through normalization – the classic case of aged balances. The customer balance, for instance, is usually stored in several fields in the customer record: current, thirty-day, sixty-day, and ninety-day balances. And there is usually an additional field, for the total balance, which is the sum of the other four. But if the total balance is always the sum of the aged balances, its field can be eliminated. Instead of having a separate field, we can calculate the total balance (by adding the other fields) wherever we need it in the application. The reason we usually retain the total balance field is that this is simpler than calculating it: in most applications we modify it in only a couple of places (typically, when invoicing the customer and when receiving payments), but we show it in dozens of inquiries and reports. So it is simpler to update the total balance in the few places where an aged balance changes, and merely to *read* it in the other places.

It is obvious that the dependency of the total balance on the aged balances cannot be eliminated through normalization, by moving the total balance into a new file. What we do for this type of dependency, therefore, is similar to what we do when we decide *not* to normalize in situations where normalization *is* possible: we anticipate the problems that may be caused by the updating operations, and we add to the application's logic the necessary steps to prevent them. Thus, in the case of balances, we must remember to update the total balance too, when one of the aged ones is updated. And if we neglect this, we will face "update anomalies" (the total balance will no longer equal the sum of the aged ones) not unlike those that occur in unnormalized files when we ignore the effect of updating operations.

To continue this example, in most applications the aged balances themselves can be calculated, using a transactions file: we read the records belonging to a particular customer, and total the invoice and payment amounts under four different periods. So the aged balance fields too are dependent on other fields, and hence unnecessary (although the original data is now in another file). Also like the previous dependency, this dependency cannot be resolved through normalization. To prevent "update anomalies" (balance fields different from the sum of the transactions), we must either eliminate the balance fields, or ensure that they are updated whenever a record is added to the transactions file. (In this case, though, eliminating the fields is rarely practical, because it is too inefficient to calculate them by reading the transaction records every time.)

So what is the point in seeking a formal theory of normalization, if this theory would eliminate only *some* dependencies? Clearly, there is no limit to the types of field dependencies that can exist in an application – types like the ones we have just examined. In fact, we don't even think of these dependencies as a database problem, but as various aspects of the application's logic. Since most software requirements involve database fields – fields belonging to one file or to several files – it is natural to find relationships of dependency between fields. And it would be absurd to eliminate these relationships solely in order to avoid redundancy, or to avoid inconsistencies in updating operations. What we do in each case is seek the most effective design: we eliminate the dependency when practical, and deal with the updating problems as part of the application's logic when this is simpler or makes the application faster.

In the end, all field dependencies cause similar problems, and we can only deal with these problems by taking into account not just the database structures but *all* the structures that make up the application. These are not *database* problems but ordinary programming problems, similar to the many other problems we face when developing an application. And it is just as futile to search for an exact and complete theory of field dependency as it is to search for an exact and complete theory of programming. The relational theorists isolated *one type* of dependency – the type that can be eliminated by separating fields; and they naively concluded that, if we eliminate this one type, we will eliminate *all* the problems caused by dependency (or, at least, the most common problems).

This belief is reflected in the relational vocabulary (terms like "normalize" and "normal form" imply a particular, proper data format) and in the numbering system (the fifth normal form is said to be the last and most stringent one). Hardly ever are the other types of dependencies mentioned at all. Date discusses them briefly: "5NF is the *ultimate* normal form with respect to projection and join.… That is, a relation in 5NF is *guaranteed to be free of anomalies* that can be eliminated by taking projections [i.e., by separating fields].… Of course, this remark does not mean that the relation is free of *all possible* anomalies. It just means (to repeat) that it is free of anomalies that can be removed by taking projections."[9] Most authors, however, depict the process of normalization as a final refinement, as a guarantee of database validity.

Thus, by emphasizing the few dependencies that can be eliminated through normalization while disregarding the many that cannot, the relational experts make the normalization principles appear more important than they really are. Then, they use this misrepresentation to rationalize their search for a theory of normalization.

[9] Ibid., p. 334 and footnote.

8

If the theory of normalization is unnecessary, if the traditional design method permits us to avoid redundancy and inconsistencies simply by understanding the application's requirements, how do the theorists justify their lengthy discussions? By distorting the problem of database design. They describe some contrived database structures that are incorrect but hardly ever occur in practice, and then they show us how to turn them into correct ones.

The only theory they can offer us is one that studies the so-called normal forms and gives us methods to convert files from one form to another. But we need such a theory only if we normally create incorrect databases. The theorists present the incorrect databases as a common occurrence, and the concept of normalization appears then important. In reality, we can create correct databases from the start, by selecting file relationships that match the application's requirements. So the classification of normal forms and the conversion procedures have no practical value.

I will illustrate this distortion now with a few examples taken from database books. In all these situations, we will see, the correct design can be easily determined from the requirements. The authors, however, *ignore* the requirements, and start with a *deliberately incorrect* design: a single file, when several are needed. They start, that is, with a one-to-one relationship when the requirements call for one-to-many or many-to-many. They point to the problems caused by the incorrect design, and *then* they study the requirements and show us how to arrive at the correct one: through normalization.

The examples, in other words, are presented so as to make the theory of normalization, which in reality is totally unnecessary, look like an indispensable concept in database design. Moreover, their method is so lengthy and complicated that the reader is likely to miss the fact that its preciseness and formality are specious: the most important decisions – identifying the misplaced field dependencies – are still being made, not mathematically, but through an *informal interpretation* of the requirements.

Brathwaite demonstrates the second normal form with this simple problem:[10] we want to store some information about students and about the classes they

[10] Ken S. Brathwaite, *Relational Databases: Concepts, Design, and Administration* (New York: McGraw-Hill, 1991), pp. 76–77.

attend; students are identified by a student number, and we must record their name and major; classes are identified by a class number, and we must record the class location and time; a student may attend several classes, and we must be able to identify these classes.

Ignoring all we know about normalization, we note that the students and classes form a many-to-many relationship (a student attends several classes, and a class is attended by several students). So the student number and class number must be in separate files: a student file, where the student number is the key, and a class file, where the class number is the key. The student name and major are both related as one-to-one to the student number, so they must be non-key fields in the student file. Similarly, the class location and time are related as one-to-one to the class number, so they must be non-key fields in the class file. Lastly, to link the two files, we need a service file where the key is the combination of student number and class number. In a traditional database, the service file could then have two indexes: class number within student number (to select the class records associated with a student), and student number within class number (to select the student records associated with a class). But the requirements call only for the link from student to classes, so we need in fact only the first index. (It is worth noting that in a real application the link file wouldn't be just a service file; it would also have some non-key fields, for data that is related as one-to-one to its key – the student's grade, for instance.)

Brathwaite, though, attempts to implement the requirements with *one* file: the combination of student number and class number is the key, while the student name and major, and the class location and time, are non-key fields. Then, he notes the problems caused by this design: no information can be stored about a particular student unless the student is enrolled in at least one class, or about a particular class unless at least one student attends it. Also, a certain name and major will be repeated for every class attended by that student, and a certain location and time will be repeated for every student attending that class; so if these values change, several records would have to be updated.

What causes these problems, Brathwaite explains, is the dependency of non-key fields on *part* of the key: while the key includes both the student and the class numbers, the student name and major depend only on the student number, and the class location and time only on the class number. Non-key fields must depend on the whole key, so the solution is to create a separate file for the two student-related fields, with the student number alone as the key, and another file for the two class-related fields, with the class number alone as the key. What will be left in the original file is just its key, the student and class numbers. This design eliminates all the aforementioned problems.

The final database, thus, is identical to the one we created earlier, directly from the requirements. We knew all along that it was correct, simply because it reflects accurately the requirements. Now, however, we are told that it is correct because the files are in second normal form (whereas the original file, with all fields bundled together, was only in first normal form).

What is the point of this approach? Starting with one file would make sense, perhaps, if the method used to reach the final design were indeed formal and exact (in which case we could even automate the design process). But the misplaced dependencies were discovered *informally*, by interpreting the requirements. For instance, when noting that the name and major depend only on the student number, we used the same information and the same logic as we used earlier, when noting that they are related as one-to-one to the student number. With normalization as much as with the traditional method, we relied on skill and common sense, not on mathematics. Thus, if we know how to determine the relationship between two fields, we may as well use this knowledge directly to assign them to the proper files. Why bundle them first in one file, and then use this knowledge to *separate* them?

So the part that is formal – the classification of field dependencies – did not replace the need for, nor the importance of, the part that is informal. The correctness of the normalization depends, ultimately, on the correct interpretation of the requirements. The fancy terminology makes the process of normalization seem more exact than the traditional method, when in reality it is merely more complicated.

Date starts his discussion of the second and third normal forms with the following problem.[11] Let us imagine that we purchase parts from a number of suppliers, located in different cities and identified by a supplier number; the cities are identified by the city name, and each city has a status associated with it; several suppliers may be located in the same city; a supplier can sell different parts, which are identified by a part number; and we want to record our purchase orders by storing for each order the supplier number, part number, and quantity. (The requirements assume, for the sake of simplicity, that only one order exists at a given time for each combination of supplier and part number, so we don't need order numbers. Also, the requirements call for the capability to identify directly the city of a given supplier, but not the suppliers in a given city.)

With our knowledge of file and field relationships, we can translate these

[11] Date, *Database Systems*, pp. 297–303.

requirements into the following design. We note first that the city is related as one-to-many to the supplier, so we need two files: a city file, where the key is the city name, and a supplier file. In one-to-many relationships, the key of the "many" file includes usually the "one" file's key; so here it would be the combination of city name and supplier number. But the present requirements do not call for selecting the suppliers in a given city, so the key in the supplier file can be just the supplier number. We do have to select the city associated with a supplier, though, so we include the city name as a non-key field. The status is related as one-to-one to the city, so we add it as a non-key field to the city file. The supplier number is related as one-to-many to the order-related fields, part number and quantity; so these fields must be in a third file, orders, where the key is the combination of supplier number and part number.

Date, however, says nothing about these relationships. He starts by bundling all five fields (supplier number, status, city name, part number, and quantity) in one file: the orders file, where the key is the combination of supplier number and part number. And immediately he notes the consequent redundancy and anomalies: Since there must be a record in this file for every order, the information that a certain supplier is located in a certain city will be repeated for every order from that supplier; so, if the supplier relocates to another city, we will have to modify several records. Similarly, the information that a certain city has a certain status will be repeated for every order from every supplier in that city; so, if the status changes, we will have to modify several records. Lastly, we cannot store the information that a certain supplier is located in a certain city unless an order exists for that supplier.

Date then presents the solution. The first step is to separate the fields by creating a new file: the supplier file, where the key is the supplier number, and the city name and status are non-key fields. The quantity is left in the orders file. Since each combination of supplier and city appears now in only one record, the redundancy associated with the city, along with the update anomalies, has been eliminated. The solution can be expressed in terms of misplaced dependencies: while non-key fields must depend on the whole key, the city and status in the original file were dependent only on the supplier (they are the same for all the orders from a given supplier). In terms of normalization, the problem was solved because the new files are in second normal form, while the original one was only in first normal form.

But this still leaves the other redundancy: the status of a certain city is repeated in the supplier file for every supplier located in that city. Although not as bad as in the original file (where the repetition was for every *order* from every supplier in that city), this redundancy will nevertheless cause the same kind of problems. The misplaced dependency that must be eliminated now is between the status and the city (two non-key fields). So we create a new file:

the city file, where the key is the city name, and the status is a non-key field. The supplier file will then be left with only the city as a non-key field. In terms of normalization, the problem was solved because these two files are in third normal form. In other words, while the second is the highest normal form attainable for the orders file, we can attain the third for the supplier file by creating a separate city file; and a database is fully normalized only when each file is in its highest attainable normal form. (The difference between the second and third is in the type of misplaced dependency that is eliminated: on only a portion of the key, and on a non-key field.)

