Heedless programming: ignoring detectable error is a widespread hazard

Harold Thimbleby*

Future Interaction Technology Lab
Swansea University

SUMMARY

Software should be correct and robust. This paper suggests that we need forthright words for the failure of not being robust — *heedless* and *heedlessness* — and of recursively creating software (such as a compiler or virtual machine) that itself does not support dependable software development. Heedless programming is common, particularly affecting "trivial" operations such as on numbers, and extends deep into programming language design and into the use of computers more widely, thus making robust, dependable applications of all sorts unnecessarily problematic. The paper defines the problem and presents a call to action to start addressing the problems identified.

— "Study failure examples even more carefully than you study successes."
Frederick P. Brooks, Jr. [7]

KEY WORDS: Heedless programming, Dependability, Human Error, Excel, FORTRAN, Java, JavaScript, Mathematica, Design tradeoffs.

1. Introduction

The earliest written record of programming in the modern sense is of Ada Lovelace correcting a faulty program written by Charles Babbage [3]. The first textbook on programming [41], originally published 1951, says "it is, in fact, rare for a program to work correctly the first time it is tried, and often several attempts must be made before all errors are eliminated" (which wisdom now can cost £3,000 for a first edition in good condition). Nearly 50 years later, the *New York Times* [25] describes the omission of a hyphen in a program as leading to the crash of the Mariner 1 spacecraft. One character, one $18,500,000 spacecraft.

*Correspondence to: H. Thimbleby, Swansea University. Email: harold@thimbleby.net

All programmers make typing slips sooner or later, and some trivial slips have dramatic consequences. Thimbleby and Cairns [35] show that similar problems arise in everyday user interfaces: a user miskeying a number, by keying an extra decimal point and thus making the number syntactically incorrect, can in some systems get a number ten times larger by accident and without the system detecting or blocking the syntax error. In a medical application, giving a patient ten times too much morphine could easily be quietly fatal. Even without contextual information, about half of keying errors could be detected by simple syntactic checks, but many systems [33, 35] are heedless even to such obvious errors as multiple decimal points in a number. Part of the problem is that the programming environments and practices used themselves are heedless to error, and therefore end users inherit the heedlessness.

In the widest sense, then, people interact with computers whether through interactive user interfaces or through programming languages, and some user interfaces and programming languages are heedless to certain slips and errors people make. We know for certain that people will always eventually make slips, and in fact the greater expertise people have the more likely they *will* make slips [35, 39]; the question is whether the computer systems they use adequately block or manage those slips to help stop them turning into harm or causing other adverse outcomes.

FORTRAN is a venerable programming language dating from the 1950s, well-known for its use of implicit declarations, which means that a misspelt identifier can be treated as a valid identifier. This can result in programs that do not do what is expected of them. The compilers provide no guidance for the programmer to find the errors. Here, simple slips lead to errors that a programming language interprets as something other than what the user probably intended.

There is a tradeoff: most of the time, programmers benefit by having to type less code, but the risk is that less redundancy reduces the ability of the the compiler to check that what the programmer writes makes sufficient sense. FORTRAN was originally developed in an age when even compiling itself was a major research problem, and that it worked at all, certainly that it worked so successfully, might be a cause for wonder. Its historical shortcomings are easily forgiven. Sixty years later, computer science has moved on, and language design and implementation are now more mature subjects. Yet we still carry the original baggage with us: programming remains surprisingly risky. Indeed, many popular modern languages languages like PHP and Mathematica are still heedless to variable misspellings; Mathematica (discussed more below) has elevated this to a core feature of the language.

What I call "heedless programming" is a consequence of our desire for concise power and powerful features, even though in combination they may risk feature interaction and undetectable errors that lead to unexpected, unwanted and often undetected error and harm in the real world. The persistence of the implicit declaration hazard over decades of development in FORTRAN suggests that programmers prefer to live with risks than do extra work to help systems detect or otherwise manage them (for example, in FORTRAN, it would mean always declaring variables).

Worse, the problem is recursive: anybody programming has to contend with the heedlessness of the language they are using and its implementation as well as their own bugs; they pass on any oversights to anybody using their programs. If they implement programmable systems (say, spreadsheets) these oversights then get inextricably entangled with the end-users' own. Thompson [38] showed how bugs introduced deliberately into such a recursive environment

can be made to disappear from sight; the point is, non-malicious heedlessness can be just as invisible and just as hard to manage. It is just as hazardous.

"Heedless" is an unusual word; reckless, neglectful, oblivious and inattentive being more familiar near-synonyms. But, instead, giving a common word to a largely unrecognized problem might make it look like an obvious problem we all know about — and don't need to do anything about. Giving it a more distinctive name hopefully encourages more focussed thought on the issues.

What's the difference between heedless programming and ordinary bad programming? Bad programming can cause heedless programming, of course, but heedless programming more specifically creates environments where even careful users cannot detect, prevent or easily manage error: heedlessness is the outcome of a recursive failure to program carefully. For example, a spreadsheet would be heedless to error if there was no way to help stop or find errors in its use. Heedlessness, then, is the programmer's attitude to the program's users — be they programmers (for a language design or for a compiler) or ordinary users (for an interactive system). However, designers of languages and compilers have greater responsibility, for they create environments where future programmers create new environments, to be used in turn by further programmers or ordinary users.

Heedlessness is often initially invisible and unnoticed, and then as software is revised and improved it creates an increasing and often insurmountable conflict between real improvement and backwards-compatibility; it thus takes considerable willpower, not just insight, to work to sufficiently high standards the first time around. Suddenly a prototype idea has millions of users, and then the initial heedlessness has become a fixture and has absorbed millions of people's time, whether in error-recovery or in devising defenses against further error.

Systems have to be iteratively designed [21], so a key step to managing heedlessness is to test on the small scale before releasing beta products to large numbers of users. Ironically, by the time a system has a enough users to have a good chance of identifying heedless bugs, the rest of the users are relying on those very bugs — so-called "misfeatures": thus iteration, in the sense of reversing bad design decisions, is often impossible once there is a community of users. The shortcomings of Perl, PHP, CSS, JavaScript, etc, are going to persist because the design ambiguities resolve in ways users have grown to rely on, and often even start to justify!

