You’re right about the cure: don’t do that

Harold Thimbleby

A major factor of system usability is whether the system works at all. This paper discusses bugs and the social environment that allows and encourages them to exist. Many bugs are known about and accepted when software is released to users. They could have been corrected if there had been any motivation to do so. Although individual programmers are often responsible for bugs, various forces within the computing industry, including mistrust of users, drive software manufacturers to strategies that exacerbate the problem. Such methods as software manufacturers adopt ‘in defence’ not only work against users but also undermine scientific work, which in turn retards the advancement of HCI generally.

Keywords: human–computer interaction, bugs, software reliability

For a long time I have known about bugs and their occasionally disastrous effects. Bugs affect practically every computer system, from hand-held calculators up to mission control computers for space exploration. Some bugs have caused death; maybe in the not too distant future, a bug will start a major war or cause some other horrendous catastrophe. Yet other bugs seem trivial, and are perhaps all too readily dismissed.

One day, which I describe in the next section below, I had my first direct experience of a major bug. This made me feel angry and powerless. I have written this paper as a direct result. My anger stems from the fact that most bugs are completely avoidable, but users live with them, and manufacturers are — it seems — content to keep things that way.

The paper has four strands:

- A brief description of my experience using an Apple Macintosh. What I describe is accurate, but the result of a specific combination of circumstances: my experience should not be taken as a comment about Macintoshes or Macintosh software generally. In fact, I believe that by using a Macintosh I am much better off than had I been using some other sort of system! Alan Kay once said that the Macintosh was the first computer worth criticising: my criticism should be taken in that vein — after all, I
would not have been using the machinery and software I was unless, on the whole, it did a very good job! An equivalent combination of circumstances, with equally disastrous consequences — or worse — could affect anyone, at any time, and on any machine: that is the point I am driving at, and which my one-off experience serves to illustrate.

- Some bugs are clearly accidental, and reputable companies would surely take every step to rectify them. Yet standards within the industry are low, and we probably suffer more bugs than we need as a result. ‘Limitation’ is often a euphemism for bug, and both users and manufacturers are prepared to put up with unnecessary limitations. I give a selection of examples below where manuals warn the user not to do one thing, and then suggest remedies. I am often left wondering why the computer cannot fix a problem for the user if the manual writers knew how to do it! The complete denial of responsibility in most software warranties also speaks for the low quality standards that are widely accepted.

- The next part of the paper builds up to the observation that computers get better treatment than humans: when computers go wrong, it is the designer’s fault; when users ‘go wrong’ it is their fault. If you are a user and you experience a bug, you are told the cure is simply not to do it again (hence the title of the paper); if you are a computer and you experience a bug, then the cure is a fixed program, not an admonishment.

- Finally, I look towards the future. Software is a recent phenomenon, and perhaps we, as a society, both as users and as manufacturers, are too quickly forcing software products into conventional moulds. Certainly current problems, such as ‘pirating’, increase manufacturing costs which are passed on to the consumer and in turn increase the pressure to pirate. There may be alternatives. Conventional research, too, will improve standards (we hope) and perhaps eliminate some of the immediate problems with bugs.

Personal experiences

On Friday 29 May 1989 I sat down at my Macintosh to write a letter. I started to use MacWrite (a standard word processor); it crashed and took with it 64Mbyte out of my 100Mbyte disc. The program ‘Disk First Aid’, designed to rescue users from such disasters could not cope and suggested the disc was reinitialised. My last backup was on the previous Monday (it was a partial backup), though I had fortunately saved a few files on individual floppies. It took me Saturday to get back to where I was, more or less. Plus the rest of the weekend to recover emotionally!

Why is software so bug-ridden? Why is MacWrite so complex that its designers overlook major flaws? Why is the Macintosh disc structure so lacking in redundancy that First Aid has nothing left to go on? Why is the Macintosh architecture so unprotected?

I first tried a Macintosh in September 1988. I had had high expectations as the Macintosh represents the best available integrated user interface. The interface evolved from Xerox’s pioneering work, and was refined through Xerox’s Star...
and Apple's Lisa. I had bought a Macintosh Plus, 1 Mbyte RAM with internal floppy drive. This is the standard, basic system.

My Macintosh came with system software and MacWrite. Claris Corporation, who wrote MacWrite, also enclosed a demonstration disc, with demonstration software for MacDraw, MacProject and so on. I tried the demonstration software. I had to swap discs about 20 times before anything interesting started to happen, and you get more disc swaps later. I decided that if this was demonstration software, then the real stuff probably was not worth using. I decided — well, I was practically forced — to buy a hard disc, so that at least the disc swapping would go on behind the scenes. It then took me and my colleague Val Jones — that is, two people with PhDs in computer science — several hours to install the hard disc and get it working. And this is supposed to be an easy system.

