SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*
Section *Structured Programming*

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

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

www.softwareandmind.com

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

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

For disclaimers see pp. vii, xvi.

Designed and typeset by the author with text management software developed by the author and with Adobe FrameMaker 6.0. Printed and bound in the United States of America.

Acknowledgements

Excerpts from the works of Karl Popper: reprinted by permission of the University of Klagenfurt/Karl Popper Library.

Excerpts from *The Origins of Totalitarian Democracy* by J. L. Talmon: published by Secker & Warburg, reprinted by permission of The Random House Group Ltd.

Excerpts from *Nineteen Eighty-Four* by George Orwell: Copyright ©1949 George Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1949 Harcourt, Inc. and renewed 1977 by Sonia Brownell Orwell, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *The Collected Essays, Journalism and Letters of George Orwell*: Copyright ©1968 Sonia Brownell Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1968 Sonia Brownell Orwell and renewed 1996 by Mark Hamilton, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *Doublespeak* by William Lutz: Copyright ©1989 William Lutz, reprinted by permission of the author in care of the Jean V. Naggar Literary Agency.

Excerpts from *Four Essays on Liberty* by Isaiah Berlin: Copyright ©1969 Isaiah Berlin, reprinted by permission of Curtis Brown Group Ltd., London, on behalf of the Estate of Isaiah Berlin.

Library and Archives Canada Cataloguing in Publication

Sorin, Andrei
Software and mind : the mechanistic myth and its consequences / Andrei Sorin.

Includes index.
ISBN 978-0-9869389-0-0

1. Computers and civilization. 2. Computer software – Social aspects.
3. Computer software – Philosophy. I. Title.

QA76.9.C66S67 2013 303.48'34 C2012-906666-4

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

George Orwell, *Nineteen Eighty-Four*

Disclaimer

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

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

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

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

Contents

Preface

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 `GOTO` statement (often spelled as one word, `GOTO`); for example, `GOTO 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 `GOTO` 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.

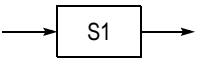

Figure 7-1

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.

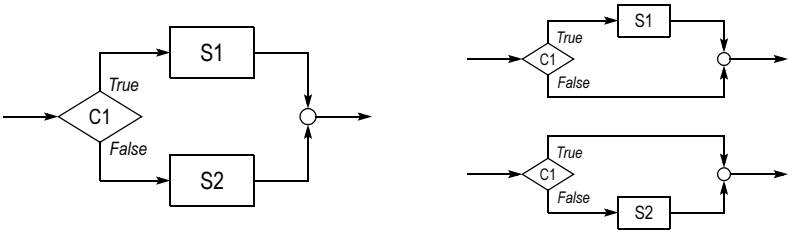

Figure 7-2

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*.

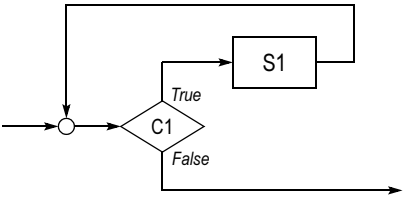

Figure 7-3

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.

