Static inspection – tapping the wheels of software

Les Hatton, Programming Research Ltd., U.K.

Article appearing in IEEE Software, May 1995

Static inspection is about the removal of obvious fault or inconsistency before a product is even tested. It forms an indispensable part of all conventional engineering disciplines except for software engineering. For example, when a micro-computer is reviewed, the reviewer will inevitably remove the case, exposing the boards to *inspect* their quality of build, and make astute judgements as to the *long-term* behaviour of the machine. Before a person buys a house, it is *inspected* for structural and many other problems.

It would be nice to think that all software engineering was subject to the same degree of care. It would be nice to think that after careful rumination by an impressive number of qualified people who together comprise a standards committee, the resulting programming language came into the world in a safe, well-defined and unambiguous manner, suitable for use by programmers necessarily less knowledgeable about the language. It would be nice to think that any resulting transgressions to this safe, complete and unambiguous language definition, would be immediately detected by a compiler before they can cause any damage, and the programmer responsible informed of this fact in a straightforward manner. In such a world, elephants fly and dreams always become reality.

In our world, nothing could be farther from the truth. In spite of the best efforts of our language experts, we continue to produce programming languages, which are not only unsafe, but frequently fail to acknowledge the fact. Compiler-writers compound the misery by providing compilers which are allowed to turn a blind eye to safety in obeisance to the great one-eyed god "Performance". Modern languages such as Ada, Fortran 90 and C++ further compound the misery by being very complex, and consequently exceedingly difficult to understand, and therefore difficult for which to write high-quality dependable compilers. As a result, much of the world's software in various languages is released full of inconsistencies, waiting for the right opportunity to develop into a real failure. These work in consort with inaccuracies in capturing the specification to reduce the reliability and safety of our software in an era when software becomes ever more pervasive with many tens of thousands and sometimes millions of lines of code in automobiles, televisions, fire alarms, medical scanners and aircraft. To see the scale of the problem, few if any organisations, have better resources or technology than NASA and yet Figure 1 shows only slow steady progress in eliminating errors, the bulk of which seems to be in improving the bad ones not the good ones.

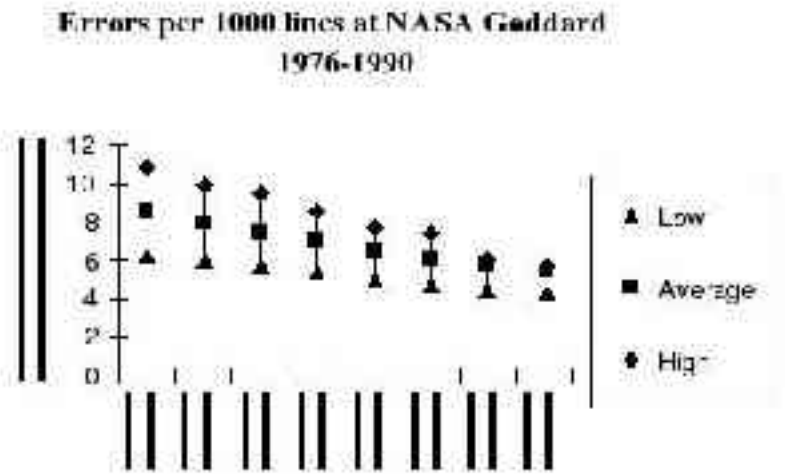

Figure 1. Graph published in the December 1991 special issue of Business Week showing the drop in software errors at NASA Goddard.

There is a nice analogy in railway engineering. It was (and perhaps still is) common practice for an engineer to walk alongside a stationary train, tapping the wheels and listening for cracks. An experienced engineer could tell not only whether a crack was likely to prejudice the safety of the train, but also whereabouts in the wheel it was. In contrast, detecting a crack dynamically, whilst the train is running was very difficult, although a small percentage of cracks couldn't be detected any other way. Nowadays, a suspect wheel would be subjected to various forms of scanning. Such "wheel-tapping" was remarkably effective. Regrettably static inspection - tapping the wheels of software - is absent from many software development environments, although there is considerable evidence as to its effectiveness, and mature tools and techniques to support its use in improving the reliability of the resulting product.

