Software Quality Measurement

BUTLER COX
P.E.P

PEP Paper 14, May 1990

BUTLER COX

P,E,P

Software Quality Measurement

PEP Paper 14, May 1990
by Martin Langham

Martin Langham

Martin Langham is a senior consultant with Butler Cox in London, where he coordinates the company's systems management consulting activities. He has broad experience in the planning, management, and implementation of technically advanced computing applications, with a particular emphasis on database management, distributed computing, and data communications.

During his time with Butler Cox, he has carried out numerous consulting assignments. Recent projects in which he has been involved include formulation of an information systems strategy for an electricity supply utility, assessment of the management aspects of an insurance company's data communications network, and preparation of recommendations for an international chemical company on the changes required to improve and standardise its European development activities. He has also been extensively involved in productivity assessments for PEP members, and is the principal author of the Butler Cox Foundation Report *Managing the Evolution of Corporate Databases*.

Prior to joining Butler Cox, he spent 10 years with BIS ASL, where he managed consulting business in the area of distributed systems. Earlier, he worked as a consultant with both ICL and Unisys, supporting major customers in both the public and private sectors.

Martin Langham has a BSc in physics from Bristol University and is a member of the British Computer Society. He is a frequent speaker at conferences and has published widely in the trade press.

Published by Butler Cox plc
Butler Cox House
12 Bloomsbury Square
London WC1A 2LL
England

Printed in Great Britain by Flexiprint Ltd., Lancing, Sussex.

BUTLER COX

P,E,P

Software Quality Measurement

PEP Paper 14, May 1990
by Martin Langham

Contents

Chapter 1

Getting the best from software quality measurement

Many systems departments have initiated software quality-assurance programmes to increase the effectiveness of applications development. The majority of systems departments find, however, that it is very difficult to direct such programmes and to justify their cost when there is no quantitative evidence of their benefits. The missing component of many software quality-assurance programmes is software quality measurement.

Software quality measurement provides essential management information to the systems department

Software quality measurement enables the qualities of applications, such as reliability, ease of use, maintainability, and so on, to be quantified in useful and consistent terms. Properly implemented, a measurement programme will help the systems department to specify and produce applications of the quality that users require, to identify where improvements might be made to the development process, and to justify the costs of a software quality-assurance programme. In effect, software quality measurement provides essential management information to the systems department.

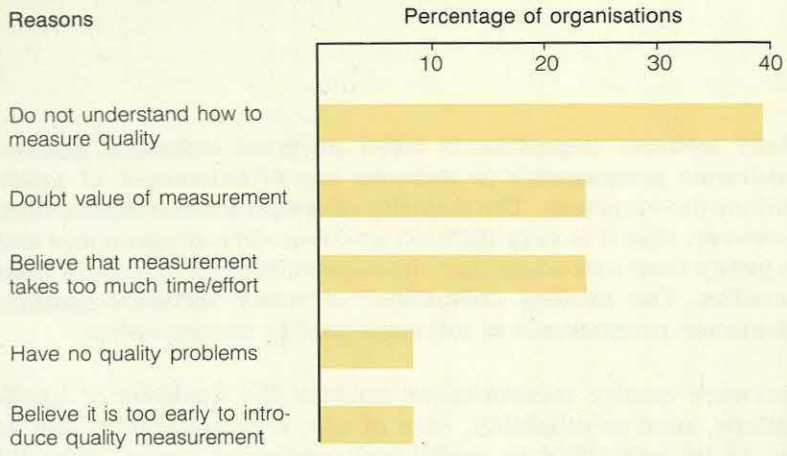
Much of the published material is too academic to be suitable for commercial use

In practice, however, software quality measurement has met with mixed success. In many systems departments, there is considerable resistance to the concept, generally based on misunderstandings about its purpose, its cost, and the level of effort required to introduce it. In others, where quality measurement programmes have been implemented, only limited benefits have been gained, because the scope of the programmes has been too narrow. While there is a wealth of material available on the subject of software quality measurement, much of it is of a very academic nature, not well suited to the commercial environment, and much of it is applicable only to particular aspects of applications development. What is required is a practical, consistent, and comprehensive approach to measuring software quality that will ensure that both users and developers are satisfied with the applications that are delivered.

THERE IS SOME RESISTANCE TO SOFTWARE QUALITY MEASUREMENT

While interest in the subject of software quality measurement is growing, many systems departments are still developing applications whose quality is never measured. They offer various reasons for their reluctance to institute measurement programmes, the most common of which, as shown in Figure 1.1 overleaf, is a lack of knowledge about how to measure quality. Most of the others are based on misconceptions about the role of quality measurement in the applications development process, and about the burden that such programmes might place on the department in terms of cost and effort.

Figure 1.1 The most common reason for not measuring software quality is a lack of understanding of how to do it



(Source: Telephone survey of PEP members)

THE OBJECTIVES OF QUALITY MEASUREMENT ARE NOT CLEARLY UNDERSTOOD

The role of software quality assurance is misunderstood

Some systems departments believe that software quality measurement is unnecessary because they do not have serious quality problems. This view is based on a misunderstanding of the role of software quality assurance, which is to produce software with *appropriate* quality characteristics, not to produce the *best* quality at any cost. Only with a carefully controlled measurement programme in place can the systems department be confident that it is achieving the levels of quality that the business requires, and can afford.

Many systems staff are apprehensive about the motives for introducing software quality measurement

Many systems staff are apprehensive about the motives of introducing a software quality measurement programme. Development managers must therefore make it very clear to everyone in the systems department and in the user departments that the objectives of the programme derive from business objectives, and are not aimed at measuring the performance of individuals. This means that quality will be assessed at the project level, not at the level of the individual developer. Individual developers will not, however, be unaffected by the introduction of software quality measurement; for a programme to work well, many of their attitudes and procedures will have to change. Their support and commitment is essential and they must therefore fully appreciate the real objectives of a measurement programme and trust that the results will be used constructively.

THERE ARE CONCERNS ABOUT THE EXTRA BURDEN ON THE SYSTEMS DEPARTMENT

Some development managers are sceptical about the value of a software quality measurement programme, arguing that it cannot produce useful results at a reasonable cost. Experience shows, however, that this is not the case.

Software quality measurement can lead to significant cost savings

Gerald E Murine, the founder of METRIQS, the first specialist software quality measurement company, claims that organisations that use his company's services have achieved considerable cost savings. In an article published in the November 1988 edition of *Quality Progress*, he quotes the results of the comparative analysis of data collected over a nine-year period on a project whose software quality was measured and similar projects whose quality was not measured. The total cost of the project, including the cost of measuring software quality throughout the development project, was reduced by 35 per cent. Most of the savings resulted from reductions in coding and module-testing effort. There were also considerable improvements in the quality of the software. Evidence from PEP assessments confirms that the more 'hard' evidence developers have about the development process and the projects in which they are involved, the better able they are to control their work.

It is important to start collecting measurement data as soon as possible

A few PEP members believe that it is too early for them to introduce quality measurement. It is important, however, to start collecting measurement data as soon as possible so that systems managers have a basis on which to plan future developments. Waiting until the development environment is stable is not justified; if software quality measurement is to be of value in the long term, it must be implemented in such a way that it is still useful in a changing environment. Collecting the data required to measure software quality properly is not too onerous a task — useful benefits can be gained by spending the equivalent of 1 or 2 per cent of development effort on gathering measurement data. In many systems departments, much of the data will already be available in a machine-readable form from systems used to manage development resources, change requests, and the correction of software faults.

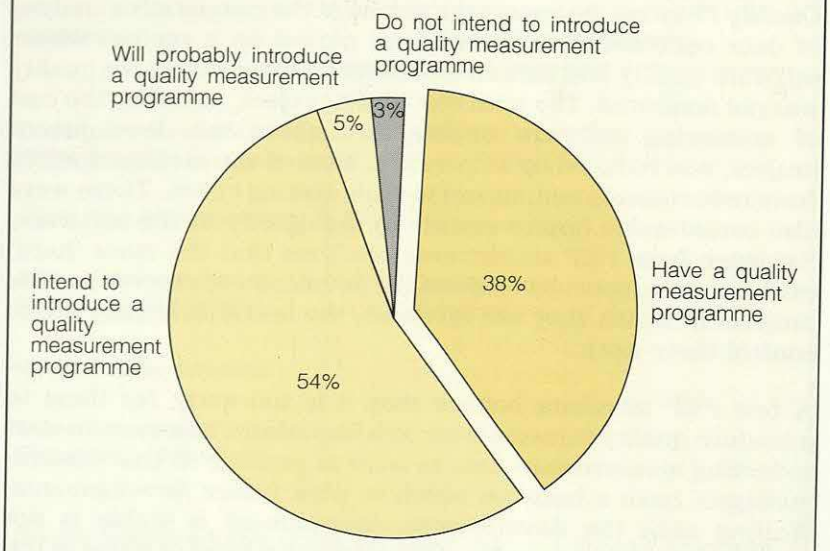
EXISTING SOFTWARE QUALITY MEASUREMENT PROGRAMMES ARE OFTEN LIMITED IN SCOPE

Our survey of PEP members showed that nearly 40 per cent already measure the quality of the software they produce, and that a further 60 per cent either definitely intend to adopt a software quality measurement programme, or will probably do so (see Figure 1.2, overleaf). However, many of those who do measure software quality measure only a few characteristics of the software that they produce and support. Most of the measures serve to assess technical qualities, such as reliability and maintainability, rather than user-perceived qualities, such as ease of use and flexibility. Figure 1.3, on page 5, shows the results of a survey of the most commonly used software quality measurements. By far the greatest emphasis is given to measuring qualities that are of little interest to users. This is consistent with the results of our own survey of PEP members' objectives in introducing software quality measurement, shown on page 6, in Figure 1.4. Top priority goes to improving the development process; improving user satisfaction ranks only third.

At present, by far the greatest emphasis is given to measuring qualities that are of little interest to users

We do not believe that this is due to a lack of interest in achieving this objective, but to a lack of communication between developers and users about software quality requirements. This, in turn, arises from the fact that software quality characteristics are rarely

Figure 1.2 Many PEP members are beginning to measure software quality



(Source: Survey of PEP members)

defined in meaningful and measurable terms. Software quality will remain an abstract concept to users unless they can specify their quality requirements in measurable units, and developers can deliver applications that demonstrably meet those requirements.

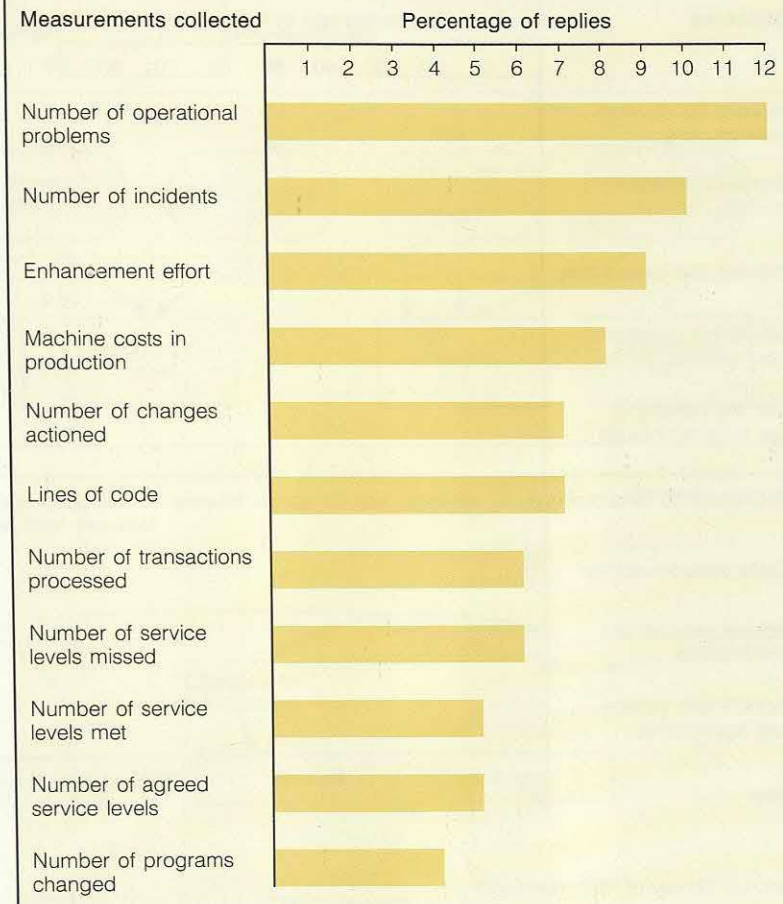
There are several types of users whose expectations of quality have to be met

The problem for developers is compounded by the fact that there are several types of users whose expectations of quality have to be met. These include the application users and their managers, systems development managers, maintenance and technical support staff, and computer operations staff. As Figure 1.5 on page 7 shows, each group of users is interested in different outputs from the systems development process. Each of these groups has its own perceptions of the characteristics of a high-quality system, and in some cases, they will have conflicting requirements.

The measurement process must be seen in a much broader perspective

Satisfying the quality demands of all the various parties will require a much broader perspective to be taken of the measurement process than has hitherto been common. As Figure 1.4 shows, most PEP members who had introduced quality measurement programmes had done so in order to improve the development process. It is this aspect of quality that is of greatest interest to systems development managers and it is in this area that techniques and methods for measuring software quality are most widely available. Other types of users, however, are more concerned with the quality of the final application, with predictions of the quality of the final application while it is being developed, or with the quality of existing applications. Only very limited techniques are available for measuring these aspects of software quality. As Figure 1.6 on page 7 shows, a quality measurement programme must cover all these aspects if it is to serve its true purpose.

Figure 1.3 Most software quality measurements are of little interest to users



(Source: Special interest group on software metrics of the Quality Assurance Forum, Meeting no. 3)

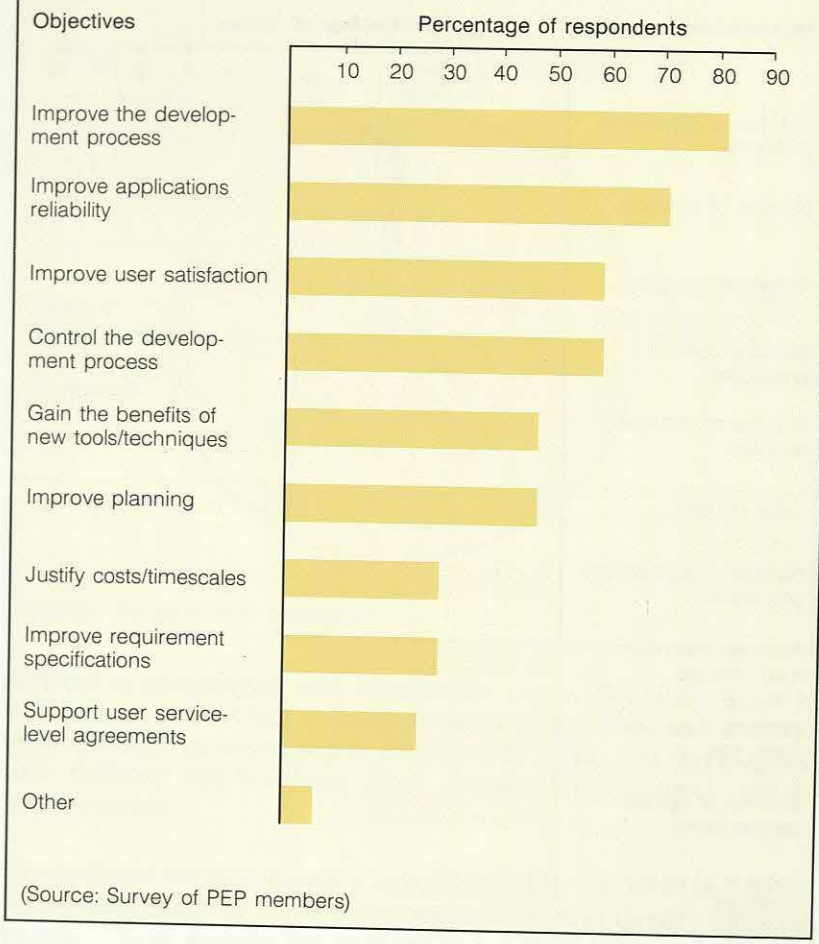
STRUCTURE OF THE PAPER

One of the major problems in applications development is that trouble spots are frequently not identified until after they have jeopardised the success of an entire project. Software quality measurements provide the development manager with valuable information that should help him to pinpoint the stages in the development process where problems are occurring, and to evaluate accurately the effects of any changes that are subsequently made to the process. In Chapter 2, we describe the role that software quality measurement can play in helping to improve the development process and to control individual projects.

