ELSEVIER

Information and Software Technology 38 (1996) 673–680

INFORMATION AND SOFTWARE TECHNOLOGY

Software evolution: past, present and future

Keith Bennett

Computer Science Department, University of Durham, South Road, Durham DH1 3LE, UK

Abstract

Much of the focus of software engineering has been on the initial development of software. The subsequent fate of the software has not been seen as a major issue. However, for much long-lived commercial and industrial software, the largest part of lifecycle costs is concerned with the evolution of software to meet changing needs. The cost-effective evolution of mission-critical software, for example, is still a major challenge. In this paper we review progress in software evolution in order to provide an overview for the conference. Much progress has been made in meeting industrial needs in recent years, and we can now think in terms of solutions rather than in problems alone. Some excellent case studies are available to demonstrate what is possible. The field is presented as a three-level model. At the top level, the way in which software evolution interacts with organizational needs and goals is addressed. The paper then concentrates on the middle, process layer, and explains the new proposed IEEE standard for maintenance processes. Finally, important technologies (particularly impact analysis) to underpin the process activities are described. This paper reviews the current state-of-the-art, and suggest some future trends for software evolution, along with key research issues.

Keywords: Software maintenance; Software evolution; Re-engineering

1. Introduction

Software evolution is concerned with modifying software once it is delivered to a customer. By that definition it forms a sub-area of the wider field of software engineering [1], which is defined as:

> the application of the systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is the application of engineering to software.

McDermid's [2] definition in the *Software Engineer's Reference Book* embodies the spirit of the engineering approach. He states that:

> software engineering is the science and art of specifying, designing, implementing and evolving — with economy, time limits and elegance — programs, documentation and operating procedures whereby computers can be made useful to man.

Software engineering is still a very young discipline and the term itself was only invented in 1968. Modern computing is only some 45 years old, yet within that time we have gained the ability to solve very difficult and large problems. Often these huge projects consume thousands of person-years or more of design. The rapid increase in the size of the systems which we tackle, from 100 line programs 45 years ago to multi-million line systems now, presents very many problems of dealing with scale. The underlying mathematics, computer science, and management techniques have found it difficult to keep up with six orders of magnitude change in scale. Civil engineers are well able to design and build both a small footbridge over a stream and a major motorway bridge over an estuary. We would not simply scale one design up or down to solve both problems. Additionally, our software processes are barely equipped to cope and we have great difficulty in building large software systems, predictably and repeatedly, to meet the expectations of customers.

Much progress has been made over the past decade in improving our ability to construct high quality software which meets users' needs. Baber [3] has identified three possible futures for software engineering:

(a) Failures of software systems are common, due to limited technical competence and developers. This is largely an extrapolation of the present situation.
(b) The use of computer systems is limited to those applications in which there is a minimal risk to the public. There is widespread scepticism about the safety of

PII S0950-5849(96)01116-0

software based systems. There may be legislation covering the use of software in safety critical and safety related systems.

(c) The professional competence and qualifications of software designers are developed to such a high level that even very challenging demands can be met reliably and safely. In this vision of the future, software systems would be delivered on time, fully meeting their requirements, and be applicable in safety critical systems.

In case (a), software development is seen primarily as a craft activity. Option (b) is unrealistic; software is too important to be restricted in this way. Hence there is considerable interest within the software engineering field in addressing the issues raised by (c). Software is increasingly being used in important applications, and the consequences of failure can be very high, whether expressed in terms of danger to humans or in terms of financial loss. For example, software problems with the Therac 25 [4] have been widely reported and analysed. Fly-by-wire aircraft control systems are another topic where the role and dependability of software is a major issue. The solution to these problems will rely on mature, well tried and understood processes, together with conservative designs, used on applications which are broadly similar to others that have been achieved successfully. Like other engineering disciplines, producing a new design for which there is little experience will always be more open ended, and difficult to plan in terms of timescales and budget (e.g. Concorde, the Channel Tunnel and myriad other examples).

