From: *Nanotechnology Perceptions*, vol. 13, July 2017

Obsolete assumptions

Geoffrey Sampson

In 2017, many Britons are wondering why information technology is serving our society so badly. It is not very many years since the collapse of "Connecting for Health", the world's largest-ever non-military software project, which spent some ten billion pounds of public money over almost a decade trying – and failing – to computerize the National Health Service.[1] This summer, we have seen that service brought to its knees temporarily, with doctors in many parts of the country unable to access their patients' records, surgical operations cancelled, and so forth, by a "ransomware" program called WannaCry, which attacked the less ambitious computer systems the Health Service has been using since Connecting for Health was abandoned (as well as attacking many other kinds of system in other countries) – the attack was only halted through an almost chance intervention by an amateur. And, just a few days later, an unexplained I.T. collapse at British Airways left thousands of passengers stranded, often separated from their luggage, at a holiday weekend.

Yet information technology no longer has the excuse of being a novel and untried technology. Computers have been in routine use throughout my adult life, and I am now in my seventies. Why do computers seem so uniquely liable to catastrophic breakdowns? We do not find that, one day, the motors of half the nation's cars or railway locomotives refuse to start, or half the nation's bridges fall down. Are software engineers an unusually irresponsible bunch of employees, who take advantage of the fact that their expertise is opaque to laymen in order to scamp their work and give themselves easy lives at their employers' or the public's expense? And if so, what do we do about it? The *Economist*, normally a sober and well-informed commentator on the commercial scene, points out that "Software-makers routinely disclaim liability for defects in their products. Changing that ... would encourage software firms to try harder."[2] By implication it suggests that the law should be

1 On this episode, see Sampson, "Whistleblowing for health", *Journal of Biological Physics and Chemistry*, vol. 12, 2012, pp. 37–43; online at <www.grsampson.net/CWhist4Health.html>.
2 "The worm that turned", *The Economist* 20 May 2017, pp. 14–15.

changed so as to force software houses to accept liability for bugs. Yet, in other contexts, the *Economist* regularly sees commercial competition as a more powerful force than government regulation for raising product standards. There is no lack of competition among the producers of computer software.

As I see it, the root of the problem is that information technology has destroyed the validity of various quite general, abstract assumptions which have been taken for granted over centuries, and society has not yet grasped how radically some of its established assumptions need to be changed. It is not that software engineers bring an irresponsible approach to their work. It might be more appropriate to call the leaders of society irresponsible in fostering unquestioning enthusiasm for applying I.T. to an increasing range of tasks – except that the leaders of society have no clearer idea than most other members of the population of how misleading old assumptions have become, so perhaps they too cannot be accused of irresponsibility.

Thinking about the computing disasters I have listed reminds me of the world's first fatal railway accident (or the first, at least, to make it into the history books). It occurred in September 1830 at the grand opening of the Liverpool and Manchester Railway, the first modern railway (in the sense that it made no use of horses but relied entirely on mechanical motive power: steam engines). The victim was William Huskisson, Member of Parliament for Liverpool and a cabinet minister until he resigned on a point of principle a couple of years earlier. He was standing on the lines talking to an even more distinguished guest, the Duke of Wellington, who was sitting in a stationary carriage, when a train came through on the track Huskisson was standing on, and killed him.

From our modern standpoint, the position Huskisson had placed himself in seems so obviously vulnerable that it is tempting to suppose he was a foolishly pompous fellow who assumed that the world would wait on the convenience of a Very Important Person such as himself. But I think that was not so. From what I have read, Huskisson was an unusually likeable man for a politician. More probably, as I see it, Huskisson had not got his mind round the idea that he was now living in a world containing heavy machinery which simply could not stop in a short distance, whether for ex-cabinet minister or for peasant. Huskisson will have known about runaway horses, but what killed him was not a runaway train. It was one hauled by a locomotive performing according to spec. But the locomotive weighed over four tons, and was hauling a train of many more tons on smooth steel rails, travelling faster than human beings had ever travelled before. Today, we are drilled from early childhood to recognize the danger of such situations. Our parents train us to look both ways before crossing the road; railways go to great lengths to keep passengers off the tracks and to warn against going on them. Huskisson perhaps had no concept of the risks associated with massive objects moving at high speed. Or if he was aware intellectually, he lacked the “gut feeling” that would have kept him safe without conscious reasoning.

Computing technology has brought into being phenomena which routinely behave in ways that are fully as counter-intuitive, for most people today, as I suggest railway trains may have been for Huskisson. Everyone understands that developing new I.T. systems requires expertise which only a minority can, or need, acquire. But they do not understand that merely living in a world containing computers requires anyone to get his or her mind round seemingly paradoxical new ways of thinking, just as living in a world containing cars requires everyone instinctively to “look right, look left, look right again” when coming to a kerb.

One of these counter-intuitive phenomena was persuasively described by Fred Brooks, in one of the few books ever written about the new technology to deserve the accolade “classic”: *The Mythical Man-Month*.[3] In most areas of life we take for granted that with tasks which require significant time and effort, there will be a rough trade-off between manpower and time taken. Primary-school arithmetic problems often take forms like “If it takes six men four days to dig a ditch, how long will it take two men?” What Brooks showed was that, if the task is to produce software to execute some reasonably complex set of functions, adding extra suitably-qualified workers does not merely fail to reduce the time needed: it increases the time, for good reasons having to do with the increased intensity of co-ordination needed between pairs of workers. As Brooks put it, “Adding manpower to a late software project makes it later”. The “man–month” is not a unit relevant for measuring the size of a software-development project, in the same way as electrical potential is not measurable in pounds per square inch.

