

Software Quality:Definitions and Strategic Issues

Ronan Fitzpatrick

MSc Computing Science (ITSM)
Advanced Research Module

Staffordshire University, School of Computing Report, April 1996

Software Quality:

Definitions and Strategic Issues

Copyright

©

1996

Ronan Fitzpatrick

All rights reserved. No part of this publication may reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission, in writing, from the copyright owner.

Abstract

This paper contains two sections relating to software quality issues. First, the various definitions of software quality are examined and an alternative suggested. It continues with a review of the quality model as defined by McCall, Richards and Walters in 1977 and mentions the later model of Boëhm published in 1978. Each of McCall's quality factors is reviewed and the extent to which they still apply in the late 1990s is commented on. The factors include, integrity, reliability, usability, accuracy, efficiency, maintainability, testability, flexibility, interface facility (interoperability), re-usability and transferability (portability). They are subdivided into external and internal quality factors. Interrelationships between the different factors are shown in Perry's model. Issues of quality management and the prioritising of these factors are included. The second section examines the strategic impact of quality from both the supplier's and the purchaser's point of view. In particular product differentiation, tendering and estimating, system acquisition and employee productivity are considered. Product differentiation is mapped to Porter's generic business strategy. The COCOMO model for software costing and estimating is used to show that quality factors influence the cost of a product. As quality impacts on all classes of people in systems, human resources and the consequences for productivity are explored. Finally, system evaluation and selection techniques involve quantitative (weighting and rating) techniques and Robson's example and the influence of quality are examined.

Contents

1 Introduction	5
2 Building quality into software products	5
2.1 Quality defined	5
2.2 Quality models	7
2.3 Quality factors	9
2.4 Other quality issues	19
2.5 Interrelationships between quality factors	20
2.6 Measuring quality.....	21
2.7 Grouping and prioritising quality factors.....	22
2.8 Quality - critique	22
2.9 Summary	22
3 The Strategic importance of quality	23
3.1 Quality and the system developer	23
3.1.1 Marketing strategy	23
3.1.2 Tendering and cost estimation.....	26
3.1.3 Human Resources and quality	29
3.1.4 Productivity.....	29
3.2 Quality and the systems purchaser.....	30
3.2.1 System acquisition.....	30
3.2.2 User productivity	30
3.3 Summary	32
4 Conclusion	32
5 References	33
6 Cyberography	34

1 Introduction

Two aspects of quality that information systems (IS) professionals need to concern themselves with are reviewed in this paper. The first focuses on what exactly must be done to a software product or what components it should contain so as to distinguish it as a quality product and elevate it from being simply a good product. The second concern addresses how quality impacts on an organisations strategic philosophy. The paper is therefore set out to reflect these two topics. In section 2 software quality is first defined. Models of quality factors are introduced and one of them is selected as a framework for explaining those factors. Finally, in section 3, the strategic contribution of quality to business issues like product differentiation, tendering and estimating, system acquisition and employee productivity is considered.

2 Building quality into software products

Many IS professionals when asked what they understand by software quality, immediately start to talk about testing. When they realise that they are limiting their understanding they include validation and verification into the formula and begin to talk about walkthroughs and reviews which are just an extension of testing. So testing is their major understanding of quality. There is a simple explanation why this is so. There is an abundance of text books with titles relation to software quality. However, closer examination reveals that the vast majority of them are concerned with quality assurance and for most authors that means testing developed code with some references to validation and verification. Perhaps this preoccupation on quality assurance stems from the international standard (ISO 9000-3 1991) where the expression quality assurance forms part of its heading and then contains a substantial emphases on guidelines for testing, validating and verifying developed product. Quality and what it means is not defined and even someone familiar with its meaning would find it difficult to locate references to it in the standard and guidelines.

2.1 Quality defined

Dictionary definitions of quality are generally focused on excellence and that should be the IS professional's focus too. Some technical authors like to describe software quality in terms of "Fitness for purpose" but more recent commercial thinking would not fully support this description.

Quality is defined by International organisations as follows:

"Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfying of given requirements".

German Industry Standard DIN 55350 Part 11

"Quality is the totality of features and characteristics of a product or a service that bears on its ability to satisfy the given needs".

ANSI Standard (ANSI/ASQC A3/1978)

"a The totality of features and characteristics of a software product that bear on its ability to satisfy given needs: for example, conform to specifications.

b The degree to which software possesses a desired combination of attributes.

c The degree to which a customer or user perceives that software meets his or her composite expectations.

d The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer".

IEEE Standard (IEEE Std 729-1983)

These standards are a long time in existence and their relevance to the late '90s might be a little too broad. The IEEE standard specifically relates to software, so, it's a good candidate for closer analysis. This standard defines quality in terms of features and characteristics. But what are these items and who defines them? It's concerned with the presence of desired combinations of [presumably, quality] attributes. So a product with say two of the desired combinations is the same quality as a product with four of the combinations? It expresses quality in terms of customer expectation. If a customer's expectation is nil doesn't that mean that a product with nil characteristics is a quality product? Finally it is based on user perception. So, if an uninformed user customer perceives a motor car that rusts, falls apart, breaks down, regularly fails to start, burns excessive oil etc, as a quality car - is it? Obviously not. So this type of definition doesn't help. Ince (1994) describes the modern view of quality:

"A high quality product is one which has associated with it a number of quality factors. These could be described in the requirements specification; they could be cultured, in that they are normally associated with the artefact through familiarity of use and through the shared experience of users; or they could be quality factors which the developer regards as important but are not considered by the customer and hence not included in the requirements specification".

For the purpose of this paper the following definition is used:

Software quality is the extent to which an industry-defined set of desirable features are incorporated into a product so as to enhance its lifetime performance.

This definition focuses on the existence of a product in the first place and that its quality has a time dimension. It also focuses on features which will enhance the product. Finally it requires the features to be incorporated from the beginning - by way of user requirements or similar specification - and not bolted on as an afterthought.

