Software Testing

BUTLER COX
P,E,P

PEP Paper 13, February 1990

BUTLER COX

P,E,P

Software Testing

PEP Paper 13, February 1990
by Nigel Saker

Nigel Saker Nigel Saker is a senior consultant with Butler Cox in London, where he specialises in project management and the specification of user requirements. He has 20 years of experience in software design and management.

During his time with Butler Cox, he has carried out several PEP assessments and conducted research for the Butler Cox Foundation Position Paper, *Legal Protection for Computer Systems*. He has also been involved in a large consulting assignment for a major US bank, developing specifications for the market-data delivery system for its new dealing room.

Prior to joining Butler Cox, Nigel Saker was a project manager with Aregon International, responsible for the design and installation of dealing-room systems in five major banks in London, New York, and Oslo. Earlier, he spent six years with Logica, advising on the selection of equipment, designing user interfaces for highly interactive systems, and managing turnkey systems development and installation. His early career was with the Meteorological Office.

Nigel Saker has an MA in mathematics from Cambridge University and an MSc in fluid dynamics from the University of Sussex.

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 Testing

PEP Paper 13, February 1990
by Nigel Saker

Contents

Chapter 1

Adopt a more analytical approach to software testing

Testing is a neglected part of the software development life cycle

Testing is an integral part of the software development life cycle, but according to our survey of PEP members, only about 20 per cent of organisations know how much software testing costs. Those who do measure costs report that testing accounts for between 25 and 60 per cent of their total development costs. In general, however, testing is a neglected part of the software development life cycle, poorly controlled, poorly managed, and characterised by limited investment in the techniques and tools that could help to provide a more efficient test environment.

The absence of a common approach to testing and the lack of control over testing, evident in many organisations, may be explained by three main factors. First, the real purpose of software testing is not clearly understood. Second, there is widespread confusion about the scope of system testing. Third, there is a lack of awareness about how useful the results of testing can be for project control.

Testing should be an integral part of the development cycle

In spite of the problems that large numbers of organisations have with managing software testing, we believe that it is possible for systems development managers to make significant improvements in the effectiveness of their testing procedures, which will provide them with a valuable tool for project control. Unfortunately, the traditional software development life-cycle model places too little emphasis on testing, and too much emphasis on the production process. This means either that testing is done less thoroughly than it should be, at the end of a project, in order to meet deadlines, or that the completion of the project is delayed because the system tests have not been properly planned in advance. The former means that the risk of introducing the system into live operation is unknown. The latter means that the benefits to be derived from introducing the system are not realised as soon as they might be. A different view of the development process is required, where testing can be carried out to a specified level, and where it can be planned as an integral part of the development cycle. This report provides guidance to PEP members on how such a process might be put into effect.

THE REAL PURPOSE OF SOFTWARE TESTING IS NOT CLEARLY UNDERSTOOD

No-one questions that software should be tested, but there is surprisingly little agreement about the reasons for doing so. When software engineering was in its infancy, it was generally supposed that the purpose of testing was to show that a system worked as its designers intended. It has since been recognised that it is impossible to prove the correctness of any but the most trivial program.

More recently, it has been argued that the purpose of testing software is to find errors, and by correcting them, to make the system more reliable. This reason is commonly given as justification for the time and effort involved in testing a system. Finding errors is certainly a useful by-product of the testing process, but if the detection of errors is viewed as the sole purpose of testing, staff will not usually be motivated to test effectively, and management will lose an important opportunity to monitor progress and control the project.

There is little agreement about the reasons for testing software

The view that is now held by the leading practitioners of software testing, and the one to which we adhere in this report, is that the purpose of testing is to provide management with information about the quality of the system being developed. It is the responsibility of management to decide what information, if any, it wants from the testing process, and to set the level of testing accordingly.