Take the language C for example. Its growth in both high-integrity and safety-critical areas has been little short of phenomenal in the last few years, in spite of a legendary reputation for allowing programmers enough rope to hang themselves. Excellent books such as [1] attempt to chart a path through the minefield, but such good advice is not frequently taken, as according to statistics published by [2], every 9th interface is incorrect on average in C and there is a statically detectable fault about every 42 executable lines *in commercially released code, irrespective of any quality system or even whether the code was safety-critical or not.* All of these were avoidable. Now some might argue that C is much worse than other languages and that this is to be expected. In fact, C is remarkably good at defining its shortcomings and numerous excellent supporting tools combine to produce a programming environment arguably as secure if not more so than any other language, including the much vaunted Ada, which although intrinsically safer, has less effective tool support, [2]. Of course, the ultimate safety of a programming language is *how safe it can be made in practice*.

So, in spite of all the emotional attachment to particular programming

languages common in software engineering, the presence of known inconsistencies in all commonly-used programming languages and their frequent appearance in released systems, suggests that real error rates in programs might be rather insensitive to the choice of programming language. In fact, this is precisely what is observed. This is evidenced by the raw statistics of [3], [4], [5], [6], [7], [8], [9] and others, whose work spans many systems written in various forms of assembly language, macro assembly language, Fortran, C and Ada. These results are brought together in Table 1.

Source	Language	Errors / KLOC	Formal methods used	Life-cycle
Siemens - operating systems	Assemblers	6-15	No	Release
IPL - language parser	C	20-100	No	Dev. only
NAG - scientific libraries	Fortran	3	No	Release
Air-traffic control	C	1.25	Yes	Release
Lloyds - language parser	C	1.4	Yes	Release
IBM cleanroom	Various	3.4	Part	Release
IBM normal	Various	30	No	Release
Loral - IBM MVS	Various	0.5*	Part	Projected
Basili & Perricone (1984)	Fortran	6-16	No	Release
Compton & Withrow (1990)	Ada	2-9	No	Release

Table 1: Error rates for systems written in various languages. The Loral data is a projected rate.

What can be done then ? From this author's point of view, there seems little point in investing massive resources in such paradigms as object-oriented design, an intuitively appealing concept for which there is unfortunately little or no empirical evidence in support of its effectiveness, when we can't even produce software which is self-consistent with respect to its language definition and which restricts itself to well-defined and well-implemented parts of that language. Perhaps we should solve this problem first. At least it has a simple solution.

References

1. Koenig, A., *C Traps and Pitfalls*. 1988, Reading, Mass.: Addison-Wesley. 146.

2. Hatton, L., *Safer C: Developing for High-Integrity and Safety-Critical Systems,*, McGraw-Hill, ISBN 0-07-707640-0, 1995.

3. Basili, V.R. and B.T. Perricone, *Software Errors and Complexity: An Empirical Investigation*. Comm. A.C.M., 1984. : p. 42-52.

4. Compton, B.T. and C. Withrow. *Improving Productivity: Using Metrics to Predict and Control Defects in Ada Software*. in *Second Annual Oregon Workshop on Software Metrics*. 1990. Oregon:

5. Davey, S., *et al*. *Metrics Collection in Code and Unit Test as part of Continuous Quality Improvement*. in *EuroStar'93*. 1993. London: BCS.

6. Hatton, L. and T.R. Hopkins. *Experiences with Flint, a software metrication tool for Fortran 77*. in *Symposium on Software Tools*. 1989. Napier Polytechnic, Edinburgh:

7. Moller, K.-H. and D.J. Paulish. *An empirical investigation of software fault distribution*. in *CSR'93*. 1993. Amsterdam: Chapman- Hall.

8. Ostrolenk, G., *et al*. *Cost- effective evaluation of a COBOL Parser using an operational Profile*. in *CSR'94*. 1994. Dublin, Ireland:

9. Hausler, P.A., R.C. Linger, and C.J. Trammell, *Adopting Cleanroom software engineering with a phased approach*. IBM Systems Journal, 1994. **33** (1): p. 89- 109.