2.2 Quality models

According to Wallmüller (1994) "one of the oldest and most frequently applied [software quality] models is that of McCall *et al.* (1979). Other models such as that of Murine & Carpenter (1984) or that of NEC (Azuma 1987) are derived from it. McCall's model is used in the United States for very large projects in the military, space and public domain. It was developed in 1976-7 by the US Airforce Electronic System Division (ESD), the Rome Air Development Centre (RADC) and General Electric (GE) with the aim of improving the quality of software products. One explicit aim was to make quality measurable.

McCall started with a volume of 55 quality characteristics which have an important influence on quality, and called them "factors". For reasons of simplicity, McCall then reduced the number of characteristics to the following eleven:

- efficiency
- integrity
- reliability
- usability
- accuracy
- maintainability
- testability
- flexibility
- interface facility
- re-usability
- transferability".

A second set of quality factors was defined by Boëhm (1978). Authors, too, have added to McCall's original list to reflect recent thinking. Ghezzi *et al.* (1991) list sixteen factors. A full list of both models is set out in table 2.1

Much has happened on the technology front since 1977 and many authors have redefined some of the eleven factors while others have added even more to better reflect recent advances in the technology. One commercial model taking this approach is the SPARDAT quality model from Germany which classifies three significant characteristics - applicability, maintainability and

adaptability. These characteristics are further sub-divided giving twenty quality factors in all. This model was created for software development in the banking environment.

Because of the format used to describe them, quality factors are often referred to as the ...ility factors.

McCall <i>et al.</i> (1976-77)	Boehm (1978)
efficiency	usability
integrity	clarity
reliability	efficiency
usability	reliability
accuracy	modifiability
maintainability	re-usability
testability	modularity
flexibility	documentation
interface facility	resilience
re-usability	correctness
transferability	maintainability
	portability
	interoperability
	understandability
	integrity
	validity
	flexibility
	generality
	economy

Table 2.1 - Software quality models

An NCC publication (no reference cited) sub-divides the quality factors into those that relate to external quality and those that relate to internal quality. It explains that "external quality is the quality of the finished product, the quality as it appears to the external world, as it comes of the end of the assembly line. Internal quality is the quality of the product as it is being constructed, while it is on the assembly line". We can extend this view by considering external quality as those factors that are clearly visible to end users while internal quality would be concerned with internal technical issues of the software. This is in keeping with the view of Ghezzi *et al.* (1991) who state that "In general, users of the software only care about the external qualities, but it is the internal qualities - which deal largely with the structure of the software - that help developers achieve external qualities". In the next section, as each quality factor is explained you will see an interplay between factors. Table 2.2 below illustrates how the external and internal sub-division might be made.

External quality	integrity reliability usability accuracy
Internal quality	efficiency maintainability testability flexibility interface facility re-usability transferability

Table 2.2 - Categorical quality factors

2.3 Quality factors

In this section the model presented by McCall *et al.* is used to explain quality factors which are commented upon in the light of developments in the late 1990s. You will see how some of the quality factors are still as fresh and as relevant today as they were in 1977. You will also see how some are redundant and have been integrated into other factors to better reflect modern practice. Under the heading of usability a whole new area of quality is taking shape and some of the topics in that domain are highlighted. Finally, you will see that each quality factor is not an isolated condition to be searched for and evaluated on its own and that in many cases there is an interplay and dependency between factors.

2.3.1 Efficiency

The volume of code or computer resources (eg. time or external storage) needed for a program so that it can fulfil its function.

McCall et al.

McCall's view of efficiency or performance is concerned with the efficient use of computer code to perform processes and the efficient use of storage resources. There are a number of techniques that can be used to achieve both of these objectives. Typical of these are:

Programming languages. Selecting the most appropriate programming language for the problem has a major impact on program efficiency. For example, business applications which require substantial volumes of reports might best be programmed in COBOL while programs requiring substantial scientific calculations might be best accommodated by FORTRAN.

Operating systems. Modern operating systems have the ability to perform multi-tasking thereby improving system performance by facilitating background operations.

Design. Strategies that address cohesion and coupling, normalisation techniques to reduce data redundancy and algorithms that optimise process time should always be employed.

Access strategies. Algorithms that optimise seek time, rotational delay and data transfer time must be continuously searched out and implemented to improve efficiency. In particular cylinder concepts and hashing algorithms are most important.

Programming techniques. Typical good programming techniques and practice like:

- Top-down design for complex problems
- Sequence, selection and iteration constructs
- Keeping local variables within procedures
- Good use of parameter passing
- Meaningful variable and procedure names
- Proper documentation

The definition used by McCall inadvertently limits resource efficiency to external storage. Developers need to be particularly careful not to ignore, to the detriment of efficiency, modern working storage facilities like CPU RAM, Video RAM or printer RAM and external storage like CD-ROM.

Modern hardware has advanced considerably since 1977 so that efficiency now applies to a much broader set of resources, both internal and external. For example, the processing power of desktop computers plays a major role in software performance. Most readers would be aware of the performance of WINDOWS 3.1 on a '386 processor as compared with its performance on a '586 processor. While software was written for a perceived stable hardware environment in the late 1970s this is not so to-day. In the definition of quality given in section 2.1, a time dimension was included by way of the product's lifetime performance. Specifiers need to be conscious that in the current climate, efficiency is influenced by technological maturity too.

Communications technologies are also part of modern efficiency. Fibre optic cables obviously support high speed data transfer rates, free of interference.

Overall it would appear that the currency of the 1977 definition of efficiency has changed substantially with technological advancements. There are many additional considerations involved.

2.3.2 Integrity

The extent to which illegal access to the programs and data of a product can be controlled.

McCall et al.

This definition of integrity confines itself to user confidence that programs and data can not be altered illegally and the access controls that can be put in place to guard against this happening.

Unfortunately programs and data can also be altered innocently by authorised users making mistakes. There is nothing illegal about this access and integrity controls must protect against that too.

So, integrity has to concern itself with controls for preventing inaccurate data entering the system and detecting it if it does. It also has to concern itself with preventing changes to the software which compromise its original purpose.