So by the time he is done, Date ends up with exactly the same database as the one we created directly from requirements with the traditional design method. The normalization method is more complicated, and we *still* depend on an informal decision: we identify the misplaced field dependencies by interpreting the requirements, the same way we identified the correct field relationships before. What is formal is only the *analysis* of these dependencies and the *conversion* from one normal form to another; that is, the work that is *additional* to the task of identifying them.

Carter uses the example of an employee file to demonstrate the fourth normal form.[12] Specifically, we have to store for each employee, in addition to his name, some data about his children and about his salary history. Thus, we need a set of fields for each child (identified by the child's name), and a set of fields for the salary of each past year (identified by the year). We will have an employee file where the employee number is key, and the name (related as one-to-one to the number) is a non-key field. And we will have two one-to-many relationships, with the employee file acting as shared "one" file: between employee and children, and between employee and salary history. In the children file, the key will be the combination of employee number and child name; and in the salary history file, the combination of employee number and year. We will then be able to select for a given employee the corresponding child records and history records; and for a given child or year, the corresponding employee record.

Carter, however, starts by showing us what would happen if we placed the child and salary history fields in the same file – a file where the key is the combination of employee number, child name, and year: we would have to repeat the entire salary history for each child. For instance, for an employee with 3 children and 10 years of history, there would be 30 records in this file:

[12] John Carter, *The Relational Database* (London: Chapman and Hall, 1995), pp. 135–150.

one record for each combination of child and year. This design, therefore, would cause redundancy and anomalies: to add or modify the data for one child, we would have to add or modify 10 records (because the same child data is stored for each year); and to add or modify the history data for one year, we would have to add or modify 3 records (because the same history data is stored for each child).

Now, *no one* would try to combine child data and salary history in one file. Carter must start with this absurd design in order to demonstrate the benefits of normalization. It is pointless to describe his actual analysis – fifteen pages of complicated principles, definitions, and diagrams related to the fourth normal form, not to mention nearly forty prior pages dealing with the lower normal forms. Briefly, that file suffers from multivalued dependencies (i.e., several fields dependent on one another). The solution is to separate it into two files, one for child data and the other for salary history – which is exactly how *we* designed the database to begin with.

The redundancy and anomalies were eliminated, we are told, because these files are in fourth normal form, while the original file was only in Boyce/Codd normal form. But *we* know that the database is correct simply because it expresses two one-to-many relationships, which is what the requirements actually called for. Carter needs an enormously complicated procedure to reach the same design that *we* reached simply by implementing, directly from requirements, the appropriate file relationships. Moreover, the critical observation that the child data and history data must be separated could only be made *informally*, by studying the requirements – just as we identified the file relationships with the traditional design method.

Date explains the fourth normal form with a more difficult example.[13] We are asked to design a database to express the relationships between the courses, teachers, and textbooks in a certain school, with the following requirements: a particular course may be taught by one or more teachers, and a teacher may teach one or more courses; a particular course may use one or more textbooks, and a textbook may be used in one or more courses; a particular course always uses the same textbooks, regardless of the teacher.

Studying the requirements, we note two many-to-many relationships: between courses and teachers, and between courses and textbooks. We need, therefore, three main files (courses, teachers, and textbooks) linked through two service files. To satisfy the requirement that a teacher may teach several

[13] Date, *Database Systems*, pp. 325–329.

courses and at the same time a course may be taught by several teachers, we create a service file where the key is the combination of course and teacher; and to satisfy the requirement that a course may use several textbooks and at the same time a textbook may be used in several courses, we create a service file where the key is the combination of course and textbook.

As usual, in order to implement the two-way links between files (course to teacher and teacher to course, course to textbook and textbook to course), the service files must provide *both* sorting sequences: teachers within courses and courses within teachers, textbooks within courses and courses within textbooks. (Thus, if we use a traditional database, there will be two indexes for each service file.) We will then be able to select for a given course the corresponding records in the teachers file, and for a given teacher the corresponding records in the courses file; and we will also be able to select for a given course the corresponding records in the textbooks file, and for a given textbook the corresponding records in the courses file.

This, then, is how a sensible database book would present the example – the problem and the solution. Let us see now how Date presents it. He starts by attempting to implement all the relationships with *one* service file – a file where the key is the combination of course, teacher, and textbook. (So there is one record in the file for each combination of values in the three fields.) But this design is absurd; it is deliberately incorrect in order to demonstrate the transition from one normal form to another. The file, Date explains, is only in Boyce/Codd normal form, and this gives rise to redundancy and anomalies. For instance, if a particular course uses two textbooks, we will need two records for every teacher who teaches that course, although all teachers use the same textbooks. In addition to this duplication, we would have to add, delete, or modify several records (one for each teacher) when adding, deleting, or modifying the information about a textbook. Expressing the problem in terms of dependencies, the design is incorrect because it permits a multivalued dependency.

But this is a gross simplification of Date's actual explanation – four pages of complicated pseudo-mathematical analysis, which is in fact incomprehensible without a good understanding of some fifty prior pages on the subject of normalization.

The solution, Date concludes, is to have two service files rather than one, and to separate the three key fields into two sets of two fields.[14] More specifically, it is the teacher and textbook fields that must be separated.

[14] It must be noted that Date does not call these files *service* files, thus suggesting that they are the main data files (i.e., tables). A real application, though, would also require some non-key fields, to store details about courses, teachers, and textbooks; and such fields would not be added to these files, because that would cause much redundancy.

The result, needless to say, is the two service files we created previously, when we implemented the database as two many-to-many relationships. The redundancy and anomalies were eliminated, we learn now, because these files are in fourth normal form.

The design method based on file relationships, we saw, leads directly to the correct database. Date describes a situation that is a good example of a fundamental database concept, the many-to-many relationship. But instead of discussing this concept, he presents a silly, deliberately incorrect design. Then, he uses this design to justify the need for the normalization theory.

And, as in the previous examples, the complexity of the normalization masks the fact that the critical step (the observation that it is the teachers and textbooks fields that must be separated) was based on an *informal* interpretation of the requirements – exactly the same interpretation that helped us to determine the correct relationships with the traditional method.

9

We saw earlier that the principles of normalization are not, in fact, required by the original relational model: they are not an extension of the formal model, but an attempt to formalize the process of database design (see pp. 732–734). The normalization theory is, in effect, an independent theory – a theory that can be applied to any system based on records, fields, and keys. Thus, we can study the normalization theory on its own, ignoring the relational model altogether. And when doing so, its character as a mechanistic delusion becomes even clearer. By way of summary, therefore, I want to recapitulate the normalization fallacies and to show how they arose from the mechanistic way of thinking that pervades the academic world.

Mechanists attempt to explain a complex phenomenon, which can only be represented with a complex structure, by breaking it down into simpler phenomena: they extract smaller and smaller aspects of it, until they reach an aspect that can be represented with a simple structure. And at that point they discover an exact theory – a theory based on that aspect alone. But this discovery is a trivial, predictable achievement; for, if we keep reifying *any* phenomenon, we are bound to reach, eventually, aspects simple enough to allow an exact theory. The discovery, nevertheless, generates a great deal of excitement, so the mechanists initiate a research program. The more elaborate their research becomes, the more confident they are about its importance. Although it is obvious to everyone that the theory explains only that one isolated aspect, the mechanists promote it as if what it explained were the original, complex phenomenon.

The phenomenon of a database comprises many aspects, of which the most important are the application's *requirements* and the *file relationships*; that is, the *actual* entities and relationships, and their *representation in software*. And these two aspects consist, in their turn, of many aspects. Among the other aspects of this phenomenon are the field dependencies, the data redundancy, and the inconsistencies (the so-called update anomalies).

The aim of the normalization theory is to find a formal, exact method for designing the file relationships from a knowledge of the requirements (or, at least, for determining whether a given set of relationships matches the requirements). Now, it may be possible to represent with one structure the relationships on their own, or the dependencies, or the redundancy, or the inconsistencies, or perhaps even a combination of them. But the database phenomenon as a whole is complex, because these aspects interact with the requirements, which in turn interact with many other aspects of the application. Thus, no mechanistic theory can represent the system that consists of the file relationships *plus* the requirements. No formal method can exist, therefore, to determine whether or not a given set of relationships matches the requirements.

Because they could not discover a theory for the *actual* database phenomenon, the software mechanists tried to discover a theory by breaking down the phenomenon into simpler ones. They noticed that the inconsistencies occur when the file relationships are incorrect; and they also noticed that the inconsistencies are related to data redundancy and to field dependencies. It is the misplaced dependencies, they concluded, that cause redundancy and inconsistencies. And since this one aspect of the original phenomenon is simple enough to represent with an exact theory, they made *it* their subject of research. The *dependency* theory is believed to be the answer to the original problem: if we study, analyze, and classify the various types of field dependencies, the mechanists say, we will discover a formal method for avoiding misplaced ones; this will then prevent data redundancy and inconsistencies; and the lack of redundancy and inconsistencies will indicate that the file relationships match the requirements.

But this logic is fallacious. The dependencies, like the redundancy and the inconsistencies, are merely one aspect of the database phenomenon. They are not the *cause* of correct or incorrect file relationships, but just a different way of viewing them. So it is absurd to study the dependencies in the hope of determining from them the correct relationships. The *requirements* are the real determinant in this phenomenon. It is only from the requirements, therefore, that we can determine other aspects of the phenomenon: when there is no discrepancy between the requirements and the file relationships, there are no misplaced dependencies, no redundancy, and no inconsistencies; and when

there is a discrepancy, we note misplaced dependencies, redundancy, and inconsistencies.

It is indeed possible to explain the relationships, the redundancy, and the inconsistencies in terms of dependencies; but this is true because they are closely related aspects of the same phenomenon, not because the dependencies *cause* the other aspects. Thus, instead of a dependency theory we could develop an equally elaborate redundancy theory, to study, analyze, and classify the various types of data redundancy; or an inconsistency theory, for the various types of data inconsistency; or a relationship theory, for the various types of file relationships. And each theory could then be used to "explain" the other three aspects, just as the dependency theory is said to explain the redundancy, the inconsistencies, and the relationships.

From the requirements, then, we can determine the other aspects, but not the other way around. The mechanists base their theory on dependencies because they mistakenly interpret them as the cause of correct or incorrect file relationships. The dependencies on their own, though, are meaningless; for, we cannot decide from a dependency alone whether or not it is misplaced. Similarly, the redundancy or inconsistencies or relationships on their own, or all aspects together, are meaningless. The real cause – what can explain all four aspects – is the requirements. The dependency theory, thus, suffers from the fallacy of confusing cause and effect. It is fundamentally wrong.

Each aspect of the phenomenon of a database has its own representation: the requirements are represented by means of business practices, the file relationships by means of diagrams or programming languages, and the dependencies by means of a system of notation peculiar to the normalization theory. Similar systems could be invented to represent the redundancy and the inconsistencies, if we wanted. Each aspect provides a different view of the database, but neither is complete; only a system embodying *all* these aspects, plus those aspects we are not even discussing here, can represent the phenomenon of a database accurately. Thus, because they form a complex phenomenon, it is impossible to describe these aspects and their relationships exactly and completely. We *can* design correct databases, but this is largely an informal procedure.