Being interested in HCI, I decided to start writing up my experiences. I wrote for half an hour or so, and then tried to save my work to disc so I could go to lunch. MacWrite would not save it. What had been happening was that I had unwittingly been running a demonstration version of MacWrite (it came amongst all the other demonstration programs). The demonstration program is identical in every way except that it does not save work to disc. Why, then, does Claris distribute a working MacWrite with a demonstration program that stops it working? Did nobody check?

Later, after my MacIIx had arrived, I bought Adobe Illustrator '88. (I now know that Illustrator is not quite the sort of drawing program I wanted, but I could not tell what it did not do from its impressive reviews — you have to try programs to find out what they do and do not do.) Illustrator came on a copy-protected master disc, presumably to ensure that I could not make cheap copies of it and give it away free to my friends. I never managed to install Illustrator; each attempt corrupted the master disc. It looks very much like copy protection is the manufacturers protecting themselves, and if their copy protection scheme has bugs in it, well, tough for the user. Adobe had already got my money! (Eventually Adobe sent me a non-copy-protected disc which solved the installation problems.)

Let us now turn from these few, quite typical, personal experiences to reflect on the situation they illustrate. You could say that a few bugs are inevitable given the complexity that users demand — or is forced on them by manufacturers providing more features in their software than their competitors. All of the bugs I have described so far could charitably be dismissed as accidents, of which I was an unfortunate victim. I do not believe they were accidents. They are not the only examples that could be mentioned from recent experience. I believe that the entire industry is set up to exploit users, and that what I have described is typical and symptomatic of a deep malaise. I believe the industry's dismissive attitude to bugs is the most serious problem facing computing today.

'Bugs may appear to be a purely technical issue, of concern only to software engineers, but not to human–computer interaction specialists. The social matrix that sanctions bugs is however a relevant concern, but ... imagine that insights from HCI lead to improvements in user productivity, error rates and job
satisfaction and so on. That just means that the inevitable bug will destroy more work and be all the more depressing for the user. HCI is mostly concerned with improvements, second order effects, effects of marginal difference to the user; bugs, then, are clearly a core HCI effect (Thimbleby, 1989a).

Problem scope

If you buy a car or a washing machine, and it does not work, then you can take it back and get a refund or a replacement. Computer programs are different, whether cheap, costing a pound, or expensive, costing several thousand pounds. Computer programs at any price often fail and the user has no redress. This is an issue that affects all computer users, from home enthusiasts to large companies.

Computer programs come in sealed plastic bags. On the bag will be a legal warning: once the bag is opened, then you are deemed to have accepted the software. Once you open the package, you cannot return the program, even if it is defective or totally inappropriate for the uses that you bought it for.

Manufacturers of programs will argue that they also supply documentation, instructions and manuals. You can read that before breaking the seal. But is this reasonable? Do you read the manual of a car before driving it? The driver’s manual actually gives you very little idea of how the car works or what it will feel like to drive. It is the same with programs: program manuals are very often not the place to start if you want to know what the program does. Furthermore, the program manual will have a legal statement that the program is supplied ‘as is’. You have bought what you have got, that is. Typically, the manufacturer will expressly disclaim, in suitable legal jargon, anything and everything that the manual says. The risks are all yours. The manufacturers want to protect themselves from mistakes in the manual or the program, so they say that the manual does not constitute a warranty or implication of the way the program works (I quote a typical disclaimer below), and opening the program to find out what it does implies an acceptance of it.

With computer programs, then, manufacturers put their customers in a ‘Catch-22 situation’. The program cannot be tried until you have irrevocably broken a seal, and until you break that seal you have no idea what the program does. Once you break the seal, you cannot return the program as unfit.

Actually the situation with computer programs is even worse than I have so far made out. Here are four more facts:

- Computer programs are terribly complex. It is not unusual for manuals to run to a thousand dense pages. Cars are trivial in comparison. It is therefore quite unreasonable to expect a purchaser to understand a program merely from the manual without trying the program out.
- There are very few accepted standards: indeed there is a tremendous variety in system design. Cars may differ in superficial qualities, like their colour and acceleration, but even the relatively narrow niche of word processing is filled with programs that are incompatible in almost every detail. Thus it is not possible to assess a program from a simple description, it usually has to be experienced in order to judge how it works.
• Computer programs are secret. For commercial reasons, manufacturers are reluctant to disclose the code of their systems. If it was released, then competitors would benefit at no cost from the original manufacturer’s development investment. In contrast, conventional industries such as car manufacture require considerable investment to tool up for a particular production; whereas copying a computer program is no harder than copying files and having a suitable compiler available, which in turn might easily be bootlegged. Computer programs are also ‘secret’ in the sense that many are so complex that they are effectively unknowable.

• Computer programs have bugs. The programs do not do the right thing in the right way. They go wrong. In particular there will be bugs in the manuals, in the sense that the program does not do what its manual says it does, or it does things the manual says it does not. It may even do completely bizarre things. It is something rather like having a car that loses a wheel only when overtaken on the inside by a 1978 car doing 59.3 mph. Of course, the manual does not tell you that. The MacWrite bug I mentioned at the start of this paper was more like the wheel dropping off without warning, just driving normally. Presumably because of a major design flaw.