In medicine, patients have problems and seek cures, often from drugs. Almost all drugs have side-effects; cures generally come at a cost or at least a risk. Some side-effects affect every patient, while some are risks that lucky patients may avoid. Similarly, heedlessness is a benefit, like a drug, but with side-effects that are unavoidable but, sometimes, on balance seem better than the problem being cured. We all want to program easily, but the cost is the side-effect of the bugs and other problems. The medical analogy is appealing, but "side-effect" already means something different in computing, so we shall stick with heedless and heedlessnes.

Dependable — "heedful" — programming is currently impossible to achieve even where we want it, even in some of the best and most widely-used systems currently available. Programmers building libraries or language designers creating standards of expression have a great responsibility to ensure that heedful programming is possible. To date, they have made choices that pander to easy expressiveness (including backwards compatibility) at the expense of safety, and once this is done, it is very hard to recover safety or dependability. The result is that the whole edifice, in the worst cases, from the foundations up is heedless, and layers

on top of heedless programming are not going to be able to circumvent the underlying latent hazards however well-intentioned originally. The evidence from user interface design, too, is that programmers are also failing to take account of the possibility of human error; they are building deceptive systems on top of deceptive virtual machines, languages and programming environments.

There is a very positive, major international initiative to make programs more reliable, the *Verified Software Initiative* [13, 17]. Although we are making progress, it remains a Grand Challenge with much more work required, until as Hoare put it, the most reliable part of a system is the software [15]. But the software never will be while we insist on making systems with too many deceptively convenient but unpredictably interacting features. Ironically, complexity and the concomitant heedlessness are irresistible temptations even in languages specifically designed for verification: they are still unusable "by mere mortals" (to quote [42]), which suggests that the idea of heedful programming needs to be taken up by the verification community as well. We need dependable features that *can* be used by mere mortals, yet who from time to time *will* slip and omit to type hyphens or whatever.

We need to raise awareness of the issues; in the first place the word "heedless" may help; in the second place more research will help; and, finally, we need a program of action. Section 7, towards the end of this paper, provides some suggestions. The reader is invited to think of more or better alternatives to my own suggestions.

2. Pragmatism, real-world engineering and effectiveness

Software has had an enormous impact on the world and it is amazingly effective, and it would be quite unrealistic to call all that achievement heedless. However, its success does not mean it should not be better, nor does it excuse many of the failures.

We could carefully evaluate the effectiveness of software and estimate the cost of correcting its defects as we understand them. One could then make an evidence-based decision whether heedlessness is an issue to be concerned with. This is a recipe to postpone action, and (as pointed out above) a recipe to increase the costs of correction.

Another approach comes from engineering. As pointed out by Ralph Nader [24], in the 1960s cars were promoted on a wide range of criteria, but safety was delegated to the user: *drivers* had accidents. While that remained the cultural norm, engineering failure could be passed off as the driver's (or insurer's) responsibility. However, as Nader showed, cars can be improved: they can be engineered with better brakes and better handling. This then reduces accidents. Crucially, it reduces accidents at negligible additional cost to the manufacturers: improvement is primarily achieved by engineering decisions. Designers have to design cars anyway, but now they prioritize safety. Similarly, in software, while we do not prioritise heedful programming, we will live in a "1960s culture" of thinking it is the user's or end programmer's responsibility to "drive safely." It partly is, but better engineering will avoid some of the problems.

Comparing programming with 1960s car manufacture sounds harsh — after all, in computing we now do have the knowledge and technology to do better. Yet consider: there is no mainstream car manufacturer producing unsafe cars while there remain thousands of software

manufacturers producing unsafe code. Cars (e.g., their engine management systems) are programmed using safer programming concepts than most programmers are familiar with.

Computer Science certainly has its Naders: Edward Lowry, for one, said presciently in 1968 [22], "I think that ANY significant advance in the programming art is sure to involve very extensive automated analyses of programs. If you want better debugging of programs, then the computer must analyze the programs and find the bugs ... If you want better documentation then the computer must analyze the programs to produce abstract representations of the programs from a variety of points of view." We have not moved forward; in 2004 Lowry wrote, "The software community has shown an aversion toward fundamental issues and the public safety that seems unprecedented in other technologies ... In effect, the software community has been keeping human minds debilitated on an increasingly large scale in order to keep them in a state of dependency. Leading organizations responsible for software safety should be regarded as a menace to public safety until their competence in software simplicity is demonstrated." [23].

— *"The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive world-wide distribution of bug-ridden software for which we should be deeply ashamed."*
Edsger Dijkstra [10]

3. Examples of heedless user interfaces

User interfaces are a visible manifestation of heedless programming.

A very familiar case of design that also happens to be heedless to human error appears in general purpose calculators. The people who need to use calculators are the very people least likely to spot incorrect results, for calculators are *supposed* to save them doing the sums themselves.

Calculators are designed to do "any" calculation, and therefore cannot detect when a user keys in the wrong calculation, say, accidentally substituting an add for a subtract. One might say this is an obvious risk, and is the user's responsibility to manage it. Often calculators are designed so that short cuts do "useful" things, and this power is often used as a defense against design criticism. But calculators allow users to enter jibberish and yet still produce some sort of result that the user might think was the correct result of an intended calculation. Worse, many treat an accidental double press like [+][+] as meaning something quirky rather than an error. Interpreting it as a feature (defining a constant?) may help sell the calculator to naïve users, but such users just get the wrong answer from time to time. If a user accidentally keys [×][−] (multiply, subtract) in succession, many calculators will take this as intended to be a subtraction, and proceed to provide -1 as the answer to 10×-11. The user almost certainly remains unaware that a slip (not rearranging the expression to suit the calculator's conventions) has changed the meaning of their calculation.

An example of a common short cut provided by many calculators is that starting a calculation with plus adds to the previous result. Arguably this makes a serial sum easier, yet it flies in the face of mathematical convention, where $+1$ and 1 have the same value.

Indeed, $+ + 7$ and $+7$ traditionally have the same value, yet most calculators take the former as a redefinition of some constant, so a subsequent "erroneous" calculation of 1+ will now add 7 to it. Or something. Sadly, there is no standard and calculators are different; *even* if this short cut is a good idea, it encourages "transfer errors," where a user understanding the feature tries to use it on another calculator, and gets a different result, and most likely, an unnoticed error.

