Marc Eisenstadt

My Hairi Bug Wa

STUDIES OF THE PSYCHOLOGY OF COMPUTER PROGRAMming and debugging [1, 2, 5, 8–12], while important in their own right, have generally overlooked the potential benefit of self-reports by programmers that reflect the phenomenology of debugging, that is, what it's really like out there from the programmer's perspective. However, two exceptions to this observation are the detailed account by Knuth of using a logbook to document all the errors he encountered over a 10-year development period working on TeX [6] and a logbook of the development efforts of a team implementing the Smalltalk-80 virtual machine [7]. Such self-reports and logs are valuable sources on the nature of software design, development, and maintenance. The work reported here seeks to expand the single-user-logbook approach to investigate the phenomenology of debugging across a large population of users, aiming to understand and address the problems of professional programmers working on very large programming tasks.

Toward this end, I conducted a survey of professional programmers, asking them to describe their most difficult bugs involving large pieces of software. The survey was conducted through email and conferencing/bulletin board facilities with worldwide access, including Usenet newsgroups, the *BYTE* Information Exchange (BIX), CompuServe, and AppleLink. My contribution was to gather, edit, and annotate the stories, categorizing them in a way that may shed light on the nature of the debugging enterprise. In particular, I looked at the lessons learned from the stories. Here, I discuss what they tell us about what is needed in the design of future debugging tools.

Test Stories

Despite the availability of industrial-strength debuggers and integrated program development environments, professional programmers still engage in far more detective work than they should have to. This is their story—along with some of the ways they might eradicate their bugs.

The Trawl

In early 1992, I posted a request for debugging anecdotes on the BIX electronic bulletin board and followed with similar messages posted to AppleLink, CompuServe, various Usenet newsgroups, and the Open University's own conferencing system. The original message is shown in Figure 1.

The trawl request elicited replies from 78 "informants," mostly in the U.S. and the U.K., including implementors of very well known commercial C compilers, members of the ANSI C++ definition group, and other commercial software developers. A total of 110 messages were generated by 78 different informants. Of these, 50 specifically told a story about nasty bugs. A few informants provided many anecdotes; a total of 59 bug anecdotes were collected. Figure 2 shows some typical replies to the original request. The full set of replies and analyses is available from the author.

> Subject: Trawl for debugging anecdotes (w/emphasis on tools side)
>
> I'm looking for some (serious) anecdotes describing debugging experiences. I want to know about particularly thorny bugs in large pieces of software that caused you lots of headaches. It would be handy if the software were written in C or C++, but this is not absolutely essential. I'd like to know how you cracked the problem—what techniques/tools you used. Did you "home in" on the bug systematically? Did the solution suddenly come to you in your sleep? A very brief stream-of-consciousness reply (right now) would be much much better than a carefully crafted story. I can then get back to you with further questions if necessary.
>
> Thanks.
> Marc

Figure 1. The original "trawl" request

Analyzing the Anecdotes

Our analysis included three dimensions: "why difficult," "how found," and "root cause." Although the root cause of reported bugs is of a priori interest, in order to fully characterize the phenomenology of the debugging experiences, I looked at more than the causes of the bugs. After summarizing the data several times, it became apparent that it would be necessary to say something about why a bug was difficult to find (which might or might not be related to the underlying cause) and how it was found (which might or might not be related to the underlying cause and the reason for the difficulty), as well as the root cause (what really went wrong).

We know something about each of these dimensions from earlier studies. Vessey [12] attempted to address the first dimension (why difficult) by asking how the time to find a bug depended on its location in a program's structure and its level in a propositional analysis of the program; his answers were that location in serial structure has no effect and that level in propositional structure is inconclusive. Regarding techniques for bug finding (the second dimension),

(Story A)
I had a bug in a compiler for an 8086 running MS-DOS. The compiler returned function values on the stack and once in a while such values would be wrong. When I looked at the assembly code, all seemed fine. The value was getting stored at the correct location of the stack. When I stepped through it in the assembly-level debugger and got to that store, sure enough, the effective address was correct in the stack frame, and the right value was in the register to be stored. Here's the weird thing: When I stepped through the store instruction, the value on the stack didn't change. It seems obvious in retrospect, but it took hours for me to figure out that the effective address was below the stack pointer (stacks grow down here), and the stored value was being wiped out by OS interrupt handlers (which don't switch stacks) about 18 times a second. The stack pointer was being decremented too late in the compiled code.