French (1986) states that the aims of these controls are:

- a. to ensure that all data are processed
- b. to preserve the integrity of maintained data
- c. to detect, correct and re-process all errors
- d. to prevent and detect fraud".

He also suggests that there are five different types of control. Manual checks which are applied to documents before computer processing, data preparation controls, validation checks, batch controls and file controls. Only the last three of these can be built into the software to ensure integrity.

Validation checks ensure that there are no empty fields where values are essential, datatypes are in keeping with those defined in the code and that data is within a reasonable range. Computer generated batch totals can be compared with manual batch totals to confirm valid data entry. File control totals are included at the end of each file and contain details like number of accounts and net totals.

In a database environment integrity takes on a special significance because of the many users accessing that data. Oxborrow (1986) explains that "the integrity of a database is maintained by means of validation rules which are applied to inputted and updated data, and concurrent access locks and rules which prevent concurrent transactions from interfering with each other".

The validation rules suggested are the same as those in the traditional file processing environment. Mechanisms are coded into that software to check for data input errors where datatypes are incorrect or a maximum/minimum range is imposed on acceptable values. Reasonableness checks for gross errors are also included. Concurrency access control is necessary to prevent two or more users accessing the same data and processing it in different ways. The lost update and the deadlock problems are typical of what must be protected against. Read-locking and write-locking solutions need to be put in place. Minimum data redundancy in the database environment supports the concept of data integrity. Entity attributes that are stored more than once are likely to become inconsistent thereby compromising the integrity of the data. Database management systems incorporate many tools which support the easy inclusion of integrity features in database systems.

In safety-critical systems integrity implies that a safety specification must be written into the requirements specification. Validation checks at the data input stage are critical. There are many newspaper accounts of hospital cases where patients have received incorrect treatment as a result of invalid input.

2.3.3 Reliability

The extent to which a program can be maintained so that it can fulfil its specific function.

McCall et al.

This definition seems to be more appropriate to a different quality factor called maintainability, which will be examined later. Reliability in engineering terms is the ability of a product or component to continue to perform its intended role over a period of time to pre-defined conditions. And the same applies to the systems environment where reliability is measured in terms of the mean time between failures, the mean time to repair, the mean time to recover, the probability of failure and the general availability of the system.

The mean time between failures - under pre-defined conditions, the average time between consecutive failures over a given period in the life of a system.

The mean time to repair - the average time to repair or maintain equipment.

The mean time to recover - the average time to return a system to operation after a failure. The time involved should include periods taken to re-instate from previous checkpoints.

The probability of failure - the use of formal methods to predict the likelihood that a system will behave in an expected way under certain circumstances. Probability of failure is most appropriate to safety-critical systems and "continuous running" systems.

Authors don't appear to distinguish between hardware failure and software failure. Both need to be considered together when establishing system reliability.

Even taking all these criteria into consideration systems reliability philosophy is very different to other branches of engineering and manufacturing. For example, Ghezzi *et al.* explain that "you do not take delivery of an automobile along with a list of shortcomings". Software on the other hand they point out comes with a disclaimer that the software manufacturer is not responsible for any damages due to software failure.

Ince (1994) tells us that reliability is normally expected to be totally present in safety critical systems. But can it? Is the software engineering profession sufficiently mature to be able to predict and properly cope with instances like the Three Mile Island accident and the Chernoble accident? See Bott *et al.* for an account of these disasters. Sommerville (1992) offers excellent advice on programming (fault avoidance, fault tolerance, exception handling and defensive programming) for reliability where the software is intended for use in high reliability situations.

The availability aspect of reliability is naturally a major concern in on-line and data-base systems. Many of these application are essential to doing business - factory applications on the McFarlan and McKinney grid - so excessive downtime must be avoided. Furthermore, after a system has been down, users need to be certain that the restored system is accurate with no loss of data. Consequently, reliability has connections and implications for the accuracy and testability quality factors which are examples of the interplay between factors.

2.3.4 Usability

The cost/effort to learn and handle a product.

McCall et al.

Usability is by its very name a quality factor in the HCI domain - usable by humans. To gain a proper understanding of McCall's perspective of usability in 1977, it is appropriate to recall the taxonomy of computers in those days and the purposes for which they were used. They were mainframe and mini computers running major DP applications. User staff were simply required to learn how to operate the system, input data, receive output and generally keep the system running. Software was developed to run on low specification monochrome monitors with simple combinations of green and black text. Software design strategies were generally file-based. So it is easy to identify with McCall's definition, as usability was mainly confined to selected users learning how to use it. The era of desktop computing was only beginning.

Usability in the late 1990s is concerned with a vast number of end-user issues and learning to use systems now embraces so many topics that some authors consider learnability to be a quality factor in its own right. Usability topics include suitability, learnability and adaptability.

Usability is an extremely difficult quality factor to define. The main reason for this is the vast variety of users with their different needs. Some will be novices while others will be experts and a quality product will support them all.

Curson (1996) advises that "usability will be defined in the forthcoming ISO 9241 standard as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". This definition suggests to me that the debate will still continue as to what exactly usability is because the definition is far too vague and omits the all important concept of user confidence. But more importantly, a definition that explains itself by re-using the term being defined is of now value at all. In this case usability is defined as the extent to which a product can be used. And what do we mean by used? Isn't that what we're trying to define? Finally the definition appears to ignore the need for a quality product to support all users no matter what their skill level.

In this paper, under the general heading of usability, topics are included that have been internationally discussed and researched. They are presented here under two headings: *general ergonomics* and *software ergonomics*. These heading are derived from the requirements of the

EU directive on the minimum safety and health requirements for work with display screen equipment (Council Directive 1990).

General ergonomics is concerned with equipment and the work environment. While not essentially software issues they must be considered since many systems have failed because the factors involved were not properly addressed. Equipment is concerned with the selection of display screens, keyboards, work surfaces and work chair. The environment like space requirements, lighting and distracting reflections should all be such that the user is as comfortable as possible. Noise, heat radiation and humidity are also considered as part of the desired environment.