In my experience, this is not an idea which the managerial types who organize the work of I.T. professionals are usually willing to entertain. Among managers it is axiomatic that task sizes can be estimated, and hence costed, in terms of man–months (assuming that the workers in question have the right skills, of course). Revising fundamental assumptions is a painful intellectual activity. Many managers would prefer to preside over projects that fail – failure can be blamed on subordinates – than

3 F.P. Brooks, *The Mythical Man–Month: essays on software engineering*, Addison-Wesley, 1975.

mentally to confront the possibility that man–months might be an inappropriate unit of measurement.

As another example, consider how software violates our assumption that similar causes produce similar results. In designing a system of physical machinery, small changes tend on the whole to yield minor differences in performance. A slight adjustment to a gear ratio, or to an operating pressure or temperature, may make a machine a little less efficient (or a little more efficient), but it probably will not lead to unrecognizable behaviour. There will be cases where making a small change leads to a tipping point so that performance is transformed, but that is not the usual situation. If one envisages the task of designing a machine as a search through a logical solution space of possibilities, then most of the time neighbouring points in the solution space will be similar in the extent to which they satisfy the task requirements, so that one can move from one possible solution to another with a sense that one is "getting warmer" even if a thoroughly adequate solution has not yet been achieved. We feel that it is natural for any solution space to be like this.

Contrast that with a software system, which may comprise many thousands, or even millions, of code lines, within which the tiniest change – say, deleting a full stop, or changing it to a comma – will quite commonly either cause the system to collapse and produce no output, or make it behave in some way entirely different from what we want. Instead of the solution space (the range of syntactically well-formed programs, in the software case) being characterized mainly by gentle gradients of relative adequacy, with just occasional cliff edges, it is almost all cliffs, so there is no "getting warm".

In consequence, even if we know that a thoroughly adequate software solution for a given task is in principle possible, there is very little reason to assume that we can locate it. That conflicts with the expectations we have formed from generations of experience with physical systems. If it is clear that some task is capable of being executed by a physical machine, then we expect that a suitably qualified engineer should be able to design a machine that carries out the task to some degree of adequacy. Probably the prototype machine will be relatively inefficient, and perhaps it will omit some minor aspects of the task specs, but once a prototype exists it can normally be incrementally improved. In the software domain, on the other hand, it is normal to be able to say "Nothing about this task is too ill-defined or otherwise unsuitable for execution by computer software – there must be some sequence of code lines which would do the job; and our team have the right qualifications for devising software code; yet despite protracted endeavours we have not succeeded in producing software that is even approximately adequate”.

For people without direct personal experience of computer technology, that situation is very hard to understand. It may be easier to imagine that the team of coders have somehow failed to apply themselves diligently to their work – they have goofed off. One can only really appreciate the true situation if one has had experience of writing code oneself, producing small programs which, before they are run, one feels quite certain will succeed in carrying out some simple task, and then finding, repeatedly, that they somewhere contain unforeseen and hard-to-identify bugs which cause the programs to behave in quite unexpected and undesired ways. Most inhabitants of advanced 21st-century societies, even if they make daily use of computers, lack this experience.

As for the non-applicability of the man–month measure, this I think is only fully appreciated by those who have tried to manage multi-person software projects, a kind of experience which is obviously far less widespread even than the experience of developing code as a solo effort.

It is not reasonable to expect software producers to accept liability for defects in their systems; liability implies that with care one can ensure lack of defects, and it is not given to human beings to ensure bug-free code. Making producers “try harder”, in the *Economist’s* words, will not alter that fact. But unless you have tried your hand at programming, you are unlikely to see why this should be a problem.

How would things be different, if there were a wider appreciation of how I.T. has undermined longstanding assumptions?

Well, it might lead us to ask ourselves how much we mind the kinds of disasters that I.T. produces. At present, the public response to WannaCry, or to the British Airways fiasco, is anger: someone ought to have ensured that health service software systems were properly defended against ransomware, and that British Airways software was not liable to whatever kind of breakdown it was that occurred in May 2017. All we need to say about these disasters themselves is that they were thoroughly undesirable and should not have happened.

But if one has a clearer appreciation of computing realities, anger is beside the point. Of course we would much prefer these things not to happen, but if we use information technology then, like it or not, things like these *will* periodically happen. Are several days of cancelled surgery, or of cancelled flights and lost luggage, prices worth paying for the advantages which stem from I.T. during the long periods when it works as intended? Are these events the worst imaginable examples of computer disaster – surely not, and if not then how far are the worst-case scenarios still prices worth paying?

We have not begun to take questions like these seriously, because most of us still assume that if programmers did their jobs properly, the prices would not need to be paid. We recognize the advantages of information technology, but we assume away its drawbacks. A software system turns out to contain flaws which produce some awful outcome, and if we ask "Was there no way the software could have avoided that?" the answer is "Sure, with a few modifications to the code it would never have happened." Our instinctive reaction is "Then for Heaven's sake in future get the code right in the first place." If we are not ourselves computing experts, we do not appreciate how unreasonable that demand is.

Presumably society will eventually come to a more mature understanding of the inherent limitations of the technology. When it does, will people still want to accept the bargain which gives us the advantages of the new technology but at the cost of periodic disasters? And if society concludes that in some cases the costs outweigh the benefits, will it be capable of unwinding the relevant aspects of computerization? These, to my mind, are genuinely open questions.