THERE IS UNCERTAINTY ABOUT THE SCOPE OF SOFTWARE TESTING

Testing should not be viewed as an activity that painstakingly turns a poorly designed system into a passable one. Nor should it be confused with debugging, which is the process carried out by programmers to diagnose and correct errors. Testing is concerned with measuring how closely a system conforms to its specification; these measurements enable managers to assess the risk of introducing a system into live operation.

At one end of the spectrum, no software testing may take place. Forty per cent of PEP members carry out no formal testing. This may be a legitimate choice in at least three situations:

It may be legitimate not to test software

- The system must be operational on a certain date, regardless of how well it has been tested.
- The system is not critical, and failures during operation will have a minimal impact on costs.
- The estimated cost of testing is greater than the benefit that can be derived from using the test results.

On the other hand, an absence of formal testing may simply be an oversight — because no-one in the organisation realises that it requires management action to get formal testing introduced. Or it may be that other informal criteria, such as "when the users stop complaining", "when the deadline arrives", or "when the system is standing up well enough", serve as the basis for deciding that a system is ready to be introduced into live operation. Unless a conscious and justifiable decision has been made *not* to carry out system testing, management is depriving itself of information that can be used to assess the risk to the business of introducing the system.

At the other end of the spectrum, where software directly affects safety, such as the flight-control system of an aircraft, management needs to be confident that the software is at least as reliable as any other critical component. In such cases, the system should be subjected to very extensive testing procedures. There is no consensus within the industry, however, about the

Where software directly affects safety, extensive testing is essential

level of testing required to achieve a given level of reliability, and the extent of testing, in practice, is limited by commercial considerations.

There must be a balance between the cost of testing and the level of confidence required in the system

In between these two extremes lie the majority of applications written by PEP members. Some testing is required, but for most applications, a lower level than that required for safety-critical systems will be sufficient. The problem is to decide how much testing is necessary to achieve, for example, a mean time between failures of at least one month. The solution requires a balance to be struck between the cost of testing and the level of confidence that is required in the performance of the system.

Statistical techniques for deriving objective measures of reliability from the number of errors discovered during testing are currently the subject of much academic debate. Several techniques have been developed, but their application requires an advanced knowledge of statistical theory. It will probably be a few years before any practical measures become available for general use. In the meantime, we can recommend only that organisations collect statistics of their own experiences of testing, and analyse these to produce their own guidelines. PEP Paper 14, *Software Quality Measurement*, will provide some guidance on this aspect of developing software.

THERE IS A LACK OF AWARENESS ABOUT THE RELEVANCE OF TESTING RESULTS TO PROJECT CONTROL

It is sometimes difficult to measure the progress of a project

The successful control of a project requires information about the progress of the development. This information must be objective, accurate, and updated on a regular basis (typically, weekly or monthly). It is easy to measure progress objectively at some stages of a project — for example, the design of a subsystem results in a document that can be checked for completeness. At other stages of the development, however, progress is much more difficult to measure objectively. The first time this problem occurs tends to be at the coding stage. A programmer announcing that he has coded a module is not an accurate or objective measure that a standard unit of work has been completed. Some programmers prefer to work by coding very quickly, and spend a long time debugging; others work by coding very carefully and slowly, but produce code that requires very little debugging.

The use of formal testing techniques can avoid these problems in the following ways:

- If inspections or walkthroughs, discussed in Chapter 3, are used to review code before the programmer attempts to run it, a uniform standard for assessing the completeness of coding will be used for all programs.
- If coverage analysers, also discussed in Chapter 3, are used to assess the percentage of code that has been executed by module tests, an objective measure is obtained of test progress, and definite criteria can be set for deciding if testing is complete.
- If system tests are carried out according to a definite plan, the progress of the tests can be measured by analysing successful

and unsuccessful test completions, and rates of error detection, as described in Chapter 4.

At all stages in systems development, therefore, the use of formal testing techniques, which give measurable outputs, either in terms of percentage completion, or success/fail status, will provide managers with accurate data about the progress of a project. This data will be much more reliable than that provided by the traditional technique of asking programmers and designers for their assessments of how far the work has progressed.

