This was the actual talk on Data Reliability given at the conference.

Here is an example of a flowchart. And here I must apologize for being in an ivory tower, and giving you such a small example. Please believe that my remarks will apply also to very much larger examples which are sometimes found to occupy the walls of a programming manager's office.

We all know that a real life flowchart stretches from wall to wall, indeed from floor to ceiling. Allow me to postulate that the following principles apply to larger cases, in fact with ever greater severity.
1. We cannot conveniently input a flowchart to a computer.
2. Nor can we conveniently obtain it as output.
3. It does not decompose readily into parts.
   It is intellectually unmanageable.
4. A local fault may have global consequences.
5. Dynamic structure change causes global consequences.
6. Non-local jumps are time-consuming.

Slide 2: And here is a data diagram, a picture of a supposed data structure. It suffers from all the same disadvantages. rep.

But an overwhelming
Too much paper - too many pages
A picture is worth a thousand words, but not when it stretches over a thousand pages.
and especially when the most interesting part of the picture is the words written above the boxes, that it contains.
An educated man or woman (is one who) learns to appreciate the conciseness and expressiveness of continuous prose.

It is only in the children's comics of our childhood that we knew every word must be surrounded by a balloon. But when we grow up, we learn to appreciate the power and conciseness and expressiveness of the written word. continuous prose.
Now I would like to survey with you the basic structuring methods for data, and point out the close analogy with structuring methods for programs. In each case I will show how the traditional voluminous picture can be replaced by a concise and expressive notation. The simplest and most important method is the direct or cartesian product, which corresponds closely to the concatenation of composition of statements in a programming language

On the left of the slide in black, an excerpt should remind you of a text from your favourite gospel on structured programming. On the right of the slide in blue, I always used to draw those boxes, one under the other, until I realised that for the purposes of reliable software it was better to draw them as a cascade.

On the right...

5. Conditional / union
6. Iteration / sequence
7. Recursion
7a. Recursion
The next data structuring tool, like it is recursion. Like recursion in program structuring, it is not often required; but when it is required, it is required rather badly. Unfortunately I had a bit of trouble drawing a picture of recursion [slide 7], which must of course, be a picture of a picture of a picture of a picture... On this slide I have tried my best to illustrate thousands answer to the saying of Confucius — even if you offer you the word that worth a thousand pictures.
But I fear that the controversy that has surrounded the avoidance or non-avoidance of jumps has been obscured by one vital fact. It is not by avoiding jumps that one obtains well-structured programs — indeed I fear that many jump-free programs have a lousy structure. The truth is that by writing well-structured programs we avoid the use of jumps — indeed I am glad to say that all wish to eventually even all desire to jump will wither away, and we attain a state of being.

The absence of jumps is the symptom, not the cause, of structured programming, not it is the outward and visible sign of an inward and spiritual grace.

So there remains the central question, what is the central issue in the structuring of programs, and how does it contribute to accuracy and reliability? I think it lies in what Dijkstra calls a "separation of concerns", the successful decision to embody one group of related concerns in a compact.

And the essence of this grace is the same as for data structuring, the conscious attempt to program in one place still the code arising from one programming decision, and to keep that code as separate as possible from that arising from other decisions.
And one of the most successful methods of separating our concerns is the conscious use of abstraction and its separation from the mass of highly relevant detail which is used to implement it. Slide 8 shows an example of one of the ways of an abstract program for printing all primes up to a million. On a machine on which the test of primeness and printing are built-in instructions, this would be a concrete program. Since such a machine does not exist, we must construct it by writing detailed programs to implement it. But at all costs we must separate keep these detailed concerns from the abstract program which they are designed to support, and use the abstract program as a guide and framework with which to organize the details. And if anyone is discontented with this small example, he may be assured that the advice I give will be even more useful for large programs, as he may not readily convince himself by replacing the constant million by a trillion or a quintillion before embarking on implementation.
Slide 9 shows an example of an abstract data structure.

Explain no details of representation...
In this talk, I have had occasion frequently to apologize for the simplicity of my examples. In conclusion, let us not offer no apology for the simplicity of the methods which I have described. I advocate, nor for the simplicity of the examples which I have used to illustrate, because I believe that my remarks apply with even greater force to them. As in my belief, I believe that a for the solution of top pro complex battery of elaborate features and facilities may be acceptable or even very attractive in the solution of simple problems that appear in the software manuals and sales literature of our leading purveyors of data base management systems. But for really large and complex problems, we should prefer to confine ourselves not only to the simplest methods. Simplicity is the unavoidable price which we must pay for reliability. Let us hope we do not find it too high a price to pay.
11. References and Pointers.

One remarkable feature of the structuring methods introduced here is that they make no mention of the reference or pointer, which are traditionally regarded as the prime means of structuring data. In this respect, references seem similar to go to statements, which have traditionally played a major role in computer programming, and which seem to be going rapidly out of fashion. In fact the analogy goes deeper. In the implementation of data structures use may be made of machine addresses, just as jumps are used in machine code to implement conditionals, and while loops and procedures. The major structuring disadvantage of the jump is that it creates new wide interfaces between distant parts of a program, which look as though they should be separate, and the slightest change to a program can propagate errors rapidly and uncontrollably along these interfaces. I suspect that the same is true of a reference, pointing from one part of a data structure to another distant part, which ought to be disjoint. And I expect that there will be yet another analogy - the recommendation to remove references from data structuring will meet as much controversy as that to remove go to's from programming; and perhaps even more so, because it runs counter to the still prevalent belief in integrated information systems, relational data bases, etc.; and suggests that it may be preferable to go back to earlier, simpler, techniques using separate files without cross references, and holding in one place all data relating to each single item of information.