A root problem for many software systems is complexity. Sometimes this arises because a system is migrated from hardware to software in order to gain the additional functionality that is easy to achieve in software. Complexity should be a result of implementing an inherently complex application (for example in a tax calculation package, which is deterministic but nonlinear; or automation of the UK Immigration Act, which is complex and ambiguous). The main tools to control complexity are modular design and building systems as separated layers of abstraction in order to separate concerns. Nevertheless, the combination of scale and application complexity mean that it is not feasible for one person alone to understand the complete software system.

2. Software evolution

Once software has been initially produced, it then passes into the *maintenance* phase. The IEEE definition of software maintenance is as follows [1]:

> software maintenance is the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a change in environment.

Some organizations use the term 'software maintenance' to refer to the implementation of very small changes (e.g. less than one day), and 'software development' is used to refer to all other modifications. We shall continue to use the IEEE definition. Software maintenance, although part of software engineering, is by itself of major economic importance. A number of surveys over the past 15 years have shown that for most software, software maintenance occupies anything between 40% and 90% of total lifecycle costs. A number of surveys have also tried to compute the total software maintenance costs in the UK and in the US. While these figures need to be treated with a certain amount of caution, it seems clear that a huge amount of money is being spent on software maintenance.

The inability to undertake maintenance quickly, safety and cheaply means that for many organizations, a substantial applications backlog builds up. The IT Department is unable to make changes at the rate required by marketing or business needs. End users become frustrated, and often adopt PC solutions in order to short-circuit the problems. They may then find that a process of rapid prototyping and end-user computing provides them (at least in the short term) with quicker and easier solutions than those applied by the MIS Department.

In the early decades of computing, software maintenance comprised a relatively small part of the software lifecycle; the major activity was writing new programs for new applications. In the late 1960s and 1970s, management began to realize that old software does not simply die, and at that point the software maintenance started to be recognized as a significant activity. An anecdote about the early days of electronic data processing banks illustrates this point. In the 1950s, a large US bank was about to take the major step of employing its very first full-time programmer. Management raised the issue of what would happen to this person once the programs had been written. The same bank now has several buildings full of data processing staff.

In the 1980s, it was becoming evident that old architectures were severely constraining new design. In another example from the US banking system, existing banks had difficulty modifying their software in order to introduce automatic teller machines. In contrast, new banks writing software from scratch found this relatively straightforward. It has also been reported in the UK that at least two building society mergers were unable to go ahead due to the problems of bringing together software from two different organizations.

In the 1990s, a large part of the business needs of many organizations has now been implemented, so that

business change is represented by evolutionary change to the software, and not revolutionary change, and that most so-called development is actually enhancement and evolution.

3. Types of software maintenance

Leintz and Swanson [5,6] understood a survey as a result of which maintenance was categorized into four different categories.

(a) *Perfective maintenance*: changes required as a result of user requests.
(b) *Adaptive maintenance*: changes needed as a consequence of operated system, hardware, DBMS, etc. changes.
(c) *Corrective maintenance*: the identification and removal of faults in the software.
(d) *Preventative maintenance*: changes made to software to make it more maintainable.

It seems clear from a number of surveys that the vast majority of software maintenance is concerned with evolution deriving from user requested changes.

The important requirement of software evolution for the client is that changes are accomplished quickly and cost-effectively. The reliability of the software should at worst not be degraded by the changes. Additionally, the maintainability of the system should not degrade, otherwise future changes will be progressively more expensive to carry out. This phenomenon was recognized by Lehman, and expressed in terms of his well-known laws of evolution [7,8]. The first law of continuing change states that: 'a program that is used in a real world environment necessarily must change or become progressively less useful in that environment'.

This argues that software evolution is not an undesirable attribute, and essentially it is only useful software that evolves. Lehman's second law of increasing complexity states that, as an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure. This law argues that things will become much worse unless we do something about it. The problem for most software is that nothing has been done about it so that evolution is increasingly more expensive and difficult. Ultimately evolution may become too expensive and almost infeasible: the software becomes known as a legacy system. Nevertheless, it may be of essential importance to the organization.