Users make slips, so sometimes calculators will provide incorrect results. The user (or the organisation in which they work) is to some extent responsible for this, and various steps can be taken to mitigate the problem: standard approaches include repeating the calculation (preferably, repeating it in a different way) and two independent calculations — even two "buddy" users — and comparing results. The inconsistent diversity in calculator design additionally means that these ruses are insufficient: several independent calculators should be used too, not just independent users.

Since there is no reason not to have a standard notation for calculators, calculators must be different for commercial reasons; manufacturers hope to lock in consumers to their brands, since diversity ensures that users are unfamiliar, and hence less safe, with the detailed operation of competitors' calculators. Calculators are different and bad at handling user error because it is more profitable, and because of the misdirection "it is the user's problem." Calculators could certainly be much better [33, 34, 35].

Microsoft Excel is the dominant spreadsheet: it bridges the gap between a passive user interface like a calculator and a programmable system. Excel has been designed to be heedless to user and programmer error, and different vendor's "equivalents" to Excel are heedless in different ways, thus making spreadsheets very difficult to use reliably [35]. For example, in Excel a syntax error such as `1..2` is treated *without warning* as zero by the function `SUM`. A syntactically correct cell that shows 1.2 is also treated as zero if the 1.2 is a string, which could be accidental but could easily be exploited maliciously. A malicious user can conceal almost any activity (for example a number can be displayed as 1,000 but be treated as zero by `SUM`); equally, errors can slip past a diligent user because of the original heedless design combined with the lack of warnings and checks provided by Excel.

4. Heedless programming languages

Programming is difficult, and perhaps unlike calculator and spreadsheet users, programmers are expected to program carefully.

Programming languages might be divided into two categories: general purpose and domain specific. One imagines that real programmers use general purpose languages, and domain experts use domain specific languages; of course such a division is not rigid. Thus, a real programmer might use Excel (an application containing a domain specific language generally used for finance), and a domain expert (say a mathematician) may choose to use Java rather than Mathematica. The notion of "domain specific" shifts over the years; in their time FORTRAN and Cobol were, and arguably still are, domain specific though they are now largely superseded by general purpose languages like Java.

For the purposes of this paper, however, we find it a useful distinction. One could argue that general purpose languages are intended to be used by trained programmers, so they may compromise dependability for generality or speed on the assumption that a trained user will know how to program correctly. In contrast, a domain specific language might be expected to provide powerful tools for the domain, while simplifying or even eliminating complex programming issues.

4.1. JavaScript (ECMAscript)

JavaScript is a very widely used programming language, in every sense: the number of programmers, the number of running programs, and the number of programs generating programs; it has also inspired many other languages, such as Macromind's Flash. Unfortunately, JavaScript has numerous design problems [9].

Here, we will explore how JavaScript handles errors in numbers badly (a heedless issue not raised in [9]), as figure 1 makes clear. The user's experience often hinges on keying numbers into form fields (e.g., for buying things off the internet), perhaps making errors and correcting them, and browsers render and interact with numbers in diverse ways. It follows that almost any web application using JavaScript is likely to create problems.

In JavaScript the built-in `parseFloat` ignores errors. For example, `parseFloat("1.2.3")` gives the incorrect value 1.2 with no error reported (the string may have been computed elsewhere, and in normal use the error would not be so obvious as illustrated in this paper). In other words, if a programmer is to use a built-in facility, they have to pre-check their data; as `parseFloat` is heedless to error, it provides no support itself to programmers. This is such a familiar problem that it is tempting to dismiss it! Yet consider `parseInt("08")`, as might easily occur originally in data representing the date for August: it gives the surprising value 0 (probably because the compiler is treating the `8` as a single octal digit, and as $0 \equiv 8 (\mathrm{mod}\ 8)$ — rather than generating an error message).

JavaScript's built-in `parseFloat` evidently reads numbers, but stops without reporting an error as soon as the numbers "finish"; thus on reading `1..23` (e.g., perhaps as keyed by a user into a form field) rather than report an error it returns the value 1, since the second decimal point obviously (!) signals the end of the number. Yet two decimal points is obviously an error. A more interesting error is `2e3` (i.e., 2×10^3) is treated by `parseInt` as 2 without reporting an error, since it treats the `e` as ending the number, although to `parseFloat` an `e` is a valid part of a number (isn't 2,000 and integer even when written `2e3`?). Another inconsistency is that all but one technique treats 0666 as 666, yet if 0666 is typed in the source code of a JavaScript program it is given the value 438 (because 0666 is treated as if in octal).

An internet search of JavaScript number parsing code on the web will reveal further variations — evidently, programmers try to solve the problems, but do not solve them well or even consistently! Thimbleby [37] gives more examples, and shows how very similar problems recur in safety-critical devices.

In short: any application that uses JavaScript to handle numeric data from users is almost certainly going to be problematic as sooner or later, users *will* make keying errors, and relying on JavaScript to do anything sensible is unrealistic. If a language like JavaScript is to be used — and on the world wide web, there is little realistic alternative — then *very* careful programming

Valid floating point numbers

Data	parseFloat	coercion	parseInt	!isNaN
.1	0.1	0.1	NaN	true
0.25	0.25	0.25	0	true
0666	666	666	438	true
8.0	8	8	8	true
87.23	87.23	87.23	87	true
99	99	99	99	true
2e3	2000	2000	2	true
␣5.6	5.6	5.6	5	true

Invalid floating point numbers

Data	parseFloat	coercion	parseInt	!isNaN
1e999	Infinity	Infinity	1	true
empty string	NaN	0	NaN	true
1..23	1	NaN	1	false
1.2.3	1.2	NaN	1	false
.1.2.3.	0.1	NaN	NaN	false
.1...	0.1	NaN	NaN	false
Text!	NaN	NaN	NaN	false
␣	NaN	0	NaN	true
2.7stuff	2.7	NaN	2	false

Figure 1. How JavaScript parses numbers: `parseInt` works differently even when there are no errors (top table), but are manifestly wrong when there are input errors (bottom table). Note that `parseInt` gives misleading results if a floating point number is well formed but has a non-zero decimal part, and it certainly behaves inconsistently with octal numerals. The columns show the results of JavaScript's `parseFloat(data)` function, coercion (as in `1*data`), `parseInt(data)`, and the predicate `!isNaN(data)`, not NaN — "NaN" being short for *Not a Number*. The results shown here may differ between browsers, another potential source of confusion.

is the only option, especially if one expects international users (that's what *world wide* means) who use commas or spaces for digit separators and decimal points. Despite an international standard (ISO 31-0), so it's a solved problem, why does it seem so unrealistic to expect a reasonable level of attention to detail and error-handling by the designers of JavaScript?

