Tales of Debugging from The Front Lines

Marc Eisenstadt

Paper Submitted to Empirical Studies of Programmers V,1993

Abstract

A worldwide trawl for debugging anecdotes elicited replies from 78 respondents, including a number of implementors of well-known commercial software. The stories included descriptions of bugs, bug-fixing strategies, discourses on the philosophy of programming, and several highly amusing and informative reminiscences. Experiences included using a steel ruler to debug a COBOL line printer listing, browsing through a punched card deck to debug an early Fortran compiler, and struggling in vain to find intermittent bugs on popular commercial products. An analysis of the anecdotes reveals three primary dimensions of interest: *why the bugs were difficult* to find, *how* the bugs were found, and *root causes* of bugs. Half of the difficulties arose from just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. Techniques for bug-finding were dominated by reports of data-gathering (e.g. print statements) and hand-simulation, which together accounted for almost 80% of the reported techniques. The two biggest causes of bugs were (i) memory overwrites and (ii) vendor-supplied hardware or software faults, which together accounted for more than 40% of the reported bugs. The paper discusses the implications of these findings for the design of program debuggers, and explores the possible role of a large repository/data base of debugging anecdotes.

Acknowledgements

Apple Computer, Inc.'s Advanced Technology Group provided a supportive and productive environment during the summer of 1992, enabling me to undertake this analysis along with other concomitant activities. Parts of this research were also funded by the UK SERC/ESRC/MRC Joint Council Initiative on Cognitive Science and Human Computer Interaction, and by the Commission of the European Communities ESPRIT-II Project 5365 (VITAL). Special thanks to Mark L. Miller (Apple Inc.) and those at Apple who provided detailed constructive criticisms or were generally helpful: Adam Chipkin, Sasha Karasik, Jim Spohrer, Gillian Crampton Smith, Dave Canfield Smith, Ted Kaehler, Jerry Morrison, and Ruben Kleiman. Mike Brayshaw and Blaine A. Price (Open University) provided valuable comments on several drafts of this paper. Blaine A. Price posted the trawl request on Usenet and collected the associated replies.

Preface

Programmers deserve better debugging tools. There are many excellent integrated program development environments available commercially, but professional programmers still have to engage in far more detective work than they ought to. How do I know? I asked them! This paper summarizes what they told me. Although the evidence is anecdotal and has been collected by a "pot luck" worldwide email trawl, it nevertheless provides a first-pass stab at a broader phenomenology of debugging, particularly from the perspective of professional programmers.

Introduction

At the first Workshop on the Empirical Studies of Programmers, Bill Curtis asked "By the way, did anyone study any real programmers?" (Curtis, 1987). Interestingly, despite the large number of both pre-Curtis and post-Curtis psychological studies of computer programming and debugging (e.g. Sime *et al*., 1973; Gould and Drongowski, 1974; Brooks, 1980; Shneiderman, 1980; Kahney & Eisenstadt, 1982; Soloway & Iyengar, 1986; Katz & Anderson, 1988; Vesey, 1989) there is still a surprising shortage of self-reports by programmers on the phenomenology of debugging–i.e. what it's like "out there in the trenches" from the programmer's perspective. Two exceptions to this are (a) the detailed account by Knuth of the log book that documented all the errors he encountered over a ten-year development period working on TEX (Knuth, 1989), and (b) a log book of the development efforts of a team implementing the Smalltalk-80 virtual machine (McCullough, 1983). Self-reports and log books like that of Knuth and McCullough are valuable sources of insight into the nature of software design, development, and maintenance. Moreover, they can serve as handbooks or guidelines for others by providing case study remiscences which highlight the strengths, weaknesses, or even outright dangers of certain approaches.

The work reported here attempts to expand the single-user-log-book approach of Knuth (who provided a much more detailed analysis than did McCullough) to investigate the phenomenology of debugging across a large population of users. The original aim of the study was to inform ongoing research in my laboratory on the development of program visualization and debugging tools (Eisenstadt *et al*., 1990; 1992; 1993), for which it seemed essential to have first-hand knowledge of the experiences of a large body of professional programmers. We had our own experiences to draw on, including extensive work with novice programmers, but it seemed important to look further afield. In particular, we had been growing increasingly concerned about the problems of scalability: academic work on program visualization and debugging environments would remain vulnerable to criticisms of "toy examples only" unless an attempt was made to address the problems faced by professional programmers working on very large programming tasks.

Toward this end, I conducted a survey of professional programmers, asking them to provide stories describing their most difficult bugs involving large pieces of software. The survey was conducted by electronic mail and

conferencing/bulletin board facilities with wordwide access (BIX, CompuServe, Internet Usenet, AppleLink). My contribution is to gather, edit and annotate the stories, and to categorize them in a way which may help to shed some light on the nature of the debugging enterprise. In particular, I look at the lessons learned from the stories, and discuss what they tell us about what is needed in the design of future debugging tools. I also explore the possible role of a large repository/data base of debugging anecdotes. The discussion throughout concentrates on the relationships among three critical dimentsions: (a) why the bugs were hard to find (c) how the bugs were found, and (c) the root causes of the bugs.

The Trawl

Raw data

On 3 March, 1992, I posted a request for debugging anecdotes on an electronic bulletin board called BIX. BIX is The “BYTE Information Exchange” hosted on a computer network centred in Boston, Massachusetts, and accessed by approximately 60,000 users, mostly in the USA. BIX has a reputation for strong technical expertise, and it was for this reason that I chose BIX to start with. The original message is shown in figure 1.

```
c.language/tools #2842, from meisenstadt
771 chars, Tue Mar  3 06:09:28 1992
Comment(s).

TITLE: Trawl for debugging anecdotes (w/emphasis on tools side)...

I'm looking for some (serious) anecdotes describing debugging
experiences.  In particular, I want to know about particularly thorny
bugs in LARGE pieces of software which caused you lots of headaches.  It
would be handy if the large piece of 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,
etc.  A VERY brief stream-of-consciousness reply (right now!) would be
much much better than a carefully-worked-out story.  I can then get back
to you with further questions if necessary.

Thanks!

-Marc
```

Figure 1. The original “trawl” request, posted on BIX. c.language is the name of the conference (major subject category), and “tools” is the name of the topic (minor interest sub-category). BIX has 60,000 members, hundreds of conferences and thousands of topics, each with thousands of messages and replies. A second message, explaining my motivation, was also posted.

During the ensuing months, similar messages were then posted to AppleLink, CompuServe, various Usenet newsgroups on Internet, and the Open University’s own conferencing system (OU CoSy). The trawl request elicited replies from 78 “informants”, mostly in the USA and UK. The group included implementors of very well-known commercial C compilers, members of the ANSI C++ definition group, and other known commercial software developers.