There are no well defined techniques to help manage the quality of the final application

The techniques that are used to improve the development process and to control individual developments are quite simple, and can be easily adapted to the particular circumstances of individual systems departments. No similarly well developed techniques exist to help manage the quality of the final application as it is delivered to the users. At present, it is often difficult to know, at the development stage, whether the quality of the final application

Figure 1.4 PEP members have a wide range of objectives for introducing software quality measurement



(Source: Survey of PEP members)

will be acceptable to users. Emerging techniques of software quality measurement will enable systems managers to take a more rigorous approach to defining users' quality requirements in measurable terms, and we suggest how they should begin to build on these in Chapter 3.

Ensuring that users' quality expectations are met is a difficult task, however, because the final application is not available for measurement until the end of the project, and it is then too late to correct any quality defects. Project managers therefore need some means of predicting whether they will achieve the agreed quality requirements of the users. This aspect of software quality measurement is the subject of Chapter 4. It is concerned with ways in which project managers can monitor the progress of an application by measuring the quality of the interim products of the development process and taking action to modify those products where the software quality measurements indicate that the final quality is likely to be unsatisfactory.

In most organisations, the development of new systems accounts for only about half of the systems department's time and effort. Software maintenance is an equally large consumer of resources, and in Chapter 5, we describe the major contribution that software quality measurement can make to improving the quality of

Figure 1.5 Different groups of users are interested in different outputs from the systems development process

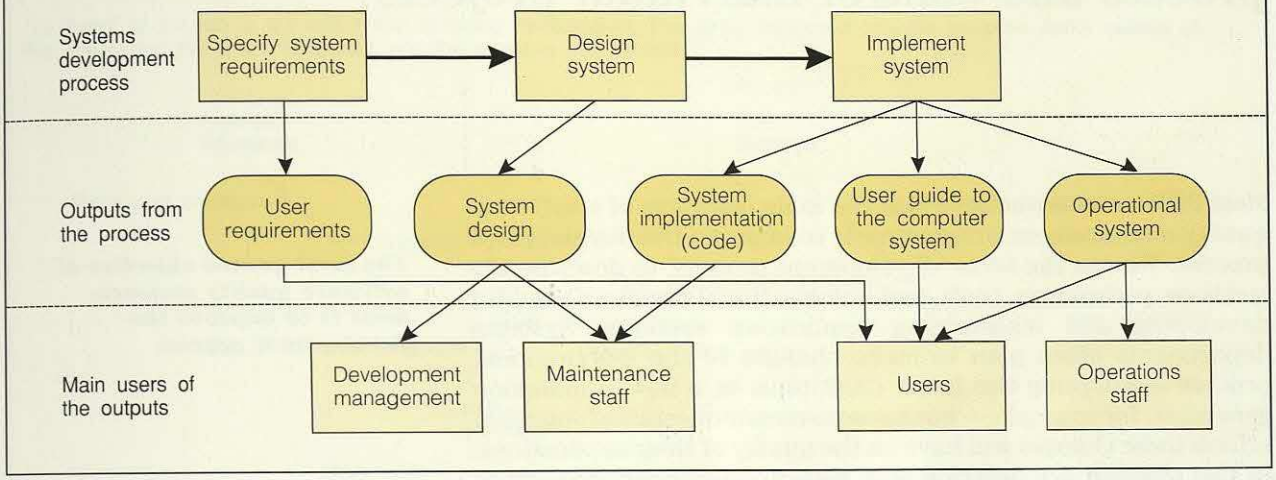
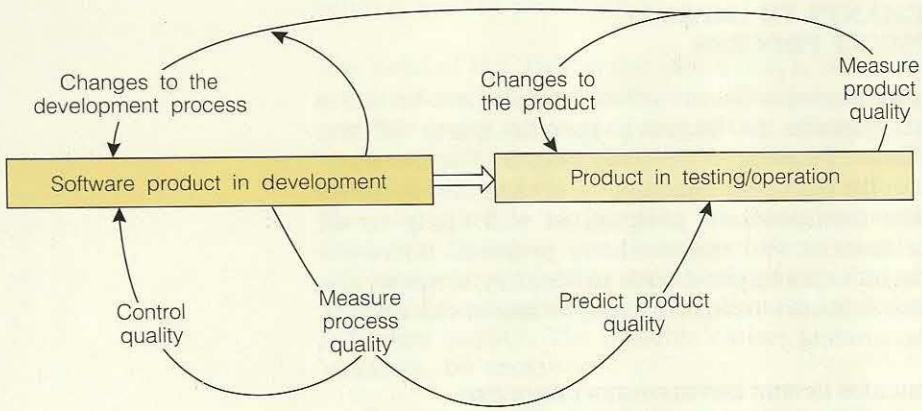


Figure 1.6 A software quality measurement programme should cover all the aspects of applications development, from the process itself to the final product



existing applications by indicating where it might be possible to reduce their complexity, and hence, the cost and effort involved in maintaining them.

RESEARCH SOURCES

Our research began with a review of the published literature on the subject of software quality measurement. We also met practitioners in the field — academics, suppliers of software quality measurement tools, quality-assurance specialists, and systems development managers in commercial organisations. We should like to offer our particular thanks to Peter Mellor of the Centre for Software Reliability, Barbara Kitchenham of the NCC, and Hugh Browton and his team, of STC Technology Limited (STL).

We also conducted telephone interviews with 20 PEP members and sent out a questionnaire both to PEP members and to other commercial systems departments. Where appropriate, the results of these surveys have been included in this report.

Chapter 2

Using measurements to improve the development process and control individual projects

Most PEP members believe that the main objective of a software quality measurement programme is to improve the development process. We use the term 'development process' to describe the methods, techniques, tools, and organisational practices used for developing and maintaining application systems. Systems departments often plan to make changes to the development process — adopting the latest CASE tools or a new application generator, for example — but have no means of establishing what effects these changes will have on the quality of their applications. In this chapter, we describe how measurements can be used to manage the systems development process, either to improve the process itself, or to provide better control over development projects as they move through the stages of the process.

The most quoted objective of software quality measurement is to improve the development process

USE MEASUREMENTS TO IMPROVE THE DEVELOPMENT PROCESS

Quite rightly, systems departments often choose first to use a measurement programme to improve specific parts of the development process. Starting in this way means that the data-collection task can be kept to a minimum and that any benefits resulting from the measurement programme will apply to all subsequent development and maintenance projects. Software quality measurements can be used both to identify areas in the process where problems are occurring, and to assess the effects of changes to the process.

IDENTIFYING PROBLEMS IN THE DEVELOPMENT PROCESS

In many systems departments, the process of developing applications is a mysterious art. Prescribed techniques are followed without question, and there is no way of quantifying either the positive contribution or the adverse effects of particular techniques to the quality of the application. Indeed, our consulting experience shows that it is quite possible for a systems department to be unaware of serious flaws in its development process, even though these flaws may well prevent it from achieving the quality and productivity objectives that it has set. This problem can be tackled in two stages: identifying the possible causes of a quality problem, and then applying the measures to identify the root cause.

Many systems departments are unaware of serious flaws in their development process

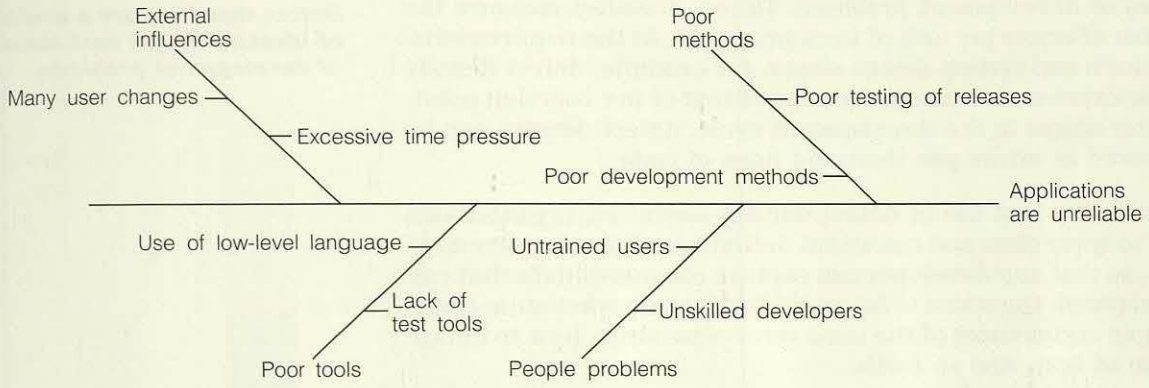
Identify the possible causes of a lack of quality

The first stage is to obtain a full understanding of the area that is causing concern so that possible causes of the problem are identified for subsequent assessment. The best tool for this task is an Ishikawa or 'fishbone' diagram, an example of which is shown in Figure 2.1. This type of diagram was invented by Dr Kaoru Ishikawa in 1952 to control processes in the Kawasaki iron works in Japan. (Dr Ishikawa is now recognised as Japan's

Ishikawa or 'fishbone' diagrams can help to identify the causes of quality problems

Figure 2.1 An Ishikawa or 'fishbone' diagram can help to identify the possible causes of a quality problem

The 'head' of the fish is the effect that is being investigated. The large 'fishbones' are the possible major causes of that effect, and the small 'fishbones' are the possible sub-causes.



leading authority on quality control.) The Ishikawa diagram has since been widely adopted throughout the world as an aid to solving quality problems.

The head of the 'fish' is the effect that is being studied (unreliable applications, in our example). The large 'fishbones' are the possible major types of causes of that effect — for example, poor tools or methods, people problems, or external influences. The small 'fishbones' are, in turn, the possible sub-causes, and so on. The objective is to identify the truly important causes, and fishbone diagrams are therefore often used in brainstorming sessions where possible causes of poor quality are discussed and analysed, and subsequently explored further using measures of software quality. The possible causes to be explored might, for instance, be that:

- Faults are introduced disproportionately in one stage of the development process.
- Most of the faults are of one type.
- Changes requested by users during development cause many of the faults.

Measurement data can help to identify which of the possible causes is the real culprit

Use measurement data to identify the root cause

The second stage is to use measurement data to identify which of the possible causes of the problem is the real culprit. Each cause to be investigated will need a carefully designed 'experiment' to collect data and analyse it. Two points are very important in ensuring that the investigation produces a result that management can rely on:

- Sufficient data should be collected and analysed to produce statistically significant results. A description of 'statistical significance' is beyond the scope of this paper. A useful primer on statistical analysis is listed in the bibliography.
- The conditions under which the data is collected should not change.

These two conditions imply the need for a stable development process in which each change is carefully considered. Progress is gradual but sure. This approach to process improvement is typical of the Japanese approach to quality management.

Measures of defect densities can often be used to identify the root causes of development problems. Defect densities measure the number of errors per unit of work produced. At the requirements-definition and system-design stages, for example, defect density can be expressed as errors per page of text or per function point. At later stages in the development cycle, defect density can be measured as errors per thousand lines of code.

Defect densities are a useful way of identifying the root causes of development problems

To make the best use of defect density measures, organisations need to apply clear and consistent definitions of the measurement units, so that any developer can capture consistent data that can be compared. Questions to be considered include whether to count multiple occurrences of the same error separately, how to define a page of text, and so forth.

If common definitions of defect densities are established, organisations can compare the quality of their development work with that of other organisations to identify the areas of their development process that produce an above-average number of errors. Such industry comparisons are frequently used in PEP assessments to provide insights into the strengths and weaknesses of a PEP member's development process.

Figure 2.2 is a diagram that was used to analyse the development stages at which errors originated and were identified in one development project. About half of the errors originating at the functional-design stage did not, for example, come to light until the testing stage. This type of analysis can be used to identify where inspections and walkthroughs are (and are not) an effective means of improving quality.

ASSESSING THE EFFECTS OF CHANGES TO THE DEVELOPMENT PROCESS

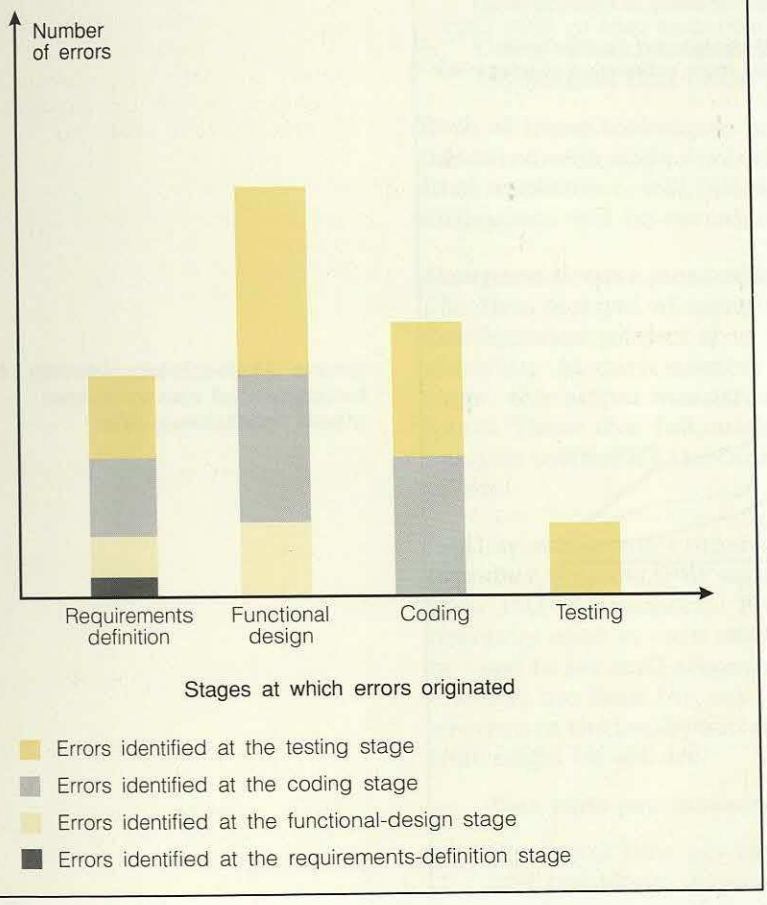
Once the root cause of a quality problem has been identified, the systems department will make changes to the development process in an attempt to resolve the problem. Quality measurement data can be used to assess whether the change has the intended effect. A paper published in the September 1988 edition of *Quality Assurance* explains how the IBM CICS support team at IBM United Kingdom Laboratories, Hursley Park, did just this.