Formal testing techniques will provide managers with accurate data about the progress of a project

IT IS IN THE INTERESTS OF MANAGERS TO INSTITUTE FORMAL SOFTWARE TESTING

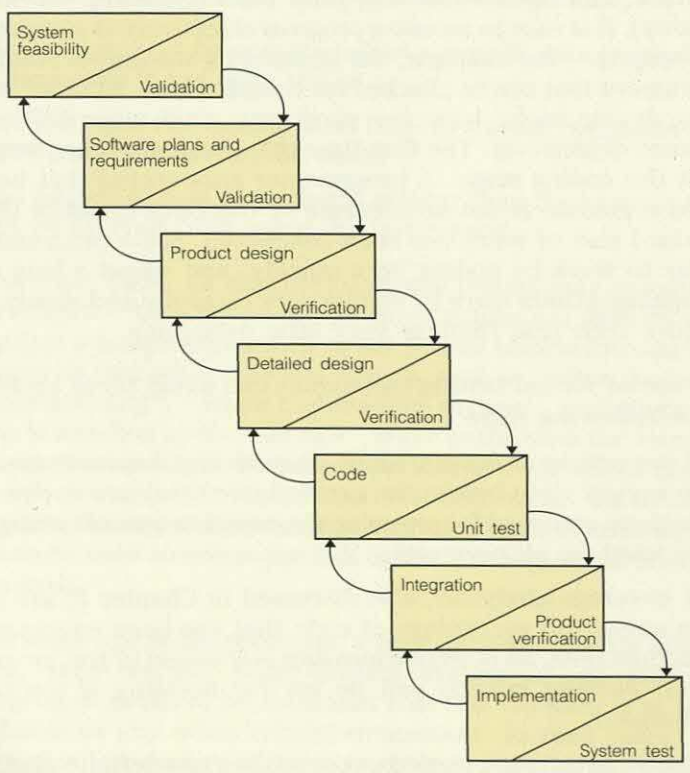
As a first step in improving their development procedures, systems development managers should consider what, if any, their policy is towards testing. They will then be in a position to choose an appropriate approach to software testing.

EXISTING LIFE-CYCLE MODELS FOR SOFTWARE PRODUCTION ARE INADEQUATE

There are well established and widely used life-cycle models for the software-development process, the best known of which, first presented in 1970, is the so-called 'waterfall model', illustrated in Figure 1.1. The main feature of this model is that development proceeds through a series of well defined phases. In an ideal development, each phase is verified and proved error-free

In the traditional life-cycle model, testing is viewed as a secondary activity

Figure 1.1 In the traditional 'waterfall' model of the software-development life cycle, testing is viewed as a secondary activity



before the developers proceed to the next. In practice, some iteration is required when errors introduced in one phase are not detected until a later phase; this iteration process is represented in Figure 1.1 by the upward arrows.

The shortcomings of the approach implied by the waterfall model have become apparent in recent years. The most significant are that testing is viewed as a secondary activity, added on to the end of each phase, and that system testing is not planned until the final development phase.

TESTING THROUGHOUT THE DEVELOPMENT LIFE CYCLE WILL RESULT IN BETTER PROJECT CONTROL

There are two ways of shortening the development timescale

The development timescale can be shortened in two ways. The first is by improving the efficiency of carrying out each phase of the development cycle (that is, completing it with less effort, and hence, in a shorter time); many members are already doing this by using tools such as fourth-generation languages and code generators. The second is by a better scheduling of activities; to achieve this, we recommend the use of an alternative life-cycle model in which there is greater emphasis on testing.

This alternative model is illustrated in Figure 1.2, and shows development occurring in three main parallel streams of activities. In each development stream, the first objective is to produce specifications. The second is to specify what to test. The third is to develop the test environment. Only then are the components assembled ready for testing. Testing is thus carried out at the end of each development stream, and measures different aspects of the development in each stream, as described in Figure 1.3, overleaf. The primary focus is on the testing activity rather than on the production activity, and the outcome of each stream of activity is both a product and a measurement of its quality.

