SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 6, 293–299 (1976)

Software Malpractice — A Distasteful Experience †

MICHAEL J. SPIER

Department of Software Engineering, Digital Equipment Corporation ‡
146 Main Street, Maynard, Massachusetts 01754, U.S.A.

SUMMARY

A sequence of events is described, leading to the severe deterioration of an initially well conceived and cleanly implemented compiler. It is shown how an initial "optimization" implanted a latent bug in the compiler, how the bug was subsequently activated through innocent compiler modification, and how the compiler then deteriorated because of the incompetent correction of the bug manifestation. This exceedingly negative case history is presented in the hope of conveying to the reader a better feeling for the complex problems inherent to industrial software production and maintenance. The difficulty in proposing any constructive (and complete!) software engineering methodology is known and acknowledged; the study of an episode such as described in this paper might help put the difficulties, with which we are confronted, into better perspective.

KEY WORDS GOTO-less Programming, Program Correctness, Software Engineering Methodology, Software Production and Maintenance, Structured Programming.

PRE SCRIPTUM

We recognize nowadays that the more programs we write for applications of increasing complexity and sophistication, the more uncertain we become of our ability to write these programs such that they be as "good" as possible. It has become fashionable for our colleagues to propose various remedies —often dogmatic in nature— which are supposed to enhance the "goodness" of our programs. Thus, for example, we are encouraged to adopt "structured programming". Nobody *definitely* knows what it is — although popular myth has it that a total ban on GOTO statements will automatically impart beneficial "structure" to our code.

"Goodness", of course, is in the eyes of the beholder. I must confess to a nagging suspicion that some of the proposed remedies come from sources more concerned with classroom toy programs than with the realities of everyday software production. The suggested remedies are typically exemplified by means of some neat little exercise (*e.g.*, enumeration of prime numbers, bubble

† This paper describes a certain compound software bug which was encountered by the author at a certain point in his career. It must be emphasized that this case history relates neither to the author's present employment with Digital Equipment Corporation, nor to any DEC software product.

‡ This paper was produced by the author in machine readable form and typeset, using Digital Equipment Corporation's Typeset10 text processing and composition system. The camera ready manuscript was submitted to this journal for direct reproduction. Type was set on a Graphic Systems, Inc. C/A/T photocomposition machine at Digital's Marlborough plant facilities.

Received 10 March 1975
Revised 15 July 1975

sort, matrix inversion, eight queens problem), perhaps very suitable as a classroom illustration but a far cry from the real world programs where we hurt. It is nigh impossible to relate the sample program to the real one (*e.g.*, operating system monitor, airline reservation system, inventory control system, payroll program) such that the remedy might be evaluated in respect of its actual context of application.

What most proposed remedies have in common is their insistence on *how to write* the program. Little or no attention is paid to the remaining issues inherent to everyday software production. In reality, a program has to be designed, coded, debugged, verified for an acceptable level of reliability (I hesitate to use the word "correctness"), modified, improved both in the sense of functionality and in the sense of performance optimization, and all the while be kept at a stable level of lucidity in order to allow for its continued evolution.

The totality of issues which have to be taken into consideration is staggering. The development of a comprehensive software engineering methodology is a challenge whose immensity defies our bravest attempts. I suspect that too few of our academicians really grasp the larger implications of the problem, tending to polarize their attention on what in reality are insignificant issues (*e.g.*, the great GOTO controversy), while failing to see areas of difficulty which literally beg for attention (*e.g.*, how many of our experts in programming linguistics have ever designed a non-interpretive language where the symbolic debugging facility is an integral part of the language specification?)†

Old chinese proverb say *"one picture worth thousand words."* It occurred to me that by describing an exceedingly negative —though, unfortunately typical— case study of how foolishness, compounded by ignorance and topped by incompetence resulted in a most incredible compound software bug, I might succeed in conveying a feeling for the problems that afflict the software industry.

I came across this bug a good many years ago. I doubt that the affected program, indeed the computer itself, are still in use. It happened well before I entered DEC's employment and is unrelated to DEC and its products. Still, he who sits in a glass house is wise not to cast stones. Thus the following has been fictionalized sufficiently to convey the essence of the story without pointing an accusing finger at any identifiable culprit.

THE STORY

Back in those days when core memory was scarce, random access memory was unknown and card readers and tape drives proliferated, there was a compiler. It consisted of 23 tape-to-tape subcompilation passes. The compiler was initially well designed and coded in as modular a fashion as could be expected under the circumstances.