Figure 2 shows a typical reply to the original request. In that reply, the informant describes many important facets of his debugging experience: (a) the background context (MS-DOS 8086 compiler), (b) the symptom (wrong value on stack); (c) how he approached the problem (single-stepping using assembly-level debugger); (d) why it was hard (the change happened more quickly at run-time than he could observe while single-stepping, plus he had the wrong model of which way the stacks grew, which made it harder to understand the behaviour); and (e) what the root cause of the problem was (the address *below* the stack pointer was being wiped out by the operating system interrupt handlers, and the pointer was being decremented too late in the compiled code).

```
I had a bug in a compiler for 8086's running MSDOS once that stands out
in my mind.  The compiler returned function values on the stack and once
in a while such a value 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 thru 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 some 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 (that
don't switch  stacks) about 18 times a second.  The stack pointer was
being decremented too late in the compiled code.
```

Figure 2 A typical debugging anecdote. A representative sample of the raw data is presented in Appendix A.

A total of 110 messages were generated by 78 different informants. Of those, 50 informants specifically told a story about a nasty bug. A few informants provided several anecdotes, and in all a total of 59 bug anecdotes were collected. A second collection of anecdotes, which will be used subsequently for a top-down analysis by comparing it against the first collection, yielded an additional 58 messages from 47 more informants, of whom 45 told specific debugging stories. A few representative raw anecdotes are presented in Appendix A. The next sections presents a detailed summary of the raw data, followed by an analysis and discussion of the lessons learned.

Condensed data

The stories and discussions which were posted on the various networks made fascinating reading, but they needed to be digested in some way to make them more meaningful and accessible. I entered summaries of the incoming data into a large spreadsheet-cum-database, experimenting with the use of a variety of different data fields in an attempt to succinctly characterize the data. Table B-1 in Appendix B is an excerpt from that database, selected to show (a) the five fields (other than ID number) which emerged as persistent and relevant throughout the study, and (b) all 36 of the anecdotes which contained enough detail to yield an entry in each field. The relevant entry for the anecdote shown earlier in Figure 2 can be found in Table B-1 alongside ID "U6".

The contents of Table B-1 is itself the result of several iterations of the analyses reported in the next sections. The reader may find it useful to browse the

appendices before proceeding further in order to gain an overview of the style and content of the informants reports.

Analysis of the anecdotes

Dimensions of analysis: why difficult, how found, and root cause

Although the "root cause" of reported bugs is of *a priori* interest, in order to fully characterise the phenomenology of the debugging experiences I needed to look at more than the causes of the bugs. In particular, it was necessary to say something about why a bug was hard 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). Thus, three dimensions became the critical focus for the ensuing analysis:

- *Why difficult:* This identifies the reason that the debugging experience itself was tricky or painful, e.g. perhaps the bug rendered the debugging tools unusable.

- *How found:* This identifies the diagnostic methodologies or techniques that were used in resolving the problem, e.g. single-stepping the code or inserting hand-tailored print statements.

- *Underlying cause of bug:* This identifies the root cause of the bug which, when fixed, means that the programmer has either totally solved the problem or else has gone far enough to regard the problem as being "in hand".

We know something about each of these dimensions from previous studies, although only Knuth's study really addresses the phenomenology of "debugging-in-the-large". Vesey (1989) attempted to address the first dimension (why difficult) by asking how the time to find a bug depended upon its location in a program's structure and its level in a propositional analysis of the program (answers: location in serial structure has no effect, and level in propositional structure is inconclusive). Regarding techniques for bug finding (second dimension), Katz and Anderson (1988) reported a variety of bug-location strategies among experienced Lisp subjects in a laboratory setting. In particular, they distinguished among (i) strategies which detected a heuristic mapping between a bug's manifestation and its origin, (ii) those which relied on a hand simulation of execution, and (iii) those which resorted to some kind of causal reasoning. Goal-driven reasoning (either heuristic mapping or causal reasoning) was predominant among subjects who were debugging their own code, whereas data-driven reasoning (typically hand simulation) was predominant among subjects who were debugging other programmers' code. In all cases bug fixing was not particularly problematic, once the bugs were located. However, that study involved programs which were typically about 10 lines long, and it is worth noting that a whole new set of problems arise for programming-in-the-large (the informants who mention program size typically speak of thousands of lines). In particular, the need for a bottom-up data

gathering phase, which helps the programmer get some approximate notion of where the bug might be located, becomes apparent.

As far as root causes are concerned (dimension three), two main approaches to the development of bug taxonomies have been followed: a deep plan analysis approach (e.g. Johnson *et. al.*, 1983; Spohrer *et al*., 1985) and a phenomonelogical account (e.g. Knuth, 1983; du Boulay, 1986). Johnson *et. al.,* worked on the premise that a large number of bugs could be accounted for by analysing the high level abstract *plans* underlying specific programs, and specifying both the possible fates that 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. (1985) refined this analysis by pointing out the critical nature of bug interdependencies and at problem-dependent goals and plans. An alternative characterization of bugs was provided by Knuth's study. In particular, Knuth's analyses uncovered the following nine (problem-independent) categories: A= algorithm awry; B= blunder or botch; D= data structure debacle; F= forgotten function; L= Language liability, i.e. misuse or misunderstanding of the tools/language/hardware ("imperfectly knowing the tools"); M= Mismatch between modules ("imperfectly knowing the specifications", e.g. interface errors involving functions called with reversed args); R= Reinforcement of robustness (e.g. handling erroneous input); S= surprise scenario (bad bugs which forced design change, unforeseen interactions); T= Trivial typo.

For both approaches (plan analysis vs. phenomenological) the "true" cause of a bug can really only be resolved by the original programmer, because it is necessary to understand the programmer's state of mind at the time the bug was spawned in order to be able to assess the cause properly. For example, using the wrong variable could occur because the programmer really misunderstood the design of the algorithm, (i.e. he or she entered precisely the intended variable, but the intentions were mistaken, thereby falling into Knuth's category "A"), or it could be a lowly typographical error (Knuth's "T"). The manifestation is the same, but the root cause is different.

I found it informative to evolve my own categories in a largely bottom-up fashion after extensive inspection of the data, and then compare them specifically with the ones provided by Knuth. The comparison is provided in the far right hand column of Table B-1 (Appendix B) by means of a bracketed label such as "{A}" or "{L}" following relevant entries. In some cases, there is a straightforward mapping, but in others it is more subtle. For instance, there is sometimes a one-to-many mapping between my categories and those of Knuth. The reason is that Knuth knew his state of mind at the time he committed the errors reported in his log, whereas I have to rely on my interpretation of the programmer's anecdote. Even when the programmer's state of mind is clearly assessible, the assignment of blame can be awkward, because a given cause may always have an even deeper root cause. For example, the apparent root cause of a crash may be a memory address being overwritten by faulty data, and that faulty data itself may have been caused by a low level hardware fault (which itself may have a deeper cause relating to the time when coffee was spilled on the hardware in question, etc.).