The benefits of this modified approach are four-fold:

Figure 1.2 In the modified software-development life cycle, the emphasis is on testing rather than production

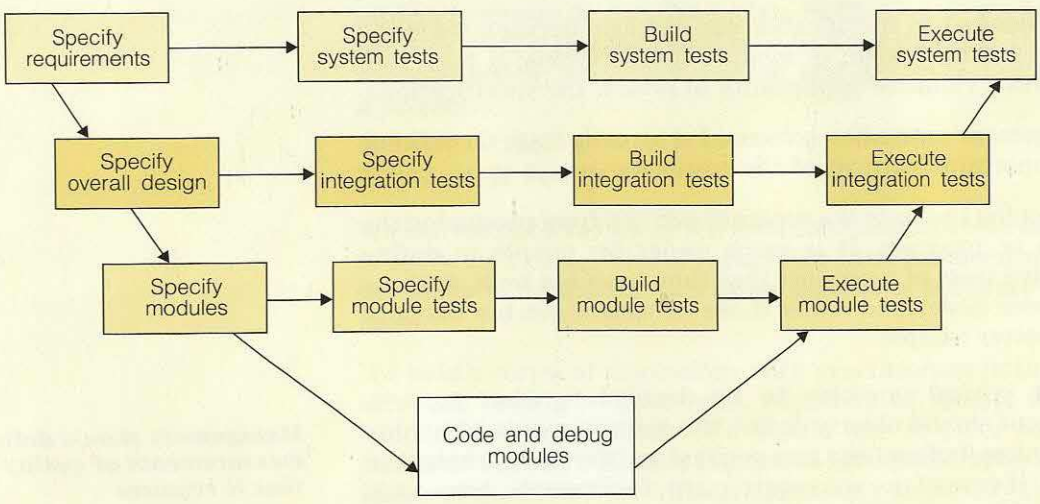


Figure 1.3 In the modified software-development life cycle, testing is carried out at the end of each development stream

Module tests

A program module is the smallest testable component of a system. Its specification comprises a definition of its input data, its output data, and the processes for transforming one into the other. The purposes of module testing are:

- To verify that the module conforms to specified standards.
- To verify that measures of the module's characteristics, such as complexity, are within specified ranges.
- To verify that the module performs its specified functions when executed with a representative sample of input data.
- To verify that each line of code and each of the possible branches have been successfully executed at least once.

Integration tests

Integration tests are designed to measure the behaviour of combinations of modules. They are of two types:

- Verifying the consistency of data definitions that are passed between the modules. This applies both to data that is passed directly, and to data that is passed via a database or shared memory.
- Verifying that all calling paths through the combinations of modules are exercised.

System tests

System tests are designed to measure the behaviour of the total system. This includes tests for some or all of the following features:

- The functionality required by the users.
- The ability to start the system.
- The ability to change the hardware configuration of the system. This particularly applies where there are back-up processors or peripherals that can be substituted in various combinations in the event of failures.
- The ability to restart the system and to recover lost transactions following a failure.
- Performance characteristics, such as response times, delays, and throughput.
- The behaviour of the system when loaded to the limits of its resources.
- The ability to prevent unauthorised users from gaining access to the system.

- By developing test specifications and the test environment concurrently with lower-level specifications or program code, the overall development time is shortened.
- Developing a test specification can highlight deficiencies in the requirements, design, or module specifications; it therefore provides a valuable opportunity to review the specifications.
- Management's attention is focused at an early stage on defining the important features of the system.
- Developing the tests is a separate activity from producing the design or program. It is much easier for people to define objective tests of a product that they have not built, and the test cases developed under these circumstances are likely to be a better sample.