4. Problems of software evolution

There are many technical and managerial problems in striving to accomplish the objective of changing software quickly, reliably and cheaply. For example, user changes are often described in terms of the *behaviour* of the software system; these must be interpreted as changes to the source code. When a change is made to the code there may be substantial consequential changes, not only in the code itself, but within documentation, design, test suites, etc. (this is termed the ripple effect). Many systems under maintenance are very large, and solutions which work for laboratory scale pilots will not scale up to industrial sized software. Indeed it may be said that any program which is sufficiently small to fit into a text book or to be understood by one person does not have maintenance problems.

There is much in common between best practice in software engineering in general, and software evolution in particular. Software evolution problems essentially break into three categories:

(a) *The alignment with organizational objectives.* Initial software development is usually project based, with a defined time scale, and budget. The main emphasis is to deliver on time within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of a software system for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements. In both cases, return on investment is much less clear, so that the review at senior management level is often of a major activity which is consuming large resources to no clear quantifiable benefit for the organization.
(b) *Process issues.* At the process level, there are many activities in common with software development. For example, configuration management is a crucial activity in both. However, software evolution requires a number of additional activities not found in initial development. Initial requests for changes are usually made to a 'help desk' (often part of a larger end-user support unit), which must assess the change (as many change requests derive from misunderstanding of documentation) and if it is viable, then pass it to a technical group who can assess the cost of making the change. Impact analysis on both the software and the organization, and the associated need for system comprehension are crucial issues. Further down the evolution lifecycle, it is important to be able to regression test the software so that the new changes do not introduce errors into the parts of the software that were not altered.
(c) *Technical issues.* There are a number of technical challenges to software evolution. As noted above, the ability to construct software such that it is easy to comprehend is a major issue. A number of studies have shown that the majority of time spent in evolving software is actually consumed in this activity. Similarly, testing in a cost-effective way provides

major challenges. Despite the emergence of formal methods, most current software is tested rather than verified, and the cost of repeating a full test suite on a major piece of software can be very large in terms of money and time. It will be better to select a sub-set of tests that only stressed those parts of the system that had been changed, together with the regression tests. The technology to do this is still not available, despite much useful progress. As an example, it is useful to consider a major billing package for an industrial organization. The change of the VAT rate in such a system should be a simple matter; after all, generations of students are taught to place such constants at the head of the program so that only a one-line edit is needed. However, for a major multi-national company, dealing with VAT rates in several countries with complex and different rules for VAT calculations (i.e. complex business rules), the changes of the VAT rate may involve a huge expense.

Other problems relate to the lower status of software maintenance and evolution compared with software development. In the manufacture of a consumer durable, the majority of the cost lies in production, and it is well understood that design faults can be hugely expensive. In contrast, the construction of software is automatic, and development represents almost all the initial cost. Hence in conditions of financial stringency, it is tempting to cut costs by cutting back on design. This can have a very serious effect on the costs of subsequent maintenance.

One of the problems for management is that it is very difficult to assess a software product to determine how easy it is to change. This means that there is little incentive for initial development projects to construct software which is easy to evolve. Indeed, lucrative maintenance contracts may follow a software system where shortcuts have been taken during its development [9].

We have stressed the problems of software evolution in order to differentiate it from software engineering in general. However, much is known about best practice in software maintenance and software evolution and there are excellent case studies such as the US Space Shuttle on-board flight control software system which demonstrates that software can be evolved carefully and with improving reliability. The remainder of this paper is focused on solutions rather than problems. The great majority of software in use today is neither geriatric nor state of the art, and the paper addresses this type of software. It describes a top-down approach to successful maintenance, addressing:

(a) Software evolution and the organization.
(b) Process models.
(c) Technical issues.

In particular, we shall focus on the new proposed IEEE standard for software maintenance process which illustrates the improving maturity of the field.

5. Organizational aspects of evolution

Like any other activity, software maintenance and evolution require financial investment. We have already seen that evolution may be regarded simply as a drain on resources, distant to core activities, by senior management in a company, and it becomes a prime candidate for funding reduction and even close down. Software evolution thus needs to be expressed in terms of return on investment.

