Forensic Software Engineering: an overview

Les Hatton
CIS, University of Kingston, UK*

December 19, 2004

Abstract

Traditional software engineering is not really a branch of engineering at all as it lacks any kind of systematic measurement framework on which to base improvement, [16]. There are many different facets to the notion of software improvement but here, the focus will be on just one, that of the prevention of defect. In this context, Forensic Software Engineering is an amalgam of techniques specifically aimed at extracting patterns of failure associated with software controlled systems, categorising them and using the information to prevent future failures of the same kind. This is in fact a classic engineering paradigm but there is a particular need in the software community to isolate it from the unusually creative but generally measurement-free mainstream. This article introduces some of the ideas and hopes.

$Date: 2004/11/26 11:32:41 $

1 Overview

The discipline known somewhat incorrectly as software engineering and more accurately as software development, spans many areas from the extremely pragmatic such as testing, all the way through to the extremely abstruse such as formal approaches to system description. Arguably the only thing all these disparate areas share in common is the humble mistake. Software systems are so complex, (even a relatively small system could quite easily have more independent paths through it than the number of elementary particles in the known universe), that whatever approach is taken to build or verify it, it will be inevitably flawed and all too frequently very flawed. At one end of the spectrum, testing seeks to eliminate defects before the ultimate user of the system takes control and at the other end of the spectrum, formal systems attempt to eliminate defect by mathematical reasoning.

In spite of the disappointments, there is a continual production line of new development methodologies each in its own way seeking the Holy Grail of zero defect. Some are highly formalised and built around increasingly labyrinthine process models such as the CMM in its latest incarnations and some attempt to short-circuit such bureaucracy under the general heading of *agile techniques*.

*L.Hatton@kingston.ac.uk, lesh@leshatton.org

This is all supported by a vast array of languages, environments and tools to the extent where it is becoming extremely difficult to know what to teach to students of the subject. This is a central characteristic of an essentially fashion based subject.

Forensic Software Engineering has a very simple premise. It simply doesn't care what development methodology, language, environment or tooling is used to build a system, it is only concerned with collecting defect data from the resulting systems to avoid injecting the same defects in future systems built with whatever technology is in use. In point of fact, the original essence of CMM level 5 is precisely this although it is becoming increasingly difficult to see as the CMM itself grows and grows and grows. Ultimately such data may then be used to produce improved techniques founded on incremental improvement.

The hope of course is that the nature of defect obeys macroscopic and as yet largely undiscovered rules. This is perhaps not an unreasonable hope. A physical system such as the atmosphere in a room is an immensely complex system with many molecules in random motion, however the whole system follows the macroscopic thermodynamic rule PV = RT (where P is the pressure, V the volume, T the temperature and R the general gas constant) across an astonishingly wide range of temperatures and pressures. If such systems could only be predicted from the movement of the individual molecules, no progress would have been made whatsoever.

There is some early hope that similar rules may exist for software systems and like their equivalents in the physical world give good predictive behaviour in the elimination of defect. It is certainly true that *the vast majority of failures in software controlled systems could have been avoided using techniques we already know how to do.* In essence, we do not have a technological problem, we have an educational problem, writ very large.

2 Some unpleasant reminders

In general, the software development community certainly does not make things any easier for itself. Perhaps the most important requirement to improve a process or way of doing something, whatever that might be, is that the rate at which the process changes naturally should be significantly slower than the time taken for process measurements to be analysed and used to modify that process. This has proven relatively easy to do in the manufacturing industries where literally incredible strides in failure avoidance have been taken against the background of relatively slowly moving processes with a solid measurement framework. So consumer devices such as hard disk drives or car engines are in some cases orders of magnitude more reliable than their ancestors of just 20 years ago. In bridge building, defect prone technologies such as the box-girder bridge are no longer used. The torsional failure mode of suspension bridges such as the eponymous Tacoma Narrows bridge is now avoided by increasing the skirt on each side of the road. The net result is that bridges are gratifyingly and reassuringly reliable, hard disk drives are frequently more reliable than the

back-up media we use to mirror them, car engines will often go 3-400,000 km. without the head being taken off and so on.