Software ergonomics is concerned with topics like how suitable is the software for the intended operations. How easy is it for users to learn and to master it. It embraces dialogue styles which vary from command line to Graphical User Interfaces. Human factors like perception, memory and human senses are included together with metaphors that aid these factors. Shneiderman's (1987) golden rules for good screen design are essential to software ergonomics. These include consistency of display, feedback and error messages, system performance, ability to reverse the users last action and similar facilities.

The whole subject of usability is very much a leading edge quality factor. Academic and industry ideas, independent standards and legal requirements are all concerned with the issue but unfortunately there seems to be no uniformity of definitions or vocabulary. Perhaps in the future this will be resolved. Sub-dividing it into new and more focused quality factors would seem appropriate. Meanwhile IS professional with an awareness of the topics mentioned here will be better equipped to specify, design, commission and evaluate software usability.

2.3.5 Accuracy

The extent to which a program fulfils its specification.

McCall et al.

Accuracy is a difficult factor to pin down because of the lack of standard terminology. It is easy to use the term interchangeably with other factors like reliability and integrity. Ince (1994) calls the factor correctness. Ghezzi *et al.* also prefer the term correctness and their definition is "A program is functionally correct if it behaves according to the specifications of the functions it should provide". That this is a often "wish-list" quality factor because for the most part program specifications are seldom available except for very high budget and safety-critical systems. In these cases it is economic to employ formal methods to confirm program accuracy. Otherwise, in as much as the various paths through a program can be measured, accuracy is confirmed through testing. Techniques include, program inspections, mathematically-based verification and static program analysers. accuracy or correctness can be achieved by using proven designs, algorithms and re-usable code.

2.3.6 Maintainability

The cost of localising and correcting errors.

McCall et al.

Maintenance and what it means has changed considerably since 1977. Finding and correcting errors is just one aspect of maintenance. Ghezzi *et al.* (1991) divide maintenance into three categories: corrective, adaptive and perfective and only corrective is concerned with correcting errors as suggested by McCall.

Corrective maintenance is concerned with removing minor bugs left after development and testing are completed. This process is also involved after other maintenance activities.

Adaptive maintenance is concerned with changing the software to reflect changes in the user's requirements. For example changes in VAT rates, income tax bands or income tax rates. Or, a user might wish to add more functionality.

Perfective maintenance seeks to improve the algorithms used in the software to enhance performance. Perfective maintenance is often influenced by technological developments.

Maintenance starts from the moment a system comes into operation and continues for the remainder of the product's life. For some products this can be twenty years or more. Maintainability is supported by good practices at all phases during the development life cycle. Typical of these practices includes:

A well defined methodology - A methodology like SSADM will guarantee that all aspects of fact finding and system modelling using DFDs and ERDs are completed and documented using standard techniques and standard forms. Some organisation might combine this approach with an end-to-end methodology which ensures that one consistent life-cycle approach applies.

Good design techniques - Cohesion and coupling techniques ensure the modular construction of systems so that all three aspects of maintenance are easier to achieve. Well structured data using normalisation techniques will also mean a minimum of data updating thus maintaining the integrity of the maintained system.

Attention to documentation - one of the most important life-cycle activities in the maintenance managers world. McCall's definition above is concerned with cost and a properly documented system will substantially reduce the cost of the corrective, adaptive and perfective functions. Offending code and its module can be easily located. Modules that are affected by change can also be easily identified and corrected. Without documentation the cost begins to rise and the maintainability of the product begins to fall.

Good programming practice - meaningful names, readable code, sequence, selection and iteration structures only, single entrance and exit points to procedures and similar policies. Maintenance is also well supported by either re-using tried and tested code or by writing new code with re-use in mind.

Maintainability is another of the internal factors not visible or obvious to the client. It is therefore incumbent on the IS professionals involved to ensure that maintainability is properly addressed as part of the requirements specification.

Finally, for the purpose of updating McCall's definition above we might define maintenance as

"The non-operational costs associated with a product after a successful user acceptance test".

2.3.7 Testability

The cost of program testing for the purpose of safeguarding that the specific requirements are met.

McCall et al.

Because of its traditional position in development models like the Waterfall or Boehm's spiral, testing is easily identified as a quality factor. The testing process is well matured at this stage. Sommerville (1992) suggest five stages - unit testing, module testing, subsystem testing, system testing and acceptance testing. However, most authors confine it to four stages and the model suggested in ISO 9000-3 names them as - item testing, integration testing, system testing and acceptance testing.

Item testing - standalone components are individually tested to ensure that they function properly. A substantial amount of item or unit testing is completed by programmers as part of their normal role.

Integration testing - brings together standalone components into modules which are tested to reflect how they link in a new environment. Integration testing is also referred to as module testing.

System testing - best performed as a full test run of the system that the client is about to receive but done without the client being present. It is the supplier's opportunity to confirm that the requirements specification has been fully achieved.

Acceptance testing - the client running the new system to ensure that it complies with the original specifications. Acceptance testing is often referred to as Alpha testing.

Testing interacts with all other quality factors. For example, to check accuracy a test plan is needed. To test reliability a test plan is needed. To test efficiency a test plan is needed and so on. So all testing must be performed in accordance with pre-defined plans, using pre-defined tests data to achieve pre-determined results. Numerous test strategies are used. They include functional or black box testing, structural or white box testing and finally residual defect estimation. These strategies can be employed using difference techniques. The options are top-down testing, bottom-up testing and stress testing.

Top-down testing - testing code that calls other, as yet unwritten, code by writing test stubs (empty routines) which can be called by the code being tested. A typical example would be testing a menu structure. Dummy stubs which represent the branched-to modules are used to test that the menu branching structure is working correctly. Later when the stubs are replaced by working modules, if things don't work properly, then the fault can be isolated to the new module and not the menu system.

Bottom-up testing - this is the compliment of top-down. In this case the modules or units are coded and unit tested, often by different developers at different sites at different times. All are eventually combined and the combining or calling structure is tested for defects.