Management should define the measurements of quality that it requires

The most critical question to be decided is what to test. Management should clearly define the measurements of quality that it requires, before tests are specified and the test environment is created. If ease of use is a requirement, for example, tests could be designed to measure how long it takes to input a transaction, how quickly the system can be learnt, and how many mistakes

are made; knowing that these aspects will be tested, the system designers will concentrate on the user interface. If accuracy of data is stated as an important requirement, the activity of specifying the tests will highlight whether all the data must have a high degree of accuracy, or whether some is less critical. Figure 1.2 shows that the decision on what to test — the requirements, design, or module-test specifications — can be taken as soon as the specifications at the beginning of each development stream are complete.

PURPOSE AND STRUCTURE OF THE PAPER

Our research reveals that there is very little agreement about how current software testing procedures might be improved to best effect. The purpose of this paper is therefore to offer guidance on how software testing might be more cost-effectively applied. It is not necessarily our objective to encourage members to carry out more testing; rather, it is to encourage members to undertake an appropriate level of testing for each system, and to test more efficiently. We make no attempt to cover the more technical aspects of testing, which are covered in several useful books on the subject, listed in the bibliography.

We believe that the immediate need for most organisations is to set up the right environment for software testing. The alternative life-cycle model suggests that there may be more effective ways of organising the staff involved in systems development, to reflect the greater emphasis on testing rather than software production. We examine the alternatives in Chapter 2.

In Chapter 3, we discuss the methods, techniques, and tools that are available to managers responsible for the testing aspects of systems development. While in many cases, they are not as well developed as those that help with other aspects of the development process, some are already in wide commercial use and are producing benefits for their users. Significant developments are likely in the future.

It is essential for managers to measure the progress of the testing process so that they can re-allocate resources promptly if the project begins to deviate from plan. In Chapter 4, we provide an analysis of the methods available to help them do this, in the module-testing, integration-testing, and system-testing phases of a project.

RESEARCH SOURCES

We carried out a review of the published literature on the subject of software testing. This revealed that little has been written on the subject, compared with other aspects of software development.

We held a series of discussions with practitioners in the field of software testing — academics, suppliers of testing tools, and commercial organisations. We should like to offer our special thanks to Professor Michael Hennell of Liverpool University, Peter Mellor of the Centre for Software Reliability at the City University, and Mike Bickers and Richard West of the Central Computer and Telecommunications Agency.

We also conducted telephone interviews with 17 PEP members. These interviews proved very valuable in highlighting the difficulties that they encountered in testing software.

Chapter 2

Organise the systems department to support testing

Most systems departments distinguish between system testing, and module and integration testing

In Chapter 1, we discussed the merits of a modified approach to the development of software, in which the conventional life-cycle model is broken down into three streams of activities that take place concurrently, and where the emphasis is on testing rather than on production. Clearly, adopting such an approach will have implications for the organisation of the systems department. In this chapter, we consider how the development teams might best be organised to improve the effectiveness of testing. For the purpose of organising testing, most systems departments distinguish between system testing, which is concerned with the functionality of a system as a whole, and module and integration testing, which is concerned with testing the behaviour of the components of a system.

We examined data collected from PEP assessments to see if there were any clear indications that particular organisation structures were producing more reliable software. The sample of data is probably too small to be significant, but any of the organisation structures for software testing described in this chapter appear to be capable of producing software of both above-average and below-average reliability. The indications are therefore that an organisation will not be able to improve the reliability of its software merely by changing the way it organises software testing. The organisation for software testing needs to be considered within the wider context of a systems department's approach to software testing as a whole.

Staff must understand the purpose of testing software

The full benefits of structuring the department and project teams to improve the effectiveness of testing will, of course, be realised only if the staff themselves fully understand the purpose of testing their software. Very little formal training on testing is given on computer training courses. Much of the training within the industry is given 'on the job', or on courses for specific skills such as programming languages or design methodologies, none of which addresses testing as a major topic. A programmer or analyst is unlikely to learn how to test effectively unless he works in an organisation that understands the nature of software testing. The limited number of courses on software testing available in the United Kingdom are listed in Figure 2.1, overleaf.