How does the software development community fare against this kind of comparison ? Well, in the same period, tens and probably hundreds of programming languages have come and gone. Universities teach languages which may never be used in the world of commercial systems. The argument frequently offered for this is that it is only to teach concepts but unfortunately in my experience, the notational complexity of most programming languages can make it very difficult indeed for students to extract concept. Those languages which are used are, since the year 2000, no longer controlled by any kind of internationally standardised validation process to ensure they behave according to the standards they claim to follow. Can anybody imagine a serious engineering profession throwing away its quality control procedures for arguably the most important tool it uses ? If the reader doubts this absolute reliance, it suffices to ask the question how many developers in practice inspect the output of the compiler ? Apart from one or two of the most safety-critical, the answer will be "no" and even with safety-critical systems, the majority of the generated code will not be inspected.

The situation is mirrored with operating systems. For a time until Linux came along, it looked as though the phenomenal reliability of Unix systems would be simply consigned to the past to be replaced by the all too frequently unreliable and constantly changing generation of consumer operating systems where even basic security is wanting. In 2004, the spectacle presents itself of the world's most widely used operating system breaking applications in an attempt to reach even the most basic of acceptable security levels and in so doing managing to block only incoming traffic whilst effectively disabling an excellent third-party application which does block bi-directional traffic, [18]. Worse, it is being deployed in places where its manufacturer specifically denies any form of suitability such as combat management systems in warships including those which control nuclear missiles. Here ubiquity is being confused with reliability.

3 The need for relatively slowly moving processes

Figure 1 shows the time-honoured method of improving a system by control process feedback. In essence if a product is produced by some process and the goal is to improve the product in some way, a complementary property of the product is measured. For example, if the reliability of the product is to be improved, its failure rates should be measured. These are then analysed and the process altered to avoid that particular failure mode.

This procedure is more or less single-handedly responsible for the dramatic improvements in consumer products described above. It makes one very important assumption however: the process must not change too quickly before the results of the measurements are fed back, otherwise a form of 'drunkard's walk' results. Software development of course is beset by rapidly moving processes

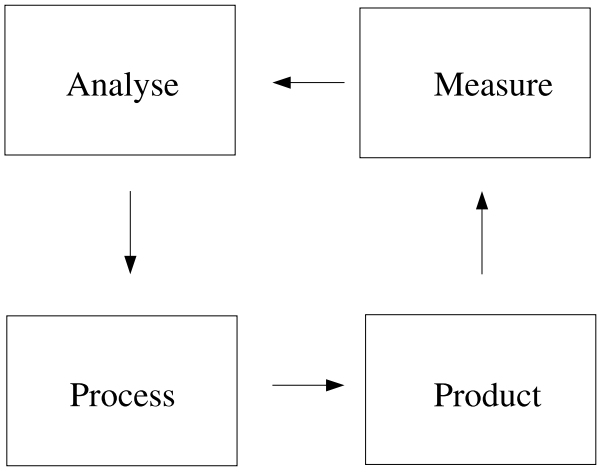

Figure 1: Control process feedback: the essence of engineering improvement

because it is essentially a fashion industry so it may not be so easy to apply although there are dramatic examples of improvements to code inspections resulting from such use, [3].

4 Potential areas for Forensic Techniques

Where and how can forensic techniques help ? To reiterate, the central requirements for any forensic technique are a suitable measurement system, a method of storing and analysing data and a reasonably slowly moving underlying process. Until relatively recently, measurement technology for software controlled systems was particularly crude or non-existent. Thanks to the efforts of centres like the Fraunhofer Institutes, [6], on the back of theoretical underpinning such as is provided by [5], some progress is being made.

The central focus of forensic techniques is the recognition, categorisation and elimination of defects. In this sense, a defect is a fault which has failed - some incorrect aspect of the system (a fault) has led to the system behaving in an unexpected way, (it has failed). Each of these phases has problematic areas.

Recognition On the face of it, recognition sounds the most trivial of phases and yet most users are familiar with the jokes which prevail about features versus defects. A defect must cause the system to behave in an unexpected way by definition but if the user does not have any expectations because the behaviour is undocumented, is this a defect ?