4.2. Heedless general purpose languages

People make errors, and people sometimes do not notice the errors they make. Eventually someone will make an unnoticed error that is then processed by computer.

In Java, the designers of the language after long experience of C *decided* to perpetuate a known problem in C [2, 32]. In C, a long integer is written with a `l` (letter l) or `L` (capital letter L) to mark it as long. Thus `11` is a short integer and `1l` (which looks almost the same) is a long integer, despite it being potentially hard to see the difference between `1` (a digit one) and `l` (a letter l). Digit `1` and letter `l` are easily confused; what seems to be a trivial problem when it is explained (as here) becomes very obscure when the context, say, in a

large program, camouflages it. Some argue this is a completely trivial problem. Indeed. If it was widely discussed in 1978 [27], why is it still haunting us over 30 years later? Why are new languages designed continuing the heedlessness, even though the designers *know* they are doing it?

Since C programs are not Java programs, there is no need to be backwards compatible with this design fault; it would have been easy to design Java to eliminate the problem. (Perhaps a word like `long` could have been used instead of a single character.) It would have been possible for the compiler to generate warnings, as an intermediate solution. Unfortunately the problem has now been euphemized as a "puzzle" [6] — that is, heedless design is presented as a fun problem for the programmer rather than as a fixable problem for the language designer! No doubt unsafe cars are more responsive and fun than safer ones; on a private racetrack, that's fine, but for any car used in public (as Java certainly is) heedless engineering is irresponsible.

4.3. Heedless domain specific languages

Although there are many domain specific languages (covering domains from music to oil exploration) to pick from, I have chosen two where the domain is well-defined and should be broadly familiar; they also have the advantage that their input and output are (predominantly) textual and hence easily reproduced in a paper.

> — *"If you can't read a program and understand what it is going to do, it is impossible to have confidence that it will correctly do what you want."*
> **Douglas Crockford** [9]

4.3.1. Mathematica

Mathematica is a very popular mathematical programming language (others include Axiom, Maple, Mathcad, Maxima, etc). Mathematical languages are distinguished by their concise approach to mathematical transformations, such as solving equations, that would be lengthy to express in conventional languages.

As mentioned earlier, undeclared variables, despite their well-known problems, are a feature of Mathematica and integral to its design. An undeclared *program* variable, say `x`, takes on the role of a *mathematical* variable. For example, the assignment `t=x+1` is valid even if `x` is not declared: it assigns to `t` the symbolic expression `x+1`; we could then later ask Mathematica to solve the equation `t=5` and we'd obtain the answer `x=4`, or we could any time later assign `1` to `x`, and `t` would become 2. A curious consequence of this approach is that `t` changes *every* time `x` changes *unless* the programmer happened to assign to `x` before the assignment to `t`! Because Mathematica thereby mixes programs with the domain it is talking about, simple programs are very powerful and concise, but complex programs, however, create many opportunities for undesirable and unexpected name capture.

Most programming languages make a clear separation of program and data by using some notational distinction, and typically the separation goes all the way down to the underlying hardware so that accidental or malicious programming cannot change the code of the running program and make it do something else. (The infamous Morris Worm was an example of data

becoming program; it crippled a significant fraction of the internet [28] — and Morris is quoted as having said he should have tested it first.)

LISP is famous for using the same notation (S expressions) for program and for data but, even so, it keeps a clear interface between them: data only becomes program when it is evaluated using `eval`.† The details are not important for the present paper, other than to note that, even in a language that goes a long way to making data have equal status to program, there is a clear interface between the two. Moreover, many programmers frown on `eval` (particularly in programs that take user input) because it can cause security and other problems, as arbitrary data can be evaluated.

In contrast, in Mathematica there is *no* distinction between program and data. There are over 20 features to control evaluation (`Defer`, `Unevaluate`, `Hold`, `HoldRest`, `ReleaseHold`, `LocalizeVariables`...), many of which make non-standard evaluation invisible at the point of use. The attribute `NHoldFirst` (to give one example) stops the built-in function `N` being applied to the first parameter of a function; it is so-specific a patch it looks like it is attempting to work-around a bigger problem. It's unlikely that a normal programmer will understand how these complex features interact with each other, let alone understand the underlying language issues they are trying to manage. A good programmer has little choice but to experimentally find *ad hoc* solutions that probably create further problems for future users.

> — *"... we do not address error-handling and other software-engineering issues that require specific assumptions ..."*
>
> **Cormen, Leiserson, Rivest & Stein** [8]

4.3.2. TEX and LATEX

It seems heedlessness is everywhere we look. Consider even the paper you are looking at right now, which was published using LaTeX.

The document processing system LaTeX and its underlying engine TeX will be familiar to many — together they have become the typesetting system of choice for technical writing. Essentially, TeX is a low-level typesetting system with a macro processor to allow it to be extended with user definitions, and then LaTeX is defined using this macro processor, in the end producing a structured writing environment for general use.

TeX is "safe" in the sense that it produces a document and the author has a chance to proof read it and correct it before anything else happens. But TeX documents often have a variety of errors that arise as a direct result of the design of TeX; TeX was designed as a macro processing language, so all programs are strings to be typeset, so they inextricably mix text and program. A common problem is that macro definitions in TeX will introduce extraneous spaces into a document that are very hard to track down. In contrast, in a normal programming language, whitespace in the program has no meaning whatsoever. A more serious problem is that TeX does not provide name scoping, so authors providing library routines have to go to

†Conversely, function closures can be treated as values but in many dialects of LISP they are uninterpretable data.

extraordinary lengths to try to conceal (actually, merely obfuscate) what should be private names [36].