IBM used quality measurements to assess whether changes to the development process had the desired effect

IBM recognised the need to set levels for software reliability and defects such that customers would perceive the software as being of high quality. The company also recognised the need to define formally the development processes that would enable these objectives to be achieved. Analysis of the root causes of the defects showed that the majority of problems resulted from inadequate design. As a consequence, three changes were made to the development process:

- Extensive use was made of a high-level language (PLAS) for writing new code.
- A formal specification and design language (Z notation) was introduced.

Figure 2.2 Software measurements can be used to analyse the development stages at which errors originate and are identified



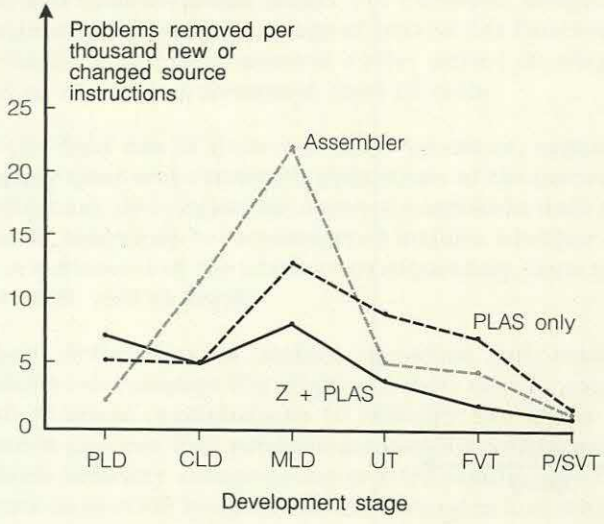
– Greater management attention was given to ensuring that development methods were complied with, and to monitoring and controlling changes to existing code.

This revised process was used for some of the development work for release B of CICS.

Three different development techniques were used — traditional techniques using English-language design to change old Assembler code, software-engineering techniques using English-language design and PLAS code, and software-engineering techniques using Z design and PLAS code. Figure 2.3, overleaf, shows one of a series of analyses that IBM made to identify the effects of using these three techniques. For each technique, it shows the number of defects removed per thousand new or changed source instructions at each of six life-cycle stages (product-level design, component-level design, module-level design, unit testing, functional verification testing, and product/system verification testing). With the combined Z design and PLAS technique, fewer defects are introduced, and they are removed in a far more uniform way throughout the development process. In addition, fewer defects are discovered at the testing stages, implying that a more reliable product will be released.

Figure 2.3 Software measures can be used to compare the effects of different development techniques on the stages at which defects are removed

The figure compares the number of problems removed at each development stage, for three different development techniques used by IBM's CICS support team. With one of the techniques (Z design and PLAS), fewer defects are introduced, and they are removed more uniformly throughout the development process.



PLD Product-level design
CLD Component-level design
MLD Module-level design
UT Unit testing
FVT Functional verification testing
P/SVT Product/system verification testing

(Source: *Quality Assurance,* vol. 14, no. 3, September 1988)

USE MEASUREMENTS TO CONTROL INDIVIDUAL PROJECTS

Relying solely on a development method to control a project has disadvantages

The conventional approach of relying solely on a development method to control a project has three main disadvantages — it is difficult and expensive to assess the extent to which the method is being applied, the highly prescriptive nature of many development methods inhibits developers' initiative and can be demotivating, and the risks of project failure are high unless methods and applications are very carefully matched. Using software quality measurement in conjunction with a development method is a more objective way of assessing the progress of a project because it can identify problems that occur during the development of an application, and indicate where corrective action should be taken.

IDENTIFYING PROBLEMS IN THE DEVELOPMENT PROJECT

Software quality measurements complement the development approach

In essence, a development method is concerned with the quality of the inputs to the development process, not the quality of the outputs. Software quality measurements assess the outputs from the development process, so that potential quality problems can be identified and corrected before the application is delivered,

and they thus complement the development method. Two techniques can be used:

- Comparing measurements made at specific points in the development process with predefined limits.
- Using measurements to identify the few components of the application that cause most of the problems.

Both of these techniques are based on the expectation that the measurements made during a project that results in a good-quality final application will follow a typical pattern, and that project difficulties will be revealed by unusual measures.

Compare actual measurements with predefined limits

At suitable checkpoints, actual measures can be compared with predefined limits

The first method of using software measurement to monitor a development project is to set predefined limits during project planning. At each suitable checkpoint, say at the end of each stage, the actual measures are compared with the predefined limits. Those that fall outside the limits are subjected to further analysis to identify the causes, and corrective action can then be taken.

It is important to set reliable and useful limits based on the norms for other projects. PEP members can use the data collected from their PEP assessments. For example, the proportion of effort normally used at each stage of a project of a particular size can be used to set staff-resource limits for each stage. A project that exceeds the limit for, say, the design stage, can be expected to overrun at the implementation stage as well. Other suitable limits that might be set are:

- Test runs per thousand lines of source code.
- Computer time per thousand lines of source code produced and tested.
- The ratio of the size of the design document to the size of the requirements specification (in pages).

These measurements can be made only at the end of a stage or after completion of a module.

Some software quality measures can be monitored continuously

Some software measurements can be monitored continuously, however, and without too much extra effort:

- The rate at which requests for changes are made and the rate at which they are implemented. Plotted over time, these measures should produce convergent lines. Divergence indicates possible problems.
- The trend in incident reports during system testing. (Sample plots were shown in PEP Paper 13, *Software Testing*.) This graph should indicate a uniform trend that declines towards the end of system testing. If the trend does not decline, problems are likely to occur.
- The stability of the requirements and the design, monitored by counting the number of design changes and measuring the number and size of modules developed that were not originally planned. High and rising levels of change indicate problems.

In PEP Paper 13, we described how past experience can be used to set predefined limits for the number of errors that can typically be expected at the system-testing stage. Testing progress can then be continuously monitored against this limit. This type of measure and those described above can be used to make frequent assessments of the 'health' of a project.

Identify the few components that cause most of the problems

It is often the case that most of the quality problems of an application are caused by a small number of the software components. The ability to identify, at the development stage, those components that are most likely to cause the quality problems will make it possible for corrective action to be taken before the application is delivered. Over time, it is possible to build up a profile of the types of software modules that are likely to cause quality problems. Two useful measures to start with are:

Over time, it is possible to build up a profile of the types of software modules that are likely to cause quality problems

- The ratio of actual module sizes to their expected sizes. High ratios may indicate modules that are too big to be developed and tested efficiently.
- The amount of computer and staff resources used to produce and test a module. A high level of resources in relation to the module's size will indicate that there are problems in developing it.

USING THE MEASUREMENTS TO INDICATE WHERE CORRECTIVE ACTION SHOULD BE TAKEN

The techniques described above can be used to pinpoint the stage of the development process or the software module in which a quality problem originates. Further analysis will be needed to establish the root cause of the problem because, in practice, there could be several possible reasons for it. Suppose, for example, that the computer time used for running tests is plotted over time, and is found to fall outside the predefined limits at the end of the system-build stage. Higher-than-normal run times could be due to the early commencement of system and integration testing, the development of code with a higher-than-usual number of errors, or the testing of processor-bound algorithms. Likewise, lower-than-normal run times could be caused by uncompleted unit testing or by the discovery of easy-to-detect errors.

Further analysis will be needed to establish the root cause of the problem

Research into using project-monitoring techniques to identify quality problems has been carried out in an ESPRIT project (REQUEST) led by STL (a sister company of ICL). This research has produced a system that is able both to identify the cause of an applications development problem and to provide relevant advice on how to correct it. The system has been produced as part of an automated quality-management system that will help a quality or project manager throughout the life of a software project. The quality-management system provides:

STL's ESPRIT research has produced a system that can identify the causes of a development problem and advise on how to correct it

- A project-planning and initiation subsystem to help create quality plans, to specify measurable quality targets and required measures, and to predict final product quality.
- A project-monitoring subsystem that uses quality measurement data to advise on project status in quality terms, as described above.

— A project-assessment subsystem that reports on the final product quality achieved and on expected maintenance and support costs implied by the level of quality achieved.

The prototype of this system runs under Unix and parts of it have been demonstrated to the author. The project-monitoring subsystem has been used successfully on several STC software development projects. We expect that this type of system will become commercially available by 1992/93.

In this chapter, we have described how software quality measurement can be used to improve the development process and to control individual development projects. The techniques are simple, well understood, and easily adaptable to the particular circumstances of individual systems departments. Unfortunately, no similarly well developed measurement techniques are currently available to help manage the quality of the final application as it is delivered to the users. In the future, software measurement is likely to play an increasingly important role in controlling applications quality, just as the measurement of elapsed time and resources is used today to monitor and control project duration and effort. It is therefore important that systems managers begin to build on the emerging techniques that are available. In the next chapter, we offer guidelines on how they should do this, to ensure that they deliver applications that meet their users' quality expectations.

Software quality measurement is likely to play an increasingly important role in controlling applications quality

Chapter 3

Defining users' quality requirements in measurable terms

In this chapter, we show how software quality measurements can be used to help manage final applications quality so that users' expectations are met. There are three stages in managing the quality of the final application: understanding the users' quality requirements, identifying an appropriate set of measures, and setting quality priorities for applications development in terms of those measures.

UNDERSTAND USERS' QUALITY REQUIREMENTS

Systems departments often state that they could produce better-quality applications if their users gave them more time. On the other hand, users often do not understand why so much development effort is spent on work that does not appear to be directly related to their requirements. The result is that neither users nor developers are satisfied with the quality of the delivered applications.

We believe that this problem arises because it is difficult for developers and users to communicate with each other about the quality requirements of applications software. In turn, this difficulty stems from the fact that software quality characteristics are not usually defined in useful and measurable terms. Applications software quality will remain an abstract concept to users unless they can specify their quality requirements in measurable units, and developers can deliver applications that demonstrably meet those requirements.

Software quality characteristics are not usually defined in useful and measurable terms

The first stage in managing the quality of an application as it is delivered to the users is therefore to understand the quality requirements of its users. The classic definition of quality is 'fitness for purpose' and this means that different types of applications, and applications used by different types of users, will have different quality 'profiles'. Producing applications with an inappropriate quality profile wastes resources and does little to satisfy users.

The first stage is to understand the quality requirements of users

There are several obstacles to be overcome before the systems department can be certain that it has identified the important criteria by which users will judge the quality of an application. In particular, users often cannot express their quality requirements in terms meaningful to development staff. The systems department must therefore define quality characteristics in terms that enable users to understand both how the characteristics are measured and what the implications of poor and high quality are. Another problem arises from the fact that there are often conflicts between the quality requirements of different groups of users. As we emphasised in Chapter 1, the systems department must be able to reconcile the often conflicting quality needs of the various groups.

Users find it easier to specify quality requirements with reference to applications they use than to abstract specifications

The best way to understand and establish users' quality requirements is to conduct a survey of users' perceptions of the quality needs for future developments. Such a survey should relate their future needs to their experience with existing applications. Users find it much easier to specify their quality requirements with reference to applications that they use regularly, than to abstract specifications of systems.

Figure 3.1 shows the results of a typical user-satisfaction survey, used by Butler Cox in its consulting work. Users are asked to rate eight qualities of an application on a scale from one (poor) to seven (excellent). The solid black line shows the average quality assessments for all the applications surveyed and the other two lines show the quality assessments for two particular applications. Such a survey can be used to identify the quality factors that are rated lowest, and to which attention should be directed for new applications. In the example shown, the 'average' line indicates that users were generally least satisfied with the ease of use of the system and its associated aspects — user documentation, training, and support. A section of the questionnaire that was used to produce these results is shown in Figure 3.2, overleaf.

COMPILE AN APPROPRIATE SET OF MEASURES

Quality measures must be derived from the business objectives of the software quality measurement programme

The set of measures that will be used for assessing the quality of an application should be derived from the business objectives of the software quality measurement programme. Unless the programme has a business purpose that is clearly expressed and agreed, it will not produce significant and long-term benefits. The objectives should apply to all applications that are being developed

Figure 3.1 User surveys can be used to identify quality factors that need to be improved

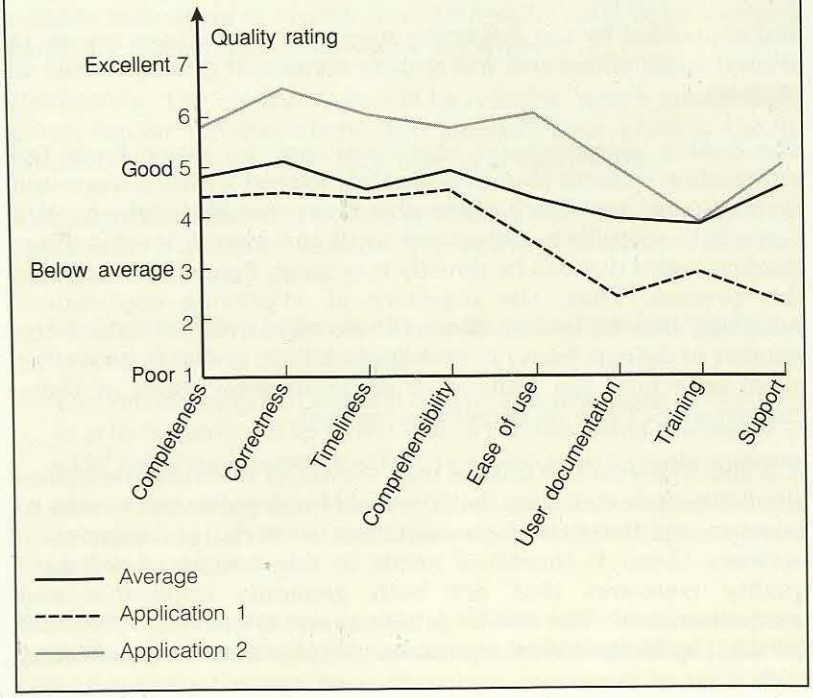


Figure 3.2 User survey questionnaires serve as the basis for identifying where quality needs to be improved

What is your opinion of the existing systems applications that you use?

The attached list contains the most important systems applications. Please select from that list up to three applications that are the most important for you and that you use frequently. Please answer the following questions for each of the applications selected by entering a score between 1 and 7 into the three columns of boxes (leave boxes blank which do not apply to you).

insufficient		poor		good		excellent
1 completely irrelevant	2	3 not very important	4	5 important	6	7 essential

	First application	Second application	Third application
3.1 Two-digit code for the application (see attached list)	☐☐	☐☐	☐☐
3.2 How important is this application for your work?	☐	☐	☐
3.3 How satisfied are you with this application?	☐	☐	☐
3.4 How do you assess this application concerning:			
— Completeness of results?	☐	☐	☐
— Correctness of results?	☐	☐	☐
— Timeliness of results?	☐	☐	☐
— Clarity of results?	☐	☐	☐
— Ease of use?	☐	☐	☐
— User manual?	☐	☐	☐
— Training facilities?	☐	☐	☐
— Support available from systems staff?	☐	☐	☐

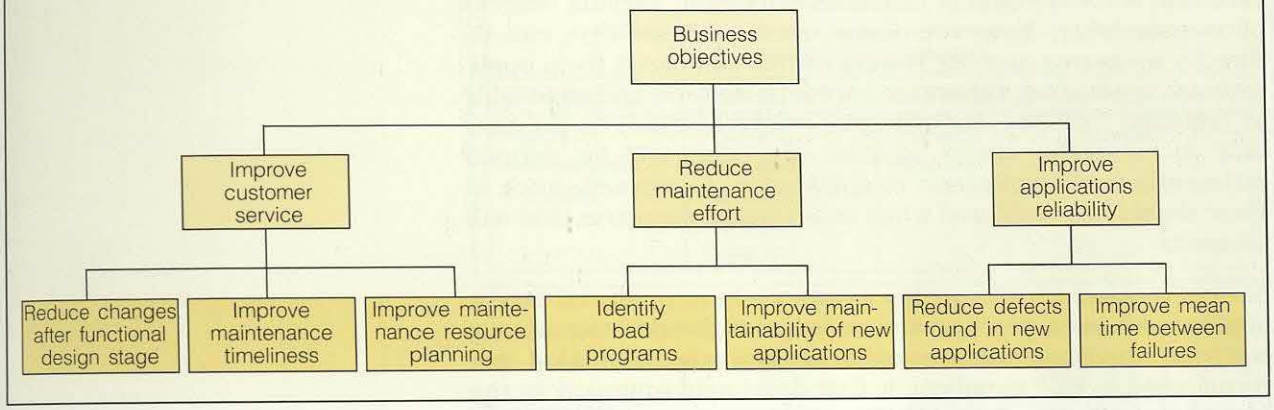
and supported by the department — most users have access to several applications and will require consistent quality across all of them.