Recently Foster [10] has proposed an interesting investment cost model which regards software as a corporate asset which can justify financial support in order to sustain its value. Foster uses his model to determine the optimum release strategy for a major software system. This is thus a business model, allowing an organization the ability to calculate return on investment in software by methods comparable with investment in other kinds of asset. Foster remarks that many papers on software maintenance recognize that it is a little understood area but it consumes vast amounts of money. With such large expenditure, even small technical advances must be worth many times that cost. The software maintenance manager, however, has to justify investment into an area which does not directly generate income. Foster's approach allows a manager to derive a model for assessing the financial implications of the proposed change of activity, thereby providing the means to calculate both cost and benefit. By expressing the results in terms of return on investment, the change can be ranked against competing demands for funding.

It is of interest to note that in Japan it would appear that software may in some circumstances be written down as a capital asset on a company's balance sheet [11]. This makes investment in supporting the software more easy to justify.

Some work has been undertaken in applying predictive cost modelling to software evolution, based on the COCOMO techniques. The results of such work remain to be seen.

6. Process models

Process management [1] is defined as: 'the direction, control and co-ordination of work performed to develop a product or perform a service'.

This definition therefore encompasses software evolution, and includes quality, line management, technical and executive processes. A mature engineering discipline is characterized by mature well-understood processes so it is understandable that modelling software

maintenance and evolutionary processes, and integrating them with software development is an area of active concern [2]. A software process model may be defined [12] as:

> a purely descriptive representation of the software process, representing the attributes of a range of particular software processes and being sufficiently specific to allow reasoning about them.

The foundation of good practice is a mature process, and the Software Engineering Institute at Carnegie-Mellon University has pioneered the development of a scale by which process maturity may be measured. A questionnaire approach is used to assess the maturity of an organization and thus also provides a metric for process improvement. More recently the BOOTSTRAP project has provided an alternative maturity model from a European perspective.

In order to promote the establishment of better understood processes, the IEEE has recently published a draft standard for software maintenance [13] and the next section describes this in detail.

7. IEEE standard for software maintenance

This new proposed standard describes the process for managing and executing software maintenance activities. Almost all the standard is relevant for software evolution. The focus of the standard is in a seven-stage activity model of software maintenance, which incorporates the following *stages*:

Problem identification
Analysis
Design
Implementation
System test
Acceptance test
Delivery

Each of the seven activities has five associated attributes; these are:

Input and output lifecycle products
Activity definition
Control
Metrics

A number of these, particularly in the early stages of the maintenance process, are already addressed by existing IEEE standards.

As an example, we consider the second activity in the process model, the *analysis phase*. This phase accepts as its input a validated problem report, together with any initial resource estimates and other repository information, plus project and system documentation if available. The process is seen as having two substantial components. First of all, feasibility analysis is undertaken in which the impact of the modification is assessed, alternative solutions investigated, short- and long-term costs assessed, and the value of the benefit of making the change computed. Once a particular approach has been selected then the second stage of detailed analysis is undertaken. This determines firm requirements of the modification, identifies the software involved, and requires a test strategy and an implementation plan to be produced.

In practice, this is one of the most difficult stages of software evolution. The change may affect many aspects of the software, including not only documentation, test suites, etc., but also the environment and even the hardware. The standard insists that all affected components shall be identified and brought in to the scope of the change.

The standard also requires that at this stage a test strategy is derived comprising at least three levels of test, including unit testing, integration testing and user-orientated functional acceptance tests. It is also necessary to supply regression test requirements associated with each of these levels of test.

The standard also establishes quality control for each of the seven phases. For example, for the analysis phase the following controls are required as a minimum:

(1) Retrieval of the current version of project and systems documentation from the configuration control function of the organization.
(2) A review of the proposed changes and an engineering analysis to assess the technical and economic feasibility and correctness.
(3) Consideration of the integration of the proposed change within the existing software.
(4) Verification that all appropriate analysis and project documentation is updated and properly controlled.
(5) Verification that the testing organization is providing a strategy for testing the changes and that the change schedule can support the proposed test strategy.
(6) Review of the resource estimates and schedules and verification of their accuracy.
(7) The understanding of a technical review to select the problem reports and proposed enhancements to be implemented and released. The list of changes shall be documented.

