"Linux and the CMM"

by

Les Hatton[1]

Version #1.4, 13th. Feb, 2000

[1] Oakwood Computing and the Computing Laboratory, University of Kent, UK lesh@oakcomp.co.uk

The Linux phenomenon

Not very long ago, few people had even heard of Linux. Today, it is on many people's lips as a viable alternative operating system to the Windows-dominated PC world. This rapid turnround in visibility is of great interest to software testers because the primary reason for the rapid take-up of Linux is its extraordinary reliability. Perhaps the most surprising aspect of Linux is that this reliability is the product of an environment which traditional software process models would probably classify as 'chaotic', a status I will confirm by performing an informal assessment against the widely-known software process model, the Capability Maturity Model originated by the Carnegie-Mellon SEI as described for example in (Humphrey 1990). Are process models that wrong or is there some aspect of the development of Linux which naturally imbues reliability ? In this article, I will try and answer these questions.

A brief history of Linux

Linux is now a sophisticated implementation of a multi-tasking, multi-threaded symmetric multi-processing operating system which is very highly portable. It is available on various architectures including Intel, Alpha, PowerPC and Sparc. It had humble beginnings and started off as a project of Linus Torvalds in 1991, then a computer science research student in Finland. He wanted to design and build a small operating system based on the architecture of Unix. He published his intention and first version on the net under the name Linux for any interested parties and the rest as they say is history. Users contribute by volunteering their time and then working on a product communicating by e-mail.

The key step forward was its link-up with the GNU project founded by Richard Stallman in the 1980s at MIT, (http://www.gnu.org/). The eventual intention of the GNU project was to produce a "free" version of Unix. The marriage of Linux with GNU completed the loop.

From then on, Linux gathered momentum rapidly. Various key technologies such as the hugely impressive Samba, an implementation of the SMB Windows network protocol have since been added by committed volunteers around the world and today, the distribution is simply stunning. Perhaps best of all, installation has been improved beyond all recognition. My last installation of Red-Hat 6.0 some

weeks ago took about 15 minutes in contrast to contemporaneous installation of Windows NT4.0 which took well over an hour including a non-diagnosable and entirely inscrutable initial failure.

Linux reliability

Before proceeding with an analysis of Linux, I will try and put a little more flesh on the assertions that Linux is reliable, at least in its common configurations. Certainly, in my own hands, during the 3400 hours or so of development work I have amassed in the three years I have been using it exclusively for development, I have had only one system crash (i.e. necessitating a reboot) with any version of Linux on a variety of machines. In contrast in the same time period, I have had on average around two crashes per 8-hour day on Windows 95 and Macintosh with extremes of perhaps 5-20 times a day if any kind of software development or communication work is done. A brief experience with Windows '98 was enough to convince me to remove it altogether. Crashes appear to occur about 5-10 times less frequently with my NT systems, (i.e. of the order of 2-3 times a week) but I do not use NT much so my data is not sufficient to estimate relative reliability against the others with any accuracy, (Hatton 1997). Figure 1 summarises.