Stress testing - involves testing systems which are designed to handle a specific load. For example, banking systems might be required to process a specified number of transactions per hour. Stress testing investigates whether the specified number of transactions can be achieved and what happens if they are exceeded. Does the system crash? Is data lost? And similar concerns.

All of this testing has cost associated with it and testability is concerned with keeping this cost to a minimum. This can be achieved by using automated testing tools, through good cohesion and coupling design strategies and through good programming techniques. McCall *et al.* (1979) suggest a complexity matrix which involves number and size of modules, size of procedures, depth of nesting, number of errors per unit time and the number of alterations per unit time.

McCall's definition of testability has stood the test of time and is still very much applicable today.

2.3.8 Flexibility

The cost of product modification.

McCall et al.

This is McCall's second element of maintenance and over the years as the maintenance function has taken on new meaning the flexibility quality factor has been fused into maintenance.

Recent interpretation of flexibility would be more associated with adaptability, ie. being able to change or reconfigure the user interface to suit users' preferences. This is a usability quality issue and is better considered in the usability section.

2.3.9 Interface facility

The cost of connecting two products with one another.

McCall et al.

Modern authors in keeping with the use of ...ility format have renamed this factor interoperability. This is the development strategy that encourages product development in a manner that it can interact with other products. For example, word processors that can incorporate charts from spreadsheets, or graphics from CAD and graphics packages, or data from databases would be considered to have high interoperability. Another example might be a payroll system - where staff are paid a commission on sales - being able to interface with a sales or order entry system. Interoperability is also the quality feature required to enable software products to interface with each other over a communications network.

Interoperability is very much dependent on an open systems policy being supported by software developers. An example of this in operation can be obtained from the Lotus Corporation (1993) white paper on Communications Architectures. In this document Lotus advise customers that they in conjunction with Apple, Borland, IBM, Novel and Work Perfect are committed to a Vendor Independent Messaging (VIM) policy. They explain that they comply with industry and de facto standards such as x.400, SQL, OLE etc, and that their products like cc:Mail and Lotus Notes have open and published Application Programming Interfaces (APIs). These interfaces enable independent software vendors to develop applications which can interface with Lotus and their partners' products.

2.3.10 Re-usability

The cost of transferring a module or program to another application.

McCall et al.

Re-usability addresses the concept of writing code so that it can be used more than once. A typical example is writing procedures or routines to receive variable parameters. The calling code passes the parameters and the called procedure processes them as appropriate. Any result from the process is returned to the calling code. The main advantage of using this approach is that once a procedure has been written and fully tested it can be used with full confidence in its accuracy thereafter. It should never need to be tested again.

Commercial libraries of re-usable code are becoming available. The C language comes with a whole series of .lib files for use with the language thus saving developers the cost of writing their own solutions. Visual Basic is another instance of commercially available windows solutions. Object oriented development techniques also support re-usability.

While re-usability is concerned with the cost of adapting code so that it can be re-used in another application there is in the first place a cost of developing re-usable code. It takes much longer and therefore costs more to develop re-usable code. Quite often budgets and timescales work against the desire to develop re-usable code.

When writing code for re-use programmers should also be conscious of the economic use of code to comply with the efficiency quality factor.

This is another of the internal quality factors where client interests, particularly in relation to copyright and ownership, need to be carefully considered. Does specially written re-usable code belong to the supplier or the customer and does code re-used from a commercial library attract licence fees? All are items for the project specification and contract documents.

2.3.11 Transferability

The cost of transferring a product from its hardware or operational environment to another.

McCall et al.

The modern expression for this quality factor is portability. Portability is the strategy of writing software to run on one operating system or hardware configuration while being conscious of how it might be refined with minimum effort to run on other operating systems and hardware platforms as well. Sommerville (1992) considers portability as a special case of software re-use "where the whole application system is re-used by implementing it across a range of different computers and operating systems.

Portability is well supported by recent advances in standards development. Operating systems like UNIX and MS-DOS are well established and are being adhered to by developers. Programming languages also have agreed standards and languages like COBOL, C, FORTRAN and PASCAL are easily ported between systems using compilers that implement the agreed standards. Windows environments like X-Windows and WINDOWS have also imposed de facto standards on developers. Consequently products developed to comply with these standards should be portable to other environments with a minimum of conversion effort.

2.4 Other quality issues

That concludes this review of McCall's quality factors. The eleven factors included here are typical of those considered by most authors. Some authors include their own individual preferences in their texts. For example, Ghezzi *et al.* include factors like robustness, reparability, evolvability, productivity and visibility. The draft document for the new ISO 9000-3 (ISO/CD 9000-3.2 1996) lists six quality features one of which is functionality. Unfortunately, there is no international or industry set of properly defined software quality factors to help to overcome this situation.

2.5 Interrelationships between quality factors

As different factors were examined in the previous section the interplay between some of them was obvious. Perry (1987) has shown that there are interrelationships between quality factors and has labelled them *inverse, neutral and direct*. Perry's model of interrelationships is shown in table 2.3

Correctness	C									
Reliability	-	R								
Efficiency			E							
Integrity			•	I						
Usability	-	-	•	-	U					
Maintainability	-	-	•		-	M				
Testability	-	-	•		-	-	T			
Flexibility	-	-	•		-	-	-	F		
Portability			•			-	-		P	
Re-usability		•	•	•		-	-	-	-	R
Interoperability			•	•					-	I

• = Inverse blank = Neutral _ = Direct

Table 2.3 - Model of interrelationships after Perry

2.6 Measuring quality

Gillies (1992) details five approaches to measuring quality from the purchaser's point of view. These are:

- Simple scoring
- Weighted scoring (or phased weighted factor method)
- The Kepner-Tregor method
- The Cologne combination method
- Polarity profiling

Because of deficiencies with these methods and because of their older date Gillies suggests a new technique called LOQUM (LOcally defined QUality Modelling) which involves a three step approach:

"LOCRT: a knowledge elicitation exercise to derive the relevant quality criteria and associated measures.

LOCREL: a further knowledge elicitation to define relationships and conflicts between criteria.