Finally, at the end of the analysis phase a risk analysis is required to be performed. Any initial resource estimate will be revised, and a decision that includes the customer is made on whether to proceed on to the next phase.

The phase deliverables are also specified, again as a minimum as follows:

(1) Feasibility report for problem reports.
(2) Detailed analysis report.
(3) Updated requirements.

(4) Preliminary modification list.
(5) Development, integration and acceptance test strategy.
(6) Implementation plan.

The contents of the analysis report is further specified in greater detail by the proposed standard. This standard suggests the following metrics are taken during the analysis phase:

Requirement changes.
Documentation area rates.
Effort per function area.
Elapsed time.
Error rates generated, by priority and type.

The proposed standard also includes appendices which provide guidelines on maintenance practices. These are not part of the standard itself but are included as useful information. For example, in terms of our analysis stage, an appendix provides a short commentary on the provision of change on impact analysis. A further appendix addresses supporting maintenance technology, particularly re-engineering and reverse engineering. A brief description of these processes is given.

8. Assessment of the proposed standard

The standard represents a welcome step forward in establishing a process standard for software maintenance which includes software evolution. A strength of the approach is that it is based on existing IEEE standards from other areas in software engineering. It accommodates practical necessities, such as the need to undertake emergency repairs.

On the other hand, it is clearly orientated towards classic concepts of software development and maintenance. It does not cover issues such as rapid application development and end-user computing. Nor does it address executive level issues in the process model nor establish boundaries for the scope of the model.

The process model corresponds approximately to level two in the SEI five-level model. The SEI model is forming the basis of the SPICE process assessment standards initiative.

Organizations may well be interested in increasing the maturity of their software engineering processes. Neither the proposed IEEE standard nor the SEI model gives direct help in process improvement. Further details of this may be found in [14]. Additionally, there is still little evidence in practice that improving software process maturity actually benefits organizations, and the whole edifice is based on the assumption that the success of the product is determined by the process. That this is not necessarily true is demonstrated by the success of certain commodity software.

9. Technical aspects of software evolution

In our description of the IEEE standard process model the need for impact analysis was identified. This is a characteristic of software evolution that is not needed in initial software development. We shall present further details of this technique as an example of the technology needed to support software evolution.

In the above process model it was necessary to determine the cost of making a change, to meet a software change request. In this section we therefore examine how impact analysis can help this activity. To amplify the analysis needed, the user-expressed problem must first of all be translated into software terms to allow the maintenance team to decide if the problem is viable for further work or if it should be rejected. It then must be localized; this step determines the origin of the anomaly by identifying the primary components to the system which must be altered to meet the new requirement.

Next, the above step may suggest several solutions, all of which are viable. Each of these must be investigated, primarily using impact analysis. The aim is to determine all changes which are consequential to the primary change. It must be applied to all software components, not just code. At the end of impact analysis, we are in the position to make a decision on the best implementation route or to make no change. Weiss [15] has shown, for three NASA projects, the primary source of maintenance changes

Requirements phase	19%
Design phase	52%
Coding phase	7%

Weiss also found that 2% derived from environment changes and 19% were planned enhancement. He noted that 34% of changes affected only one component and 26% affected two components.

10. The problem

One of the major difficulties of software evolution which encourages maintainers to be very cautious by nature is that a change made at one place in the system may have a ripple effect elsewhere, so consequence changes must be made. In order to carry out a consistent change, all such ripple effects must be investigated, the impact of the change assessed and changes possibly made in all affected contexts. Yau [16] defines this as:

> ripple effect propagation is a phenomenon by which changes made to a software component along the software life cycle (specification, design, code or test phase) have a tendency to be felt in other components.