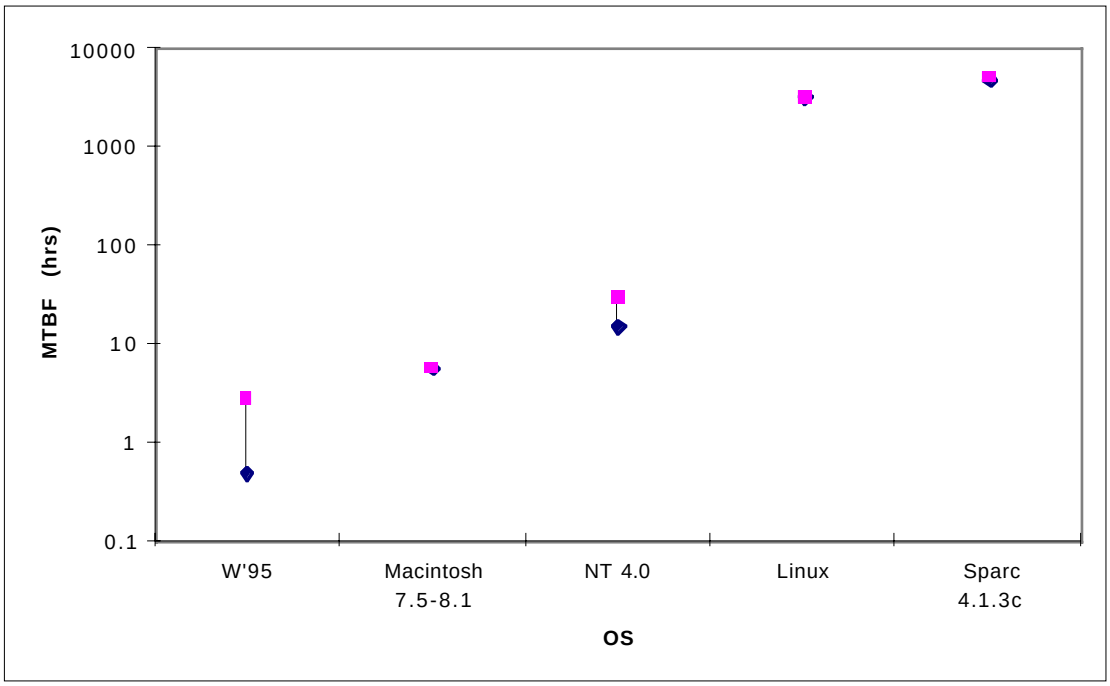

Figure 1: A comparison of my own experiences with operating environment reliability. The operating environment includes application software and in this case MTBF indicates the mean time between the appearance of a defect, including those which caused a crash. For Linux and Sparc, this is a software development environment whereas for the others it is only an 'office' environment although the lower range of Windows '95 includes salutary attempts at software development.

Looking further afield, I came across a number of articles which make specific reference to Linux reliability. For example, (Petree 1998) discusses it in a large scale power grid distribution monitoring role in Central Virginia noting that "the systems software has never failed nor caused us a single problem through thousands of hours of continuous uptime". (Shoham 1998), discusses Linux in a banking role and observes "During stress testing, we discovered 2 separate OS bugs in MVS (IBM's venerable mainframe operating system), but no problems and no bugs in the Linux servers." In an Internet application, (Ogbuji 1998) commented "for several months a single Pentium Pro-based server running Linux ran mail, DNS, central logging, IMAP, SMB and WWW for over 1000 users with little or no downtime." (Bertozzi, Broggi et al. 1999) describes how an electric car equipped with a Linux PC, CCD cameras and various other instrumentation was driven round the Mille Miglia and various public roads in Northern Italy, (a total of some 2000 km.), achieving a remarkable figure of 90% hands-off driving. The authors stated "the number of faults found in the (Linux) system components was zero over the last two years". I should note that all these articles come from the Linux Journal but the detail in which the Linux implementation is discussed add a credibility which is hard to dent.

So why is it this good ?

An informal CMM assessment

The belief of any software process model is of course that adhering to a defined set of procedures will be positively correlated with the production of software which is in some sense of higher quality. This might mean more cheaply produced, closer to the intended requirements of the user or whatever, but one implicit requirement is that the software should be more *reliable*. It is hard to imagine that an unreliable product even if produced efficiently and on time would be particularly attractive. The key question is of course, is there any evidence to justify that software process models help to produce better software by whatever criterion we happen to define as higher quality.

Basic goals of the CMM

It is not the purpose of this paper to describe the CMM in any detail - this has been done on countless occasions before. Here I will simply describe enough of the model to help us to assess the Linux development process. First I will briefly describe the goals for each level of the CMM.

The CMM is a five level model from levels 1 to 5 as shown in Table 1. Level 1 is perceived as being the lowest level and 5 the highest. The essential element of the CMM is that each level from 2 to 5 is characterised by certain attainable goals.

CMM level / name	Goals to attain for classification at this level
Level 5 - optimised	Defect Prevention Technology Innovation Process Change Management
Level 4 - managed	Process Measurement and Analysis Quality Management
Level 3 - defined	Organisational Process Focus Organisational Process Definition Training Program Integrated Software Management Software Product Engineering Intergroup Coordination Peer Reviews
Level 2 - repeatable	Requirements Management Software Project Planning Software Project Tracking and Oversight Software Subcontract Management Software Quality Assurance Software Configuration Management
Level 1 - chaotic	Being able to walk and chew at the same time Holding a simple tune

Table 1: A slightly edited version of the key goals of the CMM at each level.

As I describe the goals for each level, I will attempt to interpret how the Linux development process relates to each.

Level 2 - a repeatable process

The key goals to attain for this level are:-

Requirements management

The system requirements should provide a clearly-stated, verifiable and testable foundation for software engineering and software management. The first thing to realise here is that there is no management in Linux in the widely understood sense, although one the senior Linux engineers, Alan Cox (Cox 1999), thinks that the 'no managers in Linux' is a myth with people naturally forming classic teams of around 6 but noting that communication paths are massively more flexible. In the early

days of course, Linux was working very closely to an existing and time-proven API, but this is much less true today with the evolution of threads, symmetric multi-processing and complex applications such as Apache, (the dominant web serving software).

Software Project Planning

This provides for the development of a plan against which software activities and commitments are tracked. The development of the heart of Linux, the Linux kernel is still under the control of Linus Torvalds who maintains a plan of kernel work but it does not appear to be documented with the degree of formality the CMM contemplates.

Software Project Tracking and Oversight

In essence, the CMM requires projects to be tracked and corrective procedures to be put in place for significant deviance from plans. Linux again is far more informal although kernel work seems to be well-defined and tracked and inter-developer communication seems very comprehensive. In contrast to the kernel, development of Linux applications proceeds almost by Darwinian evolution - what works, attracts support and what doesn't work or doesn't get finished doesn't attract support and withers. I will discuss this later as I believe it is of major importance in understanding Linux and similar developments.

Software Configuration Management

Controlled and stable baselines should be established for planning, managing and building the system. In this regard, Linux is exceptionally strong and its influence should not be under-estimated as this particular issue is a common failure in companies aspiring to CMM level 2.

Unix has historically been very strong in development support and it should come as no surprise that some of the earliest GNU products included the excellent Revision Control System due to Walter Tichy. A later product building on top of RCS was added under the name CVS. There are others. In my experience, the use of one or other of these is absolutely universal amongst Linux / GNU developers and every released product has a revision number. On top of this the Red-Hat developed product RPM provides arguably the best vehicle in the industry for updating dependent software products in a strictly controlled manner.

If the reader would like to see a living example of such a regime, simply take a look at the Mozilla site, (http://www.mozilla.com/).

So in a critical and often-neglected area, Linux development is of quite outstanding quality.

The next two items simply have no equivalent within Linux.

Software Subcontract Management

Amongst other things, the prime contractor tracks the subcontractor's actual results and performance against the commitments.

Software Quality Assurance

Here, compliance of the software product with applicable standards, procedures and requirements is independently confirmed. Management issues also form a big part of this CMM category.

Level 3 - a defined process

The key goals to attain for this level are:-

Training Program

The staff and managers have the skills to do their job. Linux is very strong on this - it attracts committed and generally very capable engineers who wish to do something 'significant'. There is no training program however. It is the archetypal example of 'on-the-job' training but it is common to find exceptionally experienced developers working on it in their spare time often as cathartic response to the 'day job'. In this regard, Linux is almost an art form. Developers, proud of their ability, seek to display this under the 'open source' model.

Software Product Engineering

State-of-the-practice software engineering tools and methods are used, as appropriate, to build and maintain the software system. Linux is very strong on this and builds on the unprecedented range of fine tools developed in the precursor GNU project. In this regard, I am reminded of the19th century American clockmaker, Eli Terry. I came across this in the truly wonderful Henry Ford museum in Dearborn and I would like to share it with you as it makes breathtaking reading considering it was done nearly 200 years ago.

"In 1807 Eli Terry, accepted a contract to make an astounding 4,000 clocks in three years. He spent the first year producing special machines and gauges, which he used during the second year to produce thousands of interchangeable parts. Over the course of the third year, groups of workers assembled the clocks, successfully completing the contract to the amazement of the clock-making community."

Intergroup Coordination

The project's technical goals and objectives are understood and agreed to by its staff and managers. The project groups work as a team. Again Linux is strong on communication, (although not always on agreement !). Developers all over the world coordinate their work informally via e-mail and discussion groups.

Peer Reviews

Amongst other things, a rigorous group process for reviewing and evaluating product quality is established and used. *In this area, as in configuration management, Linux is again exceptionally well served*. Because of the open source model of development, every line of code gets inspected in detail although informally, by numerous people. I will come back to this later as one of the likely causes of the enviable reliability of Linux.

The last three items simply have no equivalent in the Linux development lexicon.

Organisational Process Focus

A group is established with appropriate knowledge, skills and resources to define a standard software process for the organisation.

Organisational Process Definition

This refers to the definition of the standard software process.

Integrated Software Management

The CMM here provides for the use of data acquired from previous projects to assist in new projects.

I will not pursue any further discussion of the higher levels of the CMM. It is clear from the above that Linux development has key failures at both levels 2 and 3 and is therefore a resounding level 1. However, in those areas where it is strong, it is in

my experience considerably stronger than just about every commercial development team I have ever seen.

What can we learn ?

So we have seen something of the development history of Linux and an informal assessment against the CMM confirms what we might have expected of a completely voluntary effort. It is firmly rooted at level1, known politely as chaotic, whilst simultaneously having areas of quite exceptional strength. What can we infer from this ? Before I attempt to make some predictions and conclusions, let me add a few more observations about Linux development which may help throw some light on the original question.

Evidence for correlation between process maturity and higher quality

The evidence for such correlation is somewhat conflicting depending on how higher quality is defined. For example, (Puttnam 1992) reports a significant correlation between attributes such as shorter time, lower cost and less staff and an increasing CMM level. In contrast, (Hatton 1995) reported that there was no obvious relationship between the statically detectable residual fault rate in a wide population of C and Fortran programs and whether or not the software was produced by following a formal process model. Since the explosive growth of the 'software process industry' in the last 10 years, it might be expected that there had been a commensurate cut in the number of delivered residual faults but there is no clear evidence of this either, (Hatton 1998). As a result of this it is difficult to argue a clear-cut case and the main effect of formal process models seems to be to reduce the not insignificant risk of nothing appearing at all, and to reduce lateness in those products which are delivered.

Individual technologies such as code inspections when performed well appear to have a much more pronounced effect on a reduction in the number of delivered defects, (Gilb and Graham 1993), (Liedtke and Ebert 1995) and many others. It is of course perfectly possible to have a well-defined and closely followed software process which does not contain such dramatic technologies. In this may lay the key to understanding the central question addressed in this paper - why is Linux so good in the apparent absence of a formal process model ?

Significant differences between Linux and the CMM

Quality of people

It has been known for a very long time that people vary dramatically in their ability to produce reliable, clear software implementations. For hard evidence of this, the experiment of (Knight and Leveson 1986) is particularly noteworthy. In this experiment, 27 implementations of the *same* algorithm were produced in the *same* programming language. The resulting programs varied between 327 lines and 1004 lines and on a million tests, the worst failed nearly 10,000 times whilst the best did not fail at all. In (Hatton 1997), another example is given of two independently applications of comparable size, (around 70,000 LOC) in the same application area, (regrettably nuclear engineering), and the same programming language, (Fortran 77). One had a heart-stopping 140 statically detectable defects per KLOC (1000 lines of source code), the other had *none*.

It is clear then that whatever process model is used, the quality of implementation staff is a crucial quality factor in software development. Linux in particular and open source software development generally attracts some extraordinarily capable and committed people. I have had personal experience of this. I have twice requested technical assistance since I started using Linux, once on graphical device implementation and once on the status of symmetric multi-processing. For the former I got two detailed responses from post-graduate computer science students who very obviously knew their stuff and for the latter an internationally known professor of parallel systems answered. In both cases, responses were received in less than an hour. My attempts to get high-quality support with commercial companies, even when paying for it, have been rather less successful and I am being charitable at this point.

Code 'inspections'

Because of its open source development model, Linux is subjected to the equivalent of code inspection by a considerable number of very capable people on a scale larger than any commercial software process I have come across. As I have already described above, inspections follow a number of different competing models with no apparent 'market-leader' but all are known to be exceptionally effective at the early elimination of defect even when carried out informally. In most commercial software inspections, the general feeling about inspections seems to be that they are a necessary evil. In Linux development, they seem to be genuinely supportive in nature. For more discussion of the effectiveness of the open-source model in this

regard, see (Raymond 1998), (http://www.tuxedo.org/~esr/writings/cathedral-bazaar/).

Kernel development

The heart of Linux , its kernel, is still controlled by one very capable person. It has therefore maintained an intellectual coherence which is quickly lost in distributed developments. The importance of this has been noted on a number of occasions, for example, by (Curtis, Krasner et al. 1988) and also by (Coplien 1994) amongst others.

Deadlines

Linux sets itself deadlines in the sense of best endeavours and general goals but in every other regard, there is no commercial pressure to finish something by a certain time. This is almost certainly a factor but I am unable to quantify it at this stage although there are certainly targets and no doubt emotional pressure.

Darwinian Software Development

I'd like to digress a little about Darwinian evolution as so much of Linux development reminds me of standard evolutionary theory, (Dawkins 1996). According to this, the shape and behaviour of species today is solely governed by how successful they are at surviving and reproducing. This is strongly reminiscent of today's software engineering industry.

In normal software development, we have a certain set of intended goals for the resulting software. We spend a great deal of time and money getting as close as possible to these goals. We very frequently fail. Sometimes we produce nothing significant, sometimes we produce the wrong thing and sometimes we produce the right thing but nobody wants it. In open-source software development, we simply produce everything usually several times and cast them onto the seas of chance. In the open-source model, anybody can look at any one of these potential programs so not only does the author take it personally and do the best they can because they can't hide behind the anonymity of object code, but also if any other person likes the concept, they can add to it or correct some defects. The next person to pick it up does the same. Each open-source program is therefore subject to something very similar to Darwinian evolution. The ones that survived you see as Linux or GNU or whatever. For every part of Linux / GNU that survived there are very

likely a large number that simply didn't make it. The ones you see are very reliable because you don't see all the ones that weren't.