Some examples

Although it would be easy to present many sad tales of bugs and their effects, we could still charitably believe them to be accidental. The manufacturers did their best, and, unfortunately, there were some bugs left. Instead of a list of such bugs, then, here is a brief list of bugs admitted by manufacturers in user manuals. We cannot be so charitably-minded with these examples. The manufacturers have released software with bugs they certainly knew about, and in some cases even knew how to fix, but did not.

The manual for MacWrite writes, ‘You cannot cut more than 100 paragraphs at a time’ (Claris, 1988, p 116)*. That statement in the manual means the designers must have known about this bug before the program was sold; they must have been content to leave it in (and it isn’t the only one they left in). Any user who wanted to delete around 100 paragraphs could easily lose count: if they were deleting 99 — OK; if they were deleting 101 — tough! Why can this word processor not have a bit of program that converts an attempt to delete, say, 170 paragraphs into two steps of 100 and 70 paragraphs each of which it could delete? After all, that is what it is expecting the user to do for it. What are computers for?

The Microsoft QuickC manual warns in several places, ‘Failure to do this [call a certain function] may cause the machine to “lock up” when another program is run’ (Microsoft, 1987, e.g. p 81).

Whitechapel’s MG1 manual warns that halting the computer, which for instance is the purpose of the standard off command, when the system is doing a disc update (which it does once every 30 s) may cause the discs to be

*Is this a bug or a limitation? All bugs are limitations, but only the expected, nondisastrous limitations are not bugs.
corrupted. The manual suggests a fix (involving modifying the operating system code, not that a user can do it).

Apple's Smalltalk-80 (v0.4) manual warns: 'Not all the Toolbox calls work properly . . . Workaround: You can encode Toolbox calls as primitive 160 as described in the 0.3 manual.' Does everybody with version 0.4 have access to the 0.3 manual? Why did Apple not use their own workaround?

The \LaTeX \ manual (Lamport, 1986) has many warnings. Here are a few examples: 'Some \LaTeX \ commands are fragile and can break when they appear in an argument . . . On the rare occasions when you have to put a fragile command in a section title, you simply protect it with a protect command.' (Why doesn't \LaTeX \ protect it for the user?) Later, 'These equivalent forms . . . The \langle and \rangle \ are fragile, \$ \ is robust.' (So they are not equivalent forms.) Or again, 'The rules describing exactly when this problem [getting certain error messages] will occur are complicated, but the solution is simple: use a command of the form \textbackslash \{size\} \{style\} . . .' So, here we are told that \LaTeX \ recognises a certain sort of tricky error, for which the cure is known (and, they say, simple), but the user still has to fix the problem.

The manual for the Saitek Simultano chess computer warns, 'However, the first eight slots in the pawn bank are reserved for simultaneous games (see section 8.3) and the last slot (h8) of the pawn bank is also reserved for the computer, so do not store any games there.'

Such explicit comments are rather easy to find in almost every user manual. Of course it is good that the user is warned at all, but would it not have been better to have fixed the bug if it was known about?

Knuth writes (Knuth, 1989) (with my emphasis added),

>'Thus, I came to the conclusion that the designer of a new system must not only be the implementor and first large-scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt \TeX \ [a big system Knuth wrote] significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.

>But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments . . .

>As I said above, manual writing provides an ideal incentive for system improvements, because you discover and remove glitches that you can't justify in print.'

It would seem that many people do not take enough care over writing manuals, nor in trying to fix the 'glitches' that honest manual writing would force them to reveal.

'You're right about the cure: don't do that'

The following quote, taken from a computer magazine 'agony column' (Espinosa, 1988), is the source of this paper's title.

Thimbleby
Q. 'I have a different Command-Shift-4 problem [...]. The printer begins at the left-hand register mark when printing a screen dump (Caps Lock Command-Shift-4), but when printing a window dump it leaves a 1-inch margin [...]. I can, of course, often avoid this problem simply by narrowing the window an eighth of an inch — but only when there's enough unused space at the right. Is there any way of reducing or removing the 1-inch printing margin? [...]'  

A. 'Guess what? That's a bug. If a window is wider than 7½ inches, it'll wrap around when you print it with Caps Lock Command-Shift-4 [...]. You're right about the cure: don't do that.'

Not long ago Unix was an exciting but unreliable system. It had security loopholes; applications programs could crash the system; filestores could easily get corrupted. But the source code of the Unix kernel was available to academics. That meant that bugs were fixed, and it also meant that many people read the code and improved it in various ways. Bulletin boards are routinely used for sharing bug fixes around the world. The effect is that the community of programmers worked, and still work, together to improve Unix. Of course, there were risks: there are now many versions of Unix, all slightly different from each other. Nevertheless it is undoubtedly true that all of these versions represent improvements on the reliability and security of the original. Now compare the initially open approach of Unix with the standard closed approach of proprietary systems. (See below for more discussion of Unix.) Apple (or IBM or ...) keep their code as secret as possible, partly to stave off clone manufacturers. When an upgrade is announced, it is sold, and therefore has to be designed, for its new features. You do not get a positive market image by admitting you have fixed appalling bugs. So new versions of proprietary software come with new features — and new bugs.