Database design entails the conversion from one system of representation to another. What we want to attain, of course, is the *software* representation; that is, the file relationships. So, if it is the requirements that ultimately determine what are the correct relationships, the only conversion worth studying is the traditional one, from requirements to relationships. Because

they failed to discover a formal and exact procedure for *this* conversion, the relational mechanists shifted their attention to the study of field dependencies. Their theory does offer a formal and exact conversion, but only from dependencies to relationships. Its exactness is illusory, therefore, because to benefit from it we must ensure first that we have correct dependencies. And the only way to attain the correct dependencies is by performing the conversion from requirements to dependencies, which is as informal as the traditional one, from requirements to relationships (see figure 7-17).

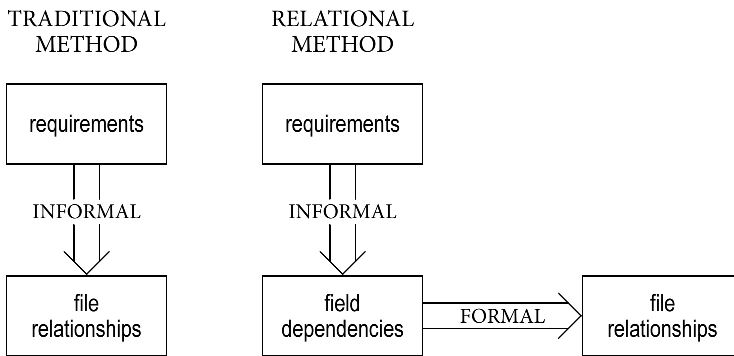

Figure 7-17

The dependency theory may appear impressive to the casual observer, but in reality an exact theory that explains relationships in terms of dependencies is a trivial accomplishment. It is not surprising that one aspect of a phenomenon can be shown to depend on another, if they are closely related. Thus, we could also discover similar theories to explain relationships in terms of redundancy, inconsistencies in terms of dependencies, dependencies in terms of redundancy, and so on. But that first step – from requirements to one of the other aspects – is always necessary, and is always informal. So we may as well use the traditional method, which entails *only* that one step – from requirements to relationships.

Of the five aspects of this phenomenon, the requirements and the file relationships are the most intuitive, and the field dependencies are the least intuitive. This is why, before the relational theory, we had no interest in dependencies; we only studied the requirements and the relationships, and sometimes the redundancy and the inconsistencies. We are asked now to replace what is the simplest method – the intuitive conversion from requirements to relationships – with a method that involves two steps, each one more complicated than our one-step method: the conversion from requirements

to dependencies – which the mechanists must perform but don't like to discuss, because it is informal – is less intuitive and already more difficult than requirements to relationships; and in addition, we have now the intricate dependency theory, for the conversion from dependencies to relationships. (It is, perhaps, precisely because the study of field dependencies is so complicated that the mechanists think it is an important discovery.)

The dependency theory is typical of the mechanistic pseudosciences. The relational mechanists settled for a dependency theory only because this is one narrow aspect of the database phenomenon for which they *could* find an exact explanation. They never proved that this theory could model the whole phenomenon. But then they forgot this limitation, and proceeded to treat the theory as if it *did* provide a formal method for database design.

In conclusion, the dependency theory – a major, thirty-year-old research program involving thousands of academics and generating a vast literature – is a worthless, senseless pursuit. No matter how exact it is, it cannot help us to determine what are the correct relationships. Thus, recalling an earlier example, the theory cannot tell us whether to assign the product description field to the product file or to the orders file. We must decide which alternative is correct during the conversion from requirements to dependencies, before we even get to use the theory. The theory may well offer us a formal, faultless conversion from dependencies to relationships, but we can only apply it after determining – informally – what *are* the correct dependencies.

The Third Delusion

1

The third delusion consists of those modifications to the relational model that are presented as *enhancements*, while being in reality *reversals* of the relational principles. These modifications were introduced when it was discovered that the model worked only with small and simple databases, and was totally impractical for serious applications. Thus, while the need for reversals constitutes an obvious refutation of the relational model, the theorists describe these reversals as *new relational features*.

The original theory defined a complete database model, and, although generally worthless, was a falsifiable concept. So, had it remained an academic treatise, it could have been regarded perhaps as a serious study. But because its supporters believed that it could have practical applications, the theory had to be modified again and again. The modifications, as we will see shortly, serve largely to restore the low-level capabilities of the traditional file operations –

capabilities which the relational model had attempted to replace with high-level features. Clearly, by the time we restore these capabilities we no longer have a relational model. The third delusion is in the belief that we can continue to enjoy the benefits promised by the original model even while reversing its principles.

The relational theory, thus, was turned into a pseudoscience when its supporters, instead of admitting that it had been refuted, decided to "improve" it: they suppressed the falsifications, one by one, by incorporating them into the model in the guise of new features. This practice rendered the theory unfalsifiable. (We examined earlier the pseudoscientific nature of the relational theory; see pp. 710–712, 713–714.) The relational model was indeed rescued, but this was accomplished by annulling the relational principles and reinstating the traditional ones. And because they were reinstated *within* the relational model, the traditional principles are now far more complicated than they were on their own. Moreover, relational systems still lack the flexibility and efficiency we enjoy with the traditional file operations.

From its simple origin, and from its mathematical ambitions, the relational theory was degraded in the end to a complicated and messy concept. What is perceived today as the relational model has little to do with the original ideas. And, although we still see the claim that the model is founded upon mathematical principles, relational systems are promoted now on the strength of features that were described originally as *informal* aspects of the model. Today's relational systems consist of large, cumbersome, inefficient, and expensive development environments, which include special programming languages and an endless list of features, definitions, principles, standards, rules, and procedures that we must assimilate. And what is the purpose of this complexity? To provide a substitute for what any programmer should be able to do by using just the six basic file operations.

2

Let us start with the concept of normalization. There are two kinds of normalization: the first normal form (1NF), and the second and higher normal forms (2NF, 3NF, etc.). 1NF was, from the start, part of the *formal* relational model; its purpose is to restrict the data stored in each field to a single item, so that the records and files match the tuples and relations of predicate calculus. The second and higher normal forms were added later, and belong to the *informal* aspects of the relational model; their purpose is to eliminate data redundancy and inconsistencies.

As we saw, whether the goal is to avoid multiple items in a field or to

eliminate redundancy and inconsistencies, we must separate the fields of the file in question into two sets, and move one set into a new file. Each normalizing step will generally increase by one the number of files in the database. Thus, although it is quite easy to normalize files, this process makes it more difficult to *access* the data. For, we must read more files and more records, in order to put back together the fields that were separated by normalization.

The idea of separating and recombining fields looks neat when presented as mathematical logic; that is, when we assume that data records can be accessed instantaneously, just like the tuples of predicate calculus. And the additional complexity caused by the separations and combinations can be justified by invoking the ultimate benefits of normalization. In real applications, however, even if we are willing to accept the additional complexity, normalization is often impractical, because of the excessive time needed to access the data.

Whether the fields were separated in order to attain the first or the higher normal forms, the only way to recombine them is with the JOIN relational operation (see pp. 702–703). JOIN creates one file from two: it combines the records of the two files, retaining only those records where certain fields relate the files in a particular way. But, while easy to use as a high-level operation, JOIN is very inefficient and hard to optimize. This may go unnoticed with small files, but in most databases its execution takes far too long to be practical. Also, applications usually need *many* normalization steps, and hence *many* JOINs later. Even a simple query may need two or three JOINs, and perhaps hundreds of times the number of disk accesses that the traditional file operations would need.[1]

So the idea of strict normalization had to be abandoned. But the theorists refer to this reversal with such euphemisms as database "optimization," or "tuning," or "tailoring." They discuss now the benefits of *denormalization* with the same seriousness, and with the same technical, impressive language, as they did the benefits of normalization before. This makes the reversal appear like progress, like an *enhancement* of the relational model. No one mentions the fact that the abandonment of strict normalization means simply a return to the informal design principles we had followed *before* the relational model: we compare in each situation the benefits and drawbacks of keeping data together in one file, with those of using two files, and we choose the more effective alternative. This is what we routinely do when creating databases with the traditional file operations.

[1] As I remarked earlier (see p. 732), we can attain the ideals sought by normalization more effectively with *traditional* databases. As a result, what is perceived as a fundamental relational principle – normalized files – is found more often in applications using the traditional file operations than in applications using relational databases.

The abandonment of the first normal form comes by way of a feature called – incredibly – *non-first-normal-form*. Abbreviated with scientific-looking terms like non-1NF, NFNF, and NF², this feature is so advanced that only a few database systems support it. Those that do are known as *extended relational* systems.

The name chosen by the experts for the new feature betrays their attitude: instead of simply stating that the first normal form – one of the fundamental principles of the relational model – has been abandoned, they present the abandonment as a new principle; and they call this principle, literally, the opposite of the original one. 1NF is still important, but now we need to impose this restriction only when convenient. Thus, the experts suppressed the falsification of an important principle by introducing a new one. In effect, the two principles, 1NF and non-1NF, cancel each other; that is, taken together they cannot possibly be serious principles. So the first normal form is now just an informal recommendation. But the experts describe this falsification of the relational model as a new, advanced relational feature.

To appreciate the significance of non-1NF, recall the 1NF restriction and its implications. For a file to be in first normal form, its fields must contain single, atomic values. Each field, in other words, must contain only one value at a time – not a list of items, or an array, or any other structure. This restriction is usually expressed by saying that the columns of a relational table must not contain *repeating groups*. The restriction to a single item per field is critical if we want to base the relational model on mathematical logic (because the elements of a tuple in predicate calculus are single items).

In most applications, however, we encounter sets of values that are so closely related that the most effective way to store and use them is as a list, or array. For example, in a file of purchased parts, we may want to store for each part a list of up to three vendor numbers, or three vendors and their selling price, or three vendors with their last price and purchase date. With the traditional file operations and a language like COBOL, we define these values, respectively, as an array of 3×1, or 3×2, or 3×3 elements. In the part record, the whole array will be treated as one field. It will be read into memory or written to disk along with the record, and, when in memory, its elements can be conveniently accessed with the same operations that programming languages provide for manipulating *memory* arrays. Thus, we can easily display or update one element or a subset of the elements, compare the three prices, change the relative position of the vendors, and so on.

In a relational database, the only way to store these values is as a separate file. The fields in the new file will be, for instance, the vendor number, price,

and date; and the key will consist of the part number and a sequence number, 1 to 3. For each record in the part file, there will be up to three records in the new file. Operations like comparing prices or exchanging the relative position of vendors, which can be performed with a couple of statements in a traditional database, will now be small programming projects (since we must combine the two files with JOINs, access the three sets of elements as separate rows but save them somehow so that we can use them together, and so on). What is worse, these operations will now take longer to execute, because of the additional disk accesses.

For a few fields, it is possible to bypass the 1NF principle; and the simplest way to do it is by simulating arrays with ordinary fields. In the previous example, we would add to the part record three, six, or nine fields, each one with its own name, and access them through whatever means a relational system provides for accessing individual fields. This method obviates the need for a second file and separate records, and solves therefore the performance problem; but it makes programming even more complicated. Simulating arrays with ordinary fields, thus, is an awkward trick that programmers must employ if they want to bypass the 1NF principle while pretending to like the relational model.[2]

The fact that we have to resort to tricks in order to avoid the inefficiency of a relational principle constitutes a falsification of the relational theory. And the final abandonment of 1NF, after thirty years of struggling to fit real-world problems into relational systems, is in effect an acknowledgment of this falsification. Presenting non-1NF as a new relational feature is how the relational charlatans suppress the falsification.