The criterion I have adopted for identifiying root causes is as follows: when the programmer is essentially satisfied that several hours or days of bewilderment have come to an end once a particular culprit is identified, then that culprit is the root cause, even when deeper causes can be found. I have adopted this approach (a) because a possible infinite regress is nipped in the bud, (b) because it is consistent with my emphasis on the phenomenology of debugging, i.e. what is apparently taking place as far as the front-line programmer is concerned, (c) it enables me to concentrate on what the programmers *reported*, and not try to second-guess them. In practice, the main consequence of this is that the category "memory cloberred" is identified as the root cause of numerous other problems, even though the cloberring may itself have been caused by, say, a faulty array declaration. If the faulty declaration is reported in the anecdote, then I select that as the root cause, but if some variation of memory cloberring is deemed to be the culprit, then I accept that as the root cause.

The subsections which follow describe the three dimensions of analysis (why difficult; how found; root cause) in turn.

Dimension 1: Why difficult

Categories

The reasons that the bug was hard to trap fell into five categories, as described below:

- **cause/effect chasm:** Often the symptom is far removed in space and/or time from the root cause, and this can make the cause hard to detect. Specific instances can involve *timing* or *synchronization* problems, bugs which are *intermittent*, *inconsistent*, or *infrequent*, and bugs which materialize "far away" (e.g. thousands of iterations) from the actual place they are spawned. In this general category I have also included debugging episodes in which thre are *too many degrees of freedom***.** As an example of such an episode, consider the case in which a piece of software which works perfectly in one environment, yet fails to work in another environment. If many things have changed (e.g. different hardware, different compiler, different linker), then there are simply too many degrees of freedom to enable systematic testing under controlled conditions to isolate the bug. Such testing can be done, given enough time and resources, but it is difficult—hence this category.

- **tools inapplicaple or hampered:** Most programmers have encountered so-called "Heisenbugs", named after the Heisenberg uncertainty principle in physics: the bug goes away when you switch on the debugging tools! Other variations within this category are: *long run to replicate* (i.e. the bug takes a long time to replicate on a fresh execution, so if switching on the debugging tool significantly slows down execution, then it can not really be used); *stealth bug* (i.e. the error itself consumes the evidence that you need to find the bug, or even clobbers the debugging tool); *context precludes* (i.e. 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 have coined this expression to reflect the cases in which the programmer stares at something which simply is not there, or is dramatically different from what it appears to be. This can range from syntactic problems (e.g. hallucinating a key word in the code which is actually not there) to run-time observations (looking at a value on the screen which is displayed in a different way, e.g. "10" in an octal display being misinterpreted as meaning 7+3 rather than 7+1). Such observations lead the programmer on a wild-goose chase, and can be the reason why certain otherwise simple bugs take a long time to track down.

- **faulty assumption/model or mis-directed blame:** If you think that stacks grow up rather than down (as did the informant in Figure 2), then bugs which are related to this behaviour are going to be hard to detect. Equally, if you bet your life on the known correct behaviour of a certain function (which is actually faulty), you are going to spend a lot of time looking in the wrong place.

- **spaghetti (unstructured) code:** Informants sometimes reported that the code "was in a mess" when they were called in to deal with it. There is, unsurprisingly, a 100% correlation between complaints about "ugly" code and the assertions that "someone else" wrote the code.

Results

In this and subsequent sections, 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 rather because it gives us a convenient overview of the nature of the problems that the informants chose to share with us. Knuth argues that the categories and "raw records" themselves are just as informative as summary statistics (e.g. showing the frequency of occurrence of particular error types): "The concept of scale cannot easily be communicated by means of numerical data alone; I believe that a detailed list gives important insights that cannot be gained from statistical summaries." In a similar vein, I am mainly interested in categorizing the findings, but feel that a count of frequency of occurrence is of interest as a reflection of what types of problems the population of informants chose to report.

The frequency of occurence of the different reasons for having difficulty is shown in Table 3.

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 mis-directed blame	6
spaghetti (unstructured) code	3
??? (no information)	*8*

Table 3. Why the bugs were difficult to track down.

Thus, 53% of the difficulties are attributable to just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. The high frequency of reports of cause/effect chasms accords well with the analyses of Vesey (1989) and Pennington (1987) which argue that the programmer must form a robust mental model of correct program behaviour in order to detect bugs—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 reported below.

Dimension 2: How found

Categories

The informants reported four major bug-catching techniques, as follows:

- **gather data:** This category refers to cases in which the informant decided to "find out more", say by planting print statements or breakpoints. In fact, a variety of specific data collection techniques were reported. The important thing that distinguishes these data collection techniques from the "controlled experiments" category described below is their bottom-up nature. That is, the informants may have had a rough idea of what they were looking for, but were not explicitly testing any hypotheses in a systematic way. Notice that this category is different both from Katz and Anderson's (1987) causal reasoning and from their data-driven hand simulation: it is really a hybrid of the two, because it is bottom-up, on the one hand, yet can lead directly to a causal analysis once the data has been gathered. Here are the six sub-categories reported by the informants:

    - *step & study:* the programmer single-steps through the code, and studies the behaviour, 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 which calls (say) a timer or data-structure printout both before and 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. voluminous output of 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, which highlights the difference between the two execution runs

- *conditional break & inspect:* a breakpoint is inserted into the code, typically triggered by some specific behaviour; data values are then inspected to determine what is happening

- *specialist profile tool (MEM or Heap Scramble):* there are several off-the-shelf tools which detect memory leaks and corrupt or illegal memory references, and the experts who relied on these also tended to rave about their value.

- **"inspeculation":** This name is meant to be a hybrid of "inspection" (code inspection), "simulation" (hand-simulation), and "speculation", which were among a wide variety of techniques mentioned explicitly or implicitly by informants: cogitation, meditation, observation, inspection, contemplation, hand-simulation, gestation, rumination, dedication, inspiration. In other words, they either go away and think about something else for a while, or else spend a lot of time reading through the code and thinking about it, possibly hand-simulating an execution run. "Articulation" (explaining to someone else how the code works) also fits here. The point is that this family of techniques does not involve any experimentation or data gathering, but rather involves "thinking about" the code.

- **expert recognized cliché:** These are cases where the programmer called upon a cohort, and the cohort was able to spot the bug relatively simply. This recognition corresponds to the heuristic mapping observed by Katz and Anderson. The very nature of my data gathering excercise invariably requires a cohort (rather than the informant) to have detect the bug: the informant would not have been stumped, nor would have bothered telling me about the bug, had he or she been able to identify the solution quickly in the first place!

- **controlled experiments:** Informants resorted to specific controlled experiments when they had a clear idea about what the root cause of the bug might be.

Results

The frequency of occurrence of the different debugging techniques is shown in Table 3.

Category	Occurrences
gather data	27
inspeculation	13
expert recognized cliché	5
controlled experiments	4
??? (no information)	*2*

Table 3. Techniques used to track down the bugs.

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" that the programmer

is reduced to in difficult debugging situations. Let's now turn to an analysis of the root causes of the bugs before we go on to see how the different dimensions interrelate.

