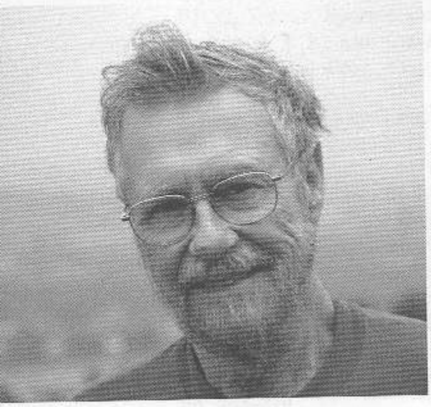
From "Goto Considered Harmful" to Structured Programming

Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
dijkstra@cs.utexas.edu



Ph.D. in Computing Science,
University of Amsterdam

Professor of Mathematics,
Eindhoven University of Technology:

ACM Turing Award,
AFIPS Harry Good Memorial Award

Member of the Royal Netherlands
Academy of Arts and Sciences

Distinguished Fellow of the
British Computer Society

Major contributions: implementation
of ALGOL 60, THE operating system,
structured programming

Current interests: formal derivation
of proofs and programs, streamlining
of the mathematical argument

Edsger W. Dijkstra

EWD 1308: What Led to "Notes on Structured Programming"

The purpose of this historical note is to describe the experiences which with hindsight seem to have influenced me when I wrote EWD 249 "Notes on Structured Programming" in 1969. The note is based on what I remember; I am sure my memory has been selective and hence don't claim the objectivity of a professional historian.

I was introduced to programming at the 1951 Summer School, given in Cambridge by Wilkes, Wheeler and Gill, and in 1952 became on a part-time basis – initially two days per week – the programmer of the Mathematical Centre in Amsterdam; the rest of the week I studied theoretical physics in Leyden.

My only model was the program organization for the EDSAC in Cambridge; I followed it closely when designing program notation, input, output and library organisation for the ARRA in Amsterdam. For the next machines, the FERTA, the ARMAC and the X1, program notation and input would very much follow the same pattern: I clearly was a conservative programmer. Add to this the ARMAC's instruction buffer with a capacity of one track of the drum having destroyed the store's homogeneity, then you will understand that I did not embark on adventurers like "autocoders".

In 1955 I made the decision not to become a theoretical physicist, but to become a programmer instead. I made that decision because I had concluded that of theoretical physics and programming, programming embodied the greater intellectual challenge. You see, in those days I did not suffer from intellectual modesty. It was a difficult decision, for I had been groomed as a first-class scientist and becoming a programmer looked like a farewell from science. When I explained my dilemma to A. van Wyngaarden, then my boss, he told me that computers were here to stay and that in the world of programming I could very well be the one to create the science that was still lacking. Getting my physics degree in Leyden became a formality to be done as quickly as possible. (As a matter of fact I no longer felt welcome in Leyden: the physicists considered me a deserter and the mathematicians, one of whom openly prided himself on "of course knowing nothing about computers", were just contemptuous.)

In the meantime, a pattern emerged for the co-operation between me and my hardware colleagues Bram J. Loopstra and Carel S. Scholten. After the functional specification of the next machine had been written down (usually by me), that document served as a kind of contract between us: it told them what machine to design and construct, while I knew what I could count on while writing all the basic software for the machine. The aim of this division of labour was that my programs would be ready by the time the construction of the machine had been completed.

Looking back I now observe that the above arrangement has had a profound influence on how I grew up as a programmer: I found it perfectly normal to program for not yet existing machines. As a byproduct it became firmly ingrained in my mind that I programmed for the abstract machine as specified in the original document, and not for the actual piece of hardware: the original document was not a description but a prescription, and in the case of a discrepancy, not the text but the actual hardware would be at fault.

At the time I regarded this division of labour and the resulting practice of programming for non-existing machines as perfectly normal. Later I read an American article on why software was always late; I remember being very amazed when I read that limited availability of the hardware was a main cause, and I concluded that the circumstances under which I had learned programming had been less common than I had assumed.

Of course I could not exclude from my designs typographical errors and similar blemishes, but such shortcomings did not matter as long as the machine was not ready yet, and after the completion of the machine they could be readily identified as soon as they manifested themselves, but this last comforting thought was denied to me in 1957 with the introduction of the real-time interrupt. When Loopstra and Scholten suggested this feature for the X1, our next machine, I had visions of my program causing irreproducible errors and I panicked.