The quality measurement objectives may be taken from the information systems plan or separately agreed with the users and systems managers. Each of the objectives should then be broken down into subsidiary objectives until the lowest level defines characteristics that can be directly measured. Figure 3.3 illustrates this process. Thus, the objective of improving applications reliability may be broken down in two objectives: to reduce the number of defects found in new applications, and to improve the mean time between failures of an application. Both of these characteristics can be directly measured.

Quality measurement objectives should be broken down into subsidiary objectives until they define characteristics that can be directly measured

It is also important to ensure that the set of measures describes all of the characteristics that the systems department wishes to measure, and that there is the minimum of overlap and interaction between them. It therefore needs to select a set of software quality measures that are both generally applicable and comprehensive. The NCC publication, *Measuring Software Quality*, by Richard Watts, provides useful guidelines for selecting such a set of measures, under six headings:

Figure 3.3 Software quality objectives should derive from business objectives and be broken down to a level where they can be directly measured



Standardisation and comparability: Ideally, measures of software quality should enable applications quality to be compared on a standard scale. However, it is not possible to compare some quality measurements between applications or between different organisations, and nor are comparisons with other organisations always essential; valuable information can be gained from identifying trends and extreme variances in the measures.

Measures need to be 'normalised' before they can be compared

Before the quality measures for different applications can be compared, the effects that the different application characteristics, such as size and frequency of use, have on the quality measure have to be compensated for. In other words, the measures have to be 'normalised' before they can be compared.

Objectivity: Quality measures should be as objective as possible so that the value does not depend on the tester's judgement during measurement, analysis, or evaluation. Measures made by subjective assessments (ease of use, for example) may still be reliable indicators of quality, but the results need to be assessed carefully before any conclusions are drawn.

Reliability: The measures should be reliable, which means that, given similar circumstances, the measurement process should produce the same result each time it is carried out.

Validity: The measures should produce a valid assessment of an application's qualities. There are three ways in which a measure can assess software quality:

- The relationship between the measure and the quality factor can be direct — as for reliability.
- The relationship can depend on an understanding of the way in which quality affects the characteristic being measured — as in using maintenance effort to assess ease of maintenance.
- The relationship may depend upon empirical validation of a relationship, such as that between design complexity and software reliability.

Economy: Quality measures should be cost effective, so that the benefit gained from using the measures is much greater than the cost of making them.

Usefulness: The measures should provide information that helps management to decide on the action that needs to be taken.

Different software quality characteristics have varying degrees of measurability, however. Some, such as reliability, can be directly measured and the results can be compared from application to application. Others are harder to measure and impossible to compare. Usability, for example, can be assessed by the user only in subjective terms, and in any case, will be defined differently from application to application. Constraints such as these should be considered when selecting the measures that will be used.

Several researchers into quality assurance have produced lists of quality measures. One of the most widely accepted among software quality-assurance experts, and the one that we recommend to PEP members, is that developed originally in the United States for the Rome Air Development Center, and known as the RADC approach. The RADC approach to measuring software quality derives from research work carried out in the late 1970s and early 1980s. RADC had been pursuing a programme intended to achieve better control of software since 1976. The programme's aim was to identify the key issues and provide a valid method for specifying and measuring quality requirements for software developed for major Air Force weapon systems.

The programme defined a set of 11 user-oriented characteristics, or quality factors — reliability, flexibility, maintainability, reusability, correctness, testability, efficiency, usability, integrity (which actually refers to security), interoperability, and portability — which extend throughout the software life cycle. These 11 quality factors were originally defined to help predict the quality of a final application as it is being developed. In Chapter 4, we show how the factors are used for this purpose. We have also devised an appropriate user-oriented measure for each of the factors so that the quality of *existing* applications can be assessed. These measures are described in Chapter 5.

The RADC programme defined a set of 11 user-oriented characteristics

Although the RADC approach was originally defined for military applications, it has successfully been applied to the development of commercial computing applications. (Full details of the RADC research have been published by the US National Technical Information Service as *Software Quality Measurement for Distributed Systems — Final Report*. Copies can be obtained from ILI, Index House, Ascot, Berkshire. The approach is clearly described in the books by J Vincent, A Waters, and J Sinclair, listed in the bibliography.)

Figure 3.4 shows how the 11 RADC quality factors match the four *quality characteristics* that we defined in PEP Paper 9. For the purpose of this paper, however, it is more convenient to classify the factors into three groups: those that are independent of a particular application, those that are specific to a class of application, and those that are application-specific.

QUALITY FACTORS INDEPENDENT OF THE APPLICATION

Six of the 11 RADC quality factors describe characteristics that can be defined independently of the application, which means that the quality measures for different applications can be

Figure 3.4 The 11 RADC quality factors can be categorised in terms of the four quality characteristics defined in PEP Paper 9

Quality characteristics			
Functional	Technical	Operational	Ease of use
Integrity Interoperability Portability	Correctness Re-usability Maintainability Flexibility Testability	Efficiency Reliability	Usability

compared. The reliability of an operating system, for example, can be compared with that of a computer game.

Reliability

Reliability is defined as the rate of failure of an application in use. The failures may be a partial or complete functional failure, or inaccuracies in results. The measure of this quality factor answers the user's question: "Will the application work when I use it and will it produce accurate results?" It can be readily measured because the frequency of application failures is often recorded by the operations department.

Flexibility

The flexibility of an application is the ease with which perfective and adaptive maintenance can be done. (Perfective maintenance is changing the software's structure to improve its performance and maintainability; adaptive maintenance is concerned with enhancing and extending systems software to incorporate the evolving needs of users.) The measure of this quality factor answers the question: "How easily can the functions of the application be changed?"

Low levels of maintenance productivity may indicate inflexible applications

A further useful measure of flexibility is a comparison of the productivity of the development staff involved in making changes with the productivity of those carrying out new development work. This comparison (which can be based on PIs for enhancements and new developments) can be used to decide whether to scrap and rewrite an application or to continue maintaining it.

Maintainability

The maintainability of an application is the ease with which corrective maintenance activity can be carried out. This measure answers the question: "How easily can faults be fixed?" A measure of the maintainability of an application is the average effort required (in developer hours) to find and fix a fault.

Re-usability

The re-usability of an application is the extent to which all or parts of the code can be re-used in other applications. This measure answers the question: "Does this application provide an opportunity to save costs by re-using its components in other applications?"

Measurement of this quality factor is difficult because the level of actual, as opposed to intended, re-use cannot be established

until long after the application has been developed. A practical measure of re-usability is the proportion of the application that is composed of re-used modules.

Correctness

The correctness of an application is the extent to which the application conforms to the stated requirements. This measure answers the question: "How faithfully have the users' requirements been implemented?" Correctness does not, however, assess the ability of the application to produce correct results. This ability is an aspect of the reliability of the application. A suitable method of measuring correctness is the number of application defects found. A defect is any difference between the application requirements and its implementation.

Testability

The testability of an application is the ease with which the application can be tested to ensure that it will perform its intended function. This measure is an important attribute affecting the reliability and the cost of maintaining an application. It answers the question: "Can I test the application thoroughly and easily?" A measure of the testability of the application is the number of test cases that are needed to test the application fully.

QUALITY FACTORS SPECIFIC TO A CLASS OF APPLICATION

Two of the 11 quality factors (efficiency and usability) describe characteristics that must be defined specifically for each *class* of application. Thus, for the class of workstation-based applications, the important efficiency measures are memory requirements and speed of calculation. For batch programs, the important efficiency measures are processor usage and elapsed time to process a given volume of transactions.

Efficiency

The efficiency of an application is measured in terms of the computer resources needed to provide the required functions with the required response time. Efficiency can be measured in terms of processor usage, disc-storage requirements, and so on. Response time is not a measure of the efficiency of an application because it is a characteristic of the particular combination of hardware and systems software in which the application runs, rather than a characteristic of the applications software. The measure of this quality factor answers the user's question: "Will the application use a reasonable (affordable) amount of computer resources?"

Usability

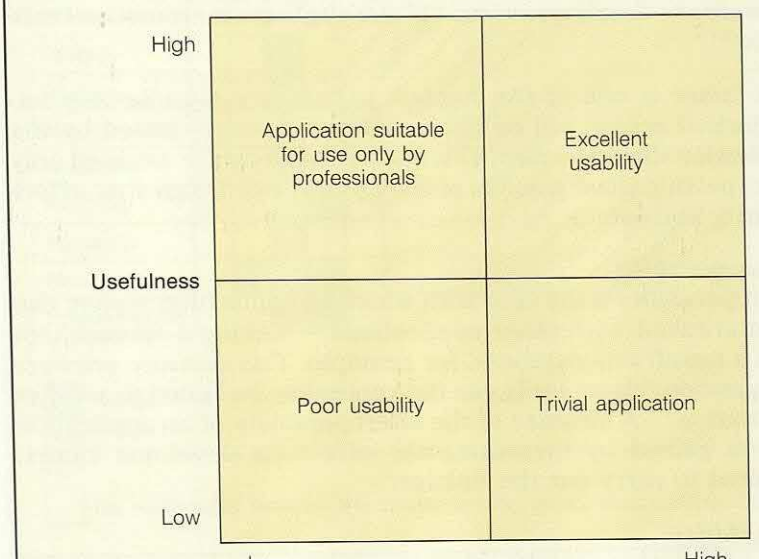
An application has a high measure of usability if it can be used easily to produce useful results. The measure of this quality factor has two components: the effort required to learn and use the application, and the usefulness of the application. It answers the user's question: "Will the application be useful to me?"

Usability is related to the effort required to learn how to use an application . . .

Research has shown that these two components of usability are a good indicator both of the likely level of use and of user satisfaction with an application. Users consider an application

to have a high level of usability when the return on the time they invest in learning the application is commensurate with the benefits obtained from using it. Figure 3.5 shows how these two components interact to define the usability characteristics of the application.

Figure 3.5 Usability is a combination of ease of use and usefulness



Ease of use can be assessed only when the application is in everyday use. A suitable method of measuring ease of use is to count the number of unfounded fault reports and the number of requests for support made by users of the application. Where ease of use is of particular concern, the application should be designed to capture instances of incorrect use automatically.

. . . and is entirely subjective

The level of usefulness of an application is entirely subjective, and can be assessed only by carrying out user surveys. Properly conducted surveys can provide a consistent assessment of the usefulness of an application and the way it is changing over time.

QUALITY FACTORS SPECIFIC TO AN APPLICATION

Three of the 11 quality factors (integrity, interoperability, and portability) describe characteristics of the application that are based on specific application requirements. For example, a portability requirement could be for the application to run on two specific computer systems. This type of requirement will differ from one application to another. These three quality factors are therefore *application-specific* and cannot be compared between applications.

The qualities of integrity, interoperability, and portability also depend on features of the application design. The best way to measure these qualities in an application is by means of a checklist

of relevant facilities. Integrity qualities, for example, can be inferred by the presence of features such as audit trails and access-control facilities.

Integrity

The integrity quality factor as defined by RADC is really about measuring the security and safety of an application. It provides a measure of the ability of an application to resist unauthorised access (security) and to protect those who use it from being harmed in some way (safety). The measure of this quality factor answers the user's question: "Is the application secure and safe to use?"

Integrity is one of the hardest quality factors to measure

This factor is one of the hardest to measure because any unauthorised access will be illicit and any damage caused by the application should be rare. This quality factor can be assessed only by inspecting those aspects of the application design that affect security and safety.

Interoperability

Interoperability is the ease with which an application system can be interlinked with other applications — linking a spreadsheet with a mainframe database, for example. This measure answers the question: "How easily can the application be linked to another application?" A measure of the interoperability of an application can be gained by measuring the effort (in developer hours) required to carry out the linkage.

Portability

Portability is the ease with which an application can be transferred from one computing environment to another. It answers the question: "How easily can the application be transferred to another software and hardware environment?" A measure of portability can be gained by measuring the developer hours required to transfer an application to a different environment — for example, the effort required to recompile and test a Cobol application to run on a different computer.

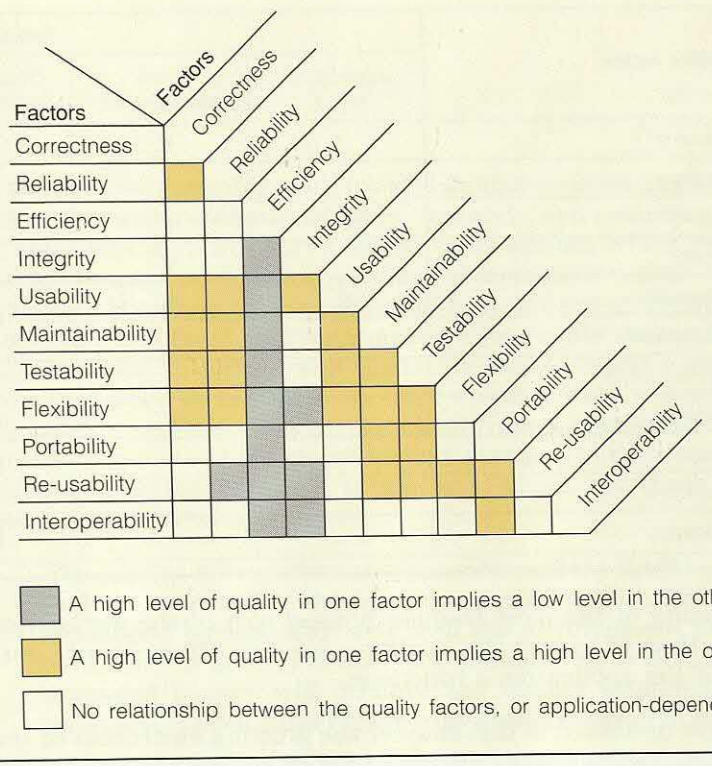
SET QUALITY PRIORITIES FOR APPLICATIONS DEVELOPMENT

When developing a new application, it is not always possible to meet all of the quality requirements desired by all groups of users. There are two main reasons for this. First, a high level of quality in one of the 11 RADC quality factors may imply a low level of quality in one of the other factors. For example, a high level of portability will usually imply a low level of efficiency, and *vice versa*. The main conflicts that can occur between the quality factors are shown in Figure 3.6. Second, the project manager will often have to make trade-offs between the time, cost, and quality of the application. The implication is that quality should not be specified at a higher level than the application warrants. For example, the maintainability requirements of the application can be reduced if the lifetime of the application is known to be short.