With non-1NF, a field in one file acts as a pointer to records in a second file. For example, if the first file contains customer records, one field may be used for that customer's invoices. But the field itself contains no information. It only points to another file: an invoice file, where the records are identified through the combination of customer and invoice numbers, and the set of invoice records associated with a particular customer record are those with the same customer number.

[2] The 1NF principle is impractical, not because it requires a second file, but because it requires a second file in *any* situation. In contrast, with the traditional operations we are free to choose, in each situation, the most effective alternative. Thus, we may decide to use a second file even to replace a *small* array, if the application must access those elements in such a way that the use of indexed data records is simpler. Conversely, if access time is critical, we may decide to use an array even if this results in a very large record size.

With this method, a record in the first file can point to any number of records in the second file. In some database systems, more than one field can act as a pointer to another file; for example, in addition to the invoice field, we can have an order field and a history field in the customer record, pointing to records in an orders file and a sales history file, respectively.

Non-1NF allows us to relate files hierarchically, by logically *nesting* one file within another. Thus, databases that utilize this feature are also known as *nested relational* databases. Nesting is not limited to one level: fields in the second file can act as pointers to further files, which become then logically nested within the second one, and so on. Non-1NF allows us, therefore, to create hierarchical file structures. And, since the original relational model does not support these relationships, new relational operations were introduced for defining and accessing the records of nested files.

The concept of file nesting, however, is not new; it is practically identical, in fact, to the way we relate files when using the traditional file operations (see pp. 683–686). The only real difference is the higher level of abstraction of the non-1NF operations. What this means in practice is that, instead of creating explicit file scanning loops like those in figures 7-15 and 7-16, we invoke some built-in functions that generate the loops for us.

But, as we know, a higher level of abstraction also has drawbacks: we are restricted to fewer alternatives. So in the end, even with non-1NF, the relational systems are not as flexible or efficient as the basic file operations. For example, with the basic operations we can nest – in different places in the application, through different fields – the same files in different ways; we can create, therefore, several relationships between the same files. Also, with the basic operations we still have the option of storing arrays directly in a record – a method that is both simpler and faster than file nesting.

The main objection to non-1NF, however, is that it is presented as a new feature while being an abandonment of the relational file-relating method and a reinstatement of the traditional one. Even the term "nesting" is old: with the traditional operations, the files are nested by nesting their scanning loops; with relational systems, the files are nested through *implicit* scanning loops. The logical relationship between files is the same.

The term "non-1NF," then, is not only silly but also misleading. For, the intent of the new feature is not to avoid the problems caused by the 1NF principle, but to replace the impractical JOIN operation. Let us examine this misrepresentation more closely.

To promote non-1NF, the experts point to the inefficiency of certain file

combinations in the original relational model. But the combinations they describe were never thought to be a consequence of the 1NF restriction. Specifically, non-1NF is recommended for files of any size, not just as a substitute for the small arrays that we may want to store directly in a record. Thus, referring to the earlier examples, we can use file nesting not just to replace an array of three vendors associated with one part, but also for a whole invoice file, where hundreds of invoices may be associated with one customer. For this type of data, though, we have *always* resorted to a second file, even with the traditional file operations, because this is the only practical way to store it. The difference between non-1NF and 1NF, then, is simply in the way we combine files: through nesting instead of JOINs. So what the experts are recommending in reality is not the replacement of 1NF with non-1NF, but the replacement of JOIN operations with the traditional concept of file nesting.

Non-1NF, in other words, is not promoted as a solution to the inefficiency of 1NF, but as a solution to the inefficiency of JOIN; that is, for any situation where we have to combine files. Thus, if we adopt non-1NF we can dispose of the JOIN operation altogether. If we want, we can replace with nested files every situation that would normally require JOINs: not just files that would be created when enforcing the first normal form, but also files that would be created when enforcing the second and higher normal forms, and even files that would be kept separate in any case. Non-1NF eliminates, therefore, the inefficiency caused by combining *any* files in a relational database. So, if it is a general substitute for the relational way of combining files, what we have now is *a different database model.*

Far from being just a new feature, then, non-1NF cancels the whole relational model. To understand this, let us take a moment and recall the importance of the first normal form. And there is no better way to start than by citing the experts themselves.

Date says that 1NF is so fundamental that the term "normalized," when unqualified, means "first normal form": "It follows that *every* normalized relation is in first normal form ...; it is this fact that accounts for the term 'first.' In other words, 'normalized' and '1NF' mean *exactly the same thing.*"[3] In Codd's original papers, too, the term "normalized" means what we call now first normal form;[4] the higher normal forms are not even mentioned. Recall also that the first normal form is the only one that is part of the *formal* relational model.

[3] C. J. Date, *An Introduction to Database Systems*, 6th ed. (Reading, MA: Addison-Wesley, 1995), pp. 289–290.

[4] See, for example, E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387.

Here are some additional statements: "At each intersection of a row and column there is exactly one value. This is the principle of *first normal form*, fundamental in the relational model."[5] "This property implies that columns do not contain repeating groups. Often, such tables are referred to as 'normalized' or as being in 'first normal form (1NF).' It is important that you understand the significance and effects of this property because it is a cornerstone of the relational data structure."[6] "Occasionally there might be good reasons for flouting the principles of normalization.... The only hard requirement is that relations be in at least first normal form."[7] "All data in a relational database is represented in *one and only one way*, namely by explicit value (this feature is sometimes referred to as 'the basic principle of the relational model' ...). In particular, logical connections within and across relations are represented by such explicit values."[8]

It is not difficult to see why the first normal form is so important to the relational model – why it is "fundamental," a "cornerstone," a "hard requirement," and a "basic principle." It is not so much the restriction to single values that is important, as the *purpose* of this restriction. By preventing us from creating any data structures within a record, 1NF forces us to keep all data in the form of tables. And if the data is restricted to tables, the methods used to access and combine the data can be restricted to operations on tables; that is, to high-level operations based on mathematical logic.

Accordingly, by annulling 1NF we also annul these restrictions: we can store, access, and combine data in other ways too. In effect, we have regained some of the freedom we enjoyed when using files through the traditional file operations: we can now relate them through the versatile hierarchical concept, as data within data. And we can use this method, not just with small arrays or structures, but with files of any size, and on any number of nesting levels. In the end, annulling 1NF permits us to create database structures that are more flexible and more efficient than those possible with the relational model.

In conclusion, the restriction imposed by the first normal form is far more significant than what it appears to be – merely preventing multiple values in a field. Its annulment, therefore, means far more than just permitting multiple values; it means the annulment of the relational model. It also demonstrates the pseudoscientific nature of this theory, as well as the dishonesty of its supporters: the impracticality of 1NF, along with the impracticality of JOIN, is

[5] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1162.

[6] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design* (Reading, MA: Addison-Wesley, 1989), pp. 32–33.

[7] Date, *Database Systems*, p. 291.

[8] Ibid., p. 99.

a falsification of the model; but instead of being abandoned, the theory is expanded – by turning this falsification into a new relational feature, non-1NF.

The mathematical foundation of the original model was predicate calculus, with its relations and tuples. Thus, if our databases no longer consist of this type of relations and tuples, it is absurd to continue to call them relational. Terms like "extended relational" and "nested relational" are simply incorrect if the new model is not "relational." The term "relational" derives from the mathematical concept of a relation; namely, a set of tuples, where each tuple is composed of single elements. And in predicate calculus the only operations are those performed on such sets through mathematical logic. It is these relations, tuples, and elements that become the files, records, and fields of a relational database. So, if we want a different database organization, or different operations, we need a different model.

As we saw under the first delusion, the mathematical claims of the relational model were tenuous in any case, since only a small part (those aspects that constitute the formal model) had indeed a mathematical grounding. And with the annulments we are discussing in this subsection – non-1NF, in particular – even that small part has disappeared. What we have now is neither an enhanced nor an extended relational model. What we have is not a relational model at all.

Non-1NF systems, then, are indeed as useful as their promoters claim; but they are useful because they are no longer relational. This is why some experts, embarrassed perhaps by this fraud, suggest terms like "post-relational," and even "object-relational," for the database systems that include non-1NF or a similar "enhancement."

Still, if not predicate calculus, perhaps another mathematical system can serve as a foundation for the new database model. And indeed, some theorists have attempted to extend the formal relational model to include non-1NF. But this is silly. For, if a mathematical system could guarantee the correctness of nested databases, then the same system would also guarantee the correctness of the nesting performed through the traditional file operations – which is identical, logically.

This, of course, is true for the original model too: nothing stops us from using the traditional data files and operations while limiting ourselves to the subset of features that parallel the relational model; and our databases would then be founded on predicate calculus, just like the relational ones.

The conclusion must be that, no matter how rigorous a formal database model is, it offers no mathematical benefits that we do not also enjoy with the

informal traditional operations.[9] The answer to this apparent contradiction is that the formal part plays such a small role in a database system that it is practically irrelevant. So, for the application as a whole, the mathematical benefits are about the same with a formal database model as they are without one. (This is the essence of the first delusion.)

By way of summary, I want to show how the software elites are presenting the non-1NF feature. A good example is the white paper published by IBM to promote one of their new database systems.[10] This paper, we are told, "discusses technical advances represented by nested relational database technology."[11] And just in case we were not sufficiently impressed by this statement, a few sentences later we are reminded that nested relational databases represent an "advanced technology."

Now, the advanced technology that is file nesting has been available since about 1970 to anyone capable of writing a few lines of COBOL. So it is clear that IBM addresses individuals who, while being perhaps programmers or managers, have very little programming knowledge. These incompetents try to develop applications, not through programming, but by buying programming substitutes. They can be impressed by a feature like non-1NF because they are always dependent on the elites for solutions to their software problems. They have problems now because they trusted the elites in the past and adopted a relational system. But they believe that the solution must also come from the elites, in the form of a new system.

The paper continues by describing the problems caused by the restriction to 1NF: "Database conformance with 1NF often increases the amount of storage used, makes maintenance more difficult, and most importantly greatly increases the processing required to produce results, while still making the schema more complex.... For some potential users of relational databases, the joins [i.e., JOIN operations] that would be required to resolve relationship relations [i.e., cross-references] in 1NF databases would affect performance

[9] Because they are restricted to higher levels, the relational operations are logically a subset of the basic, traditional file operations. Thus, we can always simulate a relational database system using a traditional file system, but not vice versa. Many relational systems, in fact, are designed simply as a high-level environment based on an underlying file management system: the relational operations are implemented as subroutines that employ the basic file operations in conjunction with appropriate loops and conditions.

[10] IBM Corporation, *Nested Relational Databases*, white paper (2001).

[11] Ibid., p. 3. Note, again, the slogan "technology," used to make something appear more important than it really is.

enough to preclude the use of relational databases.... Apart from performance considerations, 1NF relational databases also have practical limitations for many applications."[12]

This is an excellent description of the restrictions imposed by the first normal form, and by the relational model in general. Reading this, one is liable to forget that the same institutions that are so harshly attacking this model now had been promoting it for the previous thirty years as an expression of database science, and as an important aspect of software engineering. These problems had been noticed from the start, of course. So how were the millions of programmers and users who had adopted relational systems coping all these years? By constantly seeking ways to bypass the restrictions; by spending most of their time dealing with these spurious problems instead of the actual business problems; and, ultimately, by being content with inadequate and inefficient applications.