LaTeX is built on top of TeX, and while a long discussion of design issues is out of place, we note a symptom: some macros have so-called "fragile" arguments — meaning that somehow their parameters may disintegrate if the author is not heedful. Fragility arises precisely because of the well-known problems of controlling macro expansion. How can the average author be sufficiently heedful of the deep technical problems in the system they are using?

> — *"A feature which is omitted can always be added later, when its design and its implications are well understood. A feature which is included before it is fully understood can never be removed later."*
>
> **C. A. R. Hoare** [14]

5. A "power-heedlessness tradeoff"

Recurring in the preceding discussion is the idea that the unconsidered desire for power and flexibility leads to heedlessness. This is inevitable: if we forbid certain "shorthands" or "implicit features" or work arounds — Amey [1] calls it "magic" — then necessarily the average length of a program must increase, since the shorter work arounds are not longer permitted.

Mathematica gives a concrete example. To multiply two numbers, use the operator `*`. However Mathematica (following the tradition established by conventional mathematical notation) permits an implicit multiplication. Thus `3t` is short for `3*t`. This reduces typing and, albeit in a small way, makes using Mathematica more pleasurable, since one obtains running programs with less effort and sooner. On the other hand, consider some dangers:

- While `3t` is short for `3*t`, `t3` is simply a variable called `t3` and is not short for anything. The implicit multiplication rule means the programmer must learn its exceptions or be caught out. (While implicit multiplication is conventional mathematical notation it is curious that in Mathematica it does not commute, for `3t`$\neq$`t3`.)
- Mistyping a number intended to be `1.23` as `1.2.3` will be treated as 0.36, since Mathematica implicitly inserts a multiplication `1.2`$\times$`.3` — though again a programmer ought to learn why that is the implicit multiplication rather than `1.`$\times$ `2.3` or even `1`$\times$`.2`$\times$`.3`
- Mathematica's conditional statement general form is `If[test, trueArm, falseArm, neitherArm]`. If a programmer misses out any one of the commas in this test, which is easy to do and hard to spot when the program is complex, Mathematica silently provides an implicit multiplication, perhaps `If[test, trueArm, falseArm*neitherArm]`. There is no error since all the parameters to `If` are optional: a program will happily run with *no* commas (though if all of them are omitted Mathematica generates a warning message that the programmer may overlook).

The point is, there is a *tradeoff*. Programmers should not just want power, flexibility and "usability," for they come at a price, and ignoring the price leads to heedlessness. Sadly many

programmers externalise the price — the users of their programs pay the price instead. Since users generally vastly out-number programmers, the programmers should err on the side of caution, and put the extra effort of programming carefully into perspective. Moreover, as in the case of Mathematica, when those users cannot reasonably be expected to be competent programmers aware of many or even any of the deeper issues of dependable programming, even more care should be taken in design.

Another price programmers ignore is the investment in their training (since they already have it), except when they are asked to learn new ideas. Thus there is a natural tendency to resist new ideas that might improve their performance. For many programmers, type safety is a new idea, and we consider this idea next.

— *"I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."*
C. A. R. Hoare [14]

6. The value of type safety

Type safety is how well a programming language prevents type errors, such as adding a number to a string. While types help eliminate a large class of errors, most programs take input from somewhere, other programs or from human users, and then generate results. The simplest example might be reading a number: the type of this operation is String→Number, but the program, even being type-safe, may convert some strings (e.g., ones involving underflow, overflow, excess precision, or syntax errors) into arbitrary numbers.

In contrast, Mathematica, JavaScript, Perl, PHP, Excel and others are good examples of systems that seemingly gain power and flexibility by having little type safety: they rely entirely on run-time typing. In Mathematica, a variable can stand for anything, and it might be a formula or a numeric value — or a picture, or a string (which is perhaps also a formula), or tensor or any object whatsoever. This means a programmer does not have to learn about types, and therefore starts doing interesting things in the language very quickly. Mathematica does not even have the concept of a type error; for example, adding 1 to a string simply gives you 1 plus the string *as a formula*. However, if later you subtract 1 from it, you end up with the original string; or if you multiply it by zero you get zero (and, again, no error reported). This flexibility means that a type error may get cancelled out later and never be detected. C is even more extreme: everything is just a bit pattern, and if you happen to add an integer to a floating point number, you get whatever the resulting bit pattern means on the underlying hardware when it is next used.

In contrast, some programming languages such as Haskell and OCaml have strict static typing, and any program that compiles is therefore type-correct and will run without type errors. Languages like Java are intermediate: there is compile time checking, but if a programmer decides to use very general types like `Object`, then those objects can represent anything, and this can lead to problems when accidentally the objects are not what the program expects.

In summary, types and static typing in particular are a significant contribution to making programs more heedful because of the error-detection support they provide to programmers at almost negligible cost, but they do not stop programs being incorrect. One still needs to be a heedful programmer.

> — *"If you don't know where you are going, you won't know when you don't get there."*
> **Yogi Berra** [4]

7. Call to action

The persistence of heedless programming suggests some combination of any or all of the following factors are at play:

- **Ignorance, ignorance of ignorance, and recursive ignorance**.
  Kruger and Dunning in their famous article now seen as defining the Dunning-Kruger Effect [20] note that not only do incompetent people reach erroneous conclusions and make mistakes, but their very incompetence means they have no ability to understand the reality of their problems. Ignorance of standards of performance is behind a great deal of incompetence; it is evident that some programmers are incompetent (whether through lack of training or by not being resourced well enough for the complexity of the problems they face), they don't know it, and they then unwittingly shift the burden of their unnoticed problems onto their programs' users. The entire field of Human-Computer Interaction fights the corner of the end user [21, 31], but as we've seen in this paper, all programmers are caught up in the cycle of heedlessness.
- **Slippage**.
  Programmers build interesting systems, perhaps for themselves or for their immediate colleagues, and the success of the system then leads to wide uptake: a system on the web could soon have international usage on a massive scale. The homely development of the system, appropriate when it was in-house, is entirely inappropriate (illegal in the EU if it involves safety-critical applications) when the scale expands. Steve Job's well-known thought argues that the "10 seconds" an isolated programmer might dismiss is significant to the world; equally, an "insignificant" bug the programmer cannot be bothered to find or fix will impact users on a large scale the same way:-

