Alex Groce (agroce@gmail.com), Northern Arizona University

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare's *Structured Programming* is a foundational text in software engineering, programming languages, and computer science, although perhaps now a seldom-read one. This column will focus primarily on Dijkstra's part of the book, the first 80 or so pages, his famous "Notes on Structured Programming." I'll also comment briefly on Dahl and Hoare's contributions, which are well worth your time.

Dijkstra has had a bit of a (bad) reputation. He was brilliant, but he was overly mathematical, overly harsh, and (in writing) really sort of a jerk, I think, is not an uncommon point of view. I did not have the pleasure of interacting with Dijkstra in person, though I have had close colleagues who did. I think the charge against his written works is unfair; Dijkstra was sometimes wrong (aren't we all?) but his ideas were more practical, more reasonable, and less esoteric than is often believed. Yes, Dijkstra wrote "mathematically" -- at the time, almost all computer programming was to perform some series of mathematical calculations, and furthermore, if we want to understand a formal entity like a computer program, mathematics is the best (perhaps only) place we can really make a start. So, did Dijkstra (famously) oppose the use of "goto" because it made the math too hard, and he arrogantly thought programming should be driven by his mathematical notions? No. Dijkstra opposed the use of goto because he 1) knew how *hard* programming is and how limited our minds are but 2) wanted us to nonetheless *be able to understand computer programs*, which are phenomenally complicated objects.

A large part of the point Dijkstra wants to make remains, I think, absolutely essential to software engineering in the present day. If we are to understand programs, and their executions, then being able to make mappings between *program text* and *progress of execution* in a "real run" of the program is simply essential. All of Dijkstra's goto-hate boils down to a faithfulness to that principle: when we restrict the constructs we use in programs, our ability to understand "where we are" in a program may be greatly increased. I see no reason to reject Dijkstra's principle here: to the extent we cannot map between program text and program run, we are going to be in rough waters indeed. When this is impossible, perhaps because the program is concurrent, or because compiler optimizations have altered the work (we thought was) being performed almost beyond recognition, most software engineers are lost at sea.

Two points Dijkstra emphasizes are, fundamentally, not really mathematical at all, or at least only in some abstract sense of mathematics as a "science of all structure." These points are intimately related, to the point of almost being one point differently explained, and, I would argue, make up the bulk of the actual substance (vs. stepping through program examples) of Dijskstra's notes. People who have only "heard of" Dijkstra's ideas but not read him, may find both in some ways surprising. The first idea is that the way to write a program is not so much about a process of logical breaking-down (though that happens) as a metaphor in which, in order to write program X, you write it as if you had a programming language (or machine, as Dijkstra more commonly says) in which writing that program was quite easy. Then repeat, to build each layer. Dijkstra approaches this in a top-down fashion, but nothing about his

understanding of the idea makes it essentially top-down, conceptually. In fact, Dijkstra strongly implies that the more primitives your machines support, the more effectively this approach is likely to work. There is therefore a clear link between Dijkstra's ideas and those of the camp of LISP-advocates (Paul Graham is a more recent explicit proponent of this, but the idea of LISP as a programming language in which you build programming languages is longstanding) who think of programming as about building a programming language for the problem at hand, after which the task becomes trivial. That viewpoint, without much explicit acknowledgment, has become in some ways ubiquitous in modern computing, through the use of very powerful standard libraries, and the acceptance of Domain Specific Languages as normal, rather than esoteric. The second aspect of this idea is to delay design decisions as long as possible, and make it easy to change them. Otherwise, larger programs will be impossible to deal with, Dijsktra explains. "Shoving" some decisions down to the "implementation of the low-level language" is one important way to achieve this desired delay.

The remainder of the book, after Dijkstra, covers things like all the basic data structures anyone cares about (think sum and product types are a new innovation from functional languages? -- think again!) and some approaches to hierarchy. A striking thing is how much of this material will be *familiar* to anyone well read in the literature of type systems and program composition.

And, fundamentally, I think that is one reason this little book deserves to be read, not just cited as an early source for our ideas. The startling thing is not how far we have come since this book; the really starting thing is, in some ways, how little distance we have traveled. Seeing these ideas in the context of a much earlier point in computing history can make the ideas clearer, and their centrality and importance, as well as how hard they are to improve upon, clear. That's a valuable lesson for any software engineer to take home.