(Story B)
. . . I once had a program that worked properly only on Wednesdays . . . The documentation claimed the day of the week was returned in a doubleword, 8 bytes. In fact, Wednesday is 9 characters long, and the system routine actually expected 12 bytes of space to put the day of the week. Since I was supplying only 8 bytes, it was writing 4 bytes on top of the storage area intended for another purpose. As it turned out, that space was where a "y" was supposed to be stored for comparison with the user's answer. Six days a week the system would wipe out the "y" with blanks, but on Wednesdays, a "y" would be stored in its correct place.

(Story C)
. . . The program crashed after running about 45,000 iterations of the main simulation loop . . . Somewhere, somehow, someone was walking over memory. But that somewhere could have been anywhere—for example, writing in one of the many global arrays . . . The bug turned out to be a case of an array of shorts (max. value 32K) with certain elements incremented every time they were "used"—the fastest use being about every 1.5 iterations of the simulator. So, an element of an array would be incremented past 32K, back down to -32K. This value was then used as an array index . . . But the actual seg fault was happening several iterations after the error—the bogus write into memory. It took 3 hours for the program to crash, so creating test cases took forever. I couldn't use any of the heavier powered debugging malloc()s or watchpoints, because they slow a program down at least 10-fold, resulting in 30 hours to track a bug.

Figure 2. Typical debugging anecdotes

Katz and Anderson [5] reported a variety of bug-location strategies used by experienced Lisp programmers in a laboratory setting involving small (10-line) programs. These programmers distinguished among strategies that detect a heuristic mapping between a bug's manifestation and its origin, those that rely on a hand simulation of execution, and those that resort to some kind of causal reasoning. Goal-driven reasoning (either heuristic mapping or causal reasoning) predominated among subjects debugging their own code, whereas data-driven reasoning (typically hand simulation) predominated among subjects debugging other programmers' code. For the kind of programming-in-the-large being studied here, the need for a bottom-up data-gathering phase is apparent—to help the programmer get some approximate notion of where the bug might be located.

As far as root causes are concerned, two main approaches to the development of bug taxonomies have been followed: a deep plan analysis approach (e.g., [4, 11]) and a phenomenological account (e.g., [6]). Johnson [4] worked on the premise that a large number of bugs could be accounted for by analyzing the high-level abstract plans underlying specific programs and specifying both the possible fates a plan component could undergo (e.g., missing, spurious, misplaced) and the nature of the program constructs involved (e.g., inputs, outputs, initializations, conditionals). Spohrer et al. [11] refined this analysis by pointing out the critical nature of bug interdependencies and problem-dependent goals and plans. An alternative characterization of bugs was provided by Knuth [6], who uncovered nine problem-independent categories:

- Algorithm awry
- Blunder or botch
- Data structure debacle
- Forgotten function
- Language liability, or the misuse or misunderstanding of the tools/language/hardware, or imperfectly knowing the tools
- Mismatch between modules, or imperfectly knowing the specifications (e.g., interface errors involving functions called with reversed arguments)
- Reinforcement of robustness (e.g., handling erroneous input)
- Surprise scenario, or bad bugs that forced design change or unforeseen interactions
- Trivial typo

For both approaches—plan analysis and phenom-

enology—the "true" cause of a bug can be resolved only by the original programmer, because it is necessary to understand the programmer's state of mind at the time the bug was spawned to be able to assess the cause properly. I found it informative to evolve my own categories in a largely bottom-up fashion after extensive inspection of the data and specific comparison with those provided by Knuth. I adopted one criterion for identifying root causes: When the programmer is essentially satisfied that several hours or days of bewilderment have come to an end once a particular culprit is identified, that culprit is the root cause, even when deeper causes can be found. I adopted this approach because it allows a possible infinite regress to be nipped in the bud; because it is consistent with my emphasis on the phenomenology of debugging, that is, what is apparently taking place as far as the front-line programmer is concerned; and because it enables me to concentrate on what the programmers reported and not try to second-guess them.