LOCPRO: A profiling tool to display a graphical profile to represent the overall quality of the system".

A more scientific approach was proposed by both McCall and by Boëhm. Typical examples are

- a. Mean Time Between Failure (MTBF)

$$MTBF = \frac{T\text{-tot}}{N}$$

T-tot = the total time period

N = the number of failures in T-tot

- b. Complexity is measured by McCabe (1976) "as a cyclomatic number, based upon graph theory, that seeks to estimate the number of linearly independent paths through a program".

This level of detail is beyond the scope of this paper. Gillies (1992, p40-43) cites numerous references to quality measurement.

2.7 Grouping and prioritising quality factors

Perhaps because of industry concentration on testing or validation and verification little work seems to have been done about grouping or prioritising quality factors. Some authors (Daily 1992 and Ghezzi *et al.* 1991) suggest that there is an order of sequence and priority that must be applied to quality factors. They imply that usability is the first in this order. If a software product cannot be loaded, it cannot be launched and therefore cannot be used. Consequently other quality issues do not come into play. Daily suggests that "once the software is usable, correct and reliable then efficiency, compatibility (interoperability) and integrity can be considered in more detail". However it would be most important that this priority would be confined to an evaluation strategy only. A strategy for building these quality factors into a product must treat all factors equally and do so at the requirements specification stage.

2.8 Quality - critique

One of the problems with the software quality models is that they were published in 1977/78 and in the intervening years enormous technological advancements have taken place. In this paper some of them have been highlighted. Perhaps more of them need to be reviewed so as to better reflect those changes. Perhaps some new additions are also needed. It is also extraordinary that so few of the quality factors get any mention in international and national standards relating to quality assurance.

2.9 Summary

In this section the various definitions of quality have been reviewed and commented upon. The fact that these definitions are of older date and in many ways inappropriate to reflect modern technological developments encouraged a definition of quality for use in this paper. The McCall *et al.* model for quality factors was used for a detailed explanation of each quality factor. Quality was sub-divided into external and internal factors and the interplay between them illustrated as factors were explained. Further sub-division might be appropriate to reflect current interest in Human-computer interaction. Some individual quality factors were seen to be outdated by technological developments and it would be appropriate to redefine them and add some new ones to take into account modern standards and legal obligations. Daily's and Ghezzi's concept of priority among quality factors were mentioned in relation to requirements specifications.

3 The Strategic importance of quality

An organisation's focus on the strategic importance of software quality depends on whether they are producers or users of software. Software developers see themselves as either the producers of a product that will be sold in the marketplace or contractors developing a product for a specific client. Software users see software as a tool to be used to support them in the way they do business in their specific sector. Both business views are very different but in both cases quality is an underlying consideration to helping them achieve their business objectives. In this section the impact of quality on the strategic business issues of both are considered. The issues addressed are marketing strategies, tendering and estimating, human resources issues, productivity and system acquisition.

3.1 Quality and the system developer

Software developers need to consider quality from two points-of-view. The first addresses issues that relate to developing generic product for off-the-shelf sale in the High Street. The second is their philosophy relating to the management of quality and quality assurance for bespoke product development.

3.1.1 Marketing strategy

Developers producing generic products will need to map their strategic philosophy to a model like Porter's (1980) generic business strategy (Figure 3.1).

<p>Product Differentiation. By following this strategy the organisation hopes to win customers by offering "better" products or services than its competitors. The organisation adopting this strategy must focus on building unique products and services and publishing their existence</p>	<p>Overall Cost Leadership. By following this strategy the organisation seeks to win customers upon the basis of cost, for a given level of quality and service. The organisation must focus on "good" cost control, seeking cost reductions wherever possible. This cost leadership can be achieved throughout the value chain, from low unit cost raw materials to low unit cost distribution process.</p>
<p>Focus/Niche. By following this strategy the organisation is targeting particular parts of the market, such as certain customer groups or on a regional area. This basis for competition is selective but, within the niche market, competition is either on a low cost or differentiation basis.</p>	

Figure 3.1 - Porter's generic business strategies (after Robson)

This model identifies three different strategies that an organisation can employ - product differentiation, overall cost leadership and focus/niche. To implement Porter's model Robson (1994) suggests a three step approach.

- ***What basis?*** - *alternative competitive strategies*

The basis (or alternative competitive strategy) is to decide on differentiation, cost leadership or focus/niche. Reading the text in the model we can see that quality is all important - "better" products or "level of quality" being the requirement. The significance of quality as a strategic marketing technique can be seen from the following observations by Hooley and Saunders (1993).

"A prime factor in differentiating the product or service from that of competitors is quality". They continue, "of central importance is the consumer's perception of quality which may not be the same as the manufacturer's". "Quality has been demonstrated to be a major determinant of success. Indeed, Buzzell and Gale (1987) concluded that relative perceived quality (customer's judgements of the quality of the supplier's offer relative to its competitors) was the single most important factor in affecting the long-run performance of a business. Quality was shown both to have a greater impact on ROI level and to be more effective at gaining market share than lower pricing".

- ***Which direction?*** - *alternative decisions*

Under, which direction, Robson quotes seven alternatives; do nothing, withdrawal, consolidation, market penetration, product development, market development and diversification.

- *do nothing* - do nothing new, continue as before, go with the flow
- *withdrawal* - the organisation withdraws from the market - no demand
- *consolidation* - stabilise with a view to accumulating reserves for future development
- *market penetration* - generating growth within the same market
- *product development* - creating new products for the same market
- *market development* - expanding through new market uses, geographical expansion or new market sectors
- *diversification* - new products for a new marketplace

So the organisation will have to decide which directional approach best suits their situation at any particular time.

• **How?** - *alternative methods*

Having decided on which direction it is then necessary to decide to what extent quality can be incorporated into their plans. According to Robson, two of these directions - *consolidation* and *market penetration* - are achieved by increasing quality. So software developers are well advised to include quality as part of the strategic policy.