Dimension 3: Root cause

Categories

The bug causes reported by the informants fell into the following nine categories:

- **mem:** Memory clobbered or used up. This cause has a variety of manifestations (e.g. overwriting a reserved portion of memory, and thereby causing the system to crash) and may even have deeper causes (e.g. array subscript out of bounds), yet is often singled out by the informants as being the source of the difficulty. Knuth has an analagous category, which he calls "D = 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 else 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 which Knuth labels as "A = algorithm awry" and also those labelled as "S=surprise scenario". Knuth's A-vs.-S distinction can only be resolved by in-depth introspection, and is too fine-grained for the purposes of this study.

- **init:** Wrong initialization; wrong type; definition clash. A programmer will sometimes make an erroneous type declaration, or re-define the meaning of some system keyword, or incorrectly initialize a variable. I refer to all of these as "init" errors, since the program begins with its variables, data structures, or function definitions in an incorrect starting state.

- **var:** Wrong variable or operator. Somehow, the wrong term has been used. The informant may not provide enough information to deduce whether this was really due to faulty design logic (**des.logic**) or whether it was 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 meant to be trivial problems, not due to the algorithm itself, nor to faulty variables or declarations. This class of errors encompasses Knuth's "B=Blunder" and "T=Typo", which are hard to distinguish in informant's reports.

- **unsolved:** Unknown and still unsolved to this day. Some informants never solved their problem!

- **lang:** Language semantics ambiguous or misunderstood. In one case, an informant reports that he thought that 256K meant 256000, which is incorrect, and can be thought of as a semantic confusion. In another case, an informant reported a mismatch between the way a manual described some maximum value and the way in which it was actually dealt with by the compiler.

- **behav:** End-user's (or programmer's) subtle behaviour. For example, in one case the bug was caused by an end-user mysteriously depressing several keys on the keyboard at once, and in another case the bug involved some mischievous code inserted as a joke. These are really manifestations of behaviour external to the normal progamming arena, but still warrant a category in their own right.

Results

Table 3 displayes the frequency of occurrence of the nine underlying causes.

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 behaviour	2
??? (no information)	*2*

Table 3. Underlying causes of the reported bugs.

Table 3 indicates that the biggest culprits were memory overwrites and vendor-supplied hardware/software problems. Even ignoring vendor-specific difficulties, one implication of Table 3 is that 37% of the nastiest bugs reported by professionals could be addressed by (a) memory-analysis tools and (b) smarter compilers which trapped initialization errors. But what about the interaction between the cause of the bug, the reason for the debugging difficulty, and the debugging technique? That is precisely the focus of the next section.

Relating the dimensions

To understand the ways in which the three dimensions of analysis interrelate, we can place every anecdote precisely in our three-dimensional space. For expository purposes (and because multi-dimensional diagrams are hard to

discuss) let's consider just the following two-dimensional comparisons: (a) root cause vs. how found, and (b) how found vs. why difficult.

Table 4 compares root causes (row labels) against bug-finding techniques (column labels). Each cell entry shows the number of anecdotes with the given attributes. In cases where an anecdote reveals multiple attributes (say, it belonged partly to column 3 and partly to column 4), it is simply split evenly across the appropriate cells, hence the fractional entries. Tables 1, 2, and 3, incidentally, did *not* use this splitting technique, and correspond to tallies of the primary (first-reported) categories only.

CAUSE vs. HOW	gather data	inspeculation	expert recognized cliché	controlled experiments	??? (no info)	TOTALS
mem	8.50	4.00		1.00		**13.50**
vendor	4.00	3.00	1.00	3.00		**11.00**
des.logic	4.00	3.00				**7.00**
init	5.00	1.00				**6.00**
lex		1.00	2.00		1.00	**4.00**
var	2.00		.50		1.00	**3.50**
unsolved	3.00		.50	.50		**4.00**
lang	1.00		1.00			**2.00**
behav		1.00	1.00			**2.00**
??? (no info)	2.00					**2.00**
TOTALS	**29.50**	**13.00**	**6.00**	**4.50**	**2.00**	**55.00**

Table 4. Tally of root causes of bugs (rows) vs. how found (columns). Each cell entry (e.g. 8.50) is a tally of the number of anecdotes reporting that cell's row label (i.e. root cause) and column label (i.e. how found). Fractional entries reflect anecdotes which have been divided into multiple categories, so that an anecdote reporting both a "controlled experiment" and "expert recognized cliché" scores .50 in each cell. Note that tables 1, 2, and 3 only tallied the primary (first-reported) category for each dimension.

Of most interest is the relative density of anecdotes in the upper left-hand corner of the table, suggesting particularly that memory-clobbering errors could usefully be dealt with by better data-gathering tools. The density of the cell entries is *not* greater than that predictable by chance from the row and column totals alone (X^2 ,df:36, = 45.30, ns), suggesting no reliable relationship between root cause and how found, though the cell densities are nevertheless of *a priori* interest to tool developers, as discussed below.

Table 5 compares reasons for difficulty (rows lables) against bug-finding techniques (column labels). Once again, the need for data-gathering tools is highlighted, this time for dealing with cause/effect chasms and cases in which other debugging tools are inapplicable. In this case, the density of certain cell entries *is* greater than that predictable by chance from the row and column totals alone (X^2 ,df:20, = 33.50, p. < .05), suggesting in particular that data-gathering activities are of special relevance when a cause/effect chasm is involved or when the built-in debugging tools are somehow rendered inapplicable.

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 info)	4.00	2.00	2.00			**8.00**
TOTALS	**29.50**	**13.00**	**6.00**	**4.50**	**2.00**	**55.00**

Table 5. Tally of why bugs were difficult (rows) vs. how found (columns). Each cell entry (e.g. 9.83) is a tally of the number of anecdotes reporting that cell's row label (i.e. root cause) and column label (i.e. how found). Fractional entries reflect anecdotes which have been divided into multiple categories, so that an anecdote reporting *three* reasons for difficulty scores .33 in each of three relevant cells.

A niche of potential interest (and profit) to tool vendors is highlighted by looking at the relationship among the three dimensions: the most heavily populated cells are those involving data-gathering, cause-effect chasms and memory or initialization errors. The implications of this finding are discussed in the next section.

Discussion: Lessons learned

From boasting war stories to on-line repository

What intrigued me the most upon seeing the replies was the way in which complete strangers, with very little prompting and no incentive, were so articulate in their reminscences, and so forthcoming with details. These people clearly *enjoyed* relating their debugging experiences. Moreover, the depth of supplied details 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). Clearly, this is a self-selecting audience of email users and conference browsers who enjoy electronic "chatting" anyway, and some may even have felt a "macho" need to tell a good (and hence boastful) war story-- so much the better! I have no reason to distrust the sources, and the details of each story certainly have their own self-consistency. It is already widely accepted that the international computer network community is a gold-mine of information (see, e.g. *BYTE* feature article on the Internet, 1992). This collection of anecdotes suggests that it may also be a rich repository of willing subjects ready to supply detailed knowledge in a fairly rigorous manner which may then serve as a resource for others. We are now exploring the idea of developing an on-line repository which could be used both to receive new anecdotes and to respond automatically to keyword enquiries. This could be of great benefit to those with an urgent need to solve complex debugging problems, but only after an appropriate indexing scheme has evolved. This paper is a first step toward such a scheme. Note that it is not necessary to develop a definitive taxonomy. On the contrary, the stories themselves, even when only tangentially related to a specific bug enquiry, could be sufficient to trigger an insight which leads to the solution of a debugging problem.