ALLOCATE RESPONSIBILITY FOR INTEGRATION AND MODULE TESTING TO AN INDIVIDUAL OR A SMALL TEAM

Knowledge of the detailed system and program designs is required to develop integration and module tests. It would therefore be very expensive to set up independent integration and module test teams, and none of the PEP members whom we interviewed had done so. The alternative life cycle depicted in Figure 1.2 does,

Figure 2.1 There is a limited number of courses on software testing available in the United Kingdom

Course provider	Nature of course
Open University Contract Training Unit Milton Keynes MK7 6AA	One-day courses on software testing at various locations
QCC 4 Tyrone Road Thorpe Bay Essex SS1 3HF	One-day seminar on the management of testing Three-day workshop on quality review and testing Three-day workshop on acceptance testing Three-day course on auditor's approach to testing
National Computing Centre Oxford Road Manchester M1 7ED	Three-day course on software verification, validation, and testing Regular one-day events, including briefings and demonstrations of tools
Frost & Sullivan 4 Grosvenor Gardens London SW1W 0DH	Various two- or three-day seminars on specific aspects of testing
Learning Tree International Trafalgar House Hammersmith International Centre London W6 8DN	Four-day courses on software quality assurance and testing

however, suggest that the system designers should design the integration tests, and that the specifiers of program modules should design the module tests. The interviews indicated that few PEP members distinguished clearly between module and integration tests, possibly because both activities are the responsibility of the project team, and cannot involve the users.

We identified three main team structures for integration and module testing among PEP members:

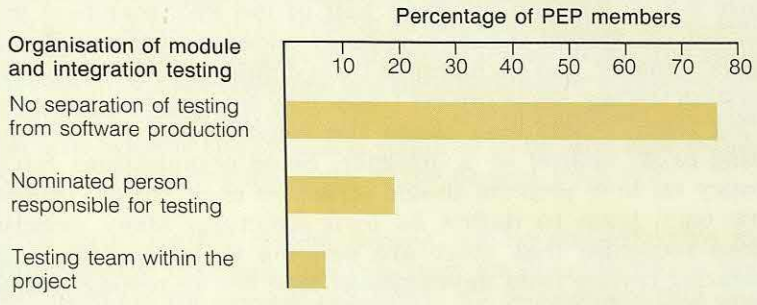
- Each person specifies and executes tests on his own work.
- A nominated person within the team is responsible for ensuring that all tests are carried out to specified standards.
- A distinct team, working under the control of the project manager, is responsible for testing.

Most PEP members do not separate testing from software production

The first of these is the most common among the PEP members we interviewed; three-quarters of them did not attempt to separate testing from production within the development team. The results of the survey are illustrated in Figure 2.2. It is interesting to note that the four members who separated the testing and development functions within the project team included the three who had also established a separate system test department.

The main problem with allowing individual programmers to test their own work is the inconsistency in quality that is likely to result. Some programmers are undoubtedly good at testing their

Figure 2.2 Three-quarters of PEP members make no attempt to separate module and integration testing from software production



(Source: Butler Cox survey of PEP members)

own modules; others, possibly because of inexperience or lack of training, perform virtually no systematic testing. Since a poorly tested module in a critical part of a system can cause considerable delays and expense during system testing, it is not cost-effective to allow uncontrolled individual module testing.

Module testing should usually be done by the programmer, with the appropriate tools

Module testing is difficult to do well. It can be very tedious for a programmer to check that each line of code and all true and false results of decision statements have been tested by a sample of test cases. It is equally, if not more, difficult for a programmer who did not write the code to carry out these tests. It is no doubt for this reason that module testing tends to be done by the programmer who wrote the code, and it is probably not effective in terms of cost or staff morale to introduce an independent module-testing team. However, the use of dynamic-analysis tools (which are discussed on page 23) can remove most of the tedium from module testing, and also provide management with a printed record of the extent of the tests. At the module level, it therefore seems practical to leave the responsibility for testing with the programmer, but to provide the tools that make the job easier and that give management greater project control.