There are marketing reasons to add features on each software release, but another powerful factor is the manufacturer's attempt to stay ahead of the software pirates who would otherwise be copying their software (Kocher, 1989). It is to a manufacturer's short-term advantage to make software as intricate as possible. The more features there are, the easier it may be to uphold a prosecution in the law courts when a competitor is found to have copied ideas. And we know, the more features, especially ones added for such reasons, the more bugs and the worse off the user. Richard Stallman (1985) has argued forcibly that our ideas of copyright, based on authorship, book and printing technology, are inappropriate for computer programs; enforcing conventional copyright, whether by legal or computing means, is damaging society at large. Third World countries generally have no copyright laws for this very reason.

Copyright does give some protection against cloning*, piracy† and industrial espionage (all of which could be addressed by different means), but it is of no direct advantage to users; in particular, sensible backup procedures infringe most copyright 'agreements'. Copyright hurts not just users, but also the originators of user interface ideas: creative developers that produce software

*Cloning: making counterfeit systems.
†Piracy: selling unlawful copies.
components for distributors are restricted in how they can reuse their own work.

Further discussion of copyright issues can be found in Samuelson (1989a and b) and Stern (1989); Brand’s chapter, ‘The Politics of Broadcast’ (Brand, 1988) is especially recommended. The point for this paper is that the present position on software copyright, although certainly complex and distracting, sanctions the legalistic expression of possessiveness, secrecy and gratuitous complexity (of which bugs are another manifestation) at the expense of the user. It is arguable that patents are anyway a more satisfactory mechanism; we will return to this point below.

To return to bugs; bugs in themselves are interesting, but more interesting is why manufacturers openly sell programs full of bugs and refuse to take any responsibility for them. In fact they actively disown bugs: they admit their manuals and programs may be wrong, but it is not their responsibility — see the ‘warranty’ (called a ‘disclaimer of warranty’) below which is taken verbatim from the Claris MacWrite user manual. I must emphasise that this warranty is not unusual in its provisions, and it is not the worst one I have seen; the intention if not the exact wording is standard for the industry.

‘A. Limited warranty on media.‡

B. Disclaimer of Warranty on Software.
The software is provided “as is” without warranty of any kind, and Claris expressly disclaims all implied warranties, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Claris does not warrant, guarantee, or make any representations regarding the use of results of the use of the software or any accompanying written materials of their correctness, accuracy, reliability, currentness, or otherwise. The entire risk as to the results and performance of the software and written materials is assumed by you. Claris does not warrant that the software will work correctly in your multiuser environment. If the software is defective, you, and not Claris or its dealers, distributors, agents or employees, assume the entire cost of all necessary servicing, repair or correction. Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

C. Complete Statement of Warranty.
The limited warranties provided in paragraphs A and B above are the only warranties of any kind that are made by Claris on this Claris product. No oral or written information or advice given by Claris, its dealers, distributors, agents, or employees shall create a warranty or in any way increase the scope of this warranty, and you may not rely on any such information or advice. This warranty provides you with specific legal rights. You may have other rights which vary from state to state.’

Of course, these are just legal hedges, protecting the manufacturer from the worst. In practice, a reputable manufacturer will probably make some concessions to user’s problems, but that is not quite what they say. You would

‡The media warranty offers to replace defective disks within ninety days.
indeed struggle hard to find a similar useless ‘warranty’ for any other sort of product. It is interesting to contrast, say, the warranties of conventional products (like glue, washing machines or cameras — the latter two often containing complex computer programs) with word processors. To take the extreme example: a can of glue may have a warranty that warns that the manufacturers can take no responsibility because the conditions of the glue’s use are beyond their control. This situation is true also of word processors. But the difference is that, with glue, the assumption is that the user is in control of the glue and its application. With any computer program the user certainly has no control whatsoever over its internal workings (the manufacturers anyway do their best to make them complex and secret), and the manufacturers pretend that they have no control either. Complex products such as washing machines and automatic cameras, even though they contain programs, have warranties that extend to all ‘normal use’ — so we are left wondering about the vacuous warranties of software, and what this says about the quality of word processors as compared with cameras.

Not only do your ‘specific legal rights’ provided by software warranties seem rather weak, but manufacturers go on to ask for money to ‘upgrade’, or to have a ‘servicing agreement’ — that is, to fix problems in programs that should never have been sold. It is a profitable business to trap users on an upgrade path. There are cases where manufacturers bring out ‘upgrades’ — that actually fix bugs — about every six months and charge over 20% of the original product cost! And that is often a limited period offer: after 30 days, you have to pay the full price. The very least they could do is distinguish between upgrades and bug fixes. It is generally reasonable to pay for an upgrade (getting more features not provided in the original version), but what actually happens is that as manufacturers fix bugs in the original, they also add new features. The user, then, cannot get bugs fixed without acquiring the upgrade, for which extra money is demanded.