Dimension 1 (Why Difficult). The reasons a bug is difficult to trap fell into five categories:

- Cause/effect chasm. The symptom is often far removed in space or time from the root cause, possibly making the cause difficult to detect. Specific instances can involve timing or synchronization problems; bugs that are intermittent, inconsistent, or infrequent; and bugs that materialize "far away" (e.g., thousands of iterations) from the place they were spawned.
- Tools inapplicable or hampered. Most programmers have encountered so-called *Heisenbugs*, named after the Heisenberg uncertainty principle in physics. The bug goes away, for example, when you switch on the debugging tools. Other variations in this category are *stealth bugs*, in which the error itself consumes the evidence, and *context precludes*, in which some configuration or memory constraints make it impractical or impossible to use the debugging tool.
- WYSIPIG (what you see is probably illusory, guv'nor). I coined this expression to reflect the cases in which the programmer stares at something that simply is not there or that is dramatically different from what it appears to be (e.g., 10 in an octal display misinterpreted as meaning 7+3 rather than 7+1).
- Faulty assumption/model or misdirected blame. If you think stacks grow up rather than down (as did the informant in Story A in Figure 2), you will have difficulty detecting bugs related to this behavior.
- Spaghetti (unstructured) code. Informants sometimes complain about "ugly" code invariably written by "someone else."

Here I report the frequency of occurrence of the different categories, not because it supports an a priori hypothesis at some level of statistical significance, but because it provides a convenient overview of the nature of the problems the informants chose to share with us. The frequency of occurrence of the different reasons for having difficulty is shown in Table 1. Thus, 53% of the difficulties are attributable to just two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools inapplicable. The high frequency of reports of cause/effect chasms accords well with the analyses of Vessey [12] and Pennington [8], who argued that the programmer must form a robust mental model of correct program behavior in order to detect bugs, but the cause/effect chasm seriously undermines the programmer's efforts to construct a robust mental model. The relationship of this finding to the analysis of the other dimensions is addressed in the following sections.

Table 1. Why the bugs were difficult to track down

Category	Occurrences
Cause/Effect Chasm	15
Tools Inapplicable or Hampered	12
WYSIPIG (What you see is probably illusory, guv'nor)	7
Faulty Assumption/Model or Misdirected Blame	6
Spaghetti (Unstructured) Code	3
??? (*No Information*)	8

Dimension 2 (How Found). The informants reported four major bug-catching techniques:

- Gather data. This technique refers to cases in which informants decided to "find out more," by, say, planting print statements or breakpoints. They reported six subtechniques:
  - Step & Study. The programmer single-steps through the code and studies the behavior, typically monitoring changes to data structures
  - Wrap & Profile. Tailor-made performance, metric, or other profiling information is collected by "wrapping" (enclosing) a suspect function inside a one-off variant of that function that calls for, say, a timer or data-structure printout both before and

When the programmer is satisfied that several hours or days of bewilderment have come to an end once a particular culprit is identified, that culprit is the root cause, even when deeper causes can be found.

after the suspect function.

– Print & Peruse. Print statements are inserted at particular points in the code, and their output is observed during subsequent runs of the program.
– Dump & Diff. Either a true core dump or else some variation (e.g., extensive print statements) is saved to two text files corresponding to two different execution runs; the two files are then compared using a source-compare ("diff") utility.
– Conditional Break & Inspect. A breakpoint is inserted into the code, typically triggered by some specific behavior; data values are then inspected to determine what is happening.
– Specialist Profile Tool (e.g., MEM or Heap Scramble). Several off-the-shelf tools detect memory leaks and corrupt or illegal memory references.

• Inspeculation. This name is a hybrid of "inspection" (code inspection), "simulation" (hand simulation), and "speculation," which were among a variety of techniques mentioned explicitly or implicitly by informants. In other words, either they go away and think about something else for a while or they spend a lot of time reading through the code and thinking about it, possibly hand-simulating an execution run.
• Expert Recognized Clichés. These represent cases in which the programmer called upon a cohort for help, and the cohort was able to spot the bug relatively simply. Such recognition corresponds to the heuristic mapping reported in [5].
• Controlled Experiments. Informants resorted to specific controlled experiments when they had a clear idea about the root cause of a bug.

THE FREQUENCY OF OCCURrence of the different debugging techniques is shown in Table 2. Techniques for bug-finding are clearly dominated by reports of data-gathering (e.g., print statements) and hand simulation, which together account for 78% of the reported techniques and highlight the kind of "groping" the programmer is reduced to in difficult debugging situations. Here we turn to an analysis of the root causes of the bugs before we show how the different dimensions are interrelated.