  > — *"Well, let's say you can shave 10 seconds off of the boot time. Multiply that by five million users and that's 50 million seconds, every single day. Over a year, that's probably dozens of lifetimes. So if you make it boot ten seconds faster, you've saved a dozen lives. That's really worth it, don't you think?"*
  > **Steve Jobs** [18]

- **Cost-benefit**.
  Maybe it does not matter. It is a fact that computers have transformed society *despite* any supposed problems of heedlessness. On balance, we have chosen to have progress without the investment in some issues of dependability that this paper would have preferred. It is likely that our cost-benefit analyses suffer from "success bias": for

obvious reasons, we rarely hear about failed software (because of litigation and because companies with failed software go bust and disappear). Moreover failing software usually fails and disappears, whereas successful software is obviously successful over a longer period of time. So perhaps it just *appears* most software works well. Careful empirical research is called for.

- **Socially**.
  We *prefer* heedlessness to the alternatives, which would include prolonged language development times, restrictions on concise expressiveness, slower uptake of cutting-edge technological developments, continual revision in the field, etc.
- **Technically**.
  We do not yet know how to design languages to avoid or manage heedlessness. For example, exception handling is still an active area of research, and the "best" way of exception handling is currently a matter of opinion, application, and domain priorities.
- **Psychologically (*resistance to change*)**.
  Although it may be possible to avoid some problems in principle, change cannot occur without retraining programmers, and that then causes knock-on problems. The persistence of the `l` to mean long integer in the transition from C to Java is a case in point: whatever gains there may have been in a safer language *may* have been lost in the retraining costs for programmers, and might more certainly have been lost in a reduction in the popularity of Java.
- **Psychologically (*cognitive dissonance*)**.
  It is well known that successful effort gives people a sense of accomplishment, but it is also true that pointless effort creates *post hoc* justification [30]. Good programmers are good programmers because they have mastered some of the problems of heedlessness; they may now think that their approach to programming is a sign of maturity and that aspiring programmers should go through the same initiation. To that extent, they have little incentive to make programming languages more heedful.
- **Human factors (HF) and user centered design (UCD)**.
  It is relatively easy for programmers to develop systems ignoring human factors such as error, information design, and interaction design — programmers typically focus on "correct" behavior and avoid the complexities of error handling, undo and recovery; getting programs to work at all is cognitively very demanding and creates "tunnel vision" that tends to ignore everything else — sometimes to the extent of claiming it is not a professional requirement for programmers! (Understanding tunnel vision is itself a human factors issue.) Teaching human factors and user centered design (see [21] and the ISO 13407: *Human-centred design process* standard) is crucial for developing systems that work with the realities of user (including other programmers') behavior.
- **Theoretically**.
  We have to live with human error, and therefore there will always be a trade off between flexibility and dependability, reliability and resilience. Without theoretical developments, it is not clear what these trade offs are, and whether some or all of the issues raised in this paper are avoidable in principle or have to be accepted as part of an implicit trade off. Without any theoretical foundation, all solutions are unavoidably *ad hoc* — and that is what is happening.

While there is no theory and no clear identification of the problems, implicit trade offs will be made from education through to language design, and there will be no guarantees that there will be an overall benefit. Further work is clearly needed on a wide range of fronts, from gathering empirical evidence on the expected costs of heedlessness through to the theoretical considerations, as well as considering the best ways to change the educational culture that trains programmers. (Kernighan and Plauger's *The Elements of Programming Style* [19] was notorious because its bad examples were taken from real programming textbooks and indicated the quality of the education of programmers.)

In computer science, the complement of improvement is backwards compatibility. How, then, can improvements be made, when it is also necessary to maintain the legacy of current products, as well as the knowledge and skills base of the people who have to move from using old systems to using better systems? The simplest answer is that we are making changes all the time; change itself is not the problem. For example, Java is changing, often in ways that are not backwards compatible, even with itself. The issue is that change, even if it does not happen all at once, still needs to point in the right direction; the purpose of this article is to raise awareness — hitherto we have not had useful words to describe either the point of departure nor the point of destination, and therefore we have not always been moving in the right direction.

We can't just ask programmers to program better and hope things will come out right, we have to tell them how, and motivate them to own the responsibility of managing the problems their program's users will have. People who design programming languages, virtual machines, web based systems, and any system with many users or for building applications (e.g., involving scripting languages), need to remain aware of the problems and best practice for controlling them. In turn, to do this we need to do more research and build better, more rigorous, programming tools.

Here are some concrete suggestions:

- **Checklists**.
  Recalling our earlier analogy with medicine, perhaps a first step should be to devise checklists, as these are known to reduce problems and improve the management of human error — Gawande's book *The Checklist Manifesto* [12] is particularly inspiring. What are the issues to cover in checklists? What are the issues both that matter and that we can do something about through improved design?
- **Use dependable languages**.
  It ought to go without saying that dependable programs should be written in languages that try to support dependable programs. Thus medical devices should not be programmed in C, and radiotherapy treatments should not be calculated in Excel. Haskell, SPARK and MISRA C/C++ are examples of programming languages much more suited to safety critical applications — and if some programmers cannot cope with them, then almost certainly they cannot program well enough to develop dependable applications in *any* language.
- **Use good programmers**.
  Good programmers will know many techniques — such as unit testing, regression testing, randomized testing, static tools (like Lint and many others), formal methods,

model checking, iterative design, and applicable standards — to help develop better programs. If programmers haven't heard of or understand the contents of, say, C. A. R. Hoare and He Jifeng's *Unified Theories of Programming* [16], or an equivalent, then they should not be working on dependable programs.