Categorisation In measurement terms, to be of use forensically, enough properties of the defect must be recorded. For example, at what stage of the life-cycle was it injected ? An area of current research is how such categorisations should be made in order to be of most use for forensic purposes. A defect might require multiple tags to be of most use. For example a failing division by zero could provide several forensic leads:-

- A requirements error, for example an attempt to invert a singular matrix without guards. A forensic response here might be to require all algorithms with potentially singular behaviour to be specifically treated in some way.
- An error of omission where the specification requires a guard but the guard is omitted for some reason. A forensic response here might be to investigate how well engineers guard against unusual conditions and to investigate if there are other places this might have happened.
- An implementation error where the programmer has simply used the wrong logic. A forensic response here might be to ask the question why was such a defect missed by the inspection teams ? Was it inexperience, (requiring more training) or perhaps an oversight (requiring a process change) ?
- A missing test. Should there have been a test case to check for this eventuality ? A suitable forensic response is to investigate the test process and test education of the engineers.

Diagnosis A further requirement for defect analysis is that the fault which has failed can actually be diagnosed. There is growing evidence, [11], that this is becoming increasingly more difficult as systems become more complex and we continue in failing to teach students the importance of failure diagnosis as a design procedure. If failures can not be related to the faults which caused them, the failures simply re-occur at some frequency. No discipline which pretends to the throne of engineering can allow repetitive failure modes to dominate its systems as is frequently the case with software-controlled systems.

The preceding comments probably understate the difficulty of acquiring high quality information from a failure as it is compounded by organisational secrecy when a failure occurs. As an example, on the day of writing this, the UK based bank *Cahoot* announced a major security problem which allowed users to log in to not only their own accounts without a password, but other people's accounts as well, [1]. This is one of a long series of Internet banking failures. The cause was stated to be "a systems upgrade" and the head of Cahoot bank stated "I believe that we need to look closely at our processes", a deliciously guarded statement. The bank was down for 10 hours. It is abundantly clear that the stated cause is completely useless for any kind of forensic work. Whilst this level of opacity of disseminated information on failure continues, forensic methods cannot help.[1]

4.1 Forensic Process Engineering

In forensic process engineering, the focus of attention is on process defects. Of course the real problem here is that many if not most software projects either, [15],

- Fail to produce anything at all, or
- Fail to produce the right product, or
- Produce the right product but too late to be of much use

The reasons are essentially human but much can be learned by analysing process failures, [2]. Even the most basic of process tracking can turn round a failing project. Figure 2 shows a very useful display for process tracking, this time on a project which is manifestly going nowhere. The display simply shows the result of asking the project managers each week for their estimated time to completion and then plotting this against the date on which the estimate was made. An ideal project would be a straight line with gradient at least -1. The reality in this case is very different. It is clear that this project, (an attempt to unify a graphical widget set portably across several platforms), is going nowhere. The project managers are simply guessing. The project was restarted under the following conditions:-

[1] It later transpired that the initially released statements were somewhat inaccurate. In fact the cause of the failure was that Cahoot, unlike other banks, enabled completion of password fields for 'user convenience'. The forensic response is quite correctly to desist from this practice.

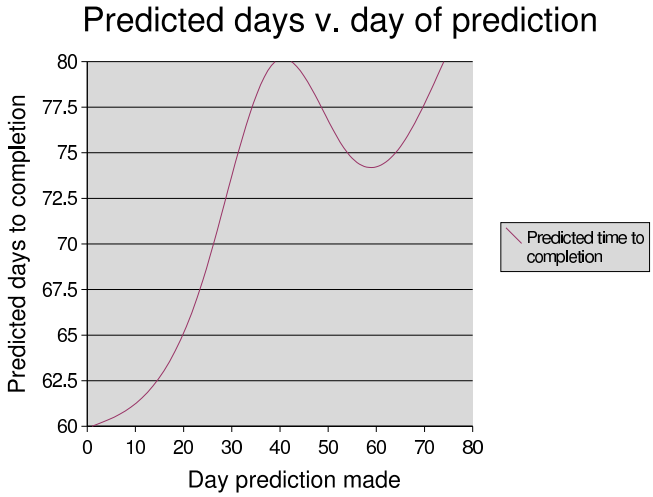

Figure 2: A plot of predicted days to finish against the day the prediction was made before intervention

- No individual's task was allowed to exceed 5 days. The rationale behind this is that programmers appear to have difficulty estimating much further than this reliably. Longer tasks had to be broken up until they could convince the project manager that they were viable,
- The project would be tracked weekly and the results published in this graphical form and displayed prominently.