Eventually, Loopstra and Scholten flattered me out of my resistance and I studied their proposal. The first thing I investigated was whether I could demonstrate that the machine state could be saved and restored in such a way that interrupted programs could be continued as if nothing had happened. I demonstrated instead that it could not be done and my friends had to change their proposal. Admittedly, the scenarios under which the original proposal would fail were very unlikely, but this could have only strengthened my conviction that I had to rely on arguments rather than on experiments. At the time that conviction was apparently not so widespread, for up to seven years later I would find flaws in the interrupt hardware of new commercial machines.

I had a very illuminating experience in 1959, when I had posed the following problem to my colleagues at the Mathematical Centre. Consider two programs that can communicate via atomic reads and writes in a shared store. Can they be programmed in such a way that the executions of their critical sections exclude each other in time? Solutions came pouring in, but all wrong, so people tried more complicated "solutions". As these required more and more elaborate counter examples for their refutation, I had to change the rules: besides the programs they had to hand in an argument why the solution was correct. Within a few hours T. J. Dekker handed in a true solution with its correctness argument. Dekker had first analysed the proof obligations, then chosen the shape of an argument that would meet them, and then constructed the program to which this argument was applicable. It was a very clear example of how much one loses when the role of mathematics is confined to a posteriori verification; in 1968 I would publish a paper titled "A constructive approach to the problem of program correctness".

And then there was ALGOL 60. We saw it coming in 1959 and implemented it in the first half of 1960. I was terribly afraid, this implementation was then by far my most ambitious project: ALGOL 60 was so far ahead of its time that even its designers did not know how to implement it. I had never written a compiler and had to achieve my goal with a machine that had only 4096 words of storage. (The latter constraint was of course a blessing in disguise, but I don't remember seeing that when we started.)

By today's standards we did not know what we were doing; we did not dream of giving any guarantee that our implementation would be correct because we knew full well that we lacked the theoretical knowledge that would be needed for that. We did the only thing we could do under the circumstances, namely, to keep our design as simple and systematic as we could and to check that we had avoided all the mistakes we could think of. Eventually we learned that we had made mistakes we had not thought of, and after all the repairs the compiler was no longer as clean as we had originally hoped (F.E.J. Kruseman Aretz still found and repaired a number of errors after I had left the Mathematical Centre in 1962).

Right from the start we expected two very different types of errors, writing errors, whose repair is trivial, and thinking errors that would send us back to the drawing board, and the distinction has helped us because one combats them with different techniques. J.A. Zonneveld and I combatted the writing errors by coding together, each with his own text in front of him. When we were done, both our texts were punched independently, the two tapes were compared mechanically and about two dozen discrepancies – if I remember correctly – showed up. The thinking errors we had tried to prevent by convincing each other why the proposed section would work. With this reasoning we would mainly discuss the workings of the compiler while the program compiled was treated as data, and this experience was helpful for later, as it made us accustomed to non-operational considerations of program texts.

The whole experience made me receptive to what later would be called modularization or divide-and-rule or abstraction. It also sensitized me to the care with which interfaces have to be chosen and to the potential scope of the programming challenge in general. It heavily contributed to my subsequent opinion that creating confidence in the correctness of the design was the most important but hardest aspect of the programmer's task. In a world obsessed with speed, this was not a universally popular notion.

I remember from those days two design principles that have served me well ever since:

(i) before really embarking on a sizeable project, in particular before starting the large investment of coding, try to kill the project first, and
(ii) start with the most difficult, most risky parts first.

My first test program was almost the empty block, say

```
begin real x end,
```

not the most difficult example, but my 4th test was a double summation in which the nested calls of the summation routine were introduced via the parameter mechanism, while the summation routine itself had been defined recursively. In passing we had demonstrated the validity of what became known as Jensen's Device.

After this implementation interlude I returned in fairly general terms to the still open problem of the proper coordination of, in principle, asynchronous components. Without being directly involved I had witnessed a few years earlier the efforts of coupling all sorts of punched card equipment to the X1 and had been horrified by the degree of complication introduced by the inclusion of real-time commitments. For the sake of simplicity I therefore insisted on "timeless" designs, the correctness of which could be established by discrete reasoning only.