Quality should not be defined at a higher level than the application warrants

The conflicts in quality priorities occur because of the conflicting requirements of the main groups who 'use' an application — the application's users, their managers, the development managers, the maintenance and support teams, and computer operations

Figure 3.6 A high level of quality in one factor may imply a low level in another factor



staff. Figure 3.7, overleaf, shows which of the 11 quality factors are of most interest to each of these groups. Because of the complexity of these conflicting interests, most systems departments will need to select just two or three of the quality factors that they need to control during development. The cost of controlling more qualities than this becomes prohibitive.

The choice of quality factors will depend on the nature of the system, but they must be limited in number

Usually, the most important quality factors to concentrate on are the three that will increase user satisfaction through reduced costs and better service — maintainability, flexibility, and reliability. The choice will, however, depend on the nature of the system in question. Typical quality requirements for specific types of application are:

- Systems with a long life: maintainability, flexibility, and portability.
- Publicly accessed systems: usability, integrity, and reliability.
- Systems that can cause damage to property or lives if they go wrong: reliability, correctness, testability, and integrity.
- Systems that use advanced technology: portability.

The cost of providing high quality may be minimised by developing an operational-use profile

A valuable technique for minimising the cost of providing high (user perceived) quality is to develop an operational-use profile. Such a profile shows the expected level of use of each of the functions of the application. Suppose, for example, that an application has two main functions, one of which will be used for 90 per cent of the time and the other for 10 per cent of the time. In this situation, it is obviously better to concentrate on improving

Figure 3.7 Different groups of users have different quality needs

Quality factor	Group of users				
	Application users	User management	Development management	Maintenance and support	Operations
Reliability		✓			✓
Efficiency					✓
Usability	✓	✓			
Integrity		✓			✓
Correctness	✓	✓			
Interoperability			✓	✓	
Maintainability			✓	✓	
Flexibility		✓	✓	✓	
Portability				✓	
Testability			✓	✓	
Re-usability			✓		

the quality of the most frequently used part of the application rather than to spread the quality-assurance effort evenly over both of the application's functions.

We have described in this chapter the growing awareness of the role of software quality measurement in controlling the quality of the final application at the development stage, and we have discussed some of the approaches that the systems department might take to ensure that users' quality expectations are met. It is a difficult task, however, because the final application is not available for measurement until the end of the project. Project managers therefore need some means of *predicting* whether they will achieve the agreed quality requirements of the users. We turn our attention to this aspect of software quality measurement in Chapter 4.

Chapter 4

Using measurements to predict the quality of the final application

Once an application is complete, it is too late to correct any quality defects

In the previous chapter, we provided guidance on setting quality objectives with reference to users' demands, and compiling a set of measures that will serve as the basis against which project managers can assess whether those objectives are likely to be achieved. However, it is not practical to wait until an application has been completed and then measure its quality, because it is then too late to correct any quality defects. Project managers therefore need to be able to predict whether they will achieve the quality requirements of the users from assessments of the interim products of the development process, so they can take corrective action before it is too late. There are three main approaches to this task:

- *The quality-of-conformance* approach uses the RADC quality factors to monitor the quality of the application being developed by assessing how well the development process conforms to good practice. The assumption is that a well managed project will produce the quality intended. This approach concentrates on the *conformance* aspect of quality.
- The *quality-of-design* approach, developed by Tom Gilb, sets quality targets and builds these into the design of the system. This approach concentrates on the *design* aspect of quality.
- The *empirical* approach predicts the quality of the final software by using validated empirical relationships between the characteristics of the interim products of the software being developed and the quality of the final product. This approach also concentrates on the design aspect of quality, but is more objective than Gilb's approach because it relies on established empirical relationships between the characteristics and final quality.

In each case, the measurements made during the development of a project are used to predict the quality of the final application while it is being developed, and to identify trends and patterns that could lead to improvements in the development process. Figure 4.1, overleaf, depicts the essential differences between the three approaches.

SET AND MEASURE QUALITY CONFORMANCE TARGETS

In the quality-of-conformance approach, a project is subjectively assessed for its conformance to quality standards

This approach tackles the problem of predicting final quality during development by subjectively assessing the development project for its conformance to quality standards to establish whether it will produce the required quality. Each of the 11 RADC quality factors is progressively subdivided until it can be expressed as *quality criteria* that can be assessed during development. As Figure 4.2, overleaf, shows, the criteria may be broken down further into *subcriteria* and then into measurable *attributes* of

Figure 4.1 There are three approaches to predicting the quality of the final product

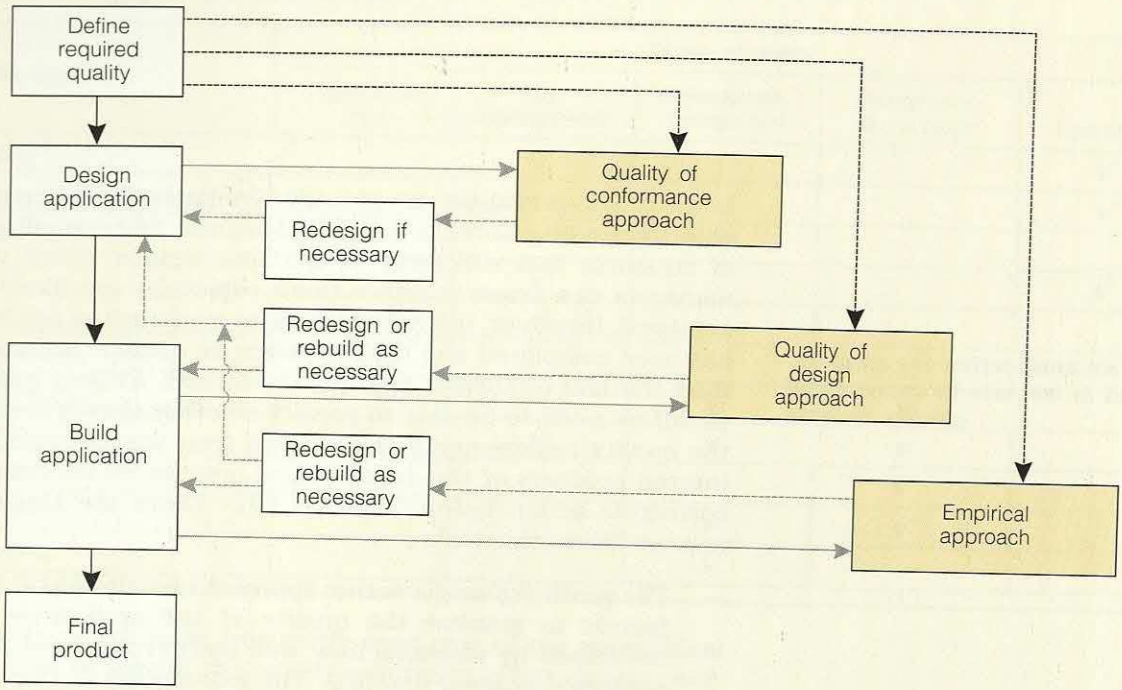
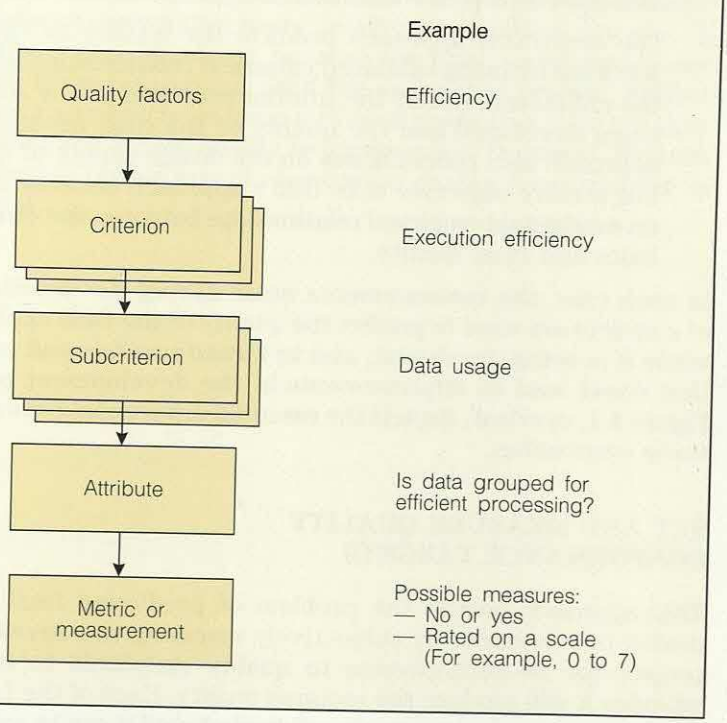


Figure 4.2 Each quality factor is broken down into measurable attributes



the application. In the example shown, one of the quality criteria for efficiency is execution efficiency, which in turn can be divided into several subcriteria, one of which is data usage. The

measurable attribute of data-usage execution efficiency is the extent to which the data is grouped for efficient processing. This can either be assessed simply as 'yes' or 'no', or rated on a scale — for example, from 0 (not grouped at all for efficient processing) to 7 (optimised for efficient processing).

The criteria used to assess one of the quality factors may also apply to other factors. Figure 4.3 lists the measurable attributes that are used to assess the extent to which the completeness criterion is met. This criterion contributes to the quality factors of reliability, correctness, and usability.

Figure 4.3 The measure of completeness contributes to the assessment of the reliability, correctness, and usability quality factors

The completeness quality criterion has nine measurable attributes:

- Unambiguous reference (input, function, output).
- All external data references defined, computed, or obtained from external source.
- All defined functions used.
- All referenced functions defined.
- All conditions and processing defined for each decision point.
- All defined and referenced calling sequence parameters agree.
- All problem reports resolved.
- Design agrees with requirements.
- Code agrees with design.

Each of these attributes should be rated on the same scale (for example, 0 to 7); the measure of completeness is the average of the nine ratings.

A simple example will help to illustrate how the RADC approach might work. Suppose that the quality factors of a car are defined as maximum speed, economy, and safety. The 'speed' quality factor can be broken down into the quality criteria of low wind resistance, low rolling resistance, high power/weight ratio, lightness of construction, and so on. It is obvious that many of these criteria also apply to the economy quality factor. Subdividing the rolling-resistance criterion into measurable attributes produces wheel-bearing friction, energy losses through the tyres and suspension, and so forth. These attributes are scored on a numerical scale — usually subjectively — and the results for all the attributes are added together to create a score for the quality factor. The score is then expressed as a percentage of the total possible score, and is interpreted as follows:

Quality criteria are broken down into measurable attributes, which are scored on a numerical scale

- 95 to 100 per cent: There is a high probability of meeting the quality targets.
- 90 to 94 per cent: Progress should not be impeded, but the items responsible for reducing the score should be dealt with.
- 60 to 89 per cent: Problems are likely to result in a poor-quality final product and increased costs; immediate action is required to identify and rectify the problems.
- 0 to 59 per cent: There are insurmountable problems; the project should be re-organised, or the development approach should be changed.

Many organisations that develop large, complex, software systems (particularly in government, military, and telecommunications systems) have invested considerable effort in implementing the RADC approach to software quality measurement. In particular,

several Japanese IT companies have applied this approach to a range of applications and claim that it has led to considerable improvements in quality. As Figure 4.4 shows, the approach also resulted in significant reductions in effort and cost — in various projects, companies noted a 50 per cent reduction in testing effort, a 46 per cent reduction in coding effort, a 25 per cent reduction in specification effort, and a 33 per cent reduction in cost.

In Japan, the quality-of-conformance approach has resulted in significant reductions in effort and cost

Figure 4.4 Japanese companies have gained considerable benefits from using the RADC approach to software quality measurement

	Project A	Project B	Project C
Software type	Operating system (Assembler)	Cost control (Cobol)	Business application (Cobol)
Phases	All	Design, code	Design, code
Factors controlled	Usability	Correctness Reliability	Correctness Reliability
Elements measured	28	15 (Design) 22 (Code)	9 (Design) 8 (Code)
Number of software quality measurement staff	4	3	1
Proportion of total project cost	25%	20.8%	12.7%
Results	25% reduction in specification effort 33% reduction in cost	50.8% reduction in testing effort	46.2% reduction in coding effort

(Source: *Quality Progress*, November 1988)

The RADC approach does, however, have several significant drawbacks:

- There is no proven connection between the interim measures and the quality of the final product.
- The measurements and scoring of the attributes are largely subjective.
- Collecting the data is labour-intensive and difficult to automate.
- The approach is valid only for a development process that is similar to that used in the RADC study (requirements analysis, preliminary design, detailed design, implementation, and test and integration).

SET AND MEASURE QUALITY DESIGN TARGETS

During the development of an application, developers are continually making conscious or unconscious design trade-offs between the various quality attributes of the final application. An example of such a trade-off is to improve operational efficiency at the expense of ease of maintenance. Setting appropriate design objectives therefore affects the way in which development staff approach their work and helps to channel development effort in the appropriate way.

An experiment has been carried out to assess the effects of setting different objectives on the results of application development.

The results of applications development are clearly affected by the design objectives that are set

Five groups of programmers were given different design objectives for the same application: to minimise memory usage, to minimise the number of statements, to maximise output clarity, to use the least effort, and to make the program as understandable as possible. When the five final applications were compared, each of them met its specific design objective. The results of this experiment, which were described in *Goals and Performance in Computer Programming*, by Gerald E Weinberg and Edward L Schulman, of the School of Advanced Technology at the University of New York, showed that "programming performance can be strongly influenced by slight differences in objectives". The conclusion of the paper was that "No programming project should be undertaken without clear, explicit, and reasonable goals [design objectives]".

Managing the 'producibility' of the application design allows the quality of the final application to be predicted

David Card, author of a recent book, *Measuring Software Design Quality*, states that the objective of the system designer is to create a *producible* design. A producible design will have the qualities of simplicity and ease of understanding, and is most likely to lead to a reliable and maintainable application. Managing the producibility of the application design allows the quality of the final application to be predicted. The ideas set out in his book represent the leading edge of academic thinking on software quality assurance. As far as we are aware, no-one has actually managed to achieve the ideals set out in this book. One approach that goes some way towards them is that developed by Tom Gilb, the well known quality metrics guru and consultant. His approach, which is closely related to his idea of design by objectives, attempts to predict and manage the quality of the final application by estimating the contribution that each aspect of the design makes to the quality objectives.

In the quality-of-design approach, quality objectives are set at the design stage in measurable terms

With Gilb's approach, quality objectives are clearly defined at the design stage in measurable terms. Figure 4.5 provides an example of the definition for the 'reliability' quality factor. This detailed definition of each quality measure ensures that it is as meaningful as possible. The quality objectives are usually broken down into several subsidiary quality objectives, each of which can be measured. Usability, for example, can be broken down into ease of learning, operator error rates, and so on. This means that the quality objectives for the final application are expressed in directly measurable terms. The success or failure in meeting the application quality objectives can thus be directly assessed at the end of the project.