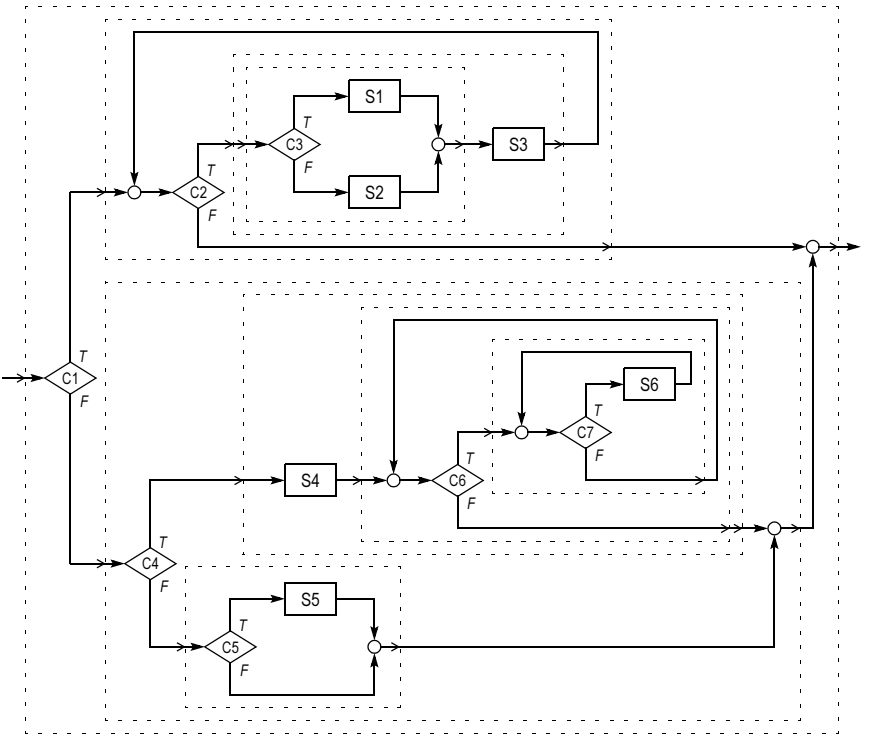

Figure 7-4

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.

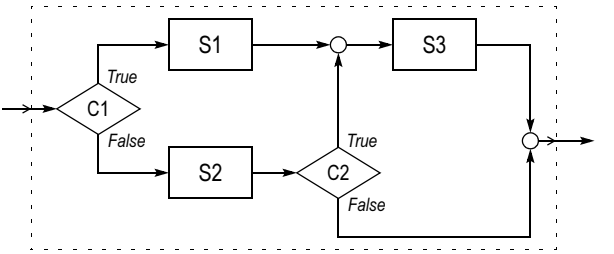

Figure 7-5

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.

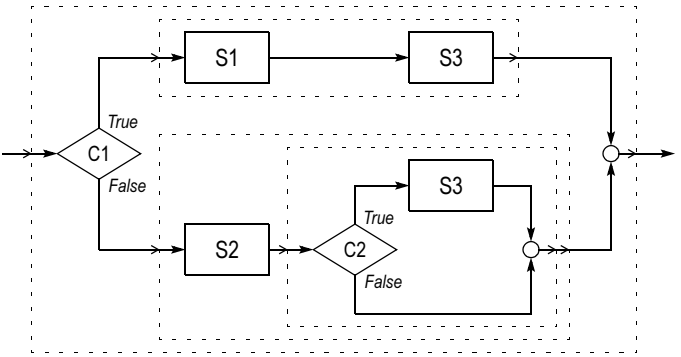

Figure 7-6

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.

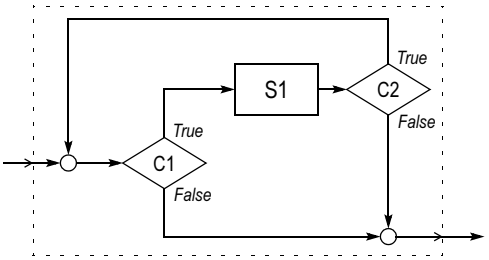

Figure 7-7

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]

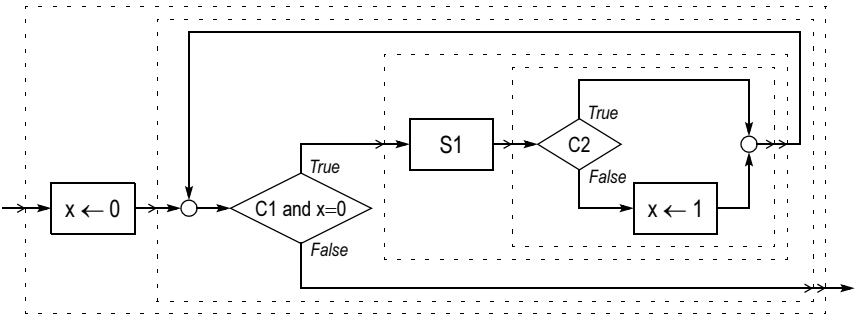

Figure 7-8

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.