Table 2. Techniques used to track down the bugs

Category	Occurrences
Gather Data	27
Inspeculation	13
Expert Recognized Cliché	5
Controlled Experiments	4
??? (No Information)	2

Dimension 3 (Root Cause Categories). The bug causes reported by the informants fell into nine categories:

• Mem (memory clobbered or used up). This cause has a variety of manifestations (e.g., overwriting a reserved portion of memory, thereby causing the system to crash) and may even have deeper causes (e.g., array subscript out of bounds), yet was often singled out by the informants as the source of the difficulty. Knuth has an analogous category he calls Data Structure Debacle.
• Vendor (vendor's problem, hardware or software). Some informants report buggy compilers or faulty logic boards for which they either need to develop a workaround or wait for the vendor to provide corrective measures.
• Des.logic (unanticipated case, faulty design logic). In such cases, the algorithm itself has gone awry because the programmer has not worked through all the cases correctly. This category encompasses both those Knuth labels Algorithm Awry and those he labels Surprise Scenario.
• Init (wrong initialization, wrong type, definition clash). A programmer sometimes makes an erroneous type declaration, redefines the meaning of some system keyword, or incorrectly initializes a

variable. I refer to all of these as "init" errors.

- Var (wrong variable or operator). The wrong term is used. The informant may not provide enough information to deduce whether this error is due to faulty design logic (des.logic) or is a trivial lexical error (lex), though in the latter case trivial typos are normally mentioned explicitly as the root cause.
- Lex (lexical problem, bad parse, or ambiguous syntax). These are trivial problems, not due to the algorithm itself or to faulty variables or declarations. This class of errors encompasses Knuth's Blunder and Typo and are difficult to distinguish in the informants' reports.
- Unsolved (unknown and still unsolved to this day). Some informants never solved their problems.
- Lang (language semantics ambiguous or misunderstood). In one case, an informant reported he thought 256K meant 256,000, which is incorrect and can be thought of as semantic confusion.
- Behav (the end user's or programmer's subtle behavior). In one case, the bug was caused by an end user's mysteriously depressing several keys on the keyboard at once; in another case, the cause was mischievous code inserted as a joke.

Table 3 shows the frequency of occurrence of these nine underlying causes, identifying the biggest culprits as memory overwrites and problems with vendor-supplied hardware/software. Even ignoring vendor-specific difficulties, an implication of Table 3 is that 37% of the nastiest bugs reported by professionals could be addressed by memory-analysis tools and smarter compilers that trap initialization errors.

Table 3. Underlying causes of the reported bugs

Category	Occurrences
mem Memory clobbered or used up	13
vendor Vendor's problem (hardware or software)	9
des.logic Unanticipated case (faulty design logic)	7
init Wrong initialization; wrong type; definition clash	6
lex Lexical problem, bad parse, or ambiguous syntax	4
var Wrong variable or operator	3
unsolved Unknown and still unsolved to this day	3
lang Language semantics ambiguous or misunderstood	2
behav End user's (or programmer's) subtle behavior	2
??? (No Information)	2

Relating the Dimensions

To understand how the three dimensions of analysis are interrelated, we can place every anecdote precisely in our three-dimensional space. For expository purposes, consider just a single two-dimensional comparison: how found vs. why difficult. Table 4 compares reasons for difficulty (row labels) against bug-finding techniques (column labels). Cells with large numbers are noteworthy; indeed, the magnitude of certain cell entries is greater than what is predictable by chance from the row and column totals alone (X^2,df:20, = 33.50, p. < 0.05), suggesting that data-gathering activities are of special relevance when a cause/effect chasm is involved or when built-in debugging tools are inapplicable.

Table 4. Tally of why bugs were difficult (rows) vs. how they were found (columns). (Each cell entry [e.g., 9.83] is a tally of the number of anecdotes reporting that cell's row label, or the bug's root cause, and column label, or how the bug was found. Fractional entries reflect anecdotes divided into multiple categories, so an anecdote reporting three reasons for difficulty scores 0.33 in each of the three relevant cells.)