An intriguing thought (due to Brand, 1988; see also Thimbleby, 1989b) is that users should subscribe to programs: though the detailed effects (would the first system have to be bad enough to encourage users to pay for a series of upgrades?) need to be worked out carefully. Certainly, interactive systems, such as drawing packages, hypertext etc., naturally lend themselves to subscription — not only for the inevitable bug fixes but also for getting clip art, new databases and so on.

Talking about bugs

Arabs have many words for camel; Eskimos have many words for snow; Aborigines have many words for hole; we have many words for vehicle (car, lorry, bus, sedan . . .) A language has lots of words for concepts that its speakers are interested in. Eskimos live in a land of snow, and they need to talk precisely about it.

Bugs cripple programs. Problems such as deadly embrace, stack overflow, subscript out of range, real underflow, division by zero, are all well-documented.
Bugs also cripple users' work. So how many words do we have for 'interface bugs'? One: bug. Everything else is circumlocution; that is, nobody is interested in interface bugs.

We have seen that manufacturers have hedged themselves around by legal protection so that they do not need to worry, but it is interesting to ask why users do not seem to worry either. Why is there not a revolt? Why do consumers not take mass action?

There are two reasons why consumers are complacent. Programs, even ones with bugs, are terribly useful and many people would find it hard to think of them being much improved. I can do things with a computer there is no way I could do without one. I put up with its quirks: yes, sometimes it goes wrong and wastes time, but mostly it is worth it. A more subtle reason is due to cognitive dissonance. People naturally try to reduce cognitive dissonance, so if you unavoidably do something silly often enough (causing dissonance), you make up a good reason for doing it (reducing the dissonance) — otherwise you might go mad. That is why bad programs are popular. If you regularly use a bad, nasty program, you have to make up a really good reason for doing so. Typically, you set yourself up as an expert. Now everyone looks to you for advice about this silly program. So you turn a potentially bad purchase into an opportunity to get esteem. Another view is the charm effect: the user invests a lot of faith in a program to solve their problems and does not want to be disillusioned. You have a 'magic potion' and you will use it however difficult it turns out to be. You are like Hans Christian Anderson’s Emperor after buying his expensive new clothes.

If the bad programs around were improved, had their bugs fixed, whole professions and career structures could be destroyed! Many people currently earn their living solving the essentially spurious problems that arise out of poorly designed software. It would have been better if the bugs and low standards of programming had never been accepted in the first place. Users are just making the best of a very bad situation: the manufacturers are to blame for the problem, and possibly for taking advantage of the trap they get advisory services into. Consumers become committed to bad systems because they are bad: it takes so long to understand a program that soon you do not want to waste that investment by getting a 'better' program. It is now quite common for computer systems to be sold on the strength of their faults — that is, that they have the same faults (and possibly a few more). It is called 'compatibility'.

Some 'pseudo' bugs are in programs to enforce certain policies. My bank takes several days to clear cheques; the library cannot tell me what books I have out unless I am prepared to pay — a consequence, apparently, of the Data Protection Act. And if you complain, you are told that 'the computer is programmed that way'. To make life easier for the management, to better exploit their computer users. They can only do this while everyone accepts 'bugs' — in this case, restrictive practices implemented by computer — as inevitable.

It is certainly very worrying that the computer industry is so complacent about bugs, when not using them as lame excuses for poor or restrictive design. It is terrifying that the prevailing social climate is not just complacent but
encourages bugs. It is something of an intellectual challenge to design a decent bug. And it is quite acceptable to do so. Both the law and social pressure are very weak in this area: crime is often fun, but usually there are various pressures to behave sociably. Not so in computing.

Brian Reffin Smith has suggested some more words for bug (Smith, 1986). A 'smug' is a bug you do not worry about; a 'hug' is a deadly embrace; a 'mug' is a silly bug; a 'thug' is a bug that is destructive. A 'plug' perhaps should be a bug deliberately installed to help sell a system. Even with these words and with a proper taxonomy, we still need to change the way bugs are talked about. 'Programs have bugs', we like to say. But we ought to say that programmers make bugs*; bugs don't come out of nowhere, they are caught from sloppy programmers. (Weinberg (1971) argues that this is a consequence of programmers reducing cognitive dissonance; his cure, egoless programming, has yet to catch on.)

We do in fact have some words for special sorts of deliberate 'bug'. These are virus, Trojan Horse, worm, bomb. Unlike the bugs we have been talking about, these things are deliberately, intentionally introduced by clever, malicious programmers. (See Reid, 1986; Stoll, 1988; Thomson, 1987 and Witten, 1987.) Some of them are very destructive, and destructive in ways outside the jurisdiction of the law. By giving you a virus I can destroy your company records, possibly your company as well. And not be held responsible, and not even feel responsible. It was an intellectual challenge to get the virus to work, and besides it is your fault that it destroyed your data, as you should backup every day.