We can also explain the additional flow-control structures by counting the

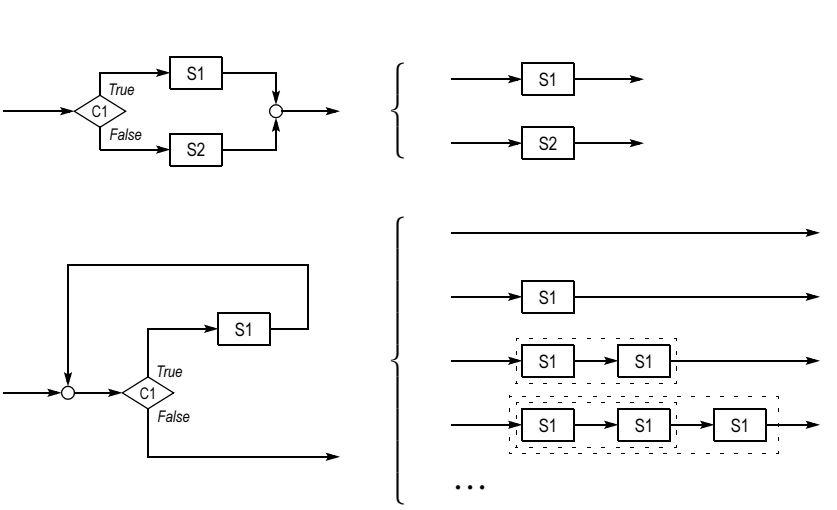

Figure 7-9

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. The links to those structures are officially reinstated by the theorists in the

[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).

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 software. They want software systems to be simple hierarchical structures, but

[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.

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, because the effort of performing the transformations – the essence of the

[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.

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

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

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]).

Although they proved it through mathematical logic, we can demonstrate

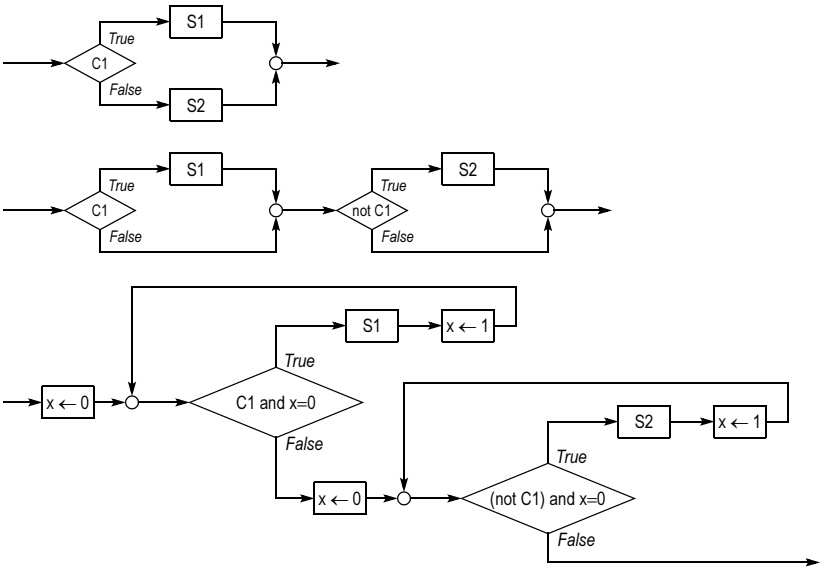

Figure 7-10

this reduction with flow diagrams (see figure 7-10). In the first step, the conditional construct is reduced to two consecutive, simpler conditional constructs. The new constructs have only one operation each, *S1* and *S2*,

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

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.

The theory of structured programming, thus, is inconsistent if its principles

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

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 generalized loop mechanism. 3. A binary-decision mechanism."[9] "Böhm and

[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.

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?

To understand the reason, recall that characteristic feature of structured

[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.

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 supposed to guarantee *mathematically* (that is, with no qualifications) the

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

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]

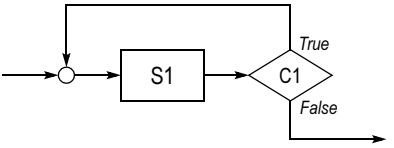

Figure 7-11

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]

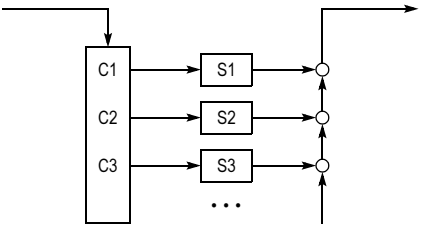

Figure 7-12

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 non-standard 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 flowcharts, 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 `GOTO`s 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.