As a very simple example, a maintainer may wish to remove a redundant variable X. It is obviously necessary to remove all applied occurrences of X too, but for most high level languages the compiler can detect and report undeclared variables. This is, hence, a very simple example of an impact which can be determined by *static analysis*. In many cases, ripple effects cannot determine statically, and dynamic analysis must be used. For example, an assignment to an element of an array, followed by the use of a subscripted variable may or may not represent a ripple effect depending on the particular elements accessed. In real large programs containing pointers, aliases, etc. the problem is much harder. We shall define the problem of impact analysis [17] as: 'the task for assessing the effects for making the set of changes to a software system'.

The starting point for impact analysis is an explicit set of primary software objects which the maintainer intends to modify. He or she has determined the set by relating the change request to objects such as variables, assignments, goals, etc. The purpose of impact analysis is, hence, to ensure that the change has been correctly and consistently bounded. The impact analysis stage identifies a set of further objects impacted by changes in the primary sector. This process is repeated until no further candid objects can be identified.

In general, we require traceability of information between various software artefacts in order to help us assess impact in software components. Traceability is defined [1] as:

> Traceability is a degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor–successor or master–subordinate relationship to one another.

Informally, traceability provides us with semantic links which we can then use to perform impact analysis. The links may relate similar components such as design documents or they may link between different types, e.g. for a specification to code.

Some types of traceability links are very hard to determine. For example, altering in even a minor way the source code may have performance implications which cause a real-time system to fail to meet a specification. It is not surprising that the majority of work in impact analysis has been undertaken at the code level as this is the most tractable. Wilde [17] provides a good review of code level impact analysis techniques.

Many modern programming languages are based on using static analysis to detect or stop a ripple effect. The use of modules with opaque types, for example, can prevent at compile time several unpleasant types of ripple effect. Many existing software systems are unfortunately written in older languages, using programming styles (such as global aliased variables) which make the potential for ripple effects much greater and their detection much harder.

More recently, Munro and Turver [18] have described an approach which has placed impact analysis within the overall software maintenance process. The major advance is that documentation is included within the objects analysed; documentation is modelled using a ripple propagation graph and it is this representation that is used for analysis. The approach has the advantage that it may be set in the early stages of analysis to assess costs without reference to the source code.

Work has also been undertaken recently to establish traceability links between HOOD design documents [19] in order to support impact analysis of the design level.

In a major research project at Durham, formal semantic preserving transformations are being used to derive executable code from formal specifications and in reverse engineering to derive specifications from existing code. The ultimate objective is to understand maintenance at the specification level rather than at the code level, and generate executable code automatically or semi-automatically. The transformation technique supports the derivation of the formal traceability link between the two representations, and research is underway to explore this as a means of enhancing the ripple effect across wider sections of the lifecycle (see e.g. [20] for more details).

11. Research questions

Although software evolution and software maintenance tend to be regarded in academic circles as of minor importance, they are of major commercial and industrial significance.

There are many interesting research problems to be solved which can lead to important commercial benefits. There are also some grand challenges which lie at the heart of software engineering.

Firstly, how do we change software quickly, reliably, and safely? In safety critical systems, for example, enormous effort is expended in producing and validating software. If we wish to make a minor change to the software, do we have completely to repeat the validation or can we make the cost of the change proportionally in some way to its size? There are several well-publicized cases where very minor changes to important software have caused major crashes and failures in service. A connected problem lies in the measurement of how easily new software can be changed. Without this, it is difficult to purchase software in the knowledge that a reduced purchase price is not to be balanced by enormous maintenance costs later on. Almost certainly a solution to this problem will involve addressing process issues as well as attributes of the product itself.

In practice, much existing software has been evolved

in ad hoc ways, and has suffered the fate predicted by Lehman's laws. Despite its often central role in many organizations, such legacy systems provide a major headache. With management and technical solutions the problems of legacy systems also need to be addressed, otherwise we shall be unable to move forward and introduce new technology because of our commitments and dependence on the old.

It is often thought that the move to end-user computing, open systems and client service systems has removed this problem. In practice, it may well make it considerably worse. A system which comprises many components, from many different sources by horizontal and vertical integration and possibly across a widely distributed network, poses major problems when any of those components change.