The result is shown in Figure 3. The project ran over by 10% but is clearly in control throughout in stark contrast to Figure 2. This is scarcely rocket science but is the kind of simple control which is all too often missing in software projects big and small.

4.2 Forensic Product Engineering

Forensic product engineering simply covers failures of the software product itself. As was intimated by the divide by zero example above, these may arise for a variety of reasons. However, once again, the primary focus is to identify patterns of failure leading to preventative methods. There are many opportunities for these as so many product failures occur repetitively, indeed programming language development generally fosters them as the following example illustrates.

Precedence Precedence in programming languages is a perfect example of a repetitive mode failure. Dennis Ritchie one of the original authors of the C programming language is reported in [17] as identifying commonly occurring failures as long ago as 1982, based on the precedence table of C [2] having some of its levels not intuitively 'obvious'. Not only do such failures still occur regularly, [8], but the C precedence table forms the heart of all the C-like languages such as C++, Java, Javascript, Tcl/Tk, PHP and Perl developed from C in subsequent years. As a result the same problems occur in these languages also. In other words, not only has no particular effort been made to remove this repeating failure mode (for good reasons at the time) but the problem has been amplified for reasons of 'backwards compatibility'. Backwards compatibility is an interesting concept and is largely absent from other engineering disciplines because it inhibits, and in extreme cases like the IT industry, may entirely prevent the ability to learn from mistakes. The reader may like to pause a moment and imagine a world where mainstream engineering industries were forced to repeat past failures by a policy of backwards compatibility. This subject is considerably expanded in another paper in preparation by the author.

Note that product failures can be glaringly obvious, such as a program crash or insidious, leading to subtle but expensively misleading failures. It is the author's opinion with some justification, [7], that the output of most significant mathematical simulations is at least tainted and perhaps fatally compromised by software failure.

[2] which has an awe-inspiring 15 levels and is still eclipsed by Perl and PHP which have 21 and 23 levels respectively

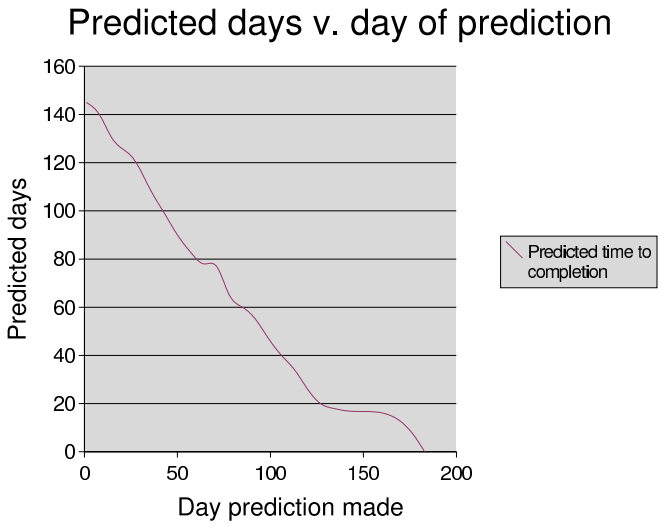

Figure 3: A plot of predicted days to finish against the day the prediction was made after project restarted

4.3 Forensic Systems Engineering

In this context, the word *Systems* is used to cover the environment in which software products are developed and includes but is not limited to:-

- Operating System Environment
- Security
- Arithmetic implementation quality
- Compiler and tool quality

In some cases, forensic work may only serve to reject an alternative rather than help improve it. This might occur for example, when selecting an operating system with a mean time between failures of greater than 1000 hours which currently seems to preclude all but the Unix like operating systems, [9]. In other cases, forensic work in this area simply overlaps product analysis as with security where many problems occur because of *buffer overflow*, a programming fault. In yet other cases such as with arithmetic implementation or compiler quality, actions by the community at large such as the decision in April 2000 which essentially terminated independent quality validation of compilers for programming languages as described earlier, lead to the forensic conclusion that they be reinstated as quickly as possible or that users of compilers must do it for themselves. With arithmetic implementation quality, this is at least possible through publically available downloads of *paranoia*, [12] or *embedded system paranoia*, [10].

