Structured programming and structured design as art forms

by EDWARD YOURDON

Yourdon Inc.
New York, New York

It is generally known that structured programming was initially promoted in Europe through the efforts of Edsger Dijkstra and others. However, its introduction to the USA is primarily due to the efforts of IBM; furthermore, much of the American programmer’s knowledge of structured programming is based on such superficial discussions as the December 1973 Datamation articles.

The consequences of this superficial exposure are beginning to be realized: programming without the GO-TO still offers innumerable ways of writing incomprehensible programs. More important, it is now being realized that structured programming imposes a rigid discipline on the coding of a program, but leaves important design questions to the whim of the programmer. Thus, we see a number of “structured” programs whose modules share local working storage with one another, or whose code manages to ALTER the code in other modules; the result is that an innocent change to one module causes unpredictable problems in another module.

Because of these problems, there is growing interest in a related discipline known as “structured design.” Based on work carried out by Larry Constantine in 1965-68, structured design concentrates on the relationships between modules; it introduces the concepts of “coupling” and “cohesiveness” to help quantify the “goodness” of a design. When carried out properly, a program implemented with structured design is substantially easier to maintain than a program implemented with either a “random-design” approach or even a “structured programming” approach. This is particularly interesting since structured design does not impose any rigid disciplines on the code within a module. Thus, structured design combined with structured programming should be an almost unbeatable combination.

Unfortunately, structured design is not yet a precise science. For example, coupling is easily understood by programmers; however the strategy of minimizing intermodule coupling often contradicts the programmer’s instincts and habits: for reasons of efficiency or convenience, he is often unwilling to relinquish his use of global variables, COMMON data areas, and common control blocks. Cohesiveness is similarly misunderstood: the programmer learns that a module called EDIT-AND-UPDATE has “sequential” cohesiveness, and that EDIT-ALL-TRANSACTIONS has “logical” cohesiveness—and that neither module is considered “good” when compared to the ideal module possessing “functional” cohesiveness. Since the programmer often doesn’t understand the various levels of cohesiveness anyway, and since it may contradict design strategies he has employed for years, he may ignore the suggestions of structured design altogether.

To summarize, programming has improved since the introduction of structured programming and structured design. However, it is unreasonable to think that programming has suddenly become a rigorous science—and it is foolish to think that the application of structured programming is guaranteed to lead to “perfect” computer systems. Our programmers still need more education in the “art” of structured programming; they need more experience in the application of structured programming and structured design to real-world systems; and finally, they need “rules” of structured design that are as rigid as the “no-GO-TO” rule is in the realm of coding.