The application quality plan is developed by assessing the contribution that each design feature makes to each of the quality

Figure 4.5 With Gilb's approach, measurable quality objectives are set at the design stage

The 'reliability' quality factor might, for example, be defined as mean time between failures, with the following characteristics:

Measuring unit:	Days
Measuring tool:	Problem-management system
Worst case:	1 day
Planned level:	5 days
Best case:	30 days
Current level	3 days
Consequences of failure:	Service-level agreement penalties

objectives. This process uses a quota-control table, an example of which is shown in Figure 4.6. In the example, the objective for the 'ease of learning' quality factor would be expressed in a measurable way — users should be able to learn how to use the application in less than 30 minutes, for instance. The contributions that a variety of design features make to meeting this objective are then assessed. Thus, in the example, the assessors believe that providing single-line help messages will enable all users to learn how to use the system in less than 30 minutes. However, providing full-screen help messages is, in the view of the assessors, likely to result in only partial achievement of this quality objective — only about one-in-three users would be able to learn how to use the system in less than 30 minutes. Thus, the quota-control table can help to identify those design features that are likely to make a significant contribution to a range of quality objectives. If the 'total' for each quality factor is less than 100 per cent, it is unlikely that the objective can be met, even if all the design features are included in the system.

The application quality plan is developed by assessing the contribution that each design feature makes to each of the quality features

Figure 4.6 In Gilb's approach, the application quality plan is developed by assessing the contribution that each design feature makes to each of the quality objectives

Design feature	Quality factor	
	Ease of learning	Reliability of user operation
Single-line help	100%	50% ± 40
Full-screen help	33%	20% ± 10
Automatic help, based on the difficulty the user is experiencing	10%	10% ± 5
Colour screens	10%	10% ± 5
Total	150%	90% ± 60

Gilb's approach suffers from the weakness that there is no established relationship between a design feature and the quality of the final application. The connection has to be established by the subjective judgement of experienced system designers. Gilb suggests that the task of quantifying each quality objective is made easier by breaking down the objectives into simpler ones, in a manner similar to that required by the RADC approach.

MEASURE QUALITY USING VALIDATED EMPIRICAL RELATIONSHIPS

Neither the RADC nor Gilb's approach to predicting final application quality is completely satisfactory because neither is based on objective relationships between the measures of the interim products of development and the quality of the final product. A better approach is to make use of empirical relationships, established by research, between the characteristics of the interim development product and final product quality. Such an approach is more objective and is easy to apply because the relationships are expressed as simple mathematical formulae.

The empirical approach is based on identifying objective relationships between measures of the interim products and the quality of the final product

An important relationship is that between the complexity of the system design and the maintainability of the final application. By ensuring that design complexity does not exceed a predefined level, the project manager can ensure that the maintainability of the final application reaches the target quality level.

In general, the complexity of an application is a good indicator of its likely quality. Complexity (or its opposite, simplicity) is a quality criterion that contributes to seven of the 11 RADC quality factors — correctness, reliability, efficiency, maintainability, testability, flexibility, and re-usability. In addition, unlike many of the other quality criteria, it does not adversely affect efficiency so that trade-offs between these eight quality factors and efficiency do not have to be allowed for when considering complexity.

The complexity of an application is one good indicator of its likely quality

It is important, however, to distinguish between the complexity of the problem that the application has been designed to solve and the complexity of the application itself. A complex problem does not necessarily imply a complex solution, although the task of developing a simple solution for a complex problem may be difficult.

Complexity can be measured in two ways: by measuring the complexity of the design based on the interconnections between the modules and programs that make up a system, or by measuring the complexity of the code itself. A measure of design complexity is most beneficial in detecting potential quality problems at an early stage, when they can be fixed quickly and cheaply. Unfortunately, the collection of the basic data required to measure design complexity can be laborious, and at present, there are no tools available to carry out this task automatically. Measuring code complexity has the disadvantage that it cannot take place before the code has been written. If a module turns out to be too complex, further work is needed to simplify it. However, tools are available for measuring the complexity of code written in the more popular third-generation languages.

A measure of design complexity is most beneficial in detecting potential quality problems at an early stage

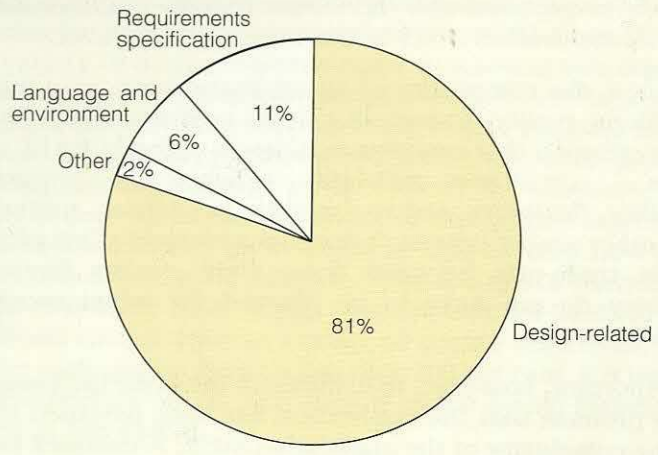
MEASURING DESIGN COMPLEXITY

A high proportion of software errors discovered late in the development life cycle have their origins in the system-design stage. Figure 4.7, overleaf, shows the distribution of the sources of software errors for three actual development projects. The majority of the errors are related in some way to the design of the system. Reducing design complexity is highly likely to reduce the number of design errors, thereby increasing the likelihood of delivering a high-quality system. Design complexity measures define the success of the systems designer in developing a producible design.

Many schemes have been suggested for measuring system-design complexity. Many of these are based on assessing the modular structure of a system; the most widely recognised is the information-flow complexity measure proposed by S M Henry and D G Kafura, of the University of Wisconsin-La Crosse, and Iowa State University, respectively, which provides a measure of module coupling (that is, the links between modules). This measure is calculated from the information 'fan-in' and 'fan-out' for each module. (Details of this measure can be found in

Measuring the complexity of information flow is a widely recognised way of assessing system-design complexity

Figure 4.7 The source of most software errors lies in the design of the system



(Source: Weiss, D M, and Basili, V R. Evaluating software development by analysis of changes. *IEEE Transactions on Software Engineering*, vol. 11, no. 2, February 1985)

'Software Structure Metrics Based on Information Flow', published in *IEEE Transactions on Software Engineering*, vol. SE-7, no. 5, September 1981.)

Information fan-in is calculated from the number of modules that call a module, the number of common data items read by the module, and the number of parameters input to the module. Information fan-out is calculated from the number of modules called by a module, the number of common data items amended by the module, and the number of parameters output from the module.

Modules with high fan-in and fan-out values should be avoided

Modules with a large fan-out value tend to control a large number of other modules and therefore may have too many functions. Modules with a large fan-in value are used by many other modules and therefore, ideally, need to have a single clearly defined function, and to be reliable and operationally efficient. They should therefore be kept small and simple. Thus, from a software-design viewpoint, modules with high fan-in or fan-out values indicate areas where the application may be badly modularised. Such designs should be reviewed before coding commences.

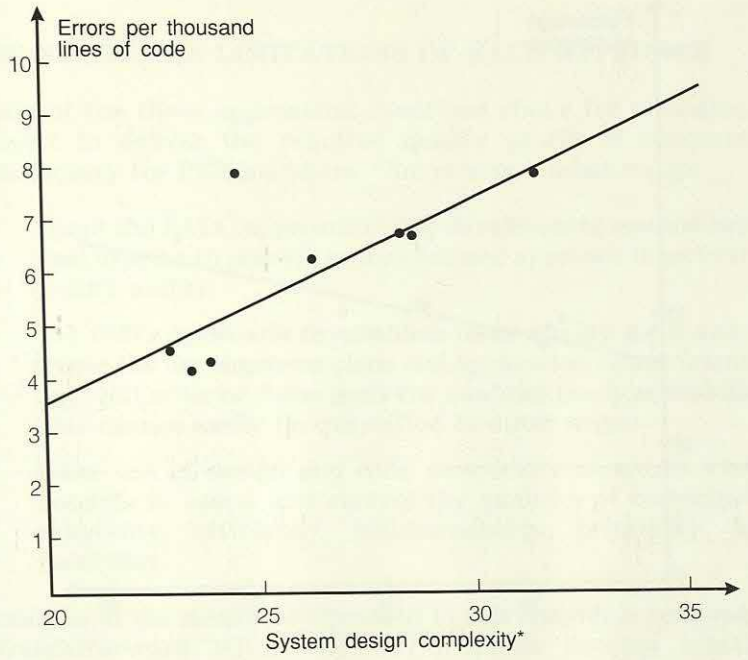
Recent studies have shown that modules with high fan-out values are likely to lead to a variety of quality-related problems. One of these studies analysed 226 programs in a communications system. The results showed that of the 40 programs that had the highest fan-out values, 82 per cent had a significant problem such as higher-than-average error change rates, or were judged to be of above-average complexity.

There is a high correlation between design complexity and error rates

Further work on refining this measure of design complexity has shown that there is a high correlation between design complexity and error rates. Figure 4.8 shows the results of this analysis for eight applications. The complexity of a module is also correlated with the ease with which it can be changed. Figure 4.9, on page 36, shows the proportion of changes (for the same eight

Figure 4.8 The complexity of system design correlates with software error rates

The figure shows the correlation between system design complexity and errors per thousand lines of code for eight applications.



* Complexity was defined by complex mathematical formulae involving number of modules, module format, and intra-module complexity.

(Source: Card, D N, and Glass, R L. *Measuring software design quality.* London: Prentice-Hall, 1990)

applications) that were difficult to make, plotted against module complexity (measured by McCabe's complexity measure, which is described briefly below). This clearly shows that less complex modules have a smaller proportion of hard-to-implement changes.

At present, measures of design complexity are expensive to collect and have limited application

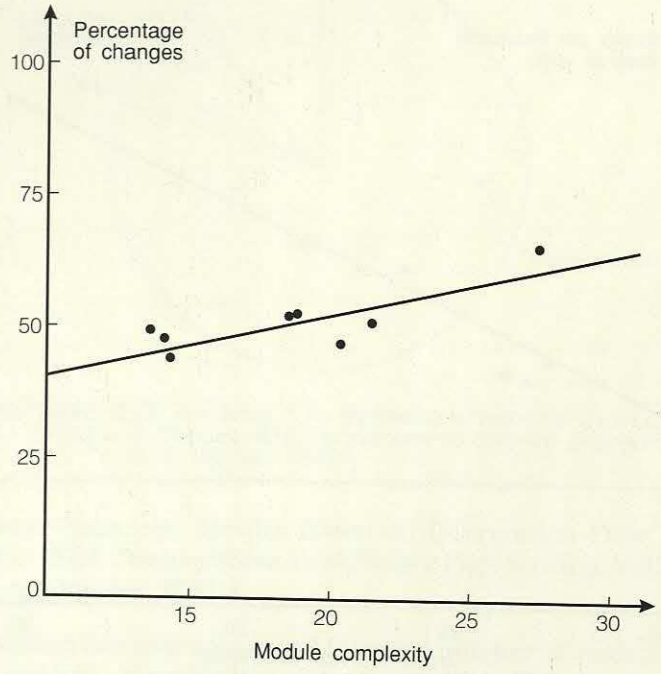
Measures of design complexity show considerable promise for the future but at their present state of development, they are expensive to collect and they have only limited application. Fan-in and fan-out, for example, can be calculated only where a modular approach to development is used. This usually implies the use of third-generation languages, as the modular approach is less common with fourth-generation languages. In general, few measures are available for assessing the design complexity of an application written in a fourth-generation language. Systems departments can, however, gain many of the benefits of using a measure of design complexity by establishing design guidelines that prohibit the construction of systems with modules that would have high fan-in and fan-out values.

MEASURING CODE COMPLEXITY

Measures of the complexity of an application are easier to collect at the coding stage, and are a more reliable indicator of

Figure 4.9 Module complexity can be used to predict software maintainability

The figure shows the correlation between module complexity and the proportion of difficult-to-implement changes made to eight applications.



(Source: Adapted from Card, D N, and Glass, R L. *Measuring software design quality.* London: Prentice-Hall, 1990)

applications quality. The most common measure of code complexity is McCabe's complexity measure. This simple measure is defined as the number of decision statements in a section of code, plus one. Code with a McCabe value higher than 10 seems to have disproportionately more bugs than code with values of less than 10.

The commonest measure of code complexity is McCabe's complexity measure

Many studies have shown that programs with high levels of complexity also have high error rates. One study of the Unix operating system, for example, showed a very high correlation (0.98) between modules with a high number of errors and modules with a high level of complexity. In another study, 47,000 lines of Fortran code were analysed to assess how well McCabe's complexity measure could predict the number of actual changes to modules based on data for a year's error reports. An almost perfect correlation (0.9985) was found between the complexity measure of a program and the number of changes made.

Programs with high levels of code complexity also have high error rates

Another measure that can be used to obtain an assessment of code complexity is the ratio of object code instructions to source code statements. The expectation is that programs using very powerful source-language statements will have a higher ratio than those using simple statements. Experience suggests that applications written in a simple language are more reliable and easier to maintain. Thus, programs with high ratios of object-to-source-code size should be carefully evaluated.

Applications written in a simple language are more reliable and easier to maintain

Measurement of code complexity can be a useful technique for improving the reliability and maintainability of application code. Its use, however, is limited to third-generation procedural languages such as Cobol because there are, as yet, no well established measures of the complexity of code written in fourth-generation languages.

RECOGNISE THE LIMITATIONS OF EACH APPROACH

None of the established approaches to predicting final quality is ideal

None of the three approaches described above for managing a project to deliver the required quality profile is completely satisfactory for PEP members. Our recommendations are:

- Adapt the RADC approach to the development method being used in order to provide a more focused approach to software-quality audits.
- Use Gilb's approach to establish clear quality goals and to create the development plans and application-design features that will achieve these goals for qualities (such as usability) that cannot easily be quantified in other ways.
- Make use of design and code complexity measures where possible to assess and control the qualities of correctness, reliability, efficiency, maintainability, testability, and flexibility.

Analysis of the measures discussed in this chapter is reasonably straightforward while the development process remains unchanged. However, PEP members should bear in mind that advances in system development tools (integrated computer-aided software engineering, in particular) will lead to significant changes in the applications life cycle. They should therefore take great care in planning their software quality measurement programme to ensure that future changes in the development process do not invalidate a valuable set of software measurement data.

Software quality measurement programmes should be able to cope with changes in the development process

We have been concerned in this chapter with describing ways in which project managers can monitor the progress of an application by measuring the quality of the interim products of the development process, and taking action to modify those products where the measures indicate that the final quality is likely to be unsatisfactory. In most organisations, however, software maintenance accounts for about the same amount of time and effort as the development of new systems. The contribution that software quality measurement can make to improving the quality of existing applications and thereby reducing the cost of their maintenance is the subject of the next chapter.

Chapter 5

Measuring the quality of existing applications

So far in this report, we have concentrated on showing how software quality measurements can be used as applications are developed — to improve the development process itself, to control an application as it is developed, and to predict the quality of the final product. Much of the development department's workload, however, is concerned with maintaining existing applications. In this chapter, we show how software quality measurement can be used to monitor and improve the quality of existing operational applications, thereby leading to a reduction in maintenance costs.

Software quality measurement can be used to monitor and improve the quality of existing applications

The quality of existing applications must, of course, be measured in terms of the users' satisfaction with the final systems that are delivered to them. This should not be too onerous a task since a small number of well chosen measures will provide the basis of a comprehensive measurement programme. The data collected in this process should be made widely available, and in the form that most clearly demonstrates the point of a particular analysis. The analyses will be of interest not only to those who collect and analyse the data, but also to development staff, development managers, and user managers. It is in the interests of all those involved to use software quality measurements to detect difficult-to-maintain or unreliable applications and to determine which applications can have their maintainability improved.

