A position paper on Software Reliability.

The term "software reliability" does not occur in my active vocabulary --by which I mean that I almost never use it-- , although I think that it occurs in my passive vocabulary --by which I mean that I believe to know what people mean by it when they use it-- . The only conclusion that I can draw from this observation, is that "software reliability" as I understand it is in my opinion not a very fruitful notion.

I call a tool "reliable" when it is safe to use by virtue of the fact that, when used, it acts as intended, or, more precisely, reacts upon our inputs as intended. With this meaning of "reliability" I assume all of us to agree that "reliability" is a great, even indispensable virtue of all tools worth building. Consequently, "software reliability" would certainly be a notion, important enough to devote a panel discussion to.

But is the notion fruitful? We can try to decide that by dissecting it, by analysing what we can conclude when a tool is not reliable.

When we try to do that, we come to the conclusion that there are two vastly different kinds of tools. On the one hand we have tools like a hammer or even a bicycle, tools that we learn to use without explicit knowledge of their properties: being able to solve the differential equations of motion is not a prerequisite for the ability to use a hammer or to ride a bicycle. We learn to use those tools by the experience of using them. At the other end of the spectrum we have tools like mathematical theorems. Although their virtuosic use again requires experience in using them, this time "just experience" is insufficient: the safe usage of a theorem requires explicit knowledge of it, we need to know precisely under which conditions we are allowed to draw which conclusion, for that is all the theorem is about!

The tools at both ends of the spectrum are so radically different that there is little point in trying to abstract from that difference. Hence we have to make up our mind: is a piece of software more like a hammer or more like a mathematical theorem? For various reasons I am convinced that it is more appropriate to consider a piece of software, in its capacity of a tool,

like a mathematical theorem than like a hammer.

Over the intended range of its applications the hammer's reactions are a continuous function of its inputs; besides that, in his choice of inputs the user of a hammer has very few degrees of freedom. These two circumstances make it possible to learn subconsciously, i.e. from experience, how to use a hammer.

In the case of a typical program, where the input typically consists of character sequences, it is exceptional when the intended reaction is anything like a continuous function of the inputs, and the tremendous processing power of machines has increased the number of degrees of freedom in the possible inputs in the typical application considerably. As a consequence I don't regard the kind of tool as embodied by a computer program as one that, like a hammer or a bicycle, one can learn to use by the experience of using it. A computer program, considered as a tool, is a very different kind of tool indeed!

A computer program is a tool that one can use by virtue of the knowledge of its explicitly stated properties. Those stated properties are known as "its functional specification" or "its specification" for short. The specification specifies, in one way or another, what the execution of the program achieves, without fixing how this desired net effect is being achieved. In this sense the specifications have been described as " logical firewall between user and implementor": it is a contract between the program composer and the program user. The program user is in his right as long as his use of the program is justified by its specifications, the program designer is not to blame as long as correct executions of his program meet the specifications.

This indispensable division of responsibilities, however, reveals why "software reliability" is too crude a term to be of much usefulness. For its reveals two very different ways in which a program can manifest itself as an unsafe tool to use.

Firstly, it may be that the specifications are to blame. The specifications may be poorly stated --in which case the program user can be written down as a fool, for no one should build his application on quicksand-- or the specifications, although quite clear and unambiguous. describe a tool that is either clumsy to

use in general, or just hardly adequate for his task --in the second case the user is probably regarded best as a victim of circumstances-- .

Secondly, the specifications may be perfect, but even correct executions of the program may fail to meet them --and in this case certainly the program designer has to be blamed-- .

(In the case of poorly stated specifications, both program designer and program user share the blame for any disastrous application: both of them should have realized the ambiguity of the specifications.)

We can carry the analogy between program user and program designer one stage further: in the same way as in which the specifications allow the user to rely upon the program regardless of how the executions achieve the specified results, the specifications allow the program designer to design a program meeting them, regardless of how , that is in which interpretation and in which environment, his program is going to be used. But here the symmetry between program user and program designer ends dramatically.

As soon as in the specifications the inputs and the outputs are treated as values that are divested from the interpretation given to them in the intended environment of program usage, the specifications are indeed "a logical firewall", and the question whether or not the program meets the specifications can, in principle at least, be settled by mathematical means. The rigorous separation of responsiblities isolated for the program designer a task that is within the realm of applicability of scientific methods.

At the other side of the interface provided by the specifications, the program's user can never justify the program's usage any better than he understands the environment in which he employs the program. As the vast majority of those environments is miles away from being formally defined, the question whether or not a program has been used adequately or not is nearly always a question that is too vague to be amenable to scientific treatment.

And now we see, why "software reliability" is a fruitless notion. Thanks to the existence of the interface as should be provided by the functional

specification it covers two completely different questions: the formalized question whether a program is correct, i.e. whether it meets its specifications, and the unformalized question whether a tool meeting those specifications is in such-and-such unformalized and ill-understood environment a pleasant tool to use. Correctness is a scientific issue, pleasantness is a non-scientific one, and its therefore confusing to try to deal with both of them in a single sweep. And that is why the term "software reliability" has been banned from my active vocabulary.

Finally, for those who haven't observed it. I would like to draw attention to the fact that I have refused to call the one issue more important than the other: I have refused to do so because I wouldn't know how to justify such a judgement. I state this, because at gatherings like this I have often seemed to observe the general feeling that, the scientific issue now being well-isolated, the non-scientific issue now remains as the key problem; I would like to point out that such a general feeling is no more than a reflection of the circumstance that at such meetings --like nearly everywhere, for that matter-- the truely scientificly inclined are a minority.

Scientists largely prefer to confine their attention to the scientific issue, and as long as they don't ignore the existence of the non-scientific issues, I think that that constraint is correct. The scientific issue may have been well-isolated, we are far from having solved all its problems, and the scientist's attention is primarily required where he can contribute more than anybody else. As far as the non-scientific issue of pleasantness is concerned, there is little reason to assume that the scientist is much better equipped to contribute than others. As furthermore no scientific fruits are to be expected from dealing with fundamentally non-scientific issues, the scientist is justified in experiencing dealing with the non-scientific issue not only as a neglect of duty, but even as a waste of time.

P.S. The reader is mistaken if he thinks that he can send me a copy of Imre Lakatos's "Proofs and Refutations" for my education.

Plataanstraat 5
5671 AL NUENEN

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow