The automation of software process and product quality

L. Hatton
Programming Research Ltd. Waynflete House, 74-76 High Street, Esher, Surrey, KT10 9QS, UK

ABSTRACT

Software engineering today can be compared to a manufacturing industry in which a product is generally produced without any recognisable production line. In any other engineering discipline this would be unacceptable but appears to be the norm for software producing companies. For example, the Software Engineering Institute (S.E.I.) at Carnegie-Mellon University have found that approximately 85% of all software producing companies are at the lowest level of *software process maturity*, officially deemed *chaotic*, c.f. Humphrey [1] The S.E.I. defines this to mean that they are deficient in one or more of the following key areas: Project Estimation, Project Planning, Configuration Management or Software Quality Assurance. Other studies show that software product quality, the intrinsic quality of the code itself, is similarly poor, with most commercially released packages riddled with errors *which could have been detected by inspections alone*, Hatton [2]. The net result is that users are subjected to unreliable software, public safety may be prejudiced and software developers pay far more in both development and maintenance than they need.

It has often been argued in the past that this is the nature of software engineering and users should accept this. This is no longer the case. Whilst there is no magical solution, tools already exist to quantify product quality and automate many key aspects of both inspection and testing. Other tools exist to automate the process of change around recent standard software process models such as ISO 9001 and the Carnegie-Mellon CMM (Capability Maturity Model). This paper will discuss such tools and how they

can be integrated to automate software process management with software product quality control. Experiences with such an environment over the last two years will be presented as a case history.

INTRODUCTION

It has been proven through a number of studies around the world that adopting an appropriate Quality Assurance philosophy can significantly improve productivity. For example, results published in the 11th Feb., 1991 edition of Business Week of a 12 month study of a Japanese software company adopting quality assurance techniques in the production of software borrowed from their other manufacturing industries, and a comparative US software company show the following:

	Lines of source delivered per man year of work	Technical failures per 1,000 lines during first 12 months
USA	7,290	4.44
Japan	12,447	1.96

The role of measurement must not be underestimated. In any quality system associated with manufacturing, a key element is the notion of measuring the efficiency of the process used and using that information to improve the process. Such measurements may be based on the process itself or on the products produced by the process. This is the cornerstone of the statistical techniques pioneered by W Edwards Deming and used with such dramatic success by Japan over the last 40 years or so. It is these techniques which form the basis of the recently emerged Carnegie-Mellon CMM (Capability Maturity Model) 5 level model, which is rapidly assuming importance in the U.S. and elsewhere. In Europe, the pre-eminent quality standard to which people aspire is based around the international standard, ISO 9001, which although a general manufacturing standard, has been translated via the U.K. TickIT initiative into the software vernacular in the form of ISO 9000-3. The growing importance of such standards is based around the fact that the E.C. now endorses ISO 9001 and the D.o.D. endorses the CMM. Mandatory compliance seems just around the corner.

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

It is generally acknowledged that management is impossible without objective measurement to check its progress. This applies to projects in all disciplines. If working practices and the reliability of what is produced are to be improved, it is vital to define the initial state and the final goal and to measure progress continually in order to get there.

The problem with software of course, is what precisely must be measured and how can the measurements be used.

PROCESS AND PRODUCT QUALITY

As has been mentioned, the software *process* is the process whereby a software *product* comprising both code and documentation is constructed. The overall relationship is shown below.

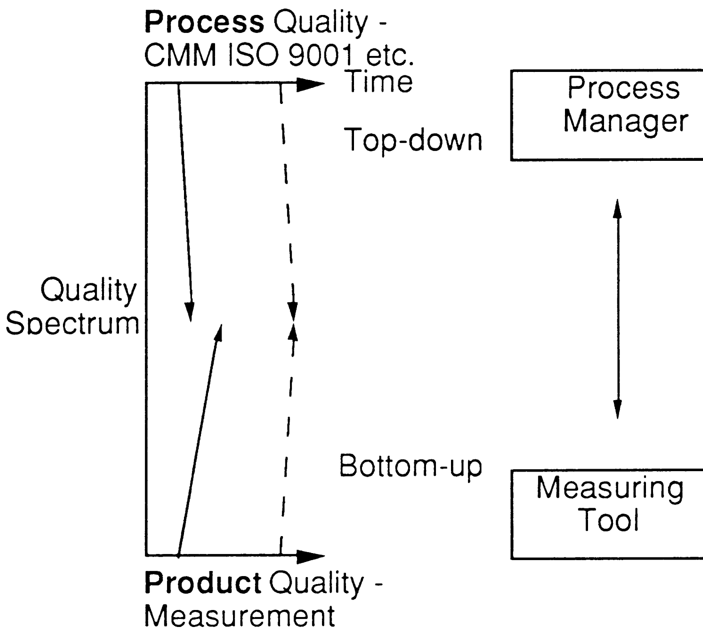

Figure 1

The left hand part of this diagram is intended to show that the improvement of modelling using physical measurement is iterative. In essence, as in all measurement based sciences, measurement drives modelling. As understanding grows and models become more sophisticated, so the models can be used to infer better

measurements and so on. The right hand side of the diagram shows how to automate this methodology for software development. The Process Manager performs process control which includes but is not limited to change and configuration control and uses the product measuring tool to determine whether the product is conformant and therefore acceptable. *It is vital that this quality control function is done automatically in conjunction with the management of change and configuration, as will be discussed later.*

For a programming language, the product is source code and the measurement tool might be an *automated inspection* tool used for measuring the objective quality of the source code itself. Such quality could be defined in a number of ways. It might be for example, the degree of dependence on unsafe parts of the underlying language or a measure of its complexity and resulting maintenance cost implications. For other entities such as design documents, the concepts are precisely the same, although the quality measures would be different.

MEASURING THE PRODUCT

For source code, product measurement tools may take many forms but should support a Quality Assurance philosophy in the following key areas:

- Automatic Standards Enforcement
- Portability checking
- Reliability checking
- Complexity analysis
- Architectural analysis

The first issue, that of maintaining programming standards automatically, is closely interlinked with the concept of accountability in ISO 9001. Since Quality Assurance requires *"verifiable adherence to standards which are widely believed to improve software quality"*, the importance of a programming standard cannot be overemphasised. For example, Hatton [2] found that companies currently break their own programming standards once in about every 168 lines, a dismal state of affairs which calls into question the point of having a standard in the first place.

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

The second two issues together relate to a ***safe subset*** of the language. Working with such a subset is known to eliminate many classes of errors at the coding stage, saving time and resources that would otherwise be spent debugging and testing. Hatton [2] found by analysing millions of lines of C and Fortran from around the world, that dangerous abuses of the programming language occur at about the rate of once every 80 lines or so in C and once every 150 lines or so in Fortran on average. *These statistics were compiled by measuring commercial production code* and are cause for great concern particularly when it was found that safety-critical codes were no better than non safety-critical codes in this area.

Many large organisations (such as IBM and AT&T) have found that dense, complicated code is inherently less reliable, less readable and less maintainable than well-modularised code. It is also much harder to test properly. This is self-evident, but without tools able to make such assessments of code quickly, it is not possible to tell which parts of the code are likely to cause problems. A number of standard complexity metrics are known which can be used to measure the readability of code, the testability of the logic and the reusability of the components, (c.f. for example, Akiyama [3], Boehm [4], Conte, Dunsmore & Shen [5], Fenton [6], Hatton & Hopkins [7], McCabe [8], Nejmeh [9], Shooman [10], Brandl [11]). By calibrating these metrics on packages with a known maintenance history, (Hatton and Hopkins [7]), the relationship between complexity (quality) and maintenance costs is found to obey rules similar to the below: (See Fig. 2)

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

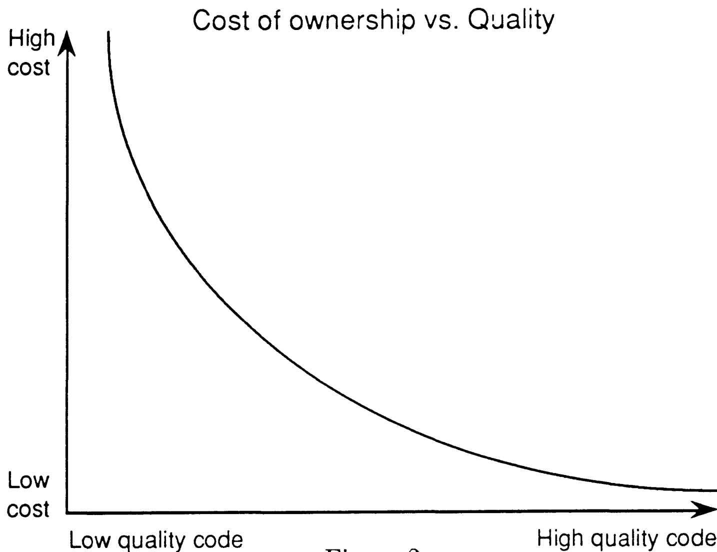

Figure 2

As in many other branches of engineering, the 80:20 rule seems to hold up well. *In general, 80% of the maintenance budget will be spent on the 20% most complex modules.*

By comparing complexity metric values against those measured for populations of source code, statements of *relative* quality can be made, such as "this module is in the worst 20% of all software in a particular demographic on several key metrics", implying an expected higher than average maintenance costs over the life-cycle. This technique was termed *Demographic Quality Analysis* by Hatton [2] who reported that this technique closely correlates with human experience. It has the additional advantage of transparency and compares favourably with telling a programmer for example, that their function has a cyclomatic complexity of 43, a statement of little value in isolation.

The enforcement of safe subsets

People would like to believe that a programming language is a well-thought out and reliable development tool. Nothing could be further from the truth. Most languages are full of compromise and contain features which are simply too dangerous to use. The situation is akin to giving a carpenter a hammer with a loose head and then wondering why people get hurt using it. A classic example is the language C, which has one of the most rapidly

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

growing base of users of any language as well as one of the loosest heads.

Even though the language is governed by a recent international standard, programming in C is accompanied by the following problems impacting reliability, portability and upwards compatibility:

- Some parts of the language are *implementation-defined* in the sense that the behaviour must be specified by the compiler writer but it can be different from machine to machine.
- Some parts of the language are *undefined* in the sense that the standard declines to specify what is supposed to happen.
- Some parts of the language have changed "quietly" compared with the parent de facto standard which most C in use still adheres to and will now behave subtly differently.
- Other parts of the language are fundamentally *unsafe* to use although defined.
- C++ the "descendant" of C, contains many features which are syntactically identical but have different semantics. Hence an identical program compiled with a C compiler can do something different if compiled with a C++ compiler.

Compiler writers are not compelled to inform users of these issues and as a result, few programmers are even aware of them and *yet they are all detectable before compilation and easily avoidable*. As a result of this ignorance, *every 9th. interface in commercially released C on average has a fault of some kind in it*, (Hatton [2]). The fault might not have reared its ugly head yet but one day it probably will and the system will misbehave. They are simply faults, or bugs waiting to happen. C is by no means alone in this but its rapid spread into even safety critical areas gives great cause for concern.

If follows from the above that a minimum requirement in the improvement of quality of source code generally is the enforced removal of any features with the above properties.

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

CONTROLLING THE PROCESS

We have seen above that many common problems with software products are detectable at a very early part of the code life-cycle. So why do they still appear with such alarming frequency in the final product ? The answer is simple and two-fold:

1. Appropriate tools were not available during development

or

2. Appropriate tools were available but their use was ad hoc because of the absence of process, or enforced methodology of use.

This is the area in which process management comes into its own. It should never be forgotten that:

- *Process control has no effect without product measurement.*
- *Product measurement is pointless in the absence of a process for making use of such measurements.*

In its simplest form, Process Control is about the management of Change, Configuration *and Quality.* Each one has a vital part to play and each one is inextricably related to each other and the principle Process Models, CMM and ISO 9001. The philosophy of software process control is very simple:

- *Define what you are going to do.*
- *Do it.*
- *Check what you did according to defined standards.*
- *Record what you did.*

The intricate relationship between Change, Configuration and Quality in a Software Process can best be depicted in layered form:

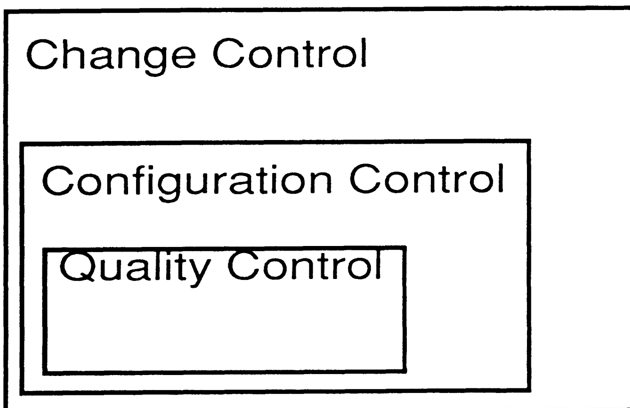

Figure 3

This diagram illustrates a number of things. The most important issue is that configuration and quality control are more primitive actions and should be largely invisible and entirely automatic. Configuration control relates directly to the product being modified and direct access to configuration without reference to the governing requirement for change is an invitation to "hack", albeit reversibly, violating an important principle of quality standards, that of *no change without requirement.* Many companies in the author's experience do not even have configuration control inviting irreversible hacking- true chaos. Quality control is in a similar position. If it isn't done automatically by the process control system, it will be applied erratically, if at all, as evidenced by the findings of Hatton [2] on the adherence to internally defined programming standards described earlier.

In contrast, the management of Change including requirements, specifications, documentation and code is the visible framework of the software process. The *Change* process is categorised by a request for some change or other maturing through a number of states starting with proposal as shown below.

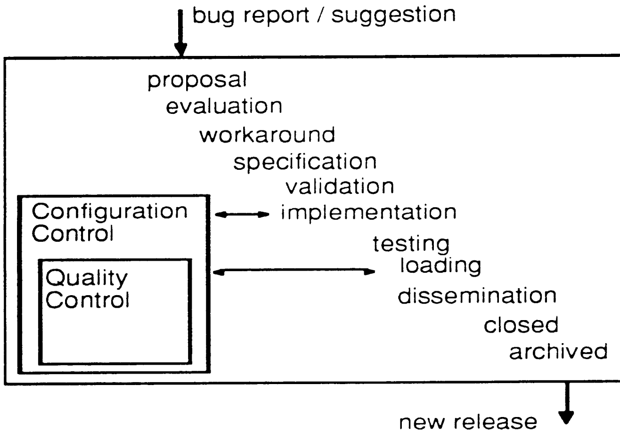

Figure 4

Unfortunately, there is no agreed nomenclature for this yet although the IEEE Software Engineering standards are particularly useful. Each change state will correspond to a *documented procedure* in the ISO 9001 sense and state change must require sign-off of responsibility for the previous state and sign-on of responsibility for the new state. During the *Change* process, the product to be changed will only be affected after the change has been validated perhaps several times, so that requirements and specifications match. Only at the implementation stage of Change

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

will the product actually be changed. At this point *Configuration* control is used to control the product versions directly. When the product is released into the loading state, *Quality* Control is applied so that *only* compliant software can proceed to the next state.

The automation of the *Quality* Control in the above process is carried out by the Process Manager which runs a series of user-defined scripts, the results of which define whether the product is compliant or not. The content of such scripts will be discussed next as part of a case history but the implied life-cycle of a source code component within the configuration control system subject to automatic quality control is shown below:

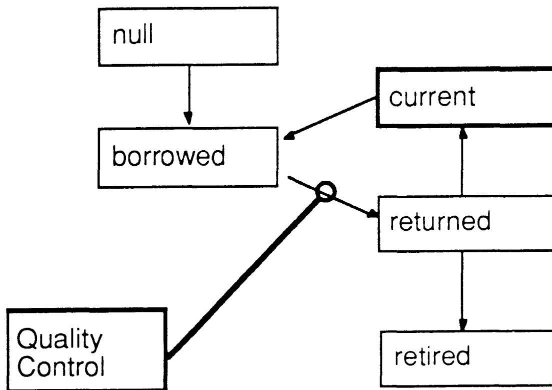

Figure 5

New modules always pre-exist in the state NULL and existing modules are in the state CURRENT.

A CASE HISTORY OF PROCESS AND PRODUCT QUALITY AUTOMATION

Programming Research Ltd. (PRL) is a relatively small organisation which develops product quality measurement tools for C, C++ and Fortran (QAC and QAC Dynamic, QAC++ and QA Fortran) and a process manager (QA Manager) which automates significant parts of levels 2 and 3 of the Carnegie-Mellon CMM. These products are used at many sites around the world in various industries including aerospace (e.g. NASA, European Space Agency, McDonnell-Douglas), telecommunications (e.g. AT & T), chemical modelling, (e.g. Shell), Earth Sciences (e.g. BP, Mobil) and Government (e.g. U.K. M.O.D.). The point here is that there

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

are many external users and effective Process Control is extremely important.

In January 1991, PRL implemented an internal process automated by QA Manager whose goal was to guarantee compliance with key parts of the CMM and to form a basis for ongoing metrics extraction. Other goals included a systematic improvement policy for existing software (written in either C or C++), a rigidly defined programming standard for new code, and a mechanism to encourage re-use.

The architecture of QA Manager is largely as indicated above in the discussion of Process Control with QAC and QAC++ used as the product quality measurement tools.

Phase 1: July 1991 - September 1991

The first stage undertaken was to define an *incremental* programming standard for *existing* code and an *absolute* programming standard for *new* code and a mechanism for systematically evolving these standards (the Software Engineering Process Group, (SEPG), a CMM concept), i.e. turning the screws periodically. This approach ensures that the overall problem is capped and then systematically reduced.

A number of items were selected split up into different levels - internal standards transgressions (level 4), complexity warnings (level 7), ANSI violations (level 8) and ANSI C constraint violations (level 9) and the C measurement tool QAC was configured to detect just those. The constraint violation category might be surprising but some so-called ANSI compilers allow certain constraint violations in C. The standards for new and existing code were enforced as follows:

New code

$$(l_4 + l_7 + l_8) * 50 \leq L \qquad \text{AND}$$

$$l_9 = 0$$

where L is the number of executable lines and l_i is the number of occurrences of items at level i. In essence, this means no more than one item of any level per 50 executable lines and NO constraint violations.

Existing code

$(l_4 + l_7 + l_8)$ must decrease AND

$l_4 \leq l'_4$; $l_7 \leq l'_7$; $l_8 \leq l'_8$ AND

$l_9 = 0$

where l'_i corresponds to the previous release and l_i to the current release. Hence no level may increase and the total number of violations must decrease. Again NO constraint violations are allowed.

The Process Manager was then configured to run QA C with these measurement criteria and reject any modules at the implementation stage of the Change process.

Phase 2: October 1991 - December 1992

Phase 1 of the improvement program was relatively unambitious to encourage cultural acceptance. The principle enemy of many attempts at standards maintenance is over-ambition. This normally results in complete rather than partial failure. This second phase was considerably more ambitious than the first because cultural acceptance had been achieved and significant progress made. The SEPG then mandated the following:

More items were added at the previously enforced levels internal standards transgressions including indentation and other stylistic warnings (level 4), complexity warnings (level 7), ANSI violations (level 8) and constraint violations (level 9) and two new levels were added, unsafe features (level 3) and obsolescent and upwards compatibility features (level 5). This latter category included items which had a different meaning in different dialects both of C and C++. The effect of adding items is most felt on new code. Existing code is not so affected as it must simply be better than the previous version when *both are measured for the same items*. The standards for new and existing code were now enforced as follows:

New code

$(l_3 + l_4 + l_5 + l_7 + l_8) * 100 \leq L$ AND

$l_9 = 0$

In essence, this means no more than one item of any level per 100 executable lines and NO constraint violations.

Existing code

$(l_3 + l_4 + l_5 + l_7 + l_8)$ must decrease AND

$l_3 \leq l'_3$; $l_4 \leq l'_4$; $l_5 \leq l'_5$; $l_7 \leq l'_7$; $l_8 \leq l'_8$ AND

$l_9 = 0$

Again no level may increase and the total number of violations must decrease. NO constraint violations are allowed.

The above required a simple change to the shell scripts driven by the Process Manager.

Phase 3: January 1993 - December 1993

At the time of writing, this has yet to come into force but the SEPG will mandate the following:

More items added at the previously enforced levels. In addition, a number of individual items from level 3 had emerged from analysis of many failures in C programs as simply too dangerous to allow at all. These became known as "killer items". The standards for new and existing code will be enforced as follows:

New code

$(l_3 + l_4 + l_5 + l_7 + l_8) * 100 \leq L$ AND

$l_9 = 0$ AND

NO killer items.

In essence, this means no more than one item of any level per 100 executable lines and NO constraint violations or killer items.

Existing code

$(l_3 + l_4 + l_5 + l_7 + l_8)$ must decrease AND

$l_3 \leq l'_3$; $l_4 \leq l'_4$; $l_5 \leq l'_5$; $l_7 \leq l'_7$; $l_8 \leq l'_8$ AND

$l_9 = 0$ AND

NO killer items.

Again no level may increase and the total number of violations must decrease. NO constraint violations or killer items are allowed.

Again a simple change to the shell scripts driven by the Process Manager is all that is required. Currently, these shell scripts total around 230 lines of Unix Bourne shell compatible commands. Since QA C is around the same speed as the compiler, the implied lag in returning a source code component successfully to the Process Manager is at most twice the compilation time, this worst case arising for a return of a modified component, necessitating a quality *comparison.*

So what has been achieved ? First of all, dramatic reductions in the numbers of these warnings have been achieved in the various packages under the control of the Process Manager, (which currently comprise some 200,000 lines of C and C++) to the benefit of all the packages concerned.

Second, a less obvious but highly desirable benefit has been that the automatic enforcement of standards concentrates the programmer's mind wonderfully as sub-standard code is simply rejected. The result has been a dramatic increase in the average programmer's fluency with C and in particular, knowledge of its strengths and weaknesses.

Third, it has frequently been observed that enhancements which cause the number of complexity warnings to *increase* (necessitating action) force the programmer to do something which normally is difficult to achieve, viz. inspecting familiar code with the same critical eye as a third party. The reason for this is that unlike say an implementation defined syntactic issue, complexity warnings arise on *structural* issues necessitating a more global inspection. This has been without exception beneficial with many occurrences of the "What on earth was I doing it that way for ..." syndrome leading to substantial reductions in complexity of previously over-complex components. The programmer cannot simply ignore it as the Process Manager will not accept it unless the problem is resolved.

As a result of the efforts of the last two years, PRL is converging rapidly to the following:

- Zero detectable static faults.

- Systematic and progressive re-engineering of existing over-complex code and control of complexity in new code.
- Verifiable adherence to a set of well-defined standards known to improve product reliability, portability and maintainability.

The first of these will achieved in Phase 2. It is expected that the second and third will be achieved by Phase 3. The recent introduction of a Dynamic Reliability checking tool, (QA C Dynamic) will allow maximum run-time error rates during regression testing to be defined for each product and made accessible to the Process Manager to enable an enforced convergence to zero. This is planned for Phase 3.