12. Conclusions

We have described a three-level approach to considering software evolution, in terms of the impact on the organization, on the process, and on technology supporting that process. This has provided a framework in which to consider evolution. Much progress has been made in all three areas, and we have described briefly recent work on the establishment of a standard maintenance process model. The adoption of such models, along with formal process assessment and improvement, will do much to improve the best practice and average practice in software maintenance.

Thus, we have presented software evolution not as a problem but as a solution. However, there are still major research issues of strategic industrial importance to be solved. We have defined these as: firstly to learn how to evolve software quickly and cheaply; and secondly how to deal with large legacy systems. Where a modern technology such as object-oriented systems claim to improve the situation, this is largely a hope, and there is yet little evidence that they do indeed do so. As usual, there are no magic bullets, and the Japanese principle of *Kaizen*, the progressive and incremental improvement of practices, is likely to be more successful.

Acknowledgements

Much of the work at Durham in software maintenance has been supported by SERC and DTI funding, together with major grants from IBM and British Telecom. I am grateful to colleagues at Durham for discussions which led to ideas presented in this paper, in particular to Martin Ward. A number of key ideas have arisen from discussions with Pierrick Fillon. Thanks are due to David Hinley and to Cornelia Boldyreff for reading drafts of this paper.

References

[1] IEEE Std. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, New York, 1991.
[2] J. McDermid (ed.), Software Engineer's Reference Book, Butterworth-Heinemann, 1991.
[3] R.L. Baber, Epilogue: Future developments, in J. McDermid (ed.), Software Engineer's Reference Book, Butterworth-Heinemann, 1991.
[4] N.G. Leveson and C.S. Turner, An investigation of the Therac-25 accidents, IEEE Computer, 26 (July 1993) 18–41.
[5] B. Lientz, E.B. Swanson and G.E. Tompkins, Characteristics of applications software maintenance, Comm. ACM, 21 (1978) 466–471.
[6] Leintz, B. and E.B. Swanson, Software Maintenance Management, Addison-Wesley, 1980.
[7] M.M. Lehman, Programs, lifecycles and the laws of software evolution, Proc. IEEE, 19, 1980 pp. 1060–1076.
[8] Lehman, M.M., Program evolution, Information Processing Management, 20 (1984) 19–36.
[9] D.S. Walton, Maintainability metrics, Proc. Centre for Software Reliability Conference, Dublin, 1994 (available from the Centre for Software Reliability, City University, Northampton Square, London EC1V 0HB, UK).
[10] J. Foster, Cost factors in software maintenance, PhD Thesis, Computer Science Department, University of Durham, 1993.
[11] K.H. Bennett, Software maintenance in Japan, Report published by the UK Department of Trade and Industry, September, 1994.
[12] M. Dowson and J.C. Wilden, A brief report on the international workshop on the software process and software environment, ACM Soft. Eng. Notes, 10 (1985) 19–23.
[13] IEEE Standard for Software Maintenance (unapproved draft). IEEE ref. P1219, IEEE, New York, 1994.
[14] D.S. Hinley and K.H. Bennett, Developing a model to manage the software maintenance process, Proc. Conf. on Software Maintenance, Orlando, FL, IEEE Computer Society, 1992.
[15] D.M. Weiss, Evaluating software development by analysis of change, PhD dissertation, University of Maryland, USA.
[16] S.S. Yau and S. Liu, Some approaches to logical ripple effect analysis, Technical Report, SERC, USA, 1987.
[17] N. Wilde, Software impact analysis: Processes and issues, Technical Report, Durham University, 7/93, 1993.
[18] R.J. Turver and M. Munro, An early impact analysis technique for software maintenance, J. of Software Maintenance: Research and Practice, 6 (1994) 35–52.
[19] P. Fillon, An approach to impact analysis in software maintenance, MSc Thesis (to be submitted), University of Durham, 1994.
[20] M.P. Ward, Abstracting a specification from code, J. of Software Maintenance: Practice and Experience, 5 (1993) 101–122.