The nested relational model, the paper tells us, eliminates the 1NF problems. Non-1NF is such an important feature, in fact, that all relational systems will soon support it: "Because of the limitations of 1NF relational databases, especially for storing complex data structures, all commercial relational databases have begun adopting extended relational technology; however, IBM has a technological lead of several years over its closest competitor."[13]

The shallowness of the non-1NF issue is seen in the pretentious description of file nesting. For example, one of the reasons why IBM's "extended relational technology" is more advanced than the competing ones is that "the IBM nested relational implementation, unlike others, is not limited to a single nested table."[14] With the basic file operations, as we know, it is just as easy to nest several file scanning loops as it is to nest one, simply because programming languages allow us to combine file scanning loops in any way we like. But with nested relational databases, this trivial capability is presented as a major technological advance, currently available only from IBM. Again, only ignorant practitioners can be impressed by such claims.

Finally, the paper reminds us (three times[15]) that the relational model has a rigorous mathematical foundation, which guarantees correct results when using the relational operations. And, the paper assures us, research has shown that this guarantee is not compromised by the annulment of the 1NF principle: "Analysis has proven that the resulting model is equally robust."[16] Such analysis and proof are senseless, though, because the relational model is *not* robust even *with* 1NF. As we saw under the first delusion, its mathematical foundation is irrelevant in practice. It is precisely because the mathematical foundation is

[12] Ibid., p. 7. [13] Ibid., p. 14. [14] Ibid. [15] Ibid., pp. 3, 7, 14.
[16] Ibid., p. 7. The paper cites several sources, where presumably the proof can be found.

irrelevant that annulling an important principle like 1NF indeed makes no difference.

And we are expected to feel even better after reading that nested relational databases have been "accepted by the academic community as adhering to a valid relational model."[17] But we saw that it is wrong even to *call* the new model relational. In any case, this statement is hardly reassuring if we remember that the same academic community also advocated other theories that failed (structured programming and object-oriented programming, in particular), and that, just like the relational model, those theories were rescued by being turned into pseudosciences.

Thus, by promoting pseudoscientific software theories, the universities help software companies to sell worthless development systems, and help incompetent programmers and managers to control corporate computing.

3

I began the discussion of the third delusion with the non-1NF issue because this is the most flagrant of the relational reversals – a reversal that marks, in effect, the end of the relational theory. But 1NF is merely the latest principle to be annulled. At this point, most relational principles had already been forsaken, because, like 1NF, they had been found to be impractical. In the remainder of this subsection, I propose to study the other reversals.

The abandonment of the second and higher normal forms (2NF, 3NF, etc.) came by way of a new relational principle, called *denormalization*. At first, database designers and programmers simply ignored the stipulation to fully normalize their files, when this was too complicated or too inefficient. But the theorists were condemning this practice. Before long, though, even they realized that strict normalization is impractical, and that the decision whether or not to normalize a particular set of files depends ultimately on the situation: on the type of data stored in these files, on the file relationships, and on the way we plan to use the files in the application.[18]

But instead of admitting that the idea of strict normalization had failed, the theorists reacted, as pseudoscientists do, by turning this falsification of the

[17] Ibid., p. 14.

[18] The term "normalization" refers usually to *all* normal forms; but here, in the discussion of the second and higher normal forms, I use "normalization" to refer only to them.

relational model into a new relational principle – denormalization. The new principle says that we must first normalize all files, as before; then, we must denormalize (that is, restore to their previous state) those files that should not have been normalized in the first place.

Both the principle and the term, "denormalization," are absurd. All we needed was a statement acknowledging that normalization was annulled as a relational principle and is now just an informal concept. The very term "normalization" should have been abandoned, in fact. After all, normalizing some files and not others is what we had been doing all along, with the traditional file operations, and we didn't need a special term to describe this activity. With the relational model we have now *two* principles for this activity, and *two* terms. We are told that normalization is as important as before, and that denormalization is the process of *improving* the results of normalization.

Clearly, the theorists invented the second principle in order to suppress the fact that the first one had failed. The two principles, normalization and denormalization, in effect cancel each other. But the theorists managed to make this return to what we had before the relational model look like an *enhancement* of the model.

Here is a typical explanation of the new principle: "Denormalization is the 'undoing' of the normalization process. It does not, however, imply omission of the normalization process. Rather, *denormalization* is the process whereby, after defining a stable, fully normalized data structure, you selectively introduce duplicate data to facilitate specific performance requirements."[19] What this sophistic verbiage is trying to say is that, while normalization is generally desirable, strict normalization is impractical; in other words, what we always knew. Now, however, we can no longer simply allow some data duplication from the start (when we know from experience that the application would otherwise be too slow). Instead, we must first normalize the whole database, and then "selectively introduce duplicate data to facilitate specific performance requirements." Actually, in both cases we address the same problem and end up with the same database. The pompous language serves to mask the fact that the principle of strict normalization – a fundamental relational requirement – has been falsified.

Here is how two other experts present this reversal: "The general idea of normalization is that the database designer should aim for relations in the 'ultimate' normal form (5NF). However, this recommendation should not be

[19] Fleming and von Halle, *Relational Database Design*, p. 440.

construed as law. Occasionally there might be good reasons for flouting the principles of normalization."[20] "There are, however, exceptions to [strict normalization].... We recommend that data models *always* be designed in third normal form, but that the physical data-base designer be permitted to deviate from it if he has good reasons and if the data administrator agrees that no serious harm will be done."[21]

A critical aspect of the idea of denormalization, then, and what the experts keep stressing, is that denormalization does *not* constitute the annulment of normalization. Normalization remains as important as before, and what we must do is both normalize and denormalize the database.

Here is another example of this doubletalk: "Data denormalization is constrained so that it does not alter the basic structure of the conceptual schema. It only makes adjustments to the basic structure for operational efficiency."[22] Denormalization, thus, consists in *adjusting* the database design, but without *altering* it. This is silly, of course, since adjusting something will also alter it. A database either is or is not normalized; so, if we denormalize a normalized database we necessarily end up with an unnormalized one, regardless of whether we call this process "adjustment" or "alteration." Not so, says Brackett: "A common misconception about data denormalization is that it results in a return to the unnormalized business schema that began the data normalization process.... However, this is not the situation. Data denormalization produces denormalized data, not unnormalized data."[23] In reality, there is no difference between the two: both "denormalized" and "unnormalized" mean simply data that is not fully normalized, violating therefore this relational principle.

The theorists, thus, are defending their deviation from strict normalization by claiming that denormalizing the database after fully normalizing it is different from simply leaving some of the files unnormalized in the first place. One method, they tell us, constitutes an exact design process, while the other is merely an informal decision. But this would be true if denormalization were indeed an exact process. In practice, though, the decision to denormalize a file can be no more exact than the decision to leave a file unnormalized to begin with. Recall the previous quotations: "[the designer is] permitted to deviate from [strict normalization] if he has good reasons and if the data administrator agrees that no serious harm will be done," and "occasionally there might be good reasons for flouting the principles of normalization."

[20] Date, *Database Systems*, p. 291.

[21] James Martin, *Managing the Data-Base Environment* (Englewood Cliffs, NJ: Prentice Hall, 1983), p. 216.

[22] Michael H. Brackett, *Practical Data Design* (Englewood Cliffs, NJ: Prentice Hall, 1990), pp. 155–156.

[23] Ibid., p. 156.

Informal comments like these can hardly be described as an exact method of denormalization.

Brackett starts by promising us an exact method: "Conceptual schema are converted to internal schema through a denormalization process following a precise set of rules depending on the physical operating environment."[24] But the "precise set of rules" never materializes. All we find on the subsequent pages is a list of cases where denormalization is beneficial, and a reminder to deal carefully with the consequent problem (redundancy and inconsistencies). For instance, this is how Brackett describes one of the cases of denormalization: "This situation creates redundant data and those redundant data must be consistently updated or the quality of the database will deteriorate rapidly.... Other data entities may be denormalized for operational efficiency based on these criteria.... Each situation must be carefully evaluated to assure that the logical model is not compromised and that any redundant data are routinely and consistently updated."[25]

So what Brackett is describing as denormalization is not "a precise set of rules" but an informal process – a process no different from what we do with *traditional* databases: we study the application's requirements, allow redundancy and inconsistencies when it is impractical to eliminate them, and deal with the consequent problems by adding special checks and operations to the application's logic.

Thus, to cover up the failure of strict normalization, the theorists were compelled to invent the absurd principle that we must first normalize the database and then denormalize it. And they defended the principle with the absurd claim that this method is exact while the traditional, simpler method – creating the correct database directly from the requirements – is not. In reality, both methods entail the same decisions and result in the same design.

We saw under the second delusion that the process of normalization is presented by the theorists as a formal design method, while being in fact as informal as the traditional method. It is informal because it must be based, ultimately, on the same decisions as those we make when designing the database directly from the requirements. Now we see that the process of denormalization too is informal, despite the claims that it is exact. Only we, by studying and interpreting the requirements, can determine whether strict normalization is practical in a given situation, and, if not, what operations must be added to maintain data integrity.

[24] Ibid., p. 155. [25] Ibid., pp. 157–158.

In conclusion, both normalization and denormalization are perceived as formal design methods, when in fact both are informal. So, to appreciate the new delusion, denormalization, we must ignore the previous one: we must believe, with the theorists, that normalization is indeed an exact process. Judging it from *their* perspective, therefore, denormalization is a delusion; for they did not stop promoting normalization when they introduced the concept of denormalization. They continue their research in what they believe to be formal and exact concepts – the dependency theory, the classification of normal forms – even while praising the virtues of denormalization, which is informal. They are oblivious to the absurdity of promoting these two methods at the same time: no matter how exact is the process of normalization, when we modify its result by adding the inexact process of denormalization the final result is bound to be inexact. So what is the point in seeking a formal and exact normalization theory while also permitting denormalization?

It is in order to resolve this self-contradiction that the theorists introduced the principle that we must denormalize the database only after fully normalizing it. This principle appears to justify the need for both processes, when in reality it shows that we need neither.

Earlier, to justify the need for normalization, the theorists distorted the problem of database design. Instead of determining the correct design simply by studying the application's requirements, we were asked to do two things: create a deliberately incorrect database, and then normalize it to make it correct. And now, to justify both normalization and denormalization, we are asked to do *three* things: create an incorrect database, normalize it to make it correct, and, finally, denormalize it to make it practical.

The traditional design method allows us to create, not only correct databases, but also efficient ones. For, the same skills that help us to create a correct, fully normalized database also help us to decide when this would be inefficient. Thus, we can create a correct *and* efficient database at the same time, directly from the requirements. We don't need a denormalization theory any more than we need a normalization one.

Finally, and quite apart from the delusions already discussed, the need for denormalization means that we are again preoccupied with the *efficiency* of the database operations – contrary to the claim that the relational model shields us from the physical implementation of the database. We must study each situation and seek the most effective solution, instead of implementing the requirements through formal methods and high-level operations, as the relational theory had promised us. We must accept, rather than avoid, the "update anomalies"; and we must add special checks and operations to deal with them. In other words, we have returned to what we had been doing all along, with the traditional databases. The theorists describe denormalization

as database "optimization"; but if the optimization consists in a deviation from fundamental relational principles, this description is merely a way of denying that the relational model has failed.

4

One of the relational model's promises was that we could restrict ourselves, in all database work, to the high-level relational operations. And this promise too had to be annulled. In the end, the relational systems became practical only after reinstating the low-level capabilities of the traditional file operations; specifically, the means to manipulate fields and records through traditional programming methods, and the means to link them to other low-level entities in the application. Let us examine this reversal.