Comparison with fine-grained study

As part of a series of investigations on the nature of programming and debugging environments, I have also looked in detail at what it's like to work with an apparently "modern" and "friendly" program development environment: HyperCard. I kept a detailed diary of several lengthy debugging sessions, and then analysed the problems and difficulties I experienced (Eisenstadt, 1993). In particular, the "why difficult" dimension was analysed at a more a fine-grained level of detail, revealing eight fundamental problems. Table 6 overleaf lists each of those problems (left-hand column), shows which coarse-grained "why difficult" category they correspond to (middle column), and identifies a possible solution (right-hand column).

Fine-grained Problem	Coarse-grained "Why difficult" Category	Proposed Solution
1. The link between an error message and the offending source code line has to be deduced by the user (whereas it could be provided for free).	Cause/effect chasm; Tools inapplicable or hampered (long run to replicate);	**S1:** Computable relations should be computed on request, rather than be deduced by the user.
2. Performing a routine debugging/inspecting action involves dealing with many disruptive subgoals (e.g. switching modes to enable specific machine states).	Faulty assumption (typically about what "mode" the debugging tool is in)	**S2:** Atomic user-goals should be mapped onto atomic actions.
3. Access to the interpreter (and other features) in the middle of a break is disallowed.	Tools inapplicable or hampered (context precludes using debugger)	**S3:** Allow full functionality at all times.
4. The behaviour of built-in functions can not be monitored easily.	Tools inapplicable or hampered (context precludes using debugger)	**S4:** Viewers should be provided for "players" (any evaluable expression) rather than just "variables".
5. There is no meaningful coarse-grained view of execution.	Tools inapplicable or hampered	**S5:** Provide a variety of navigation tools at different levels of granularity.
6. Traversal of indirect data influences (data flow) requires too much detective work.	Cause/effect chasm	**S6=S1:** Computable relations should be computed on request, rather than be deduced by the user.
7. Understanding the precise conditions under which a particular event happens (control flow logic) requires too much detective work.	Cause/effect chasm	**S7=S1:** Computable relations should be computed on request, rather than be deduced by the user.
8. The "inner state" of an object can be deceptively different from its apparent state, requiring extra detective work to uncover.	WYSYIPIG: what you see is probably illusory, guv'nor	**S8:** Displayable states should be displayed on request, rather than be deduced by the user.

Table 6. Relationship between fine-grained problems identified using self-report (left column) and coarse-grained categories of the current broad survey (middle column). Prospective solutions to each of the eitht fine-grained problems are also identified (right-hand column).

Although the scope of the two studies was rather different, it is nevertheless gratifying that most of the problems found in the fine-grained study fell into the two most popular studies reported in this coarse-grained study of the current paper (namely, tools hampered and cause/effect chasm).

Solutions

What programmers really need, of course, are smarter compilers and debuggers. But the analyses presented throughout this paper suggest that we can be more precise than simply demanding "smartness" from tool developers. For one thing, 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 be one which employed some data-gathering or traversal method to resolved large cause/effect chasms in the case of memory-clobbering errors (indeed Purify, described below, does precisely this). Secondly, we can propose solutions to the "why difficult" problems by considering the specific cases brought to light by the fine-grained study described above. One way or another, all of the problems mentioned in Table 6 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 problems 1, 2, 3, 6, 7, and 8, and each of these problems can be addressed specifically.

The proposed solutions presented in Table 6 are not necessarily easy to implement, but there are an increasing number of tools appearing both in the research community and in the marketplace which illustrate aspects of these solutions. For example, consider the idea of computing and displaying important relations and states on request, rather than relying on the programmer's deductive skills (solutions S1=S6=S7 and S8). The software tool Purify (Hastings & Joyce, 1992) analyses run-time memory leaks in C programs on Sun workstations 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 harder dataflow traversal problem than that required to deal with indirect pointer traversing such as that reported by several informants. Solution S3 (allowing full functionality at all times) is effectively provided in many modern Lisp implementations. Solutions S4 and S5 (viewers and granularity) have been explored at length in (Brayshaw & Eisenstadt, 1991) and (Eisenstadt et. al., 1993). That leaves S2 (atomic user-goals should be mapped onto atomic actions), which is increasingly addressed in commercial debugging tools, but still requires significant research input.

Summary and 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. Half of the difficulties arose from just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. Techniques for bug-finding were dominated by reports of data-gathering (e.g. print statements) and hand-simulation, which together accounted for almost 80% of the reported techniques. The two biggest causes of bugs were (i) memory overwrites and (ii) vendor-supplied hardware or software faults, which together accounted for more than 40% of the reported bugs. The analysis pinpoints a winning niche for future tools: data-gathering or traversal methods to resolved large cause/effect chasms in the case of memory-clobbering errors. Other specific solutions, all of which emphasize issues of "directness" and

"navigation" were developed by comparing the current study with a fine-grained self-report study. The investigation highlights a potential wealth of information available by worldwide email, and indicates that it may well be possible to establish an on-line repository for perusal by those with an urgent need to solve complex debugging problems.

References

Brayshaw, M. & Eisenstadt, M. (1991). A Practical Graphical Tracer for Prolog. *International Journal of Man-Machine Studies*, **35**(5): 597-631.

Brooks, R. E. (1980). Studying Programmer Behavior Experimentally: the problems of a proper methodology. *Communications of the ACM*, **23**(4):207-213.

du Boulay, J. B. H. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, **2**(1):57-73.

Eisenstadt, M. (1993). Why HyperTalk debugging is more painful than it ought to be. Submitted to British Computer Society HCI'93, London, September 1993. Also available as: Technical Report No. 103, Human Cognition Research Laboratory, The Open University, Milton Keynes, UK.

Eisenstadt, M., Domingue, J., Rajan, T., & Motta, E. (1990). Visual Knowledge Engineering. *IEEE Transactions on Software Engineering*, **16**(10):1164-1177.

Eisenstadt, M., Keane, M., & Rajan, T. (Ed.). (1992). *Novice Programming Environments: explorations in human-computer interaction and artificial intelligence*. East Sussex, UK: Lawrence Erlbaum Associates.

Eisenstadt, M., Price, B. A., & Domingue, J. (1993). Software Visualization As A Pedagogical Tool. *Instructional Science*, **21:** 335-365.