Examples of the three alternative strategies in the marketplace are:

Differentiation. A very successful Irish payroll system which was developed to reflect social insurance, income tax and other criteria which were specific to Irish employment. This product had many usability quality factors that competing products - which had been developed for other countries - were missing. The developers were able to charge a higher price because of their differentiation strategy.

Low cost leadership. There is a very well known microcomputer operating system which has been persistently sold by its developers at a low and affordable cost as a strategy to gain market penetration for their product. The same developers also produce a full range of office automation products which run on this operating system. So the low cost leadership strategy has achieved the desired market penetration and has also supported their efforts to achieve a dominating position in the marketplace. Another example relates to the Internet. Service providers are prepared to make browsing software for the Internet available free of charge to fee-paying subscribers to their internet service.

Focus/Niche. The industry standard product for professional desktop publishing on the Apple range of computers is an excellent example of focus/niche. It provides full functionality for the tasks to be completed and is the leader by far in the marketplace. However, having only the one product the developers need to be conscious of their vulnerability. There are other specialist applications (the professions like engineering, medicine, auctioneering etc.) that fit into the focus/niche category.

3.1.2 Tendering and cost estimation

If the developers are producing product on a bespoke basis then entering into contracts will be a major part of their business. Obviously being ISO 9000 certified will be in the developers favour when seeking enquiries. Current European Union requirements stipulate that those seeking to submit tenders for projects within the European Community must be ISO 9000 certified. It also follows that organisations that are ISO 9000 certified for tendering purposes will have little difficulty marketing their generic products on the High Street.

It is easy to agree that quality costs money. Most people can quote a motor car example. So we should expect that software quality factors will play a prominent part in software costing and estimating. And so they do. Two tables prepared by Boëhm (1984) - see figures 3.2 and 3.3 - as part of the Constructive Cost Model (COCOMO) model for software estimating, list factors like re-use, reliability, conformance with external interface specification (interoperability) and storage constraints. All of these are specific quality factors. Also to be considered as part of the estimating process is the all embracing "need for software conformance with pre-established requirements". Good estimators will want to consider all quality factors in their calculations irrespective of whether they are stated in the requirements specification or not.

Group	Factor
Size attributes	Source instructions Object instructions Number of routines Number of data items Number of output formats Documentation Number of personnel
Program attributes	Type Complexity Language Re-use Required display Display requirements
Computer attributes	Time constraint Storage constraint Hardware configuration Concurrent h/w development Interfacing equipment/software
Personnel attributes	Personnel capability Personnel continuity Hardware experience Application experience Language experience
Project attributes	Tools and techniques Customer interface Requirements definition Requirements volatility Schedule Security Computer access Travel/rehosting/multi-site Support software maturity

Figure 3.2 - Factors used in cost estimating models. (Adapted by Ghezzi *et al.* from Table II of Boehm 1984)

Feature	Mode		
	Organic	Semi-detached	Embedded
Organisational understanding of product objectives	Thorough	Considerable	General
Experience with working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specification	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures and algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	< 50 KDSI	< 300 KDSI	All sizes

Figure 3.3 - COCOMO software development modes. (Adapted by Ghezzi *et al.* from Boëhm 1984)

3.1.3 Human Resources and quality

Human resource management is part of all good strategic policy. Brendan Lawlor, Quality and Methods Manager at Kindie Banking Systems in Dublin, believes that quality is achieved through people. He is supported by Gillies (1992) who holds the following view:

- It is people and human organisations who have problems to be tackled by computer software.
- It is people who define the problems and specify the solutions.

- It is (currently) people who implement designs and produce code.
- It is (currently) people who test code.
- It is people who use the final systems and who will make judgements about the overall quality of the solution.

So software developers need to put in place human resources policies as part of their strategic plans. As part of this strategy the following issues might be considered:

- Selecting only top talent to work on a development team.
- Matching staff skills to tasks to be completed.
- Understanding that the special career progression needs of computer people are not necessarily the same as for other staff.
- Creating teams with balanced age, experience and skills.
- Removing mis-fits who are not achieving with the team.

3.1.4 Productivity

And finally, productivity is improved through focusing on quality factors and through a quality assurance system. Quality factors like re-usability and portability improve the production process in that development time and testing time are both reduced. Developing with maintenance in mind will reduce the time and cost of maintaining products in their later life. Object oriented techniques, too, will improve productivity through well organised class libraries and re-use philosophies.

In relation to the implementation of ISO 9000, Macfarlane (undated) explains that the first evidence [of productivity] appears when an engineer says "It's really nice to always be able to get a current copy of the functional specification". Obviously a published quality policy and quality plan, understood by all employees will make for a more efficient workforce. Improved productivity ensues.

3.2 Quality and the systems purchaser

Similar quality topics are of interest to the system purchaser but from a different perspective. In this case the issues are evaluation and selection of generic products or specifying user requirements as part of a contract. Later these requirements will be managed from a user's viewpoint during development and then tested to ensure that the quality requirements have been built-in.

3.2.1 System acquisition

Those charged with the responsibility for acquiring an organisation software applications will certainly want to focus very much on all of the quality factors. They will also want to review and evaluate a number of different candidate applications. For evaluating and selecting

software products Robson (1994) suggests that some form of scoring and ranking (weighting and rating) must be used. Her general principles are:

- 1 Select the criteria
- 2 Associate importance % weights with each criteria
- 3 Score individual candidate systems in terms of how criteria are satisfied
- 4 Calculate each candidate's rating (Summation of all scores x weight)
- 5 Select the one with the highest score

Figure 3.4 shows Robson's partial weighted selection tree for a small office automation project. Under the software heading the criteria are recognisable as quality factors. It would be appropriate to extend this list to include a full range of quality factors.

IS professionals also need to be able to specify quality requirements for bespoke products. In this situation, all of the quality factors should be considered, those appropriate included, and criteria for measuring and acceptance defined.