Each pass was a distinct program, loaded from a library tape to overlay its predecessor in memory. A certain subset of memory was reserved as a common communications area between passes, containing both data and code. A software (*i.e.*, macro-) implemented CALL/RETURN mechanism existed. The CALL instruction would push the proper return address onto a software managed stack and transfer control to the designated subroutine. The RETURN instruction would transfer control to the locality indicated by top-of-stack, after having popped that datum from the stack to reveal the underlying return address. The stack resided in the common communications area; its size was barely sufficient to accomodate the collection of return addresses corresponding to the longest observed sequence of calls. There existed neither stack overflow nor stack underflow condition detection provisions.

† From a purely pragmatic point of view, I find little appeal in a miraculous new language, if the corresponding debugging tool talks back to me in octal or hexadecimal. If that is the only debugging tool available, then by sheer instinct of survival I would opt for machine language code, for else I would be penalized by having to learn the compiler's code generation idiosyncrasies.

Initial Coding

The compiler was modularly structured, having an iterative main program residing permanently in the common area:

```
main:   subroutine;
        prologue;
        while   (compilation is to proceed)
        begin
                load    (next program overlay);
                call    {current pass};
        end;
        epilogue;
end     main;
```

where a pass was of the general form

```
pass_i: subroutine;
        perform necessary computation;
        return;
end     pass_i;
```

As can be seen, the above is the rudimentary representation of a well conceived program which, upon return from the current pass, would test certain common variables to determine whether or not the compilation should proceed. In the affirmative case it would bring in the next program overlay and call it.

Each logical database (*e.g.*, the symbol table) typically required two tape drives; an input tape containing the current version and an output tape onto which the updated version was copied. Two subroutines residing permanently in the common area were FLUSH and SWITCH, each parametrized for any given database. FLUSH guaranteed that the entire updated version of the relevant database was completely recorded on the corresponding output tape. SWITCH rewinded the relevant tapes and made the necessary common variable updates to effect the functional permutation of the pair of tapes.

Initial Foolishness

Then, a bright programmer made a stupendous intellectual discovery whereby he could instill additional (subjective) "goodness" into the compiler. He observed that FLUSH and SWITCH were of the form

```
flush:  subroutine;
        perform flushing;
        call    switch;
        return;
end     flush;
```

```
switch: subroutine;
        perform switching;
        return;
end     switch;
```

and decided to optimize the compiler by removing all space- and microsecond-"wasteful" calls, which in his judgement were functionally superfluous. In a recent article, Knuth[1] encouragingly recommends this foolishness, and even provides the recipe for its realization, to quote [*ibid.* pp. 280-282]: *"If the last action of a procedure* **p** *before it returns is to call procedure* **q**, *simply GOTO the beginning of procedure* **q** *instead."* And indeed, that is exactly what our wise guy did

```
flush:  subroutine;
        perform flushing;
        goto    switch;
end     flush;
```

```
switch: perform switching;
        return
```

not merely with respect to these exemplified subroutines, but systematically throughout the entire compiler. With admirable perseverance, that anonymous optimizer invested a tremendous amount of work (to him, perhaps, a labor of love)† in order to save a few words of memory and an unmeasurable amount of CPU time. The unmeasurably "optimized" compiler continued to perform its function as reliably as ever.

Further Foolishness

Another "improvement" took place. Passes were no longer CALL-ed. Given that the end-of-pass locality was known, and that a pass always RETURN-ed to a fixed location in the main program, that program underwent transformation:

```
main:   subroutine;
        prologue;
        while   (compilation is to proceed)
        begin
                load    (next program overlay);
                goto    {current pass};
loop:   end;
        epilogue;
end     main;
```

where a pass was now of the general form

```
pass_i: perform necessary computation;
        goto    loop;
```

Following this "improvement", the compiler continued to perform reliably. In view of the fact that this new modification is fraught with subtle traps (as will be demonstrated in the following), yet did not affect the compiler's reliability in any discernibly adverse manner, I suspect that this too was the handiwork of our original optimizer, whose professional competence (as distinct from his level of wisdom) must be acknowledged with the greatest respect.

Ignorance

Time passed. Our genial optimizer went on to accomplish bigger and better things in life. The compiler was now being maintained by programmers who were well below the optimizing wizard's level of technical competence. The programming language evolved —as programming languages invariably do— and new linguistic features had to be incorporated into the compiler. Additionally, the compiler was gradually being improved, for both object code optimization and enhanced error detection facilities.