Recall the original relational model. The database, we are told, must be perceived as "tables and nothing but tables." The relational operations can be assumed to occur instantaneously, and can therefore be treated like the operations of mathematical logic: all we have to do is reduce the database requirements to logical expressions where the operands are tables, and the relational operations (along with standard logical operations) combine in various ways tables and portions of tables. No matter how large or how small, a data file can be treated simply as a table with a number of rows and columns. In particular, if we need just one record of a given file, we must create a new table with just one row; and if we need one field, we must create a table with one row and one column. Also, there is no way to modify the tables. Updating the database was thought to be a relatively simple and infrequent aspect of database work, so the operations that add, delete, or modify records were expected to be informal, like the traditional ones. We must be careful when modifying the database, of course; but we don't need the formality and precision of mathematics, as we do for queries.

The original model, thus, permits only database queries. Consequently, the only database language we need is one that provides the means to formulate queries through the relational operations. In their naivety, the theorists believed that a model shown to satisfy some simple queries on small files could serve as the foundation of practical database systems: for applications with files of any size and queries of any complexity. Moreover, they later believed that the same model could be extended to cover all aspects of database work, including the updating operations and the design process. The fact that simple queries look neat when expressed as mathematical logic was enough to convince the theorists that all database programming could be restricted to high-level operations and to the notion of tables.

Today, after all the reversals, the relational systems are no longer restricted to "tables and nothing but tables." Rather, they provide, in a very complicated manner, the means to link individual fields and records to the other entities in the application. In addition, the database language, SQL, has grown from a set of simple query operations into an elaborate (although quite primitive) programming language. The relational systems, thus, have restored the means to manipulate, through programming, the low-level database entities. So they have restored exactly what the traditional file operations and programming languages had been doing all along, in a much simpler way – and what the original relational model had claimed to be unnecessary.

We need to access low-level database entities for two reasons: because this is the only way to implement the *details* of a database operation, and because this is the only way to *link* the database structures to the other structures that make up the application. It is obvious, therefore, why the traditional file operations are indispensable: in addition to allowing us to access the low-level database entities, they can be used from a programming language; and through this language we can create the critical, low-level links between database entities and the other types of entities in the application.

These two qualities are both necessary and sufficient for implementing any database requirement; and it is precisely these two qualities that are lost in the relational model. Thus, since it is impossible to implement serious applications without accessing and linking the low-level elements of the application, it is not surprising that the modifications needed to make the relational systems practical consisted in restoring both the low-level operations and the means to use these operations through a programming language.

So, like all systems that offer us high-level starting elements, the relational systems became in the end a fraud. When promising us higher levels, the software charlatans tempt us to commit the two mechanistic fallacies, reification and abstraction (see "The Delusion of High Levels" in chapter 6). In the case of relational systems, the claim was that we could separate the database structures from the other structures that make up the application; this would allow us to start from higher levels of abstraction within these structures, greatly simplifying database work.

With the traditional development method, all we need is a programming language and a few libraries of subroutines (for mathematical functions, display operations, database management, and the like). The software charlatans have replaced this simple concept with the concept of *development environments*: large and complicated systems that lure ignorant practitioners with the promise

of high-level, built-in operations. These operations, we are told, function as prefabricated software subassemblies: they already contain within them many of the low-level operations that we would otherwise have to program ourselves. But, in fact, only trivial requirements can be implemented by combining high-level operations. So the systems must be continually enhanced, with more and more features. And what are these features? They are means to deal with low-level entities, precisely what the systems had originally attempted to eliminate.

Thus, instead of admitting that the restriction to high-level operations failed as a substitute for traditional programming, the software charlatans rescue these systems by turning their falsifications, one by one, into new features. The systems keep growing and appear to become more and more "powerful," but this power derives from reinstating the low-level, traditional concepts. By the time enough of these concepts are reinstated to make the systems practical, there is nothing left of the original promise. For now we must deal with the low levels again. What is worse, because the low levels were introduced within the high-level environment, they are much more complicated than they are when available directly, through a traditional language. So, in the end, programming is even more difficult than before.

Returning to the relational systems, the need for low levels emerged when the notion of *data integrity* was introduced. Data integrity became an issue in the relational model only when the model was expanded to include *updating* operations. As long as it permitted only queries, there was no need for integrity checks, because, within the scope of the model, the data never changed. Since the operations that add, delete, or modify data records were expected to be similar to the traditional ones, and to be performed outside the model, the validity checks accompanying these operations were also expected to be performed outside the model. Once the relational model was adopted for serious database work, however, the updating operations, along with the problem of data integrity, could no longer be ignored.

The normalization principles too, we saw earlier, were needed only when the relational model was expanded to include updating operations. Tables did not have to be normalized in the original model, because no data inconsistencies can arise when we restrict ourselves to queries and to the high-level relational operations. We also saw how both the attempt to formalize the process of normalization, and the idea of strict normalization, failed. In the end, the only way to design a correct database is informally, by studying the application's requirements. All that the theory of normalization accomplished was to add to the traditional design problems the complicated concepts of normal forms

and field dependencies. The critical part – the need to determine whether two given fields must be in the same file or in separate files – remained unchanged. With the traditional design method or with the relational one, we can decide in which file to place a new field only by discovering the low-level links between the database entities and the other entities in the application.

The formality and the neat classification of normal forms can be seen, therefore, as a failed attempt to raise the level of abstraction in database design: instead of having to study and understand the application's requirements, it was believed that we could attain the same goal by knowing only how to convert files from one normal form into another – an easier, largely mechanical, task.

But regardless of its failure, the normalization theory was silly because it addressed only a small number of data inconsistencies; specifically, only those that can be prevented by placing fields in separate files (see pp. 755–757). Since most data inconsistencies cannot be eliminated simply by separating fields, we must deal with them through the application's logic: to ensure that an updating operation does not cause inconsistencies, we add various checks, restrictions, or further updating operations. An example of a situation where the updating problems cannot be solved through normalization, we saw, is the requirement for the balance field in the customer record to match at all times the amounts present in that customer's transaction records. Although technically redundant, the balance field is useful because it obviates the need to recalculate the balance by reading the transaction records every time. Thus, instead of avoiding the redundancy, we ensure that the field remains correct by adding to the application's logic some operations to update it whenever a transaction is added, deleted, or modified.

The requirement to match the balance field and the transaction records is, in effect, a database integrity rule. So the notion of integrity was the answer to the updating problems that could not be solved through normalization; that is, to practically *all* the updating problems that can arise in an application. A whole new class of relational features had to be invented – features totally unrelated to the original model – in order to move the data validity operations from the application, where they are normally performed, into the database system. The sole purpose of these features is to permit us to do through a new language, in the database system, what we had been doing all along through a traditional language in the application. Thus, the operations that update the customer balance field, previously mentioned, would no longer be part of the main program; they would be written instead in a special language, and made part of the database environment.

The problem of data validity is well known. Whenever a database field is modified, the application must verify that the new value is correct within the current context. Similarly, when a record is added, the value of each field in the record must be correct. But there is more to the validity problem than verifying the value of individual fields. For example, the application must verify that a record may be modified at all, or added or deleted, in a given situation. Also, adding, deleting, or modifying a given record often affects other records and other files, so the application must perform additional operations if the database as a whole is to remain correct. Generally, all the specifications and restrictions known as business rules – which are reflected in the various processes implemented in the application – can be described, if we want, as integrity rules.

Data validity, thus, is closely related to the application's requirements: what is correct in one situation may be incorrect in another. Just like the "anomalies" they tried to eliminate through normalization, the problems that the relational theorists are discussing under "integrity" are problems we always faced. And we never thought of them as *database* problems, but as a natural part of application development. The so-called integrity problems are merely one aspect of the challenge of programming: if we fail to take into account certain requirements, some data may become incorrect – inconsistent, redundant, invalid – when the application is used. The problems that cause incorrect data are similar to those that cause incorrect operations. In both cases the application will malfunction, and in both cases the reason is that it does not reflect the requirements accurately.

We saw earlier that files cannot be said to be normalized in an absolute sense, but only relative to the application's requirements. For example, if the product description does not change from one order to the next, the product and orders files are normalized when the description field is in the product file; but when the description may change, they are normalized when the description field is in the orders file. Similarly, the validity criteria cannot be defined in an absolute sense, but only relative to the application's requirements. Some examples: A certain date may be deemed too old in one part of the application, but not in another. Deleting a transaction record may be permitted if certain conditions hold, but deemed invalid otherwise; elsewhere in the application, though, we may have to prevent the deletion under all conditions. Creating a new transaction record may generally entail adding a record to the history file too, and failing to do so would result in an incorrect history file; sometimes, though, when this is not a requirement, it is *adding* the history record that would result in an incorrect file.

Clearly, validity issues like these are part of the application's logic. It is absurd to treat them as a special class of operations just because they are

concerned with the correctness of database entities. We also modify *memory* variables in the application, and they too must remain correct; yet no one has suggested that – in order to safeguard the correctness of memory-based entities – we extract these operations from the application, restrict them to high levels, and design special systems and languages to perform them. If we were to do this for every type of entities and operations, we would no longer need applications and general-purpose languages. Performing and combining various types of operations, including those concerned with data validity, is precisely what applications are for, and what programming languages are designed to do. In any case, the operations that validate the database, as much as those that modify it, must necessarily access *low-level* entities. So the idea of separating them from the application, incorporating them into a database system, and restricting them to high levels is senseless, and bound to fail.

In conclusion, the integrity features added to the relational database systems were totally unnecessary. Their real purpose was to rescue the relational model from refutation. Here is how: The promise had been a model that satisfies all our database needs through high-level operations. The existing data validity functions, however, required *low-level* operations. Moreover, they required *programming*, so they could not be implemented at all in a relational system. Asking us to depend on traditional programming for a critical aspect of database management was, thus, a falsification of the relational model. To save the model, the theorists were compelled to move these functions *from* the application, where they belong naturally and logically, *into* the database system. The integrity features are a fraud because this move is said to complement the high-level database operations, when in reality the new functions require low levels, and programming.

The integrity features, then, were the expedient through which low-level capabilities could be added to a relational system. Instead of recognizing the need to deal with low-level entities as a falsification, the theorists solved the problem by annulling the restriction to high-level operations. Using the issue of data integrity as pretext, they turned a blatant falsification into an important new feature. This feature is so important, in fact, that no serious database requirements can be implemented without it. And all this time, they kept praising the power of the relational model: annulling the restriction to high levels, they say, is an *enhancement* of the model.

The first integrity functions were limited to simple validity checks. Here are some examples: The *attribute integrity* functions check that the value placed in a field is correct with respect to the definition of the field (valid numbers in a

numeric field, valid dates in a date field, etc.). The *domain integrity* functions check that the value placed in a field is correct when the field is restricted to a range of values (a number must not be larger than 1,000, for instance, a date must not be older than 30 days ago, etc.). The *referential integrity* functions check that the relationship between two files remains correct when the files are modified; typically, they are used to prevent the deletion of a record in one file while there exist records in the other file related to it through their key.

To use an integrity function, the programmer specifies the event that is to invoke the function at run time (this event is known as *trigger*), the conditions and values that make up the constraint, and the action to take in case of error (display a message, prevent a change or deletion, etc.). Triggers may be included in the application when a certain field is modified, after a record is added to a certain file, before a record is deleted, and so on.

Validity checks like these can be easily implemented in the application, of course, using the conditional constructs or exception-handling features available in most languages. So it is not at all clear why a database system must provide these checks in the form of built-in functions. Still, if we agree that higher levels of abstraction are sometimes beneficial, these functions do provide a good alternative for specifying and enforcing certain validity criteria.

Only simple checks, however, can be implemented through standard, built-in functions. This is true because all we can do in a standard function is specify a few conditions and values and the action to take, while most integrity checks entail *combinations* of conditions, values, and actions. Thus, the checks we need in a typical application may affect several fields and files, may require a unique piece of logic, and may need some data that resides in the application, not in the database.