A number of other benefits have accrued from automated process control and process and product metrication. First and foremost an average re-use ratio of 40% has been achieved in the sense that for every 60 lines of a new product which must be written, 100 lines are delivered. Even very different products such as the Process Manager and the product measurement tools have re-use ratios of 28%. The figures break down as follows:

Product	Reusable lines	Total lines	Re-use Ratio
QAC	38,500	85,000	45 %
QA Fortran	32,700	71,700	46 %
QA Manager (X)	11,100	39,100	28 %
QA Manager (M)	11,100	41,700	27 %
QA C++	38,500	49,200	78 %
QA C Dynamic	10,300	28,300	36 %

One likely cause for this very high and increasing re-use ratio is that recreating the wheel is often more difficult than using an existing solution because the absolute coding standard is so high. In other words, if the programmer must recreate the wheel, it has to be a very good wheel ! This fact coupled with the requirement that all code is maintained by the Process Manager (QA Manager), giving easy browsing access to all components are compelling reasons for *not* recreating the wheel.

Finally, the rapidly growing re-use ratio strongly suggests that the SEPG should set unusually tough standards for re-usable components in the quality system in a future process review. This is now planned.

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

CONCLUSION

It has been shown that both Software Process Control and Product Quality measurement can be systematically carried out by existing tools and that such tools can be used to automate a high quality process and automatically enforce Quality Control. As a result, the following goals can be achieved:

- Zero detectable static faults.
- Systematic and progressive re-engineering of existing over-complex code and control of complexity in new code.
- Verifiable adherence to a set of well-defined standards known to improve product reliability, portability and maintainability.

To the above, a mechanism has also been described ensuring that

- Dynamic reliability failures occur at less than or equal to N per execution minute on standard tests, (where N will ultimately be zero).

can also be enforced.

Measurement of millions of lines of commercially released code around the world suggests that code developed to achieve the above goals will be of a significantly higher quality than the average with dramatically reduced maintenance costs. The lower levels of maintenance now being experienced by the case history company suggest that this is already paying handsome dividends.

ACKNOWLEDGEMENTS

I would like to thank my colleagues, in particular Sean Corfield, Andrew Ritchie and Norman Clancy, who contributed so much to the process control and product measurements described here. With their enthusiasm and capability, automated process and product control to the standards enjoyed by modern manufacturing industries has become a very successful reality for software also.

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

I would also like to thank the many companies who have contributed invaluable feedback.

REFERENCES

1. Humphrey, Watts S."Managing the Software Process", Addison-Wesley, 1990.

2. Hatton, L "Population analysis of C and Fortran quality using automated inspection: 2 - Deep flow analysis and static fault rates", submitted to IEEE Transactions on Software Engineering, 1992.

3. Akiyama, F (1971) "An example of Software System Debugging", Proc. IFIP Congress '71, Lbujlana, Yugoslavia, American Federation of Information Processing Societies, Montvale, N.J., 1971.

4. Boehm, B W "Software Engineering Economics", Englewood cliffs, New Jersey, Prentice Hall, 1981.

5. Conte, Dunsmore & Shen "Software Engineering Metrics and Models", The Benjamin/Cummings Publishing Co Inc Menlo Park, California, 1986.

6. Fenton, N E "Software Metrics: A Rigorous Approach", Chapman & Hall, 1991.

7. Hatton L, Hopkins T R "Experiences with FLINT, a software metrication tool for Fortran 77", Symposium on Software tools, Napier Polytechnic, Edinburgh, June 1989.

8. McCabe, T A "A complexity measure", IEEE Trans. Softw. Eng, SE-2,4 (Apr. 1976), 308-320.

9. Nejmeh, B A "NPATH: A measure of execution path complexity and its applications", Comm. ACM, Vol 31. no. 2, (Feb. 1988), 188-200.

10. Shooman, M L "Software Engineering", McGraw-Hill, 1983.

11. Brandl, D L "Quality measures in design", ACM Sigsoft. Software Engineering Notes, vol 15, no. 1. 1990