You can get virus detection programs (some of which themselves carry devious infections). A virus detection program searches your work for recognised viruses and attempts to save you from their effects. Now guess how most copy protection schemes work? You've guessed it: by doing devious things indistinguishable from virus infection. So if you run a virus detector, you run the risk of it wrecking your expensive software. So what does the user do?

When a computer divides by zero, it is in trouble. There has been some mistake in its program. Now when a program has that sort of bug, the computer complains, usually it stops working and crashes. The programmer recognises this sort of problem — it is one he has been taught to try and avoid. The program will be fixed, because the computer cannot continue until it is.

Why is it a bug to divide by zero, but not to delete the user's data? Why are complaining users told to do things differently, but programs are redesigned when computers have problems? Why is it neat to write short programs for computers, but it is of no interest to give users an easy time?

Aren't these interesting observations? If a computer finds a bug, then the bug is serious. But if a human discovers a bug, then that is the human's fault. They should not have been doing that anyway! Humans are too flexible and adaptive,

*There may be bugs in the requirements specification, or even intrinsic in the application that the programmers are not responsible for. See Earnest (1989).
they can cope. 'You’re right about the cure: don’t do that’ was a perceptive comment!

In short, **computers get a better deal than humans, their users.** Computers impose strict standards with which manufacturers cannot fail to comply. Humans can be exploited, by large manufacturers hiding behind the law and by some programmers hiding behind their immaturity and lack of social conscience. Immaturity (or whatever it is) is not always the programmers’ fault; it is part of a more general attitude to work. One reason why programmers deliberately and so complacently leave bugs they know about in programs is that they are not taught well enough. Some of the problems start at school. Is it not a good idea to teach children computing at school? To write their own programs? Then they can get jobs in the information technology sector.

Actually, no. An analogy will help: Is it not a good idea to teach children building at school? To build their own houses? Then they can get craftsman jobs in the building trade. The problem is you teach children using building blocks and imagination; yet you build houses out of bricks, gas pipes, glass, girders and rules. Not many skills learnt about building blocks in the playroom help build real buildings. The trouble with school computing, indeed all our attitudes about computing, is that we think it is easy. It is very easy to write a program, and if it goes wrong it is very easy to fix it, almost so easy it’s trivial and is not worth the bother. That is what children learn at school, that is what university graduates think, but it is only true in the building block world of the playroom.

Real programs are far more complex. Cavalier attitudes to programming result in badly designed programs, built out of ill-conceived building blocks. Real programs are full of accidental bugs which arose because their designers simply did not understand the way large programs should be put together.

Our analogy between building and computing runs deeper still. If you want to build a house, you have to satisfy various planning authorities that the building is adequate, safe, and acceptable for the neighbourhood. If the living room has an unsupported wall running over it on the first floor, there must be some engineering calculations that the joists can support the load. Such calculations are quite technical and they are absolutely necessary.

Many programmers cannot do mathematics. Many programmers cannot do simple algebra; that might be why they chose (or were advised) to be programmers rather than engineers. Engineering has standards: if a car or a washing machine has an engineering fault, you can send it back. It is time that computing had similar standards, and it is time some of these standards became regulations. It is time that manufacturers stopped treating computers like toys, and stopped abusing the law, treating computer users like children.

**Future prospects**

'Undo' is a simple technical fix that certainly ameliorates bug-ridden software. Undo, however, is often partially implemented — bug-ridden itself — but in the future we can imagine that both programmers' and users' standards will improve: undo in particular should get better. More generally, interactive systems will acquire sound, reliable features, both in their implementation and
in their user interface to protect the user from bugs. The fact that there will be stricter standards for auditing will make user interface facilities like undo easier to implement. It is useful that a critical awareness of bugs improves user interfaces, as with Undo, but I think that our whole approach to software needs revising, not just camouflaging; otherwise improvements will be piecemeal and restricted to certain hardware platforms as manufacturers endeavour to protect their techniques from competition.

Not long ago computer systems were shared between large groups of users, and the users would be represented by a central computing service. Any complaints about software could be addressed to manufacturers through the more powerful representation of the service. Today most users work with personal computers. Even if they work in a department, they are required to pay separate fees for their software: essentially manufacturers want to relate to users on an individual, separated basis. As separate individuals, users are powerless against the relative might of unscrupulous manufacturers. (And we have seen that few manufacturers would think of their behaviour as unscrupulous: everybody does it.) Manufacturers forbid users to share software (when previously software had to be shared because the machine it ran on was shared): this can be seen as another part of their strategy to divide and conquer users.

Users must unite, they must be represented and they must become aware that such concerted action would improve computer systems to everyone’s benefit. We have all been brainwashed that bugs are inevitable. Bugs are only inevitable if we accept over-complex systems, secrecy, non-accountability. Professional bodies, such as the British Computer Society, the Institution of Electrical Engineers, the ACM, the Institute of Electrical and Electronic Engineers and their equivalents are well placed to initiate and coordinate such action. Sadly, these are all professional societies and are more likely to promote the short-term interests of the professions and not the users (Stern, 1989). When Wolfram Research, Inc., says it ‘does not warrant the software is free from all bugs and omissions’ it is being honest (and would that everyone else was) but surely there is a more professional stance? Nothing in this world is perfect, but most professionals manage to take significant, if not legal, responsibility for their actions. The IEE and IEEE could, no doubt, use their experience in electrical engineering practice to endorse professional responsibility in software.