Gould, J.D., & Drongowski, P. (1974). An exploratory study of computer program debugging. Human Factors, **16** (3): 258-277.

Hastings, R. & Joyce, B. Purify: fast detection of memory leaks and access errors. *Proceedings of the Winter Usenix Conference*, January 1992.

Johnson, W. L. (1983). An Effective Bug Classification Scheme Must Take the Programmer into Account. In *Proceedings of The Workshop on High-Level Debugging*, . Palo Alto, CA:

Kahney, H. & Eisenstadt, M. (1982). Programmers' Mental Models of their Programming Tasks: The Interaction of Real World Knowledge and Programming Knowledge. In *Proceedings of The Fourth Annual Conference of the Cognitive Science Society*, (pp. 143-145).

Katz, I. R. & Anderson, J. R. (1988). Debugging: An analysis of bug-location strategies. *Human Computer Interaction*, **3**(4):351-399.

Knuth, D. E. (1989). The Errors of TeX. *Software—Practice and Experience*, **19**(7):607-685.

McCullough, P. L. (1983). Implementing the Smalltalk-80 System: The Tektronix Experience. In G. Krasner (Eds.), *Smalltalk-80: Bits of History, Words of Advice* (pp. 59-78). Reading, MA., USA: Addison-Wesley.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19:** 295-341.

Shneiderman, B. (1980). *Software Psychology*. Cambridge, MA: Winthrop.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1973). Psychological evaluation of two conditional constructions in computer languages. *International Journal of Man-Machine Studies*, **5**:123-143.

Soloway, E. & Iyengar, S. (Ed.). (1986). *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

Spohrer, J. C., Soloway, E., & Pope, E. (1985). A Goal/Plan Analysis of BUggy Pascal Porgrams. *Human-Computer Interaction*, **1**(2):163-207.

Vesey, I. (1989). Toward a theory of computer progream bugs: an empirical test. *International Journal of Man-Machine Studies*, **30**:123-46.

Tales of Debugging from The Front Lines

Appendix A: Selected raw anecdotes

U1

Not too exciting, but I'll bet it's awfully typical. I had a program (roughly 15,000 lines) in C, running on PCs and Unix. It does screen writes using the curses library. After a long period of development (mostly on the PCs) I started to see occasional odd characters popping up on the screen. The problems were not easily reproducible, but they gave me a queasy feeling. I started cursing the (public domain) curses library I was using on the PC. I started setting breakpoints and tracing, but any time I got a reproducible glitch, setting a breakpoint or inserting a debugging statement "cured" the glitch.

Of course, by now you've probably guessed the problem. I eventually wrote some routines that put a debugging wrapper around the standard malloc() and free() calls. The routines do the following: log every malloc() and free() to a disk file by module and line number, record the number of bytes requested, insert checking signatures at the beginning and end of every allocated chunk, and check those signatures for overwrites at every free() or when explicitly requested to do so

I found all sorts of intriguing things (all in my own code, by the way, none in the curses library). I sometimes free()ed memory twice (just trying to make sure, I guess). I sometimes overran malloc()ed buffers (usually by the infamous single '\0' at the end of a string). All in all, I think I found about 10 memory allocation/usage errors. I'm not sure exactly which were responsible for my glitches, but almost any of them had bad potential. The glitches are gone, now. I can concentrate on other problems...

U12

The worst bug I've had to pin down comes from an artificial life model I've been working with. I "inherited" the code - really awful K&R C code with absolutely no structured programming. Functions are scattered throughout C files, lots of global variables, no comments, typical bad code. The whole system is rather small, actually - 4000 lines, so it is possible for me to understand the whole thing.

But at the time of the bug, I hadn't really grokked the whole mess. The program only crashed after running about 45000 iterations of the main simulation loop. Running it this long takes about 2 hours and 8 megabytes of core. The crash was a segmentation fault.

Somewhere, somehow, someone was walking over memory. But that somewhere could have been *anywhere* - writing in one of the many global arrays, for example.

The bug turned out to be a case of an array of shorts (max value 32k) that was having 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.

It points out several things of how C can really shoot you in the foot. No overflow errors on integer operations, so 32767+1 really is -32768. No bounds checking on array operations - a[-32768] = 0; is a perfectly legal operation with really negative effects.

The actual bit of memory being written into eventually hit one of the malloc() chain data structures (lots of 4k data structs being malloced and freed), causing stupid Ultrix free() to do the Wrong Thing and trash the heap.

But of course 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 use watchpoints, because those slow a program down at least 10 fold, resulting in 30 hours to track a bug. No good.

The way I found it was to first use GNU malloc(), which has some very simple range checking features built in. That let me catch on to what was actually generating the SIGSEGV - heap trashing. I then just sort of zenned the bug, printing out data structures in the program and looking to see if they looked right. I finally found the negative number somewhere, then squashed the bug.

It took me 3 days to find.

U19

The following is a true story that happened to me about 8 years ago.

I was working on a small team developing an Ada compiler in an academic setting. I was responsible for the code generator. One day I got a bug report from another member of the group that a certain Ada program of his crashed whenever he compiled it with our compiler, and it looked like the problem was a stack underflow. (Our target machine was a Perq Systems PERQ, running a microcoded stack-oriented instruction set similar to P-Code.) Examination of the disassembled object code revealed that, indeed, the compiler was generating (subtly) bad code. There were, however, other binaries that were purportedly built from the same sources by other members of the group, and they compiled the program just fine. At first, we suspected a version control problem, that somehow the version of the compiler that I had built was constructed using different sources than the others. We then suspected differences in release levels of the compiler and linker used on the various machines. After a few quick investigations, it became clear that something really fishy was going on, so we began a more systematic investigation. We took a common set of sources, and built a binary in which we tried every combination of { compile compiler, link compiler, compile test program, link test program, run test program } on each of two machines. It turned out that the problem appeared if and only if the compiler had been linked on my machine.

We reported the problem to the hardware maintenance staff, a little reluctant to blame the hardware, but fairly confident that we had controlled for every other variable. The hardware people did not seem too terribly put off by our diagnosis that a problem that might seem so clearly to be a compiler bug was in fact a hardware problem. A

technician swapped out the CPU card of my workstation, I relinked the compiler, and the problem vanished.

U29

I once had a program that only worked properly on Wednesdays.
I had a devil of a time finding what the problem was.
At the end of one cycle it would ask you if you wanted to continue, and unless you typed a "y" it would quit. (OK, OK, you caught me it was indeed a game program.) This program would always end the game even if you typed "y" unless you were playing on a Wednesday.
On Wednesdays it would work correctly.

The code for testing if a user has typed "y" or not is not very complex and I was unable to see what the problem could be. Re-arranging the code made the problem change symptoms but not go away.

In the end, the problem turned out to be that the program fetched the time and date from the system and used it to compute a seed for a random number generator. The system routine returned the day of the week along with the date. The documentation claimed that the day of the week was returned in a doubleword, 8 bytes. In actual 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 storage area intended for another purpose. As it turned out, that space was where a "y" was supposed to be stored to compare to the users 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.