In commercial closed-source development, there is no mechanism for evolutionary adaptation based on parallel development. Note that this is *not* the same as when a manufacturer makes a pledge to improve the reliability of an existing product because somebody else's product is better. In fact, this may often be a pledge which is impossible to fulfil as unreliability tends to be a persistent property and complex products which start unreliable tend to stay that way. In this case, an attempt is being made to adapt an existing dinosaur to a possibly unattainable goal. The 'gene pool' of open-source development is incomparably richer and much more likely to succeed in evolutionary terms.

Conclusions

We can conclude that in the case of Linux, the lack of a formal software process model and its obvious level 1 CMM status has not been a handicap to the production of an exceptionally reliable system which is also capable of very rapid development. The key factors in the *reliability* of Linux seem to be (in no particular order):-

- Unusually high average capability of engineer and very high commitment
- Unusually large-scale although informal code inspection process
- Unusually effective change and configuration control
- The kernel is carefully controlled by one person although helped by a number of others.
- Normal commercial pressures to complete are absent.

What lessons can companies learn from this ? Well we cannot conclude that companies can simply relax in their process vigilance and return to the bad old days where 'programmers like to program' in the complete absence of any form of control. Open-source development in general and Linux in particular simply have different priorities. We can learn from these differences though, perhaps most importantly, that engineer competence and large-scale code inspection are yet again implicated in high-reliability development. Not only that, but distributed open development appears to produce simpler systems.

I should finally mention that Alan Cox, (Cox 1999), attributes the general success of Linux to the simple engineering virtues of:-

- Peer review
- Sharing of proven ideas
- Favouring the proven over the new
- Continuous re-evaluation and re-assessment
- Controlled regular changes

As an engineer myself, I could not agree more. Linux is obviously destined to become of major importance. Perhaps the most exciting development to myself as a researcher in high-integrity and safety-critical systems would be a Linux kernel suitable for embedded control systems. The open-source model would then give collective responsibility for this most important development area and lead to a level of inspection and visibility which could provide the degree of reliability necessary in such applications.

Acknowledgements

I'd like to thank Brian Marick for the many helpful comments he made during the review of this paper. My thanks also to the Linux development community and especially Alan Cox who gave me some very useful feedback.

References

Bertozzi, M., A. Broggi, et al. (1999). "Autonomous Vehicles." Linux Journal(March 1999): p. 40-45.

Coplien, J. O. (1994). A new look at process, quality and productivity. Fifth Borland International Conference.

Cox, A. (1999). Comments on Linux v. CMM, e-mail.

Curtis, W., H. Krasner, et al. (1988). "A field study of the software design process for large systems." CACM **31 (11)**(November): p. 1268-1287.

Dawkins, R. (1996). Climbing Mount Improbable. London, Penguin Books Ltd.

Gilb, T. and D. Graham (1993). Software Inspections. Wokingham, England, Addison-Wesley.

Hatton, L. (1995). Safer C: Developing for High-Integrity and Safety-Critical Systems., McGraw-Hill.

Hatton, L. (1997). “Software failures - follies and fallacies.” IEE Review **43**(2): p. 49-54.

Hatton, L. (1997). “The T experiments: errors in scientific software.” IEEE Computational Science & Engineering **4**(2): 27-38.

Hatton, L. (1998). “Programming technology, reliability, safety and measurement.” IEE Computing and Control Engineering Journal **9**(No. 1, Feb, 1998): 23-27.

Humphrey, W. S. (1990). Managing the Software Process, Addison-Wesley.

Knight, J. C. and N. G. Leveson (1986). “An experimental evaluation of the assumption of independence in multi-version programming.” IEEE Transactions on Software Engineering **12**(1): 96-109.

Liedtke, T. and C. Ebert (1995). On the benefits of reinforcing code inspection activities. EuroStar'95, London, November, 1995.

Ogbuji, U. (1998). “Linux for Internet Business Applications.” Linux journal(55 (November)): p. 42-45.

Petree, V. (1998). “"Virginia Power Update".” Linux Journal(54 (October)): p. 48-53.

Puttnam, L. H., Myers, W. (1992). Measures for excellence: reliable software, on time, within budget. West Nyack, NY, Prentice-Hall.

Raymond, E. (1998). The Cathedral and the Bazaar.

Shoham, I. (1998). “Linux in Banking.” Linux Journal(56 (December)): p. 69-71.