- **Coding standards**.
  Coding standards are lists of "good practice" rules that programmers should follow; a good example is Joshua Bloch's book on programming well in Java [5]. Better still would be to read such standards to improve the design of languages — so the coding standards are enforced by the language and language editors themselves — rather than to push the problem onto end programmers. Seen like this, the purpose of coding standards is to make themselves obsolete.
- **Radical honesty**.
  Richard Feynman, the Nobel Prize-winning physicist, is famous for his speech at CalTech on radical honesty: "details that could throw doubt on your interpretation must be given. You must do the best you can — if you know anything at all wrong — to explain it" [11]. If you don't you are certainly wasting other people's time, and perhaps endangering them. For programmers, writing manuals should be part of this process, and moreover in programming we can revise the design of the system to make the manuals better — for example, rather than describing the problem with using `l` to confusingly indicate long integer literals, Java could have been re-designed so that a whole paragraph in its definition wasn't needed. Literate programming and systems like JavaDoc can be used similarly for internal documentation: not just for documentation but as a driver for reflection and improving implementation.
- **Reflect, fully, honestly, early, and review**.
  One of the obvious things to do is to write radically honest documentation and user manuals, then read them carefully, and redesign systems to avoid having to write warnings in the manuals. Writing documentation is a constructive part of the development process, not something to be done after the system is stabilized. The problem with `l` used to indicate long numbers is a case in point (see §4.2).
- **Education**.
  It is routine to ignore human error even in good in computer science education. To the extent that heedlessness is unavoidable then we have to train programmers to cope with the risks it entails; in fact, since heedlessness is unavoidable in today's environments, then we have to train programmers to cope with them even if in an ideal world it would not be necessary. What work arounds, coping strategies or management strategies should programmers use? When should we use independent programming teams, when is it worth incurring the extra cost of inserting checking code ... and so on. With improved education, we may be able to mitigate future problems, as today's students start to develop improved languages and programming environments.
- **Stories, shared experience and turning hindsight into foresight**.
  People do not program more carefully — and language and library developers do not program more carefully — just because it is a Good Idea or just because their managers want them to. Programmers will be more heedful when there are memorable stories about heedless programming and its consequences, when there is a culture that

remembers the value of good programming, and when programmers want to and are empowered to do better. *Analyzed* stories, with morals about the necessary behavioral changes needed, are essential to building and sustaining this culture. (Summers [29] gives some insights for influencing programmers, and Patterson *et al* [26] give widely applicable insights in effective change.)

- **Regulation**.
  In the two industries we have compared programming to, car manufacture and medicine, there is a simple regulatory distinction between private or recreational use and public use (with variations in different jurisdictions). While we refuse to make some analogous distinctions in programming, heedless programming that is exciting and even inspirational to the individual will be dangerous to bystanders. Programmers with no medical or even programming qualifications can write programs that control and perform medical procedures on patients; this seems bizarre, even if, in our current culture, the unregulated design of compilers doesn't!

8. Conclusions

Programming is a bit like a medicine; we use it intending to make the world a better place, to cure problems with computers, but like most medical procedures it has potential side-effects. Possibly, a truly safe medicine would do nothing; if it can improve somebody's health, it can probably harm somebody else's health. Possibly, a truly safe program would be close to pointless and a truly safe programming language would only permit innocuous programs? Which is the better language: JavaScript, which is pretty heedless, but supports almost all of the world wide web, or Haskell which is much safer, but runs only a tiny fraction of the world's programs? If heedlessness, power and popularity unavoidably go together the design trade-offs will always be hard, as hard as making drugs that are enjoyable without the side-effect of being addictive!

However, what is certain is that design without considering the "heedlessness trade-off" is unnecessarily risky and too often naïvely biassed to quick consumerism — the big hit of drugs.

It's hindsight, but one wonders how much better the world wide web — or climate change modelling or mathematical programming or finance or whatever— would have been had the developers of programming environments tried to prioritize heedful programming over quick and easy flexibility and power. For example, URLs might have been readable rather than constrained by the irrelevant syntax of Unix commands.

Unfortunately explaining heedless programming is tedious, and perhaps this is the main reason why this important issue has received so little attention. Thinking about heedlessness means thinking about human error and how to handle it; most of the time, though, we'd prefer to ignore the complication of thinking about errors, and celebrate what amazing things can be done *when things go right*; the details of anticipating and managing errors is too tedious.

FORTRAN is routinely used in climate models, and, given that dependable climate modelling is one of the world's top priorities, improving itor replacing it is a priority. Excel is routinely used for calculating radiation doses for cancer patients; Excel, Mathematica, Java, C are used from finance to every kind of research and beyond. There is, from one point of view,

some sense in any powerful and flexible approach, but when used in non-trivial applications the power becomes complex: the many features and their interactions are neither memorable nor intuitive.

We cannot ignore the heedless design of programming environments; more worryingly, we cannot ignore the cultural attitude that pervades computer science and that results in heedless systems masquerading as safe systems. We cannot dismiss user error as the user's fault when the very foundations of the tools they use are riddled by heedless programs. In many areas, climate change, cardiology, finance, radiotherapy, and more, we cannot put such a burden on end users.

Programmers who write or design code for other people have a radical responsibility to be heedful of possible errors, and to allow their users to determine how to handle those errors. It seems very surprising that the user interfaces of spreadsheets, being one of the most widely-used non-expert programming systems, have not received more attention: how can users manage errors if they cannot notice them?

While systems are marketed by showing off individual features — which are easy to make impressive separately — and ignore potential feature interaction it is unlikely that market economics without regulation will drive improvement. Improvement has to come about through more careful programming, especially by the best programmers who design languages, compilers and virtual machines, being aware of their responsibilities. Underlying improvement must be the will to improve, and ownership of the responsibility.

As it will take a generation to improve the foundations everyone builds on, we need to improve education now so that programmers entering the industry (or other positions where they develop applications) do not continue to build on our widespread legacy of heedless programs (and the culture that embeds it). Education is the priority (and before that, the teachers, and before that the students who will become teachers); until education is addressed, the heedless culture will continue, and itself will undermine other initiatives to improve programming.

Acknowledgements. The author is grateful to the editor and referees and to Paul Cairns, Jon Crowcroft, Tony Hoare, Daniel Jackson and Martyn Thomas for very constructive comments that have led to numerous improvements in the exposition.