A6

Well I had this 3 day bug that nearly killed me. In the end we dont know the exact cause but have some good ideas.

Essentially I was writing some serial code which would be called from an XFCN. So I built a piece of code in Think C which would open a serial port, configure it, ask for a record from a polhemus, parse the record and close the port. I put this code in a WHILE loop with the test being button down. In this way I could simulate the action of an XFCN. The code buzzed along returning records, about 3000 or so then it crashed. So I thought it might be a malloc/free kinda problem. Checked that then ran it again. Crashed again, on a different iteration. Now there was no macs bugs being invoked and using the Think C debugger was even less useful. What the [$#@%], I thought. Being a novice Mac programmer (2 months to be exact) [Ed: new to Mac, experienced at C], I started to freak out. So I heard that ANSI code was trouble.
I removed it all and replaced it with Toolbox code. Same thing. It would run from anywhere from a few hundred to a few thousand iterations then crash. But occassionally, I would get a Macs bugs error: error number 28. Stack overflow. Ok so I checked all my optimization parameters, put prototypes on and made sure I was returning something from a routine when I was supposed to. Everything looked fine. I was on my 2nd day. I put in lots of MemError calls, checked every single return value. I made sure there was no garbage in any allocated buffers. Shut off all weird inits. Increased the memory size of my app so it would have enough. I was on my third day. The stack overflow would come from the Mac routine that runs during the VBL which checks for stack/heap collision. But my routine that was called right before this was always different. It looked hopeless. Now I did what everyone debugging should always do. I called in someone else. In my case, it was the big guns: the Mac programming gurus of the group. It was me and two down and dirty assembly

hackers giving it a whirl. They showed me all sorts of macs bugs secrets and we thought we should write some assembly to catch that runaway stack pointer. Just then, one the aforementioned mac gurus started laughing and said "you know I tried to do exactly what you are trying to do last year. I wanted to rapidly open up and close serial ports for my sound app. The program would run for about 20 minutes and crash. Try this. Write your code so you open the port once, pass state back to hypercard, read then close when done." Ok so I took the 15 minutes and did this. Lo and behold we all watched in amazement as the code ran for tens of thousands of iterations. Though the device manager should be robust enough to deal with the rapid opening and resetting and closing of serial ports, it just cant deal. So we worked around it. We are going to leave it to someone else to find the real cause of the bug.

Tales of Debugging from The Front Lines

Appendix B: The Condensed Data

Table B-1. The condensed data, showing only those 36 entries for which every field could be filled. Enries in the left hand column are coded to preserve anonymity. ID "B1a" means BIX informant number 1 supplying the first of several anecdotes from that informant. ID labels U1-U37 refer to Usenet informants, and A1-A8 refer to AppleLink informants. Entries in the rightmost column include labels such as {L} and {T} to show the most plausible mapping to the categories used by Knuth (1983). Knuth's category labels are: A=Algorithm awry; B=blunder; D=data structure debacle; F=forgotten function; L=language liability; M=module mismatch;S=surprise;T=typo. The other category labels used in the table cells are discussed in the body of the paper.

ID	Context	Symptom	Why difficult	How found	Cause category: detail {Knuth label}
B1a	New commercial software about to be shipped; Quality Assurance found crash	Should be a call to OS at specific addres, but it's missing	mis-directed blame (compiler)	inspeculation: hand-replicate compiled code; inspection of source; call in expert	init: undeclared variable 'temp' clashes w. keyword 'temp' {L}
B2	Punched card COBOL programming	executed an 'unreachable' line!	WYSIPIG (What You See Is Probably Illusory, Guv'nor)	inspeculation: visual inspection (with 15-inch steel ruler)	lex: '.' was in col 72, hence regarded as a comment! {T}
B1b	IBM Series/1 programming	Console prints "IEW1234 IMMINENT SYSTEM FAILURE"	faulty assumption (of cooperative programmer... turned out to be practical or malicious joke)	expert recognized (after grilling programmer)	behav: own program printed this out intentionally, user forgot (programmer's behaviour unpredictable... this was a practical or malicious joke)
B9	VAX Pascal program for reading/writing file of complex records	write OK, but read yields garbage	tools hampered: Heisenbug (bug goes away when debugging tools used)	gather data: step & study, print & peruse	init: read parameters should have been declared as VAR (i.e. pointer rather than value) {L}
U1	15,000 lines of C code; PCs/Unix; does screen writes using curses library	Odd chars on screen	tools hampered: Heisenbug	gather data: wrap & profile	mem: free() called multiple times; malloc() buffers overrun by /0 at end of string {D}
U3	Fileserver maintenance; C / Ultrix	open file, then try to read it, server claims 'not open'	tools hampered: long run to replicate: multiple flakey parts, so tracing/stepping slowed by other failures	gather data: conditional break & inspect: bkpt on memory access (spec. address)	mem: array of char maxlength 1024 got overrun, munging file pointer structure {D}
U6	Compiler for 8086's running MSDOS	function returned wrong value	faulty model (thought stacks grew *down*); timing	gather data: step & study:single-step assembler, observe registers	mem: address BELOW stack pointer being wiped out by os interrupt handlers; pointer decremented too late in the compiled code

ID	Context	Symptom	Why difficult	How found	Cause category: detail {Knuth label}
U9	set covering code in Fortran (spaghetti)	anomalous test results	spaghetti: other person's code	gather data: MEM probe: hand-trace & debugger trace, home in via "wolf-fence"	des.logic: array element was both a status flag & a value... '0' was ambiguous, and mis-interpreted & therefore clobbered {A/F}
U10	PC clone, debugging memory resident ('TSR') programs	crash after 20 minutes, but would *not* crash when the debugger was switched on	tools hampered: a) long run to replicate (w. lotsa printout); b) Heisenbug	inspeculation: 'dedication'/observation	mem: bounds overrun; TSR wrote above top of memory into program... didn't happen under debugger which occupied some of that memory {D}
U11	called foo(1); but in definition of foo(X); assigned X = 2	1 =2	WYSIPIG semantics	gather data: print & peruse	init: famous FORTAN prob.. redefined 1 to be 2!!!! {L/M}
U12	Artificial Life; 4000 lines of unstructured K&R C code	Crash (segmentation fault) after ~45,000 iterations; 2 hours	1) spaghetti: other person's code; 2) tools hampered: long run to replicate (watchpoints etc. slowed downx10); 3) cause effect chasm	gather data: wrap & profile (GNU malloc() range-checking); trace data flow, print out data structures looking for oddball (= a kind of dump & diff)	mem: array of shorts (max value 32K) incremented every 1.5 iterations until > 32K, then this value was used as an array index!; bounds checking on array operation would have noticed, since 32676+1 -> -32768, ouch negative array index {D?/S/L}
U14	IBM kernel development for AIX v3	once every ~20,000 iterations, SIGTRAP killed traced proc	cause/effect chasm: infrequent; tools hampered: long run to replicate; Heisenbug	gather data: step & study (problem went away with new compiler)	unsolved: it's never been solved (new AIX released, prob went away)
U15	Port of large financial planning package from PC to Mac	random wrong answers (only for large models)	tools hampered: long run to replicate	gather data: print & peruse	init: uninitialized variable, on the PC version it is set to 0, but on Mac may be set to whatever was in that location previously {L/F}
U16	binary i/o package	strings were gibberish	cause/effect chasm: infrequent	expert recognized cliché & suggested discriminating test	lang: compiler (MSC) derived alignment constraints from base type rather than full type {L}
U17	cpu-intensive nighttime job doing big citation index search	job WITHOUT i/o mysteriously terminated by console interrupt	cause/effect chasm: infrequent; dump showed nothing	inspeculation: gestation, thinking about logic; realizing it wasn't a fluke	des.logic: if main acct. idle while bkgnd job has a file locked, -> os kills job (hack to avoid deadlock) {S/A}