Literate Programming (Knuth, 1984; Thimbleby, 1986a) is a technique for combining program code with its internal documentation (‘how it works,’ rather than ‘how to use it’ documentation). It makes documentation easier, perhaps more fun, and easier to keep consistent with program code. It has been used to publish two large, successful programs (Knuth, 1986a; Knuth, 1986b). Never before have such large programs been published in their entirety for public scrutiny. One wonders whether software manufacturers would dare publish their programs. Nevertheless, techniques such as literate programming could be used internally within companies. Programs would then get better peer assessment, and, incidentally be easier to maintain over a variety of machines (source code is more portable than machine code).

Literate programming, incidentally, may make a significant contribution to the copyright debate on software by making it more ‘tangible’ (Thimbleby,
Copyright, however, is not the only way for manufacturers to protect their work in law. Patenting is another option available for protecting commercial interests. Indeed, patents are a stronger form of legal protection than copyright: they can be infringed without copying or knowledge of the patent (Jakes and Yoches, 1989). It is possible to patent original ideas in software, but not if the manufacturer has released the software before it is fully developed, so in practice patent law is of little help to manufacturers if they release premature versions of their software for commercial advantage. Taking more care over software, which inevitably slows down the time to market, would make patenting more readily available as a protection for manufacturers.

In the future, the law may change. The law relating to computer crime in Britain is under review at present, and the effect of vested interests will no doubt influence the outcome. In some countries it is already illegal to gain unauthorised access to a computer or install viruses. Now imagine that the bug I mentioned at the start of this paper was instead caused by a virus (and we lived in the USA), then someone would have committed a serious crime. In fact, I can repeat the crash, so it is in fact not a virus but a bug. Yet the effect on me as a user is exactly the same in either case. It is surprising, then, that the law does not seem to relate to bugs as such. It may be that the law does not need to recognise software bugs as a new concept, but certainly until it does, users will have a very hard time gaining redress — or conversely, finding software that is reasonably free of bugs. The current state is rather like there being a law against murder but not against manslaughter — for the victim, the outcome is indistinguishable. Laws are certainly required so that a user can sue the manufacturers collectively, rather than first having to identify the specific piece of system that failed. What if it is a hardware fault? What if it is a problem of combinations of software?

The Free Software Foundation (Stallman, 1985) is one group that have reacted against the conventional commercial approach to software development that has been criticised in this paper. They develop software and distribute it in source form for a nominal distribution fee. They encourage users to report — and fix — bugs. Providing the software in source form for those who are interested is an effective way of fixing bugs before users encounter them when programs fail.

When Unix first became available it was provided in source form to academics for a modest licence fee, and indeed the attention it received greatly improved it and led to its present position of industry-wide acceptance. Its success has led, in turn, to more restrictive licencing policy from AT&T, and to a backlash from organisations such as the Free Software Foundation and the Open Software Foundation.

Free and open software attacks bugs in a number of ways. Making the source code freely available makes it available for scrutiny and improvement; making programs freely available creates a large user base, which in turn is more likely to detect bugs. The freedom for anyone to fix these bugs, without going through the hassle of central updates, means that the software will improve faster than proprietary software. The danger, though, is the ever present threat of malicious or merely careless programmers who introduce bugs into the software. There
still need to be checks and some form of copyright. One very effective way to distribute software at low cost and to retain copyright in a sensible non-oppressive way is to distribute it on write-once media. Most people do not have the facilities to make CD-roms (high capacity read-only compact discs) and this will help stop people introducing bugs; also, CD-rom discs are in a sense a 'conventional' medium that can be treated by the copyright laws — and other social copyright conventions — in the usual way. A vast amount of software is already available on CD-rom, and it is pleasing to see that some distributors emphasise that the software has been as thoroughly tested for bugs and viruses as possible (BMUG, 1989). It would also have been nice to see that the software had also been checked rigorously, but that is another point.

Perhaps one reason why nobody criticises the current sad state of the software industry is that most software is not good enough to be worth criticising. Users have their jobs to do, and using computers as such is a small part of that. Furthermore, most software is not sold in enough copies to have a large user base. Overall, then, lack of criticism from users is not surprising. On the other side, programmers have a hard enough time just getting their programs together, let alone worrying about responding to criticism that probably demands complete rewrites. Clearly substantial criticism must be available to designers much earlier in the design cycle. Formal methods are therefore a most promising approach. Formal methods occur at the beginning of design, before commitments and investments become irreversible; they are concise and comparatively cheap to alter in response to criticism; but they are mathematical. In fact, most formal development methods are too mathematical for widespread adoption in the industry. But mathematics has one very desirable feature: it is meant to be criticised. Mathematics as a discipline has chosen and developed an obligation to prove its results. This is essentially what is lacking in programming; in programming there is not even a distinction between conjectures and theorems. It is to be hoped that, if not formal methods as mathematical activities, formal methods as high standards are rapidly adopted. We note that formal descriptions of systems do not reveal as much as source code: it may be possible to publish formal descriptions of systems without seriously compromising commercial interests. Publishing formal specifications would not only help reduce bugs as the specifications go under public scrutiny (also the fear of having bugs exposed in this way would increase standards!), but it would also help users (perhaps with the aid of brokers) make an educated choice between systems with different specifications.