A continuous compiler recoding effort was underway. Certain passes were modified. Other passes were split in two and recoded from scratch. The programmers performing this work had no clear understanding of the compiler's underlying coding conventions, especially in respect of the magnificent "optimization" job which was done earlier. They did however assume that any systematically replicated pattern of code presented a safe way in which to program by mimicry. This is a most reasonable assumption which any seasoned programmer makes intuitively and typically follows through with success. In this case, however, our brave programmers innocently added the straw that broke the camel's back.

The recoding and/or modification effort inevitably required the services of "optimized" subroutine pairs such as FLUSH and SWITCH. For example, the modified compiler now evolved

† "Don't underestimate the value of love in labors of love; redirect aesthetics but don't destroy them." Knuth, private communication.

to the point where under certain circumstances SWITCH was to be invoked independently of FLUSH. If certain errors had been detected, a state variable was set to indicate that actual code generation was to cease, while further syntactic and semantic analysis should proceed for diagnostic purposes. The time consuming FLUSH-ing was then deemed unnecessary, however a *pro forma* SWITCH-ing still had to be done to satisfy file system requirements. In fact, the analogous skipping of predecessor subroutines happened with respect to various other such subroutine pairs. The technical problem confronting a program modifier was: "How is a label — such as SWITCH— attained?" The solution to the technical problem was obvious. Namely to look up existing code, and upon detection of a universal pattern of label reference to faithfully mimic that pattern. SWITCH and various other subroutines were all very clearly and systematically attained by a GOTO statement. Thus everybody invoked those subroutines *via* GOTO.

Unfortunately, the necessary underlying condition for the successful usage of the "optimization" technique described earlier is *that there exist at least one valid return address on the stack!* † The once reliable compiler started crashing when a sequence of GOTOs led to a single unmatched RETURN statement (*e.g.*, in SWITCH), provoking a stack underflow condition and sending control to the locality indicated by the value of the word which happened to immediately precede the base-of-stack locality.

Incompetence

This bug manifestation was extremely difficult to analyze. Not all CALLs were removed from the compiler. The optimizing wizard knew his business (**proof:** following his contribution, the compiler's reliability remained unaffected), and saw to it that each logical path contained a matching number of CALLs and RETURNs. Being unaware of this critical convention, the modifying programmers violated it by ommission.

The stack was common to all passes. The actual violation of the implicit convention need not necessarily have happened when the stack was empty! Given that by virtue of the *Further Foolishness* LOOP was now attained *via* a GOTO statement, **the possibility existed for a stack underflow in pass i to cause successor pass j>i to crash.** The effect of the crash was clearly discernible; the cause of the crash may well have been located in an overlaid segment of code. These bug manifestations were highly capricious, depending upon the flow of control as dictated by specific input sequences.

Management became alarmed. Everybody went into frantic "debug mode", busily poring over memory dump listings. Eventually it was discovered that the crashes must be due to a stack underflow condition. Rather than analyze the problem to determine its cause, a quick and dirty solution was implemented to negate the problem's effect. The main program was modified in the following grotesque manner:

```
main:   subroutine;
        prologue;
        goto    loop;
        while   (compilation is to proceed)
        begin
                load    (next program overlay);
                goto    {current pass};
loop:   call    cont;
cont:   end;
        epilogue;
end     main;
```

† Another necessary condition is that the subroutine "invoked" with a GOTO statement not have any formal stack frame allocation, if such allocation is performed by the CALL instruction.

where every passage through MAIN would push onto the stack a dummy return address pointing to CONT (which is functionally equivalent to LOOP) such that any extraneous RETURN would effect a transfer of control to CONT rather than cause a stack underflow.

The bug manifestations known to be caused by the stack underflow condition were definitely "fixed". A new bug manifestation came into being when an *insufficient number* of the erroneous control paths was exercised. Specifically, the erroneous control paths produced a *deficiency* of return addresses on the stack. The new version of MAIN produced a *surplus* of return addresses on the stack, exactly equalling the number of times that control passed through LOOP. The stack was barely sufficient to accomodate the collection of return addresses corresponding to the longest observed sequence of CALLs. When an insufficient number of erroneous control paths was exercised, the stack would now *overflow*. The new symptom was recognized, and "fixed" by increasing the size of the stack by a number of words equalling the number of compiler passes! These extremely distasteful modifications were made by someone who must have been a most incompetent programmer. Understanding neither the error's cause, nor the sheer assininity of his solution, he went ahead and "fixed the bug" to the general satisfaction of his superiors.