Almost unavoidably the model of Cooperating Sequential Processes emerged: sequential processes with (by definition!) undefined relative speeds and hence, for the sake of their cooperation, equipped with some primitives for synchronization.

Another opportunity for simplification was presented when we recognized that the timing aspects between a piece of communication equipment and the program that used it were completely symmetrical and independent of whether we had an input or output device. Needless to say, this unification helped a lot.

When we got involved in the design of the THE Multiprogramming System, scaling up slowly became a more and more explicit concern. It had to. Within IBM, and possibly elsewhere as well, circulated the concept as a supposed law of nature that "system complexity" in some informal sense would grow as the square of the number of components; the reasoning behind it was simple – each component could interfere with every other component – but if it were true it would de facto rule out systems beyond a certain size. This evoked my interest in systems structured in such a way that "system complexity" in the same informal sense would not grow more than linearly with the size. In 1967, the expression "layers of abstraction" entered the computer lingo.

Let me close the discussion of this episode by quoting the last two sentences of EWD 123 "Cooperating Sequential Processing" (September 1965):

> If this monograph gives any reader a clearer indication of what kind of hierarchical ordering can be expected to be relevant, I have reached one of my clearest goals. And may we not hope that a confrontation with the intricacies of Multiprogramming gives us a clearer understanding of what Uniprogramming is all about?

In 1968 I suffered from a deep depression, partly caused by the Department, which did not accept Informatics as relevant to its calling and disbanded the group I had built up, and partly caused by my own hesitation about what to do next. I knew that in retrospect, the ALGOL implementation and the THE Multiprogramming System had only been agility exercises and that now I had to tackle the real problem of How to Do Difficult Things. In my depressed state it took me months to gather the courage to write (for therapeutic reasons) EWD 249 "Notes on Structured Programming" (August 1969); it marked the beginning of my recovery.

EWD 249 tries to synthesize the above mentioned ingredients of the preceding decade. It mentions on an early page "the program structure in connection with a convincing demonstration of the correctness of the program", mentions as our mental aids "(1) Enumeration, 82) Mathematical Induction, (3) Abstraction", and about the first and the last I quote (from EWD 249-14):

> Enumerative reasoning is all right as far as it goes, but as we are rather slow-witted it does not go very far. Enumerative reasoning is only an adequate mental tool under the severe boundary condition that we only use it very moderately. We should appreciate abstraction as our main mental technique to reduce the demands made upon enumerative reasoning.

I had had two external stimuli: the 1968 NATO Conference on "Software Engineering" in Garmisch-Partenkirchen and the founding of the IFIP Working Group on „Programming Methodology". Thanks to the ubiquitous Xerox machine, my typewritten text could spread like wildfire, and it did so, probably because people found it refreshing in the prevailing culture characterized by the 1968 IBM advertisement in Datamation, which presented in full colour a beaming Susie Mayer who had solved all her programming problems by switching to PL/I. Apparently, IBM did not like the popularity of my text; it stole the term "Structured Programming" and under its auspices Harlan D. Mills trivialized the original concept to the abolishment of the goto statement.

Looking back I cannot fail to observe my fear of formal mathematics at the time. In 1970, I had spent more than a decade hoping and then arguing that programming would and should become a mathematical activity; I had (re)arranged the programming task so as to make it better amenable to mathematical treatment, but carefully avoided creating the required mathematics myself. I had to wait for Bob Floyd, who laid the foundation, for Jim King, who showed me the first example that convinced me, and for Tony Hoare, who showed how semantics could be defined in terms of the axioms needed for the proofs of properties of programs, and even then I did not see the significance of their work immediately. I was really slow.

Finally a short story for the record. In 1968, the *Communications of the ACM* published a text of mine under the title "The goto statement considered harmful", which in later years would be most frequently referenced, regrettably, however, often by authors who had seen no more of it than its title. This text became a cornerstone of my fame by becoming a template: we would see all sorts of articles under the title "X considered harmful" for almost any X, including one titled "Dijkstra considered harmful". But what had happened? I had submitted a paper under the title "A case against the goto statement", which, in order to speed up its publication, the editor had changed into a "Letter to the Editor", and in the process he had given it a new title of his own invention! The editor was Niklaus Wirth.