Why vs. How	Gather Data	Inspeculation	Expert Recognized Cliché	Controlled Experiments	??? (No Info)	Totals
Cause/Effect Chasm	9.83	3.00	1.50	2.50		**16.83**
Tools Hampered	9.83	2.00		2.00		**13.83**
WYSIPIG	2.00	2.00	1.50		2.00	**7.50**
Faulty Assumption	2.50	3.00	1.00			**6.50**
Spaghetti	1.33	1.00				**2.33**
??? (No Information)	4.00	2.00	2.00			**8.00**
Totals	**29.50**	**13.00**	**6.00**	**4.50**	**2.00**	**55.00**

More than 50% of the difficulties are attributable to just two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools inapplicable.

A niche of potential interest (and profit) to tool vendors is highlighted by looking at the relationships among the three dimensions; the most heavily populated cells involve data-gathering, cause/effect chasms, and memory or initialization errors.

A side effect of our study is the realization that complete strangers, with little prompting and no incentive, are not only articulate in their reminiscences but forthcoming with details. These people clearly enjoyed relating their debugging experiences. Moreover, the depth of the details supplied seemed to be independent of whether I had explicitly posted my motivation (as I did on BIX and AppleLink) or not (as was the case on Usenet and CompuServe). This is clearly a self-selecting audience of email users and conference browsers who enjoy electronically chatting anyway; some may have even felt an inner need to tell a good (hence boastful) war story—so much the better. I have no reason to distrust the sources, and the detailed stories certainly exhibit their own self-consistency. It is already widely accepted that the Internet is a gold mine of information. Our collection of anecdotes suggests it may also be a rich repository of willing subjects ready to supply in a fairly rigorous manner detailed knowledge that may then serve as a resource for others. These stories, even without a definitive taxonomy, could provide a valuable adjunct to frequently asked question (FAQ) repositories on the World-Wide Web and in such growing Tbyte archives of stories and postings as those at http://www.dejanews.com. However, while FAQ and generic Usenet discussion-group repositories are wonderful resources, they can be frustrating to access when an urgent debugging need arises.

Ways Forward

It would be easy to say that what programmers really need are more robust design approaches, plus smarter compilers and debuggers, Fortunately, the analyses presented here suggest we can be more precise than simply demanding "robustness" from programmers/designers and "smartness" from tool developers. We have identified a niche that really needs attention; the most heavily populated cell in our three-dimensional analysis suggests that a winning tool would employ some data-gathering or traversal method for resolving large cause/effect chasms in the case of memory-clobbering errors (e.g., Purify from Atria, Inc. does precisely this). Secondly, we can propose solutions to the "why difficult" problems by considering the specific cases brought to light by the stories themselves. One way or another, most of the problems in the stories are connected with "directness" and "navigation." For example, the need to go through indirect steps, intermediate subgoals, or obtuse lines of reasoning plagues the user encountering the most frequent problems in Table 1. A possible way forward, described in more detail in a comparative fine-grained analysis I undertook in [3], involves paying heed to the following advice: *Computable relations should be computed on request rather than deduced by the user*. A software tool can perform important and complicated deductions on the programmer's behalf, thereby liberating the programmer from some tedious work.

Purify analyzes run-time memory leaks in C programs (e.g., lost memory cells, overflowed arrays) by patching the object code at link time and pinpoints the root cause of the leak by traversing many indirect dataflow links back to the offending source code. Thus, it already solves a much more difficult dataflow traversal problem than that required to deal with indirect pointer traversing, such as that reported by several informants, and suggests a highly promising direction for development of future tools. Memory leaks will also be less prevalent as programmers adopt such languages as Java that prevent arbitrary memory access.

Other advice includes: *Displayable states should be displayed on request rather than having to be deduced by the user.* Minimizing deductive work is an important aspect of such tools as Zstep 95, described by Ungar et al. in this issue. And: *Atomic user goals should be mapped onto atomic actions.* In other words, the tool should try to infer the programmer's likely intentions, so requently occurring "reasonable" behaviors on the part of the programmer can be anticipated, yielding a concomitant reduction in wasteful "fine-tuning" activities (e.g., those requiring the pro-

grammer to deal with digressions and irrelevant subgoals just to get the tools working). As detailed in Fry's article in this issue, there are several key steps in this direction:

- *Allow full functionality at all times.* Debugging environments that prevent access to certain facilities make matters worse. (The Kansas/Oz environment described in Smith et al.'s article in this issue pushes this notion to its logical limits.)
- *Viewers should be provided for "key players" (any evaluable expression) rather than just for "variables."* Several articles in this issue, particularly Baecker et al.'s, take to heart the notion that more than variable-watching is at issue during the debugging process.
- *Provide a variety of navigation tools at different levels of granularity.* Changing granularity is a hallmark of the system underlying the work of Domingue et al. described in this issue, allowing programmers to see appropriate views at appropriate times.