REFERENCES

1. Amey, P. (2001), Logic Versus Magic in Critical Systems, *Reliable Software Technologies — Ada-Europe 2001 6th Ada-Europe International Conference*, Lecture Notes in Computer Science, **2043**, Craeynest, D. & Strohmeier, A. (Eds.), Springer-Verlag.
2. Arnold, K. & Gosling, J. (1996). *The Java Programming Language*, Reading: Addison-Wesley.
3. Babbage, C. (1864). *Passages from the life of a philosopher*, p.136.
4. Berra, Y. (1999). *The Yogi Book: I Really Didn't Say Everything I Said*, Workman Publishing Company.
5. Bloch, J. (2008). *Effective Java*, Second edition, Prentice Hall.
6. Bloch, J. & Gafter, N. (2005). *Java Puzzlers — Traps, Pitfalls and Corner Cases*, Addison-Wesley, 2005.
7. Brooks, Jr., F. P. (2010). *The Design of Design*, Addison-Wesley, 2010.
8. Cormen, T. H., Rivest, R. L., Leiserson, C. E. & Stein, C. (2009). *Introduction to Algorithms*, 3rd ed, MIT Press.

9. Crockford, D. (2008). *JavaScript: The Good Parts*, O'Reilly.
10. Dijkstra, E. (2001). The end of computing science? *Communications of the ACM*, **44**(3), p92.
11. Feynman, R. P. (1974), Cargo Cult Science, Caltech commencement address. Also in Feynman, R. P. and Leighton, R. (1992). *Surely You're Joking, Mr. Feynman! Adventures of a Curious Character*, Vintage.
12. Gawande, A. (2010). *The Checklist Manifesto: How To Get Things Right*, Profile Books.
13. Hoare, C. A. R. (2009). The Verified Software Initiative: A Manifesto, *ACM Computing Surveys*, **41**(4), pp22:1–22:8.
14. Hoare, C. A. R. (1981). The Emperor's Old Clothes, Turing Award Lecture, *Communications of the ACM* **24**(2), pp75–83.
15. Hoare, C. A. R. (2007). The Ideal of Program Correctness: Third Computer Journal Lecture, *Computer Journal*, **50**(3), pp254–260.
16. Hoare, C. A. R. & Jifeng, He (1998). *Unified Theories of Programming*, Prentice Hall.
17. Hoare, C. A. R. & Misra, J., editors, (2009). Special Issue on Software Verification, *ACM Computing Surveys*, **41**(4).
18. Jobs, S. (1983) Quoted in Hertzfeld, A., www.folklore.org/StoryView.py?project=Macintosh&story=Saving_Lives.txt
19. Kernighan, B. W. & Plauger, P. J. (1978) *The Elements of Programming Style*, 2nd ed., McGraw Hill.
20. Kruger, J. & and Dunning, D. (1999). Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments, *Journal of Personality and Social Psychology*, **77**(6):121–1134.
21. Landauer, T. K. (1996). *The Trouble with Computers: Usefulness, Usability and Productivity*, MIT Press.
22. Lowry, E. S. (1970). Report on a 1968 NATO Conference on Software Engineering Techniques, edited by Buxton, J. N. & Randell, B., The Kynoch Press.
23. Lowry, E. S. (2004). Software Simplicity, and Hence Safety — Thwarted for Decades, International Symposium on Technology and Society
24. Nader, R. (1973). *Unsafe At Any Speed: The Designed-in Dangers of the American Automobile*, revised ed., Bantam Books.
25. New York Times (2010). For Want of Hyphen Venus Rocket Is Lost, Tuesday, April 13, 2010 http://select.nytimes.com/gst/abstract.html?res=FA0913FA3C5E147B93CAAB178CD85F468685F9
26. Patterson,, K., Grenny, J. Maxfield, D. McMillan, R. & Switzler, A. (2008). *Influencer*, McGraw-Hill, New York.
27. Ritchie, D. M., Johnson, S. C., Lesk, M. E. & Kernighan, B. W. (1978). The C Programming Language, *The Bell System Technical Journal*, **57**(6), pp1998–2019.
28. Spafford, E. (1989) Crisis and aftermath, *Communications of the ACM*, **32**(6), pp678–687.
29. Summers, S. (2004). How many lightbulbs does it take to change an engineer? *Aircraft Engineering and Aerospace Technology*, **76**(1), pp47–50.
30. Tavris, C. & Aronson, E. (2008). *Mistakes Were Made (but Not by Me): Why We Justify Foolish Beliefs, Bad Decisions and Hurtful Acts*, Pinter & Martin Ltd.
31. Thimbleby, H. (1990). You're Right About the Cure: Don't Do That, *Interacting with Computers*, **2**(1), pp8–25.
32. Thimbleby, H. (1999). Java: A Critique, *Software—Practice & Experience*, **29**(5), pp457–478.
33. Thimbleby, H. (2000). Calculators are needlessly bad, *International Journal of Human-Computer Studies*, **52**(6), pp1031–1069.
34. Thimbleby, H. (2008). Ignorance of interaction programming is killing people, *ACM Interactions*, pp52–57, September+October.
35. Thimbleby, H. & Cairns, P. (2010). Reducing number entry errors: solving a widespread, serious problem, *Journal of the Royal Society Interface*, **7**(51), pp1429–1439, doi:10.1098/rsif.2010.0112.
36. Thimbleby, H. (2010). Signposting in documents, *Computer Journal*, **54**(7), pp1119–1135.
37. Thimbleby, H. (2010). Think! Interactive Systems Need Safety Locks, *Journal of Computing and Information Technology*, **18**(4), pp349–360.
38. Thompson, K. (1984). Reflections on Trusting Trust, *Communications of the ACM*, **27**(8), pp761–763.
39. Wickens, C. D. & Hollands, J. G. (2000). *Engineering Psychology and Human Performance*, Third Edition, London: Prentice Hall International.
40. Wilkes, M. V., Quoted in Campbell-Kelly, M. (2011). Historical Reflections: In Praise of 'Wilkes, Wheeler, and Gill,' *Communications of the ACM*, **54**(9), pp25–27.

41. Wilkes, M. V., Wheeler, D. J. & Gill, S. (1982 reprint). *The Preparation of Programs for an Electronic Digital Computer: With Special Reference to the EDSAC and the Use of a Library of Subroutines*, Tomash Publishers.
42. Woodcock, J., Larsen, P. G. & Fitzgerald, J. (2009). Formal Methods: Practice and Experience, *ACM Computing Surveys*, **41**(4), pp19:1–19:36.