Quality measurement data should be made widely available

CREATE A USER-ORIENTED SET OF MEASURES

Once an application has been implemented, it is important to verify that it does, in fact, meet the users' quality requirements; the systems department must be able to demonstrate that it is delivering the quality that was expected. In principle, user satisfaction can be defined (on a scale of 0 to 1) as the quality of the delivered application divided by the quality that the users were expecting.

To do this, systems departments need to present software quality measurements in terms that are relevant to users' perceptions of the quality of the application rather than in terms that indicate the need for support from the systems department. For example:

Software quality measures must be presented in terms that are relevant to users

- The reliability of an application should be measured in terms of failures per hundred hours of operation, rather than in terms of faults per thousand lines of code.
- The flexibility of an application should be specified in terms of the time taken to act on a request for a change, rather than in terms of the total number of changes made.
- Errors should be classified by the impact they have on users rather than by the type of design or coding fault that caused them. Telling users that a particular class of error will cause

an application to be unavailable for use for several hours is much more meaningful than telling them that the problem was caused by a coding error.

User-oriented measures therefore need to be devised

In our research, we were unable to identify a comprehensive set of measures for assessing the quality of existing applications. We have therefore taken each of the 11 RADC quality factors, described in Chapter 3, and devised an appropriate user-oriented measure for it. (The RADC quality factors were originally defined to help manage and predict quality during the development of an application.) In devising the measures, we have been conscious of the need for standardisation and comparability (the two most important measurement selection criteria described in Chapter 3). The measures we suggest are set out in Figure 5.1, overleaf. It lists the basic data items that need to be captured from the application in order to produce the measures. The final 'calculation' column gives the formula for calculating each quality measure from the basic data items. The calculations ensure that the measures are normalised so that they can be used to compare the quality of different applications, where practical.

As in a database application, each of the basic data items must be carefully defined at the lowest level and in detail. For example, it is usually necessary to define the basic unit of staff effort as hours, to allow for the effects of overtime or different working hours in different locations. The relevant part of the draft (and, as yet, unapproved) IEEE standard P-1061/D20 for a software quality metrics methodology (see Figure 5.2, on page 41) provides a useful template for the descriptions of the basic data items.

A small number of well chosen basic measurements should be adequate for most programmes

Since the same basic data items are needed for several of the software quality measures, a comprehensive programme for measuring the quality of existing applications can be based on a small number of well chosen basic measurements. These are listed in Figure 5.3, on page 41.

STORE THE INFORMATION IN AN EASILY ACCESSIBLE FORM

Even in large systems departments, the complexity and volume of the basic data items needed for measuring the quality of existing applications is not large, and many of them will be available in the management information systems already in use in the systems department (see Figure 5.4, on page 42). The data can easily be stored and manipulated on a spreadsheet system. As basic data items are entered into the spreadsheet, the quality measures can be calculated automatically. Staff who collect the information will thus receive instant analyses of it, but the database can also be easily distributed to development staff and managers to carry out their own analyses. Currently, it is not common practice for user managers to receive software measurement information (see Figure 5.5, on page 42), but if they are to make informed judgements, they, too, should receive the analyses.

It is not common practice, at present, for user managers to receive software measurement information

Although the responsibility for data capture, and perhaps first-stage analysis, resides with development staff, the quality-assurance function must take overall responsibility for the management of the data and the production of the overall

Figure 5.1 Most qualities of existing applications can be quantified directly or indirectly

Quality factor	Measure	Basic data items	Calculation
Reliability	Mean time to fail (MTTF)	Hours of use (H) Number of failures (N_1) Application size (lines of code or function points) (S)	MTTF = $N_1 \div (H \times S)$
Flexibility	Effort to implement a change in requirements (F)	Developer hours per change (He) Function points or lines of code changed (Sc)	F = He ÷ Sc
Maintainability	Effort required to diagnose and respond to a fault (M)	Number of faults fixed (N_2) Developer hours for fixing faults (Hf)	M = Hf ÷ N_2
Re-usability	Proportion of application consisting of re-used code (R)	Size of re-used code (Sr) Application size (lines of code or function points) (S)	R = Sr ÷ S
Correctness	Conformance to requirements (C)	Number of defects (D) Application size (lines of code or function points) (S)	C = D ÷ S
Testability	Effort required to test the application fully (Ef)	Number of test cases (T) Application size (lines of code or function points) (S)	Ef = T ÷ S
Efficiency	Online efficiency (E)	Number of transactions (Tr) Computer resources used: — CPU time (R_1) — Disc transfers (R_2) — Network traffic (R_3) — Etc	$E_1 = R_1 \div Tr$ $E_2 = R_2 \div Tr$ $E_3 = R_3 \div Tr$
Usability	Ease of use (Ea)	Number of unfounded fault reports (N_3) Hours of use (H) Number of users (U) Application size (lines of code or function points) (S)	Ea = $N_3 \div (H \times U \times S)$
Integrity	Access-control quality	Subjective ratings for: — User-access control — Database-access control — Memory protection — Recording and reporting access violations — Immediate notification of access violations	None (use the subjective ratings)
Interoperability	Effort required to link the application to another (EI)	Effort in hours to link applications (HI) Application size (lines of code or function points) (S)	EI = HI ÷ S
Portability	Effort required to transfer the application to another environment (Ee)	Effort in hours (Hc) Application size (lines of code or function points) (S)	Ee = Hc ÷ S

analyses. The analyses can be presented in a variety of ways, illustrated in Figure 5.6, on page 43:

- *Trend line charts* show the variation in a software measure (such as reliability) over time. They are useful for determining if there is a trend or pattern in the occurrence of a specific type of error.

- *Histograms* show frequency of data by various categories and classifications. They are used in PEP assessments, for example,

Figure 5.2 The description of data items in the draft IEEE standard for a software quality metrics methodology can be used as the basis for describing the basic data items

Note that the draft IEEE standard from which this is an extract has not yet been approved.

Item	Description
Name	Name of the data item.
Metrics	The metrics that are associated with the data item.
Definition	Unambiguous description of the data item.
Source	Location where data originates.
Collector	Entity responsible for collecting the data.
Timing	Time(s) in life cycle at which data is to be collected. (Some data items are collected more than once.)
Procedures	Methodology used to collect data (e.g., automated or manual).
Storage	Location where data is stored.
Representation	The manner in which data is represented; its precision and format (e.g., Boolean, dimensionless, etc.).
Sample	The percentage of the available data that is to be collected.
Verification	The manner in which the collected data is to be checked for errors.
Alternatives	Methods that may be used to collect the data other than the preferred method.
Integrity	Who is authorised to alter this data item and under what conditions.

Figure 5.3 A programme for measuring the quality of existing applications can be based on a small number of basic data items

Number of application users Hours of application use Number of transactions processed Computer resources used (CPU time, disc transfers, network traffic, . . .)
Number of application failures Number of application defects Number of unfounded user fault reports Number of faults fixed
Developer hours required per fault Developer hours required per change Developer hours required to link the application to another Developer hours required to transfer the application to another environment
Size of application, Size of change } Lines of code or function points
Number of test cases Size of re-used code

to show the distribution of Productivity Indexes (PIs) over a range of projects.

— *Pareto diagrams* are a particular type of histogram that can be used to show errors by type and frequency. They can be

Figure 5.4 Many of the basic data items required for measuring the quality of existing applications will already be available

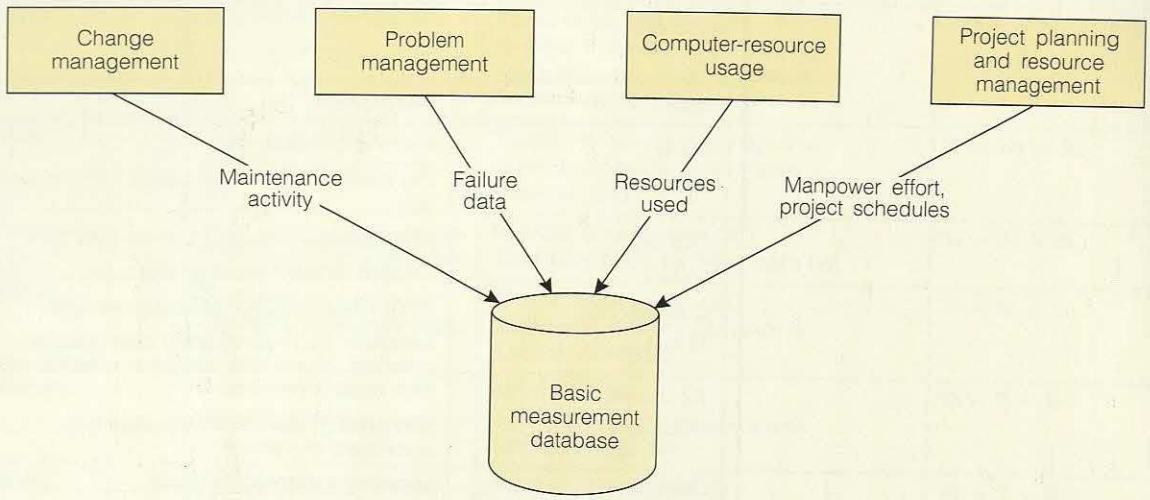
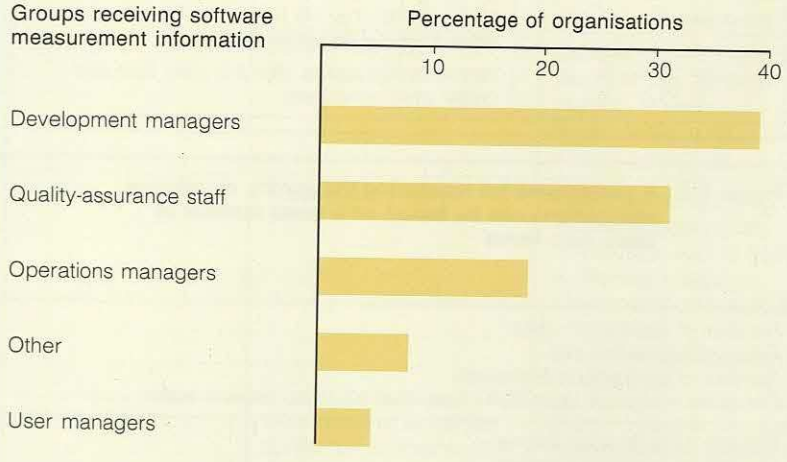


Figure 5.5 At present, software measurement information is sent mainly to development managers and quality-assurance staff

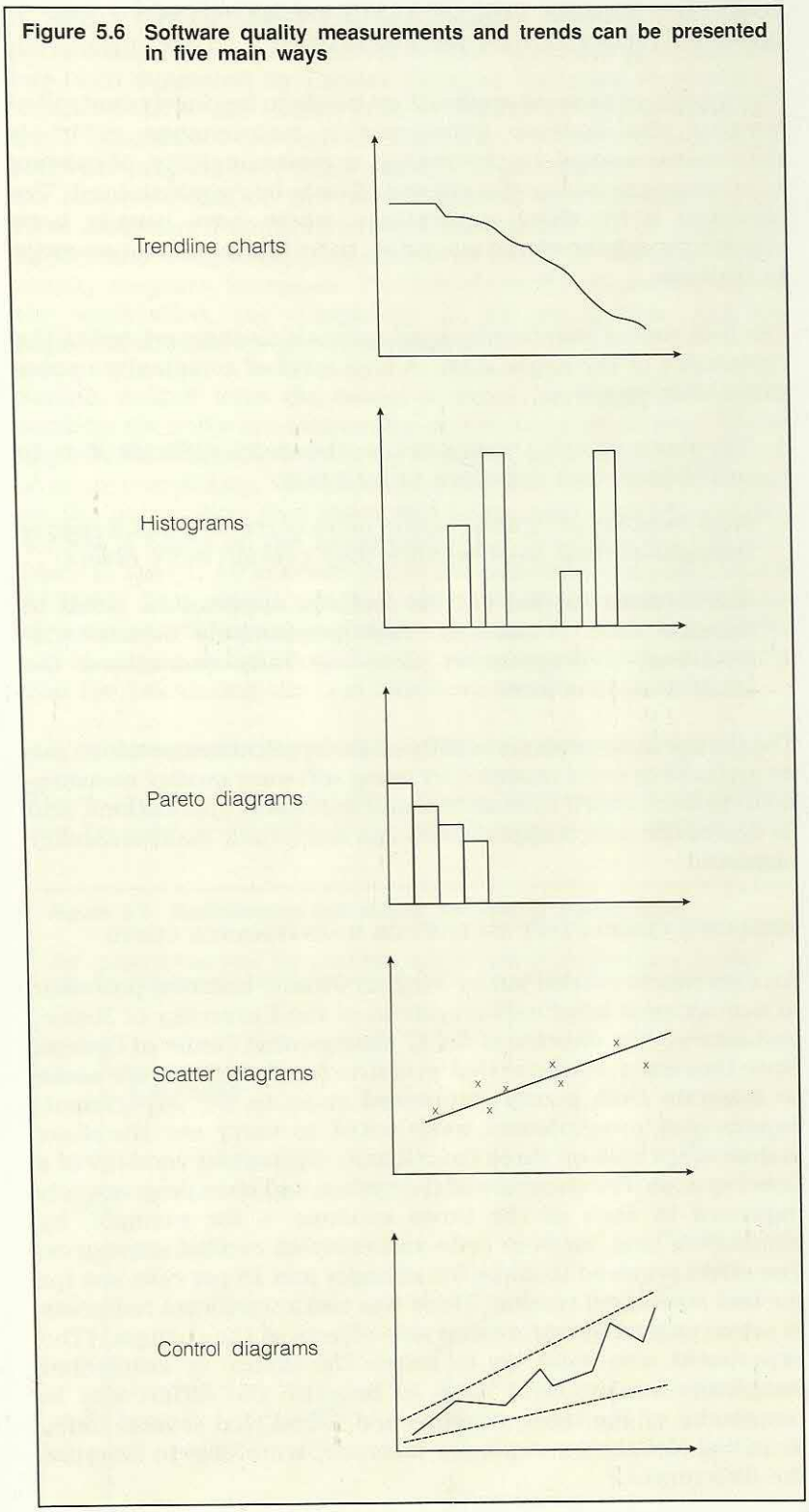


(Source: Survey of PEP members and of members of the Quality Assurance Forum)

used to highlight the few but vital error types that account for the greatest number of actual errors, and the many but trivial error types that account for few errors.

- *Scatter diagrams* show the existence (or lack) of a relationship between two factors. If a straight line is apparent in the plot (as in our example), there is likely to be a relationship between the factors.
- *Control charts* show a software measurement plotted over time within statistical control limits. If the plotted line exceeds

Figure 5.6 Software quality measurements and trends can be presented in five main ways



either of the limits, there is a strong possibility that something is going wrong with the development process. A control chart of error rates helps to determine if a process is either 'in control' (with only random errors occurring) or 'out of control' (with errors occurring more generally).

IDENTIFY WHERE THE QUALITY OF EXISTING APPLICATIONS COULD BE IMPROVED

The quality of existing applications needs to be closely controlled because the systems department's maintenance effort is determined as much by the quality, or maintainability, of existing applications as it is by the volume of code being maintained. The tendency is for older applications, which have usually been repeatedly enhanced and amended, to be harder and more costly to maintain.

