SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*
Section *Structured Programming*
Subsection *The* GOTO *Delusion*

This subsection examines the fallacies surrounding the *GOTO* statement and its prohibition in structured programming.

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

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

Acknowledgements

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

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

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

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

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

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

Library and Archives Canada Cataloguing in Publication

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

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

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

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

Printed on acid-free paper.

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

George Orwell, *Nineteen Eighty-Four*

Disclaimer

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

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

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

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

Contents

Preface

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

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

Throughout the book, the software-related arguments are also supported with ideas from other disciplines – from philosophy, in particular. These discussions are important, because they show that our software-related problems are similar, ultimately, to problems that have been studied for a long time in other domains. And the fact that the software theorists are ignoring this accumulated knowledge demonstrates their incompetence. Often, the connection between the traditional issues and the software issues is immediately apparent; but sometimes its full extent can be appreciated only in the following sections or chapters. If tempted to skip these discussions, remember that our software delusions can be recognized only when investigating the software practices from this broader perspective.

Chapter 7, on software engineering, is not just for programmers. Many parts (the first three sections, and some of the subsections in each theory) discuss the software fallacies in general, and should be read by everyone. But even the more detailed discussions require no previous programming knowledge. The whole chapter, in fact, is not so much about programming as about the delusions that pervade our programming practices. So this chapter can be seen as a special introduction to software and programming; namely, comparing their true nature with the pseudoscientific notions promoted by the software elite. This study can help both programmers and laymen to understand why the incompetence that characterizes this profession is an inevitable consequence of the mechanistic software ideology.

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

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

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

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

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

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

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

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

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

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

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

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.

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. 571–575). 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. 603–604). 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. 522–523), 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. 553–556, 592–593): 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. 537). 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. 604–605), 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. 592–593.) Thus, simply by prohibiting mediocre programmers from creating their own flow-control constructs, we will attain about the same results as we would by employing expert programmers.

❖

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6

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

The following discussion, thus, is not intended to promote a particular programming style, but to demonstrate the triviality of the GOTO problem, and hence the absurdity of its prohibition. This will serve as additional evidence for my argument that the GOTO prohibition was motivated, not by some valid software concerns, but by the corrupt ideology held by the software theorists. They had already decided that programmers must remain incompetent, and that it is they, the elite, who will control programming practices.

❖

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

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

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

Lastly, we must agree on a good naming system for labels. Labels are those flow-control variables that identify the statement where execution is to continue after a jump. And, since each GOTO statement specifies a label, we can choose names that link logically the jump's origin, its destination, and the purpose of the jump. This simple fact is overlooked by those who claim that jumps unavoidably make programs hard to follow. If we adopt an intelligent naming system, the jumps, instead of confusing us, will *explain* the program's logic. (The compiler, of course, will accept any combination of characters as label names; it is the human readers that will benefit from a good naming convention.)

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

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

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

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

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

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

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

❖

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

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

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

[48] Figures 7-13 to 7-16 (pp. 694, 697–699) 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.