EWD249-0

On useful structuring
by Edsger W. Dijkstra

The purpose of this minor contribution is to stress the urgency to make a conscious effort at exploiting "structure" as a useful thinking aid. Furthermore it gives some of the conclusions such an effort has led me to so far.

To start with, I take for granted that all of us acknowledge that the software failure is an undisputable fact: this recognition was why this conference on software engineering was organized, its acknowledgement is why we have accepted to participate. Hardware is rushing ahead of our programming ability and unless something drastic happens the situation will only get worse and worse. For: with more and more powerful machines becoming generally available society will be more ambitious in these applications and will be demanding more from the poor programmer who finds his tasks in the field of tension between the things to be done and the available tools. The scope of his task is just exploding.

At face value our main shortcoming is that we have let ourselves be lured into constructing elaborate mechanisms, the actual behaviour of which has grown far beyond our mental grasp or even worse: the misbehaviour of which is well beyond our control. As a professional community we play the Sorcerer's Apprentice over and over again.

Closer scrutiny reveals the current source of the trouble: viz. unstructured multitude and bigness, insufficiently organised complexity with its bastards such as Chaos, Unreliability, Unadaptability and the like. To regain control over what we are doing and what we are making constitutes for me the main challenge of software engineering.

Now, closer to the problem at hand. It would be helpful if all of us recognised that although the programmer only makes programs, the true subject matter of his trade are the possible computations evoked by them. In actual fact: the computation is the happening that has to effectuate the desired effect or in other words, when a programmer claims that his program is correct, he actually makes a statement about the computations!

Trivial as this remark may seem I must state that it has had a profound influence on my thinking and my programming. Once I was really aware of my mind's task to bridge the conceptual gap between the static program and the dynamic computation, I have restricted myself to the most straightforward sequencing clauses, finding myself in general unable to cope with programs containing go to statements. I will return to sequencing control later on, at present we note that here is an element of structure greatly assisting me in understandability of what we are mkking.

After long and, I must admit, rather painful struggles, I came to the following conclusion: doing something and knowing what you have done implies that your act is presented as a choice from what you could have done. In particular: making a program implies taking a whole class of programs into account: alternative programs for the same job or for related jobs, and programs on various levels of detail. In doing so I made the following observations which may inspire you: they seemed relevant in the light of my experience.

1. Different members of the program class can only share their correctness proof to the extent that they enjoy the same structure. In other words: comparing programs with the aim of comparing the corresponding computations is only a fruitful activity to the extent that they exhibit the same sequencing.

2. A flowchart need not be regarded as a vague sketch of what we are going to do, a sketch that only makes sense when the details have been filled in. On the contrary: at the appropriate level of abstraction it can be regarded as a pr gram existing in its own right.

3. It may very well be that certain aspects of the original problem statement are only reflected at the lower levels of greater detail: this just means that at the higher level one has a program solving a generalised problem.

4. I tend to think of the program consisting of a set of hierarchical lay rs, performing in steps the transition from what we have got into what we should like to have. The right of existence of these separate layers is that in each layer an independent abstraction is implemented: an identified choice is condensed in its coding. One of the trickiest kind of alternatives to compare turned out to be analogous to the design decision whether something shall be done by software or by a machine instruction. This observation is, I think, encouraging.

Finally, to ride another little pet horse of mine: experience has given me a strong indication that provided the software is properly structured its correctness can be claimed much more convincingly by a convincing proof of its correctness than can ever be achieved by the all too common procedure of testing and debugging. I know that the truth of this statement is doubted by many, but always by those who did not try to apply the method.