Maintenance effort is determined as much by the quality of existing applications as by the volume of code

The high cost of maintaining applications is strongly related to the complexity of the application. A high level of complexity causes three main problems:

- The more complex a system is, the more difficult it is to understand, and therefore to maintain.
- More complex programs require more corrective maintenance throughout their lives because they contain more errors.
- Maintenance carried out on complex applications tends to increase their complexity disproportionately because the maintenance programmer does not fully understand the application structure.

The decline in the maintainability of the applications portfolio can be arrested or even reversed by using software quality measurement to detect hard-to-maintain and unreliable applications, and to determine which applications can have their maintainability improved.

REDUCING COMPLEXITY TO REDUCE MAINTENANCE COSTS

An experiment carried out by Virginia Gibson, assistant professor of management information systems at the University of Maine, and James Senn, director of the IT Management Center at Georgia State University, confirms that well structured programs are easier to maintain than poorly structured ones. In the experiment, experienced programmers were asked to carry out the same maintenance task on three functionally equivalent versions of a Cobol system. The structure of the system had been progressively improved in each of the three versions — for example, by eliminating long jumps in code and complex control sequences. The effort required to make the changes was 18 per cent less for the best-structured version. There was also a significant reduction in errors caused by unforeseen side effects of the changes. (The experiment also went on to assess the ability of automated complexity-measurement tools to measure the differences in complexity of the three versions and found that several tools, including McCabe's complexity measure, were able to measure the differences.)

Well structured programs are easier to maintain than poorly structured ones

The result of this experiment suggests that it makes sense to restructure existing applications to reduce their complexity. Restructuring tools are becoming available to enable this process to be largely automated. A description of restructuring tools can be found in PEP Paper 8, *Managing Software Maintenance*, which was published in November 1988.

A model for predicting the maintenance-cost implications of periodically restructuring an application to reduce its complexity has been suggested by Capers Jones of Software Productivity Research Inc in the United States. The model's parameters and initial conditions were derived from Jones's observations of ageing software projects over many years. The parameters of the model are historical data on the rate of addition and deletion of code from maintained applications and the annual rate at which the application complexity, measured in terms of McCabe's complexity measure, increases. The initial conditions are the size of the application, the complexity of the application, and the expected lifetime of the application.

Periodic restructuring can reduce maintenance costs considerably

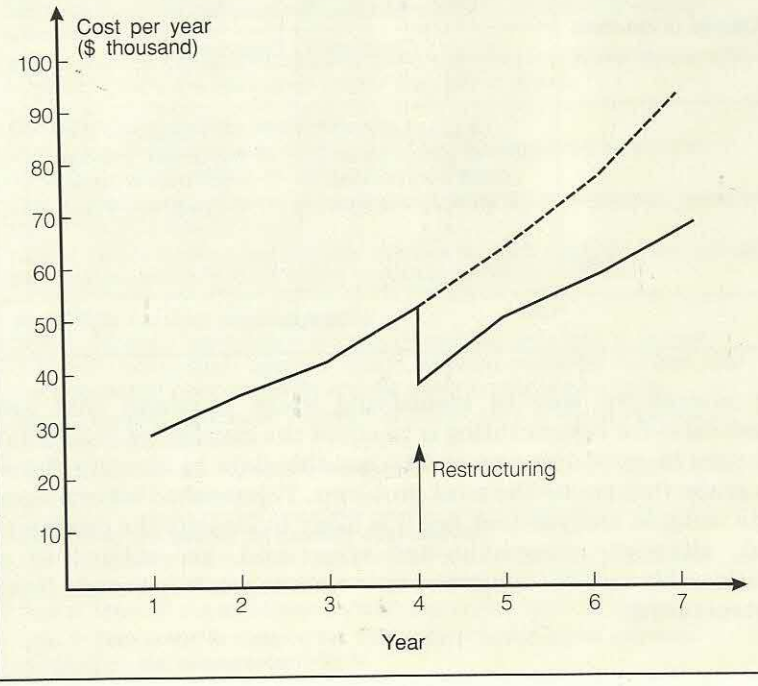
Sample output from the model is shown in Figure 5.7, which predicts the software-maintenance costs over seven years for an application originally consisting of 25,000 lines of code of above-average complexity. Two predictions are shown. The first is based on the assumption that there will be no restructuring, and the second is based on the assumption that restructuring will take place in Year 4. After seven years, the cumulative saving is nearly $70,000. This model predicts only the likely impact that restructuring will have on maintenance costs. However, it is a useful aid in setting initial levels for the maintainability quality factor and for calculating the cost implications of restructuring.

DECIDING WHICH APPLICATIONS NEED TO BE RESTRUCTURED

There are several application quality measurement tools that can be used to identify those programs that will benefit most

Figure 5.7 Restructuring can reduce the cost of maintenance

The graph shows how the predicted annual cost of maintaining a program can be reduced by restructuring it to make it less complex.



from restructuring. Reducing the complexity of these programs will also reduce the cost of maintaining them in the future.

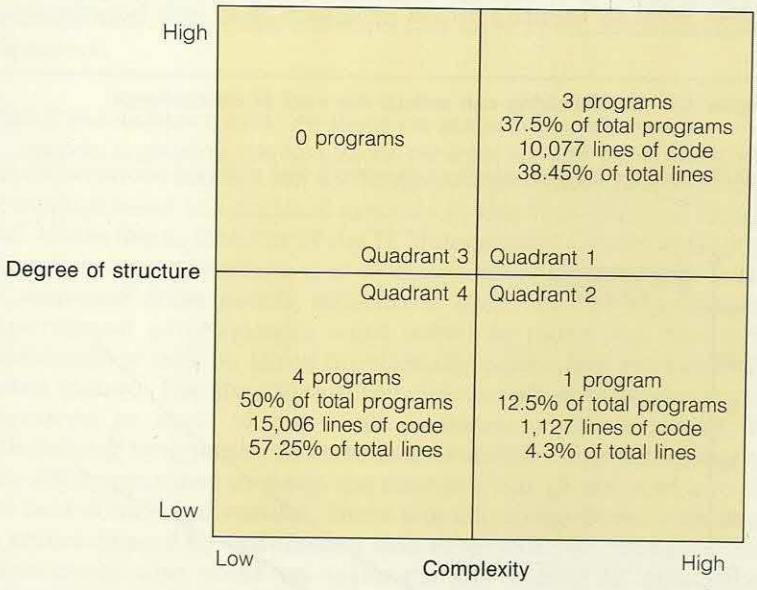
The SNAPSHOT tool from Language Technology, for example, can be used to analyse a set of Cobol programs. This tool produces a comprehensive list of measurements of the programs' structures, complexities, and their conformance with structured-programming rules. The assessments of the structure and complexity are based on McCabe's complexity measure.

The SNAPSHOT tool categorises programs according to their complexity and degree of structure

SNAPSHOT presents the results of the analysis as a series of charts, one of which is shown in Figure 5.8. This chart classifies programs according to their complexity and their degree of structure. It places programs in one of four quadrants and summarises their distribution by providing totals and percentages of the programs in each quadrant. The worst programs (those that are unstructured and highly complex) fall in Quadrant 1. The best programs (those that are well structured and of low complexity) fall in Quadrant 4.

Figure 5.8 The SNAPSHOT tool categorises a set of programs according to their level of complexity and their degree of structure

The mid-point of the x-axis represents the threshold of acceptable complexity; the mid-point of the y-axis represents the threshold of the acceptable degree of structure.



An alternative way of identifying those programs that are candidates for restructuring is to count the number of faults and changes in each program and to use this data to identify those programs that create the most problems. This method takes longer than using an analysis tool, but it is likely to identify the programs that, although reasonably well structured, are subject to a considerable rate of change and that would therefore benefit from restructuring.

Chapter 6

Putting a software quality measurement programme in place

In this paper, we have shown how a measurement programme can be used to achieve practical goals, and how each type of objective can be achieved. Figure 6.1 summarises the actions that systems managers should take to ensure that the software quality measurement programme that is put in place is practical and comprehensive and that it will have the support of both users and systems developers.

Figure 6.1 Action checklist

Improving the development process

Step 1: Identify software quality problem areas in the development process by gaining a full understanding of the area of concern and identifying all likely causes of development problems, for subsequent analysis.

Step 2: Analyse the likely causes of the problem that have been identified, using carefully selected software quality measurements to discover where the major problems lie.

Step 3: Make changes to the process that will solve the problems that have been identified.

Step 4: Use software quality measurements again to check that the changes introduced have had the effect that was intended and that no other side-effects have occurred.

Controlling a development project

Step 1: Establish measures that can be obtained during the application development process that will indicate the 'health' of the project.

Step 2: Develop organisational norms for application development measures from a historical database of measures of previous projects.

Step 3: Collect measures from the application project being managed and compare these measures with the norms established in Step 2 to identify unusual measures.

Step 4: Use the characteristic pattern of unusual measures to identify the action required to bring the application project back under control.

Defining users' quality requirements

Step 1: Understand and set priorities for users' quality requirements from surveys and from analysis of users' requirements.

Step 2: Choose a comprehensive and usable set of quality measures, based on the 11 RADC quality factors.

Step 3: Define a unique set of quality priorities for each application and choose those quality factors that will enable the quality priorities to be met.

Predicting the final product quality

Step 1: Choose a combination of the three methods described in Chapter 4 (quality of conformance, quality of design, empirical measures) that best suits the organisation's circumstances and the users' quality requirements.

Step 2: Develop the chosen approach and fine-tune it to the organisation's development environment.

Step 3: Demonstrate the achievement of the correct quality with user-oriented software quality measures.

Measuring the quality of existing applications

Step 1: Establish measures of maintainability (design and code complexity, number of user changes, application failure rates, and so on).

Step 2: Measure the application portfolio and identify the 'bad apples.'

Step 3: Take action to improve the 'bad apples' based on the problems identified by the measurement results.

Chapter 6 Putting a software quality measurement programme in place

Quality assurance has become an essential part of the software development process, and quality measurement has, in turn, become a critical element of the organisation's quality assurance task. Without reliable measures of the quality of the development process and of the final product, specifying and producing applications of the quality that users require, assessing the effects of changes to the development process, and justifying the costs of a software quality assurance programme will be virtually impossible tasks for the systems development manager.

Bibliography

Books

Card, D N, and Glass, R L. *Measuring software design quality.* London: Prentice-Hall, 1990.

Ehrenberg, A S C. *A primer in data reduction: an introductory statistics textbook.* Chichester: John Wiley, 1986.

Grady, R B, and Caswell, D L. *Software metrics: establishing a company-wide program.* London: Prentice-Hall, 1987.

Ishikawa, K. *What is total quality control? the Japanese way.* London: Prentice-Hall, 1985.

Vincent, J, Waters, A, and Sinclair, J. *Software quality assurance: volume 1: practice and implementation.* London: Prentice-Hall, 1988.

Vincent, J, Waters, A, and Sinclair, J. *Software quality assurance: volume 2: a program guide.* London: Prentice-Hall, 1988.

Standards

Standard dictionary of measures to produce reliable software. IEEE 982.1. New York: Institute of Electrical and Electronics Engineers, 1988.

Standard for software quality metrics methodology. IEEE 1061. New York: Institute of Electrical and Electronics Engineers, 1989.

Butler Cox

Butler Cox is an independent, international consulting company specialising in areas relating to information technology.

The company offers a unique blend of high-level commercial perspective and in-depth technical expertise: a capability which in recent years has been put to the service of many of the world's largest and most successful organisations.

The services provided include:

Consulting for Users
Guiding and giving practical support to organisations trying to exploit technology effectively and sensibly.

Consulting for Suppliers
Guiding suppliers towards market opportunities and their exploitation.

The Butler Cox Foundation
Keeping major organisations abreast of developments and their implications.

Multiclient Studies
Surveying markets, their driving forces and potential development.

Education
Through the Cranfield IT Institute (CITI), educating systems specialists, IT managers, line managers, and professionals to understand more fully how to apply and use today's technology.

P E P

The Butler Cox Productivity Enhancement Programme (PEP) is a participative service whose goal is to improve productivity in application systems development.

It provides practical help to systems development managers and identifies the specific problems that prevent them from using their development resources effectively. At the same time, the programme keeps these managers abreast of the latest thinking and experience of experts and practitioners in the field.

The programme consists of individual guidance for each subscriber in the form of a productivity assessment, and also publications and forum meetings common to all subscribers.

Productivity Assessment
Each subscribing organisation receives a confidential management assessment of its systems development productivity. The assessment is based on a comparison of key development data from selected subscriber projects against a large comprehensive database. It is presented in a detailed report and subscribers are briefed at a meeting with Butler Cox specialists.

Meetings
Each quarterly PEP forum meeting focuses on the issues highlighted in the previous PEP Paper. The meetings give participants the opportunity to discuss the topic in detail and to exchange views with managers from other member organisations.

PEP Papers
Four PEP Papers are produced each year. They concentrate on specific aspects of system development productivity and offer practical advice based on recent research and experience. The topics are selected to reflect the concerns of the members while maintaining a balance between management and technical issues.

Previous PEP Papers
1 Managing User Involvement in Systems Development
2 Computer-Aided Software Engineering (CASE)
3 Planning and Managing Systems Development
4 Requirements Definition: The Key to System Development Productivity
5 Managing Productivity in Systems Development
6 Managing Contemporary System Development Methods
7 Influence on Productivity of Staff Personality and Team Working
8 Managing Software Maintenance
9 Quality Assurance in Systems Development
10 Making Effective Use of Modern Development Tools
11 Organising the Systems Development Department
12 Trends in Systems Development Among PEP Members
13 Software Testing
14 Software Quality Measurement

Forthcoming PEP Papers
Selecting Application Packages
Project Estimating and Control

Butler Cox plc
Butler Cox House, 12 Bloomsbury Square,
London WC1A 2LL, England
☎ (071) 831 0101, Telex 8813717 BUTCOX G
Fax (071) 831 6250

Belgium and the Netherlands
Butler Cox Benelux bv
Prins Hendriklaan 52,
1075 BE Amsterdam, The Netherlands
☎ (020) 755 111, Fax (020) 755 331

France
Butler Cox SARL
Tour Akzo, 164 Rue Ambroise Croizat,
93204 St Denis-Cédex 1, France
☎ (1) 48.20.61.64, Télécopieur (1) 48.20.72.58

Germany (FR), Austria, and Switzerland
Butler Cox GmbH
Richard-Wagner-Str. 13, 8000 München 2, West Germany
☎ (089) 5 23 40 01, Fax (089) 5 23 35 15

Australia and New Zealand
Mr J Cooper
Butler Cox Foundation
Level 10, 70 Pitt Street, Sydney, NSW 2000, Australia
☎ (02) 223 6922, Fax (02) 223 6997

Finland
TT-Innovation Oy
Meritullinkatu 33, SF-00170 Helsinki, Finland
☎ (90) 135 1533, Fax (90) 135 2985

Ireland
SD Consulting
72 Merrion Square, Dublin 2, Ireland
☎ (01) 766088/762501, Telex 31077 EI,
Fax (01) 767945

Italy
RSO Futura Srl
Via Leopardi 1, 20123 Milano, Italy
☎ (02) 720 00 583, Fax (02) 806 800

Scandinavia
Butler Cox Foundation Scandinavia AB
Jungfrudansen 21, Box 4040, 171 04 Solna, Sweden
☎ (08) 730 03 00, Fax (08) 730 15 67

Spain and Portugal
T Network SA
Núñez Morgado 3-6°b, 28036 Madrid, Spain
☎ (91) 733 9866, Fax (91) 733 9910