ID	Context	Symptom	Why difficult	How found	Cause category: detail {Knuth label}
U18	VAX-11 FORTRAN code	mysterious behaviour of FORTRAN code	WYSIPIG lex	expert recognized cliché & suggested discriminating test	lex: TAB (1 char) replaced by 8 space (8 chars), pushed identifier past column 72, so truncated (cf. entry 49) {T}
U19	developing code generator for Ada compiler on PERQ	user complained of crash with stack underflow; other users ok	cause/effect chasm: inconsistent, many degrees of freedom (HWxcompilerxlinker xsource=2^^4)	controlled expts: exhaustively try every combination, only happened when compiler was linked on specific machine	vendor: hardware fault... after swapping CPU card & re-linking compiler, problem vanished
U20	PC clone, editor bug	crash ONLY on 486 executing wrong interrupt number	tools hampered: Heisenbug	inspeculation: 'inspiration'	vendor: int86() stores interrupt, then modifies (ok) BUT 486 instruction pipeline had ALREADY read the instruction {D/S}
U21	code inherited from others	5 old bugs (new user didn't even know it)	spaghetti	inspeculation: reformat code & visual inspection	des.logic: misc... flaws in logical flow {A}
U22a	Programming an embedded system in PL/M-86	crash.. process jumped to stack segment of another process	faulty assumption due to 'warning', not 'error' so still compiled & linked	gather data: step & study w. hed debugger	init: allocated 1 byte less than needed, e.g. char msg[1]={'h', 'i'} should be [2] {D}
U22b	implementing quicksort + print result in C; testing with printfs	out of stack space	tools hampered: error cloberred diagnostic tools!!!	gather data: print'n 'peruse	init: own use of 'write' redefined system's 'write' without warning, so qsort's output & manual trace's printf() recursed endlessly {L}
U23	portable C code with some machine-specific assembler	ran ok EXCEPT on Vax/Ultrix	faulty assumption (thought bug in own code)	a) inspeculation: hand simulation; (b) gather data: wrap & profile -> dump & diff; step & study	vendor: instruction present on older Vaxes only emulated on MicroVAX-II, emulation code had a bug in it!
U25	shorthand-to-English translation program	disk system returned wrong sector, but on different iterations!	cause/effect chasm: inconsistent; timing-sensitive (75µsec!)	gather data: wrap & profile; canonicalize (reduce to simplest replicable case)	des.logic: flip-flop set/reset side effect w. timing interaction; read(A) reads A, then read(unknown) continues to return A {A;S}
U26	Mac NetHack	misc. bugs	cause/effect chasm: inconsistent	gather data: 'Heap scramble' (provokes bugs) & 'Mr. Bus error' (tailored tease-out)	mem: double-indirect references, middle pointer ('handle') is owned by Mac OS, trouble if unlocked or invalid handle moved {D}

ID	Context	Symptom	Why difficult	How found	Cause category: detail {Knuth label}
U27	IBM 1401 w. punched cards; 8K core	dud compiler	faulty assumption (mis-directed blame, told 'didn't work')	inspeculation: book ('anatomy of compiler') + reasoning (lo mem + h'ware multiply + multiply SUBR)	mem (prog too big): mutiply SUBR pushed compiler beyond 8K... removing punched cards for this SUBR cured problem (because this model had hardware multiply)
U28a	Modifying SOS editor under TOPS10	crashed when exiting intra-line alter mode with <esc>	cause/effect chasm: intermittent	gather data: dump & diff	des.logic: different instruction for <ESC> vs, <CR> (logic error) {A}
U29	game playing program; asked 'want to continue? (y/n)'	Program worked when user input “y”, but only on Wednesdays, else always quit!!!	cause/effect chasm	inspeculation: re-arrange code (didn't help), + ?	mem: documentation said 8 bytes needed, but 12 really needed, so 6 days a week cloberred mem with blanks, but on Wednesday,'y' luckily matched 9th byte {D}
U34	large office management system	Word Perfect said 'printing', but nothing happened	cause/effect chasm: inconsistent; worked ok on similar setup	1) gather data: wrap & profile; 2) controlled experiments	unsolved: never debugged!.. failed precisely with machine A & printer B & > 1MB code & not(breakout box) ! {S}
U35	Porting graphics code to new DG machine	infinite loop	tools hampered: Heisenbug	controlled experiment: binary probe; gather data: conditional break & inspect	vendor: when arctan instruction was on a page boundary, a microcode defect caused jump to 0; since a content of 0 also means 'jump to 0', it resulted in endless loop
U36	TCP/IP network kernel for MS-DOS	Telnet hangs, but only with 1 terminal emulator, and only at one slow speed	cause/effect chasm: intermittent; speed-dependent; tools hampered: context precluded using debugger	gather data: print & peruse; step & study	des.logic: re-xmit (slow) packet, test 'already?'->neg number; old packet updated where in data stream we were {A}
U37	Porting game 'omega' from Unix to Atari ST	intermittent weirdness	tools hampered: context precluded using debugger	gather data: wrap & profile	mem: program deleted list containing ptrs to other objects {D}
A6	Developing a Mac sound application, requires rapid open/close of serial ports	crash after ~3000 iterations	cause/effect chasm: timing problem; intermittent	gather data: wrap & profile, controlled experiments; expert recognized cliché	unsolved: device mgr not robust enough to handle rapid open/reset/close of serial ports... root cause still unknown; used workaround
A7	Ampex: Unix upgraded for real-time stuff	system crash after ~2 hrs	tools hampered: error consumed evidence; long run to replicate	gather data: print & peruse; step & study with hardware bus analyzer.	vendor: custom-tuned boards-> bad data -> jump to bad address {D}

ID	Context	Symptom	Why difficult	How found	Cause category: detail {Knuth label}
A8	Kids developing Hypercard 2.0 apps	Hypercard card suddenly disappears	WYSIPIG user-action	inspeculation: lucky observation	behav: CMD-Shift-Del kills card