So, like all high-level operations, the concept of standard integrity functions can be useful if provided as an *option*, to be employed only when better than *programming* the same checks. The relational theorists, though, hoped to turn *all* integrity checks into standard functions. Their naivety is betrayed by their attempt to *classify* the integrity functions – referential integrity, domain integrity, and so forth. They actually believed that they could discover a set of standard functions that would encompass all conceivable data validation requirements (or, at least, the most common ones). Note also the pretentious names they invented to describe what are in reality simple operations. Clearly, they believed that the concept of built-in integrity functions represents an important contribution to database science. But preventing the deletion of a particular record, or ensuring that a field's value lies within a certain range, are operations we routinely perform in every application, using ordinary programming languages; and we don't need scientific-sounding terms like "referential integrity" and "domain integrity" to describe them.

The concept of built-in integrity functions failed, of course. After devising a few standard functions, the theorists had no choice but to give us the means to create freely our own functions, which is the only way to satisfy real-world data validation requirements. And, since it is only through programming that one can create such functions, new programming languages were invented – languages whose only purpose was to allow programmers to implement these functions as part of the database, rather than part of the application. Then the languages started to grow, as programmers demanded greater functionality. Means were introduced to perform calculations, to create flow-control constructs, to call subroutines, to access memory variables, to use general-purpose function libraries, and so forth. These languages provided, in other words, more and more of the very same features that were already available in the *traditional* languages.

No one noticed the absurdity of this situation. *Programming* our own functions is an alternative we always had. The promise had been, not a new language, but a higher level of abstraction. And if this turned out to be impossible, the theorists should have admitted that the only way to implement versatile data validation is through a programming language – the way we always did – and encouraged us to return to the traditional methods. What is the point in inventing specialized languages, indicating by means of "triggers" where in the application we need the integrity functions, and placing the functions in the database, when we could simply keep them as part of the application? After all, despite the multiplying features, the new languages remain inferior to the traditional ones, even in the narrow domain of database work that is their specialty. So programmers must now assimilate and depend on some new languages without deriving any real benefits. In the end, not only do the relational systems fail to provide the promised higher level of abstraction, but they make the task of data validation more complicated than before.[26]

The theorists, of course, could not admit that the concept of high-level integrity functions had failed, and that we must return to the traditional methods, because this would have been tantamount to admitting that the relational model had been falsified. So, inventing new languages was the way to cover up this falsification. Imagine an application written in COBOL, and a database system that asks us to write the data validation functions also in COBOL, but to store them in the database. Since we know that we can

[26] We hear sometimes the argument that storing the integrity functions outside the application facilitates the implementation of corporate standards, as all validity criteria are specified in one place. But this argument is tenuous. First, we can accomplish the same thing with ordinary subroutines. Second, even with the functions outside the application, why do we need new languages?

accomplish the same thing by making those functions an integral part of the application, we would reject the database alternative as absurd. If, however, it is not in COBOL but in a new language that we must write these functions – and if the language is accompanied by some new and impressive terminology, and if it is provided through a large and intricate development environment – the absurd alternative can be made to look like an important programming concept. And if we add to this the enthusiasm of the experts and the media, and the urgent needs of the companies that already depend on relational systems, everyone would perceive this concept as progress. Thus, what is in reality a *falsification* of the relational model is made to appear as an *enhancement* of the model.

Reinstating the programming capabilities, then, is what made the relational systems practical. *All* relational principles had to be annulled, as we saw earlier; but the other annulments would have amounted to nothing had the restriction to high-level database operations been maintained. The idea of programmable integrity functions was so well received because it provides the means to bypass the restriction to relational operations. Although not as useful as the traditional file operations, the operations available through the new languages do have similar capabilities. So they allow us to implement many database requirements that would be impractical through relational operations alone.

Thus, in the guise of integrity functions, programmers could now add to their applications a great variety of *low-level* file operations. Whenever a database requirement was too complicated or too inefficient to express through the relational operations, they could program it in the form of an integrity function and define a "trigger" in the application to invoke it. After all, with a little imagination any database requirement can be associated with some integrity checks or rules. For example, if we have to modify a record in such a way that a field's value is the result of calculations and conditions involving some other files and some memory variables – a task impossible or impractical through relational operations – we can program all this in a database language and call it an integrity function.

Understandably, this stratagem was very popular. Programmers praised the virtues of the relational model, but resorted to "integrity" functions and "triggers" whenever the requirements called for low-level file operations, or low-level links between database entities and other types of entities; in other words, whenever they needed the capabilities of the *traditional* file operations. They appeared to like the relational model, but what they liked in reality was the new, low-level capabilities – which contradict the relational principles.

In the end, all pretences of integrity and triggers were discarded, and these functions were expanded into the broader concept known as *stored procedures*. These procedures are general-purpose pieces of software that can be employed freely in the application. They are stored in the database, but are used like ordinary subroutines: they can be invoked from the application or from other stored procedures, can have parameters, and can return values. And, since there is no limit to the size or number of stored procedures, larger and larger portions of the application were being developed in this new fashion, in order to take advantage of the low-level capabilities of the database languages. Thus, while programmers were convinced that they were using the relational model, their applications resembled more and more those developed with the traditional languages and file operations.

So, by allowing programmers to bypass completely the relational principles, the concept of stored procedures was the final answer to the need for low-level file operations and low-level links to the other aspects of the application.

5

Although there are many relational database languages, it is SQL that became the official one. And it is through SQL, more than through the others, that the fraud of reinstating the traditional concepts was committed. Today's relational systems would be unusable without SQL. From its modest beginning as a query language, SQL has achieved its current status, and has grown to its enormous size, as a result of the enhancements introduced in order to provide programming and low-level capabilities – precisely those capabilities that the relational model had claimed to be unnecessary. Thus, today's official relational language is in reality the official means, not of *implementing*, but of *overriding*, the relational principles. Let us study this evolution.

The original relational model, we recall, was meant only for queries. And SQL (which stands for Structured *Query* Language) was the language through which programmers and users alike were expected to access the database. The original SQL, thus, allowed us to select and combine subsets of tables by specifying various criteria in the form of relational operations.

The SQL statement for queries is SELECT. This one statement, however, contains many clauses, which permit us to specify various details: the files involved in the query, the operations required to relate these files, the sorting sequence, the record selection criteria, which fields to display, and how to group the selected records for showing subtotals and the like. Thus, while neat and straightforward for trivial queries, a SELECT statement can become very long and complicated for intricate queries or queries involving several files.

The reason is that, no matter how complex, a query must be expressed in its entirety in *one* statement. Specifications that in a traditional language would be implemented naturally by combining some simple constructs must be expressed now by means of clauses and further SELECTs awkwardly nested within the various parts of the main SELECT. Moreover, in order to support real-world queries, some contrived features had to be added to SELECT. The features are, in reality, substitutes for ordinary programming concepts. But, while the traditional languages provide these concepts naturally, as diverse statements, in SQL they must all be crammed, artificially, into the SELECT statement. SQL, thus, while perceived as a modern, high-level database language, is in fact a primitive, ugly language.

Another way to include traditional operations in the SELECT statement was by making them look similar to the relational operations. For example, an operation that results in one value for a group of selected records (the sum of the values present in certain fields, or their average, or minimum) can be included through an option that creates a temporary file of one record where the fields contain the result; and an operation performed on a certain field in every record in the group (calculating the square root, multiplying by a constant, etc.) can be included through an option that creates a temporary file with the same number of records as the original group, but where the fields contain the new value. Many operations easily performed in traditional languages (mathematical and statistical functions, character string manipulation, date and time calculations, etc.) were artificially added to the SELECT statement in this fashion.

Clearly, if we have to develop real-world applications while being unable to create our own file scanning loops, and if SELECT is the only statement available, every operation that we will ever need must be included somehow in this one statement. Thus, the reason for the growing complexity of SELECT is the desire to keep SQL "non-procedural"; specifically, the attempt to provide programming capabilities while restricting these capabilities to a higher level of abstraction than a traditional language. This is an absurd quest, since, if we want the ability to implement any conceivable queries, the language must provide low-level file operations. (We examined in chapter 6 the fallacy of non-procedural languages; see pp. 442–443.) So, in the end, the entities and operations that became part of the SELECT statement had to be of the same level of abstraction as those used in traditional languages: fields, records, keys, comparisons, calculations, and so on.

What the relational theorists are trying to avoid at any cost, even if the cost is increased complexity, is code like that shown in figures 7-13 to 7-16 (pp. 680, 683–685); that is, traditional programming, where the file operations are managed through explicit flow-control constructs. An SQL SELECT statement

may well be a little shorter than the equivalent COBOL code, but it does not provide a higher level of abstraction.[27] What is different between SQL and COBOL – *implicit* loops and conditions as opposed to *explicit* ones – is the easy, mechanical part of programming. The difficult part – the overall logic, the file relations, the concept of nested loops and conditions, the links between database entities and the other entities in the application – is necessarily the same in both. With SQL or with COBOL, since the computer cannot know what we want, the only way to implement a given query is by specifying all the details. It is futile to seek a higher level of abstraction.

Thus, even when restricted to queries (and hence still within the relational model), we already note the need to enhance SQL in order to extend its usefulness beyond trivial requirements, as well as the effort to cover up the fact that this is achieved by introducing *programming* capabilities. A complex SQL query is in reality a little program, and what we are doing when creating a complex SELECT statement is programming. We would be better off, therefore, to implement that requirement as several simpler statements, linked through a flow-control structure that follows naturally and intuitively the query's logic, as we do in most languages. But then we could no longer delude ourselves that SQL is non-procedural, or that we are using only high-level relational operations. In the end, as in all mechanistic software delusions, not only did SQL fail to eliminate the need for programming, but in attempting to do this it made programming more difficult.

When the relational systems were expanded to include updating operations, SQL was enhanced with the capability to add, modify, and delete records. The respective statements are INSERT, UPDATE, and DELETE. And these statements are very similar to SELECT, in that they create an implicit file scanning loop and include clauses for various details (record selection criteria, for instance). Updating operations, we recall, are not part of the formal relational model. Thus, regardless of how we feel about SQL as a *query* relational language, the new statements cannot be judged at all by relational principles. So the fact that they are in the same contrived style as SELECT, or the fact that INSERT also permits us to bypass the relational principles altogether and process individual records, can easily be overlooked.

Recall the traditional file operations (pp. 676–679): WRITE, REWRITE, DELETE,

[27] SQL code corresponding to the COBOL code of figure 7-13 might be: SELECT P-NUM FROM PART WHERE P-NUM>=P1 AND P-NUM<=P2 AND P-QTY>=Q1 ORDER BY P-NUM. For the operations in figures 7-14 to 7-16, however, the SQL code would be far more involved, especially if we have to access individual fields from two or three files at the same time.

READ, START, and READ NEXT. We concluded that this is the minimal practical set of file operations – the operations that are both necessary and sufficient for using indexed data files in serious applications. In conjunction with the flow-control constructs provided by the traditional languages, these operations permit us to implement any conceivable database requirement. Putting it in reverse, to permit us to implement any database requirement, a database system *must* provide these operations, or their equivalent.

After the various enhancements, SQL provided four of these operations: INSERT, UPDATE, DELETE, and SELECT correspond, respectively, to the traditional WRITE, REWRITE, DELETE, and READ. Only START and READ NEXT had no SQL counterpart. READ NEXT instructs the file system to retrieve the current record in the indexing sequence and advance the pointer to the next record. It is normally used, therefore, in a file scanning loop (and START is used once, before the loop, to indicate the first record). READ NEXT was thought to be unnecessary in SQL because the four other statements create their own, implicit file scanning loops.