The Sorry Ending

The bug manifestations related to stack mismanagement were definitely "fixed" to preclude compiler crashes. Compilation would always terminate. The compiler's output ceased to be correct always, given that control would sometimes undertake certain quantum jumps in logic. The compiler produced esoteric code, and programmers were busy repairing the compiler at syntax analyzer, expression evaluator and code generator levels.

At this point I was called in to lend a helping hand. It took me a significant amount of detective work to reconstruct the sequence of events described above. I recommended that the compiler be written off as a loss, and that an effort be launched to remake the compiler, possibly using a very early version of it as a starting base. The recommendation was rejected. Firstly, the compiler was too widely used to be scrapped. Secondly, the management felt that if this compiler is impossible to maintain then any other compiler would be just as messy. Hence, they reasoned, there was no point in committing themselves to an unknown evil when they had finally resigned themselves to living with the current one.

POST SCRIPTUM

A software product has a lifespan during which it must undergo constant change. The "goodness" of the product must be maintained throughout that entire period of existence, of which the initial implementation is but a part (and a minor one, at that.) The desired qualitative excellence is unfortunately subject to continuous modification; more regrettably, it is subject to practically unavoidable quantitative restrictions.

Concerning these quantitative restrctions; *volume of code* plays an exceedingly important role when it comes to large-scale software projects. The language chosen must allow for the independent compilation of program modules of manageable size. Consequently, global code quality can no longer be controlled by the language processor alone. Hence, no single "super compiler" would provide remedy. A possible solution would be to have a multi-level language where the linking of precompiled bodies of code would be treated as a semantically meaningful *post compilation* phase.

Similiarly, *runtime efficiency* is an important a factor, as is —alas— the insistence on *memory usage minimization*, especially with regard to the ever more sophisticated expectations of small-scale minicomputer users. It is the quantitative aspect of the desire for the fastest possible execution within the smallest possible memory partition, which significantly contributes to the qualitative

deficiencies of our software. The problem exists, and no amount of indifference to it will make it go away.

We have seen how a misdirected effort to "optimize" the compiler snowballed into catastrophe. Lest the proponents of GOTO-less programming gloat and proclaim: "See, I told you! You replaced some CALLs by GOTOs and God has inflicted just punishment upon you," let us reflect on the true meaning of this episode. GOTOs are but a tool, to be wielded incompetently or in a proficient, workmanlike manner. The act of replacing CALLs with GOTOs had —in itself— not the slightest deleterious effect upon the compiler, because the perpetrator knew his business and performed with commendable expertise. His successors did what they did and caused a *potential* bug to be activated. Yet potential bugs of similiar severity lurk in practically all advocated programming languages. Consider FORTRAN, where the sequence of calls $X \rightarrow Y \rightarrow Z \rightarrow X$ triggers a well known linguistic deficiency. And just in case some ALGOL *aficionado* grins and says: "Aha! Why don't you use an intelligently designed language which supports reentrant procedures," I would suggest that ALGOL and its derivatives have their own potential traps, the most blatant of which is the allowing of default references from a nested block to some outer-block variable, a notorious source of bugs of ommission which are typically caused by innocent typos.

Give a bare machine to a programming wizard, and he will be up and running within a few weeks. Cast "pearls"† to swine, and you are no closer to that undefined Utopian "structured programming". I believe that the problem transcends the purely technical domain, and that its solution must encompass the managerial domain as well.[3] Such solution must comprise the methodologies, rules and conventions which have to be respected in order to impart certain engineering standards upon the programming profession (remember, the only *real* cause for the fiasco described earlier was the violation of a single unstated rule!) We may have paid too much attention to the way in which we express our algorithm; perhaps it is time to focus more attention on the technical and human engineering contexts within which the algorithm has to exist.

REFERENCES

1. D. E. Knuth, 'Structured Programming with GOTO Statements', *ACM Computing Surveys*, **6**, No. 4, 261-301 (1974).
2. E. W. Dijkstra, 'Structured Programming', in 'Software Engineering Techniques', *1969 NATO Conference Report*, J. N. Buxton and B. Randell (Ed.), NATO Scientific Affairs Division, Brussels 39, Belgium, April 1970.
3. M. J. Spier, 'A Pragmatic Proposal for the Improvement of Program Modularity and Reliability', *Int. J. Comput. Information Sci.*, **4**, No. 2, 133-149 (1975).

† I hope that Dijkstra[2] will excuse the pun.