It is easy to argue that formal methods are impractical to adopt on the necessary scale to have any impact on the problem. There just are not enough trained programmers, for instance, and possibly there never will be. But this section of the paper is about prospects for the future. I do not think formal methods will solve problems in interactive systems soon. Instead, as features of interactive systems become understood and formalised, and published, people will be able to criticise and improve techniques. The 'public' that scrutinises specifications does not have to be large, but it does have to be larger than it is at the moment — when effectively nothing goes under scrutiny.

This article has not mentioned minor bugs, bugs of interface presentation. In
themselves presentation bugs do not stop a user operating a system, they just make it unnecessarily difficult. I believe that such bugs will be harder technically (but easier socially) to handle than the catastrophic bugs that we have concentrated on here. It is promising, however, that several research programmes are addressed at presentation bugs (Norman, 1988; Thimbleby, 1986b). Earlier I quoted Knuth that designers should write system documentation as an intrinsic part of design; indeed there is much work on helping devise more effective documentation, and there is promising research to make more abstract psychological insights into the needs of the user explicit to the designer (Young, 1989). The future prospects in these areas, then, are very promising.

Manufacturers' attitudes will unfortunately restrict progress in research. Earnest in his excellent article writes:

'Scientific knowledge requires that work is duplicated and tested. Just when computers make it easier to copy work and so advance knowledge, manufacturers respond to the underhand practices of the marketplace by making copying harder, and impractical for most individuals.' (Earnest, 1989)

That is, scientific work relies on duplication, but standard practice in the industry makes that difficult, if not subject to the sorts of spurious legal limitations exposed here.

The future, indeed, may become blacker as the current practice in software spreads to other areas. A recent book (Amadio, 1989), for instance, displayed the following warning: 'NOTE: This package may not be returned after this protective wrapping is opened'. You cannot read the book to find out whether it is even worth the paper it is written on. The book apparently contained a computer program, but one cannot be sure. It is obviously ridiculous to restrict assessment of information — in this case, to a book — in this way. It is time that computer programs are made more accessible.

Conclusions

It is often said that the computers of the future will be invisible, just like electric motors of today are already invisible (e.g. Norman, 1988). Thus we have clocks and washing machines, not motors. Similarly clocks and washing machines often contain computers, and the user does not need to be aware of them. Is this the ideal for the future? I believe that this vision is somewhat naïve because it forgets about computation: there are other uses for computers than being control systems. We often want users to be able to benefit from the wider possibilities offered by computers. If you like the 'invisible motor' analogy, it is rather like being reminded, on the contrary, that some motors, such as electric drills, that is motors-as-tools, are very useful just because they are visible motors.

But the 'invisible computer' vision has one merit: when your washing machine goes wrong, even when the fault is caused by the embedded computer's bugs, you can get it fixed. You have a right to get it fixed and a right to compensation for consequential damage. The sooner applications like word
processors are designed as carefully as washing machines the better; the sooner user require — and get — similar protection from bugs the better.

Finally, I cannot resist quoting the warranty for my Cross ball pen. Certainly ball pens are simpler than software, but nevertheless this warranty exhibits a totally different attitude to product quality and manufacturer responsibility than we have seen in the software industry. Yet a pen is just an old fashioned word processor, supposedly superseded by computing technology . . .

'Full Perpetual Warranty Cross® Guarantee

All Cross Writing Instruments and Desk Pen Holders are unquestionably guaranteed against mechanical failure regardless of age. Any Cross product or part requiring mechanical service will be replaced or repaired when returned to the factory. Refill cartridges, lead and erasers exhausted in the course of normal use are excluded from this warranty. The Cross guarantee extends our assurance of a lifetime of writing pleasure to every owner of a Cross Writing Instrument.'

Acknowledgements

The following have made very helpful comments on earlier versions of this paper: Bruce Anderson, Margaret Boden, Paul Booth, Tom Carey, Dan Diaper, John Doyle, Charles Forsyth, Henry Ledgard, Dianne Murray, Matthew Jones, David Pullinger, Ben Shneiderman, Ian Wand, Ian Witten, Chris van Wyk, and the referees.

References

Berkeley Macintosh Users Group (1989) BMUG Publicly Distributable ROM 1


Thimbleby, H.W. (1989b) ‘No way to sell a program’ New Scientist 124, 1693, 84–85