So, with the traditional operations we use READ to access *individual* records, and READ NEXT to access in a loop a series of *consecutive* records; and to modify or delete records we use REWRITE or DELETE, either for individual records or in a READ NEXT loop. With the SQL statements, on the other hand, we access records *only* in a loop – the *implicit* loop generated by each one of the four statements. (Consequently, if we need to access a single record in SQL, we must specify selection criteria that will result in a trivial loop of one iteration.)

The most striking difference between SQL statements and the traditional operations, then, is the implicit file scanning loop as opposed to the loop that we create ourselves. So SQL statements are a little simpler, but to benefit from this simplicity we must give up all flexibility. When we create our own scanning loop, in a traditional language, we can include in the loop additional operations (to perform various tasks related to the file operation). In SQL, the only operations we can have in the loop are those provided by the statement itself, through its clauses. For example, in SQL we can specify with UPDATE the record selection criteria and how to modify the fields in these records. But with a traditional language, a loop based on READ NEXT and REWRITE can also include display operations, subroutine calls, and calculations involving both database fields and memory variables. Thus, when we create our own file scanning loop we can easily link the file entities to the other entities in the application. This is the seamless integration of the database and the application that we discussed earlier (see “The Lost Integration”).

We saw how the relational theorists crammed into SELECT various features in an attempt to restore some of the flexibility that was lost in the implicit SQL loops. But there is a limit to the number of operations that can be specified in this fashion, and in the end they had to admit that the capability to create *explicit* file scanning loops, and to control the operations in the loop, is an indispensable database feature. So they added this feature to SQL too, by way of a new enhancement: the FETCH statement.

FETCH is the true counterpart of the traditional READ NEXT: it lets us create explicit loops, and retrieve one record at a time, just as we do in a traditional language. (There is no equivalent of the traditional START: in SQL we always start from the beginning of the file, and the system will deliver only those records that passed the selection criteria previously specified with a SELECT.) FETCH, of course, is not independent. To use it we also need the capability to create explicit loops, and this capability was added to SQL by means of further enhancements: actual loop-control constructs, and a way to perform SQL statements from within a traditional language. (We will examine these enhancements in a moment.)

The mechanism through which we read one record at a time in a loop is known in SQL as *cursor*, and is identical to the mechanism known as *pointer* in the traditional file operations (see pp. 677, 678). The cursor is the indicator that keeps track of records in the current indexing sequence: each time we perform a FETCH, the system retrieves the record identified by the cursor and advances the cursor to the next record – just as it does in the case of the traditional READ NEXT. And if we do this at the beginning of each iteration, all the operations in the loop will be able to access the fields in that record. Thus, in SQL too we can now include in a file scanning loop any operations we want, and thereby link the database fields to other types of entities (memory variables, display fields, etc.). Also, when used in conjunction with FETCH, UPDATE and DELETE can now modify or delete individual records in a scanning loop – just as REWRITE and DELETE can in conjunction with READ NEXT in a traditional loop.

To appreciate the importance of the cursor, we must recall the original relational principles. For, without the means to create explicit file scanning loops, SQL would have been almost useless. The relational model specifically restricts us to high-level operations: all we can do is extract and combine logical portions of tables. The permitted operations are PROJECTION, SELECTION, UNION, JOIN, and the like (see p. 702). The original SQL SELECT statement, with its implicit file scanning loop, follows this principle: we specify the operations

through the various clauses of a SELECT, and combine them by nesting SELECTs within one another. At every step we manipulate only tables – tables that contain, usually, just some of the records and fields of an actual data file.

So the original SELECT statement (plus INSERT, UPDATE, and DELETE if we allow updating operations) is *all we need* in order to implement the relational model in SQL. This is true because in high-level queries, as the model was originally intended, we only need the means to specify which fields to list, and such details as their order and format. But if we want to employ the model for *any* conceivable query, in *any* application, we need the means to perform additional operations with these fields, not just list them. Also, we need the means to use the fields together with other data types – display and data entry fields, and memory variables. The theorists hoped at first to satisfy these two demands by adding more and more options to the SELECT statement; that is, by inventing a high-level feature for every conceivable situation. This is an absurd idea, however, and they realized in the end that the only practical solution is to permit *low-level* operations.

Thus, only trivial requirements can be implemented if restricted to the implicit file scanning loops of SELECT. It was by adding to SQL the concept of a cursor, and the means to create *explicit* file scanning loops, that we gained the two critical qualities: the capability to perform additional operations in the loop, and the capability to link low-level database entities (individual fields and records) to other entities in the application.

With the concept of a cursor, then, all the capabilities of the traditional file operations were finally available in relational systems. But this was accomplished by abolishing the relational principles: the way we use data files in SQL is now practically identical to the way we use them in a traditional language.

As SQL was used for more and more demanding tasks, it had to be enhanced with the kind of features found in general-purpose programming languages. And software vendors increasingly used *these* features – which have nothing to do with the relational principles, or with database operations – as a way to promote their database systems and attract buyers. For example, some vendors enhanced their version of SQL with the means to create conditional, iterative, and other flow-control constructs (officially abandoning, therefore, the idea of a non-procedural language). And, in addition to those functions similar to the traditional operations and subroutine libraries, already mentioned, countless expedients were provided to assist programmers in developing applications: functions for creating reports, for data entry and display, for system management, and so forth.

What is the point of these enhancements? We already had these features, in a hundred languages. The relational promise had been mathematical logic and a higher level of abstraction, not a new programming language. And if this idea turned out to be impractical, it should have been abandoned. Instead, like all pseudoscientists, the relational experts rescued their theory by reinstating precisely those concepts that the theory had attempted to replace. As a result, software vendors are competing today, not by stressing the *relational* capabilities of their systems, but by adding more and more low-level, programming features; that is, features meant to help us bypass the rigours imposed by the relational model. In other words, the value of a relational system is measured today by how good it is at overriding the relational principles.

But despite the enhancements, SQL remained inferior to the traditional languages. It was still too awkward and too inefficient for serious applications, so one more feature had to be invented: the capability to use SQL from within a traditional language. This feature, called *embedded SQL*, is the ultimate relational degradation: the most effective way for a programmer to enjoy the benefits of the traditional database concepts while pretending to use the relational model.

With embedded SQL, we implement in a traditional language the entire application, including all database requirements; then, we invoke isolated SQL statements here and there in the form of subroutines. The relational system, thus, is relegated to the role of subroutine library, and works similarly to a traditional file system. A typical use of this concept is with the FETCH statement, as explained earlier: we create a file scanning loop in COBOL or any other language, and use FETCH within the loop to read one record at a time – exactly the way we use the traditional READ NEXT. Every other operation in the loop is implemented in the traditional language. The resulting code is identical, for all practical purposes, although we employ in one case a relational system and in the other a traditional file system. We have come a long way from the idea of "tables and nothing but tables," accessed through high-level operations.

An important promise of the relational model had been that the result of a query is mathematically guaranteed to be correct: if we restrict ourselves to the relational operations – to extracting and combining portions of tables – the data in the final table will always reflect accurately the data in the tables we started with. So, if we bypass the restriction to relational operations, this promise no longer holds. Whether the new operations are added in the form of SELECT options or in the form of explicit file scanning loops, the benefits promised by the relational model are now lost. Without the restriction to

relational operations, what we have is no longer a relational model, so the resulting tables may or may not reflect accurately the starting ones.

The SQL fallacy, thus, is the belief that the relational model can be enhanced with features that contradict its most fundamental principles, and still retain its original qualities. The mathematical benefits were shown to emerge only if we restrict ourselves to the relational operations. The theorists keep adding features designed specifically to bypass this restriction, but they continue to promote the model with the original claims.

We already know that the *updating* operations lie outside the scope of the formal model, so the model's mathematical grounding is irrelevant when a relational system is used for general database work. And now we see that the model's mathematical grounding has become irrelevant even for queries. As was the case with the other modifications, the SQL features do not *enhance* the relational model but *annul* it.

The Verdict

In the end, what has the relational model accomplished? After thirty years of "enhancements," relational database programming is more or less the same as *traditional* database programming: we manipulate fields, keys, records, and files in order to create database structures. The only real difference is that the database operations have been separated from the rest of the application, and are now possible only through complicated, inefficient, and expensive database environments.

If we disregard that extreme degradation, embedded SQL, applications are now divided into two parts: the part written in an ordinary language, where the application's main logic resides, and the part written in SQL (in the form of integrity functions, stored procedures, and the like), where the database-related operations reside. More and more pieces of the application have been moved into the SQL part; this is not because they are easier to implement in SQL, though, but because they are closely related to the database operations, and keeping them together is the only practical way to link them. And this artificial separation obscures the fact that the part dealing with the database-related operations is now very similar to what it was when integrated with the application's main logic.

The relational charlatans, thus, claimed at first that we must separate the database operations from the application, and restrict the links between the two to a high level of abstraction, because this is the only way to benefit from the relational model. Then, when the separation proved to be impractical, they

restored the low-level links. They did it, though, not by moving the database operations back into the application, but by bringing into the SQL procedures further pieces of the application. They restored the links, thus, by reinstating in the SQL procedures the same low-level programming concepts that we had used in the application before the separation. So the benefits believed to emerge from the relational model are now lost even if we forget that they had already been lost, in the other annulments of the relational principles. The separation of the application into two parts is absurd because what we are doing in the SQL procedures is about the same as what we were doing before, in a much simpler way, in the application.

So, after all the "enhancements," there remains very little that is relational in the relational database systems. Programmers use SQL in about the same way that the traditional file operations are used. Only now and then, when not too inconvenient or too inefficient, do they employ the relational operations as they were defined in the original theory. But by calling files "tables," records "rows," and fields "columns," they can delude themselves that they are programming under the relational model.

It must be noted that some features are indeed found only in relational systems. But these features could easily be added also to the traditional file systems, simply because they have nothing to do with the relational model. These are the kind of features made possible by hardware and software advances – larger files, new types of fields, enhanced caching and buffering, better security or backup facilities, and the like. So, if these features are missing in a file system, it is deliberate: in their effort to make everyone dependent on complicated and expensive development environments, the software elites are doing everything in their power to discredit the straightforward, traditional languages and file systems; and refusing to keep them up to date is part of this manipulation.

It is all the more remarkable, thus, that the traditional languages and file systems, while remaining practically unchanged for the last thirty years, have been the chief source of inspiration for the features added to the relational systems. This shows, again, just how little the relational model itself had to offer.

The relational model is still described as an application of mathematical logic. And those monstrous database systems are promoted with the claim that the relational model is the only way to have rigorous databases, even as everyone can see that these systems have little to do with the relational model, and that their only practical features are those taken from the traditional languages and file operations. So, like the theories of structured programming and object-oriented programming, and like all other pseudosciences, the relational theory continues to be promoted on the basis of its original

principles even after these principles were abandoned, and hence their benefits were lost.

If we have to bypass the relational restrictions and revert to operations that are practically identical to the traditional ones, in what sense is the relational model beneficial? The theorists are committing a fraud when promoting the relational systems if, at the same time, they enhance these systems with means to override the relational principles.

The multibillion-dollar relational database industry thrives on the incompetence of the software practitioners, whose skills are limited to knowing how to use programming aids and substitutes. To repeat, the six basic file operations and an ordinary language are all we need in order to implement database requirements. Thus, only a programmer incapable of designing some simple loops and conditions can be impressed by the relational features. Every one of these features has been available – in a much simpler form, through file management systems and languages like COBOL – since about 1970.