5 Data mining and Defect Analysis

Data mining forms an important part of Forensic Software Engineering as potentially useful data is often unstructured. If data is formatted in a text format such as plain text or some structured form of plain text such as CSV files, XML, HTML or some other relatively easily accessible format, important information can often be recovered without too much effort. If on the other hand, data is buried in the ever-changing binary formats of programs like Microsoft Word, data mining is effectively crippled and such file formats should never be used to contain important data.

Defect tracking and analysis is often done poorly and yet defect data is the single most important source of information a company aspiring to improve itself can have. In an industry which frequently makes sweeping, unsupported and regrettably inaccurate statements about the potential of a new technology, the only thing an organisation can often rely on is its own data and the data needs to be sufficiently comprehensive to allow improvement strategies to be extracted, [14]. This is another area in which open source has had an important impact with the admirable *bugzilla*, [4].

6 Conclusions

Forensic software engineering is not rocket science but does require a systematic approach based around the careful acquisition of failure data, its analysis

and subsequent application to discover ways of avoiding failure, ensuring that a failure of any kind can only happen once. So many things in software development today conspire against it, including but not limited to, massive growth of systems, increased complexity and coupling, poor diagnostic procedures, reliance on fashion-based statements rather than measurement-based statements, continual and frequently unnecessary replacement of technologies - the list is long. However, the current levels of failure, their potential criticality and their massive cost, [13], all bear witness to the need to improve our understanding of the causes of failure in software controlled systems and from that, its effective prevention.

References

[1] BBC (2004) *Cahoot hit by web security scare* http://www.bbc.co.uk/1/hi/business/3984845.stm

[2] Dalcher, D (2004) *Software Forensics Centre* http://www.cs.mdx.ac.uk/research/SFC

[3] Gilb, T. and Graham D. (1993) *Software Inspections* Addison-Wesley, ISBN 0-201-63181-4

[4] Bugzilla (2004) http://www.bugzilla.org/

[5] Fenton, N. Pfleeger, S.L. (1997) *Software Metrics* PWS, Boston, ISBN 053495425-1

[6] The Fraunhofer Center for empirical software engineering (1997-) http://fc-md.umd.edu/

[7] Hatton L., Roberts A. (1994) *How accurate is scientific software ?* IEEE Transactions on Software Engineering, October 1994.

[8] Hatton L. (1995) *Safer C: Developing high-integrity and safety-critical systems* McGraw-Hill, ISBN 0-07-707640-0

[9] Hatton L. (2004a) *Forensic Software Engineering* Lecture notes: http://www.leshatton.org/Forensic_1.html

[10] Hatton L. (2004b) *Embedded Systems Paranoia: a tool for testing embedded system arithmetic* Information and Software Technology, to appear, 2004

[11] Hatton L. (1999) *Repetitive failure, feedback and the lost art of diagnosis* Journal of Systems and Software, (10), October, 1999

[12] Kahan W.M. (1983) *Documentation header of the original paranoia* http://research.att.com/ http:/www.catless.ncl.ac.uk/Risks

[13] Neumann, P.G. (2004) *The Risks Forum* http:/www.catless.ncl.ac.uk/Risks

[14] Pfleeger, S.L., Hatton L., Howell C. (2002) *Solid Software* Prentice-Hall, New Jersey, ISBN 0-13-091298-0

[15] Royal Academy of Engineering (2004) *The challenge of complex IT projects* Royal Academy of Engineering, London, ISBN 1-903496-15-2, www.raeng.org.uk

[16] Tichy, W.F. (1998) *Should computer scientists experiment more ? 16 Reasons to Avoid Experimentation* IEEE Computer, 31(5), May 1998, p32-40

[17] van der Linden, P. (1994) *Expert C programming: Deep C secrets* Prentice-Hall, New Jersey, ISBN 0-13-177429-8

[18] Zone Alarm (TM) (2004) http://www.zonelabs.com/