3.2.2 User productivity

Usability is a major concern for an organisation investing in information systems. User staff will reject software that is not suitable for the purpose intended, that is difficult to use, that is difficult to learn or that cannot be adapted to suit the users preferences and skills. All of this impinges directly on productivity and for large systems with perhaps thousands of users the productivity gains can be substantial. Furthermore, in addition to the lost cost of the unusable system, staff will become dissatisfied which could result in high staff turnover and the further cost of that. The aim should be a quality human-computer interface which makes the system transparent to the users with resulting productivity gains.

Weight	Weight	Weight	Score
20% Hardware	30% Processors	20% Cost 20% Size 10% Speed 25% Modularity 25% Compatibility	
	20% Communication devices		
	50% Output devices	50% Quality 25% Speed 25% Running costs	
40% Software	Reliability		
	Modularity & expandability	Ease of changing Ease of enhancing Portability	
	Usability	Ease of use Performance Training	
10% Vendor	Reliability Flexibility Maintenance Research and upgrades		
20% Cost	Price Implementation Financing alternatives		
10% Benchmark results	Time to execute standard task		
	Time to execute standard transmission		
	Time to make backup		
		Total Score	_____

Figure 3.4 - Robson's partially weighted selection tree for a small office automation project

3.3 Summary

In this section the importance of quality for those making strategic decision has been reviewed from the point of view of the supplier and customer. Five areas of major impact were explained viz:

Marketing strategies where quality software fits into a differentiating strategy.

Tendering and cost estimating where the value of ISO 9000 certification is most important and where quality factors are inputs into the COCOMO model for costing.

Human resources issues were explained and particularly the fact that humans are central to specifying, developing and using software.

Productivity gains to both the developer and the user result in different ways from quality policies and

System acquisition techniques that employ weighting and rating matrices which include quality factors as the criteria.

4 Conclusion

Software quality definitions as defined by international organisations during the 1970s and 80s have become a little outdated in that they do not reflect the high technological developments of the 1990s. Software quality models and factors which were first defined in the late 1970s have also become outdated as a result of industry developments. In particular the impact of the microcomputer industry needs to be reflected in models. Definitions for factors like usability, efficiency and maintainability are being or need to be redefined while new factors like suitability, learnability and adaptability are evolving. The interrelationships between quality factors as set out by Perry needs to be amended to reflect these evolutions and the impact of human-computer interaction need to be addressed.

The importance of quality for both the supplier and the purchaser was investigated in the second part of the paper. Issues covered were marketing strategies, tendering and cost estimation, human resource issues, productivity and system acquisition.

5 References

- AZUMA, M. (1987) "Software quality assurance", Vortragsmanuskript zum Vortrag, Vol 12(6), 1987 an der ETH Zürich, Germany
- BOËHM, B. (1978) Characteristics of software quality, Vol 1 of TRW series on software technology, North-Holland, Amsterdam, Holland
- BOËHM, B. (1984) "Software engineering economics", IEEE transactions on software engineering, January 1984, 10(1) p4-21
- BUZZELL, R.D. AND GALE, B.T. (1987) The PIMS principles, The Free Press, New York, N.Y., USA
- COUNCIL DIRECTIVE (90/270/EEC) 1990 Minimum safety and health requirements for work with display screen equipment, Official journal of the European Communities, L156/14
- CURSON, I. (1996) "Practice makes ... a difference: thoughts on customer centred design of usability services", Interfaces, No 31 Winter 1996/96, p 3-5
- DAILY, K. (1992) Quality management for software, NCC Blackwell Limited, Oxford, England
- FRENCH, C.S. (1986) Data processing, DP Publications, Hants. UK
- GHEZZI, C., JAZAYERI, M. AND MANDRIOLI, D. (1991) Fundamentals of software engineering, Prentice-Hall, New Jersey, USA
- GILLIES, A. (1992) Software quality: Theory and management, Chapman & Hall, London, England
- HOOLEY, G.J. AND SAUNDERS, J. (1993) Competitive positioning: the key to market success, Prentice Hall International (UK) Ltd., Hertfordshire, England
- INCE, D. (1994) ISO 9001 and software quality assurance, McGraw-Hill, Berkshire, England
- ISO 9000 (1987) Quality management and quality assurance standards, International Standards Organisation, Genève, Switzerland
- ISO 9000-3, (1991) International standard. Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software, International Standards Organisation, Genève, Switzerland

ISO/CD 9000-3.2, (1996) Committee draft. Quality management and quality assurance standards - Part 3:Guidelines for the application of ISO 9001 to the design, development, supply, installation and maintenance of computer software, International Standards Organisation, Genève, Switzerland

LOTUS, (1993) Lotus Communications Architecture:a blueprint for working together. A Lotus white paper 1993, Lotus Development Corporation, Middlesex, UK

MCCABE, T. (1976) "A complexity measure", IEEE transactions on software engineering, SE-2 No 4, p308-320

MCCALL, J.A., RICHARDS, P.K. AND WALTERS, G.F. (1977) Factors in software quality, Vols I-III, Rome Air Development Centre, Italy

MURINE, G. AND CARPENTER, C. (1984) "Measuring software product quality", Quality progress, Vol 7(5), p16-20

OXBORROW, E. (1986) Databases and database systems:concepts and issues, Chartwell-Bratt, UK

PERRY, W. (1987) Effective methods for EDI quality assurance, Prentice-Hall, New Jersey, USA

PORTER, M.E. (1980) Competitive strategy:creating and sustaining competitive advantage, The Free Press, New York, USA

ROBSON, W. (1994) Strategic management and information systems, Pitman Publishing, UK

SHNEIDERMAN, B. (1987) Designing the user interface:strategies for effective human-computer interaction, Addison-Wesley, USA

SOMMERVILLE, I. (1992) Software engineering, Addison-Wesley Publishing Company, Wokingham, England

WALLMÜLLER, E. (1994) Software quality assurance:A quality approach, Prentice-Hall International, Hertfordshire, UK

6 Cyberography

MACFARLANE, M.L. (Undated) "Eating the elephant one bite at a time:Effective implementation of ISO 9001/TickIT", <http://www.exit109.com/~leebee/case01.html>

[Word count 9560]