These suggestions are not necessarily easy to implement, but an increasing number of tools are appearing in both the research community and the marketplace, illustrating some of their key aspects. These suggestions indicating that specific debugging needs can be addressed systematically and that a detailed account of programmers' continuing problems is an important step in facilitating the evolution of appropriate solutions.

Conclusions

An analysis of the debugging anecdotes collected from a worldwide email trawl revealed three primary dimensions of interest: why the bugs were difficult to find, how the bugs were found, and root causes of bugs. Just over 50% of the difficulties arose from two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools inapplicable. Techniques for bug-finding were dominated by reports of data-gathering (e.g., print statements) and hand simulation, together accounting for almost 80% of the reported techniques. The two biggest causes of bugs were memory overwrites and faults in vendor-supplied hardware or software, together accounting for more than 40% of reported bugs.

The analysis also pinpoints a winning niche for future tools: data-gathering or traversal methods to resolve large cause/effect chasms in the case of memory-clobbering errors. Other specific suggestions emerge by analyzing the underlying issues of "directness" and "navigation." The investigation highlights a potential wealth of information available on the Internet and indicates it may be possible to establish an online repository for perusal by those with an urgent need to solve complex debugging problems. An indexed repository could organize stories in a manner more accessible than that found in FAQ anecdotes. C

Acknowledgments

Parts of this research were funded by the U.K. EPSRC/ESRC/MRC Joint Council Initiative on Cognitive Science and Human Computer Interaction by the Commission of the European Communities ESPRIT-II Project 5365 (VITAL), and by Apple Computer's Advanced Technology Group, now Apple Research Labs. Simon J. Masterton helped collect and analyze the data.

References

1. Brooks, R.E. Studying programmer behavior experimentally: The problems of a proper methodology. *Commun. ACM 23*, 4 (Apr. 1980), 207–213.
2. Curtis, W. By the way, did anyone study any real programmers? In E. Soloway, and S. Iyengar, Eds. *Empirical Studies of Programmers*. Ablex, Norwood, N.J., 1986.
3. Eisenstadt, M. Why HyperTalk debugging is more painful than it ought to be. In J. Alty, D. Diaper, and S.P. Guest, Eds. *People and Computers 8th Ed*. Cambridge University Press, Cambridge, England, 1993.
4. Johnson, W.L. An effective bug classification scheme must take the programmer into account. In *Proceedings of the Workshop on High-Level Debugging* (Palo Alto, Calif., 1983).
5. Katz, I.R., and Anderson, J.R. Debugging: An analysis of bug-location strategies. *Hum.–Comput. Interaction 3*, 4 (Apr. 1988), 351–399.
6. Knuth, D.E. The errors of TeX. *Software—Pract. Exper. 19*, 7 (Jul. 1989), 607–685.
7. McCullough, P.L. Implementing the Smalltalk-80 System: The Tektronix experience. In *Smalltalk-80: Bits of History, Words of Advice*. G. Krasner, Ed. Addison-Wesley, Reading, Mass., 1983, pp. 59–78.
8. Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychol. 19* (1987), 295–341.
9. Shneiderman, B. *Software Psychology*. Winthrop, Cambridge, Mass., 1980.
10. Soloway, E., and Iyengar, S., Eds. *Empirical Studies of Programmers*. Ablex, Norwood, N.J., 1986.
11. Spohrer, J.C., Soloway, E., and Pope, E.A. Goal/plan analysis of buggy Pascal programs. *Hum.-Comput. Interaction 1*, 2 (Feb. 1985), 163–207.
12. Vessey, I. Toward a theory of computer program bugs: An empirical test. *Int. J. Man-Mach. Stud. 30* (1989), 123–46.

Marc Eisenstadt (m.eisenstadt@open.ac.uk) is a professor of Artificial Intelligence and Director of the Knowledge Media Institute at The Open University in Milton Keynes, U.K.