We also recommend that a single team member, or on larger projects, a small team, should be responsible for ensuring that module and integration testing is carried out to specified standards, even if the actual testing is carried out by the programmers themselves. At least one of the designers of the system should be part of this team. Independent testing within a project team does not impose additional costs on a project. In fact, total development costs should fall, since more reliable modules are likely to be produced, leading to a reduction in the cost of rework during system testing and live operation.

CONSIDER THE MERITS OF A SEPARATE SYSTEM TESTING DEPARTMENT

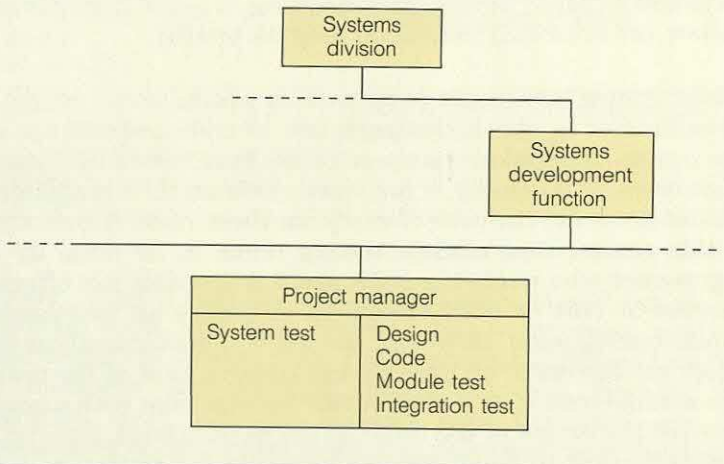
PEP members take a variety of approaches to organising the system testing function. From the 17 responses to our telephone interviews, we identified three main organisation structures.

THE PROJECT TEAM IS RESPONSIBLE FOR SYSTEM TESTING

The most common organisation structure is for the project team to be responsible for system testing. This structure, illustrated in Figure 2.3, is adopted by about half of the PEP members we interviewed. System testing is entirely under the control of the project manager, and each project team defines its own approach to system testing. Some project teams may set up a small system testing team; others may assign the responsibility for system testing to an analyst or a designer. Some organisations have a policy on how projects should structure their teams; others allow each team to define its own structure. Many organisations recognise that there are benefits to be gained from separating testing from development, and set up testing teams within the project team. One organisation ensured that the two activities remained separate by allocating a different computer for testing.

The most common case is for system testing to be controlled by the project manager

Figure 2.3 Organisation structure: the project team is responsible for system testing



Another organisation used the quality assurance department as an independent authority to carry out random tests on the software during the main-build phase. The quality assurance department can play a major role in defining and monitoring how software should be tested, but it is unlikely to have the resources to become closely involved in the design of all the systems under development. As a testing technique, random tests are unlikely to provide a useful measurement of each system's quality, and as a means of finding errors, they should not be used as an alternative to a properly defined series of tests.

Random tests are unlikely to be useful

The main benefits of placing full control of system testing with the project team are reduced costs and ease of management. In the short term, it is cheaper to allow each project team to have full control over its own testing than to incur the additional costs of a separate group of people, who have to understand the users' requirements and liaise with the project team. From the manager's point of view, assigning total responsibility for testing to the project team relieves him of the need to devote any effort to consideration of system testing.

The main disadvantages are also related to cost and management. If an organisation has several similar projects under development at any one time, it should be possible to reduce costs by developing a common testing environment, or by purchasing a set of software testing tools that can be used on all projects. Making system testing the responsibility of individual project teams means that systems development management has no independent measures of the characteristics of a system. Whether this is a problem will depend on how reliable the system is required to be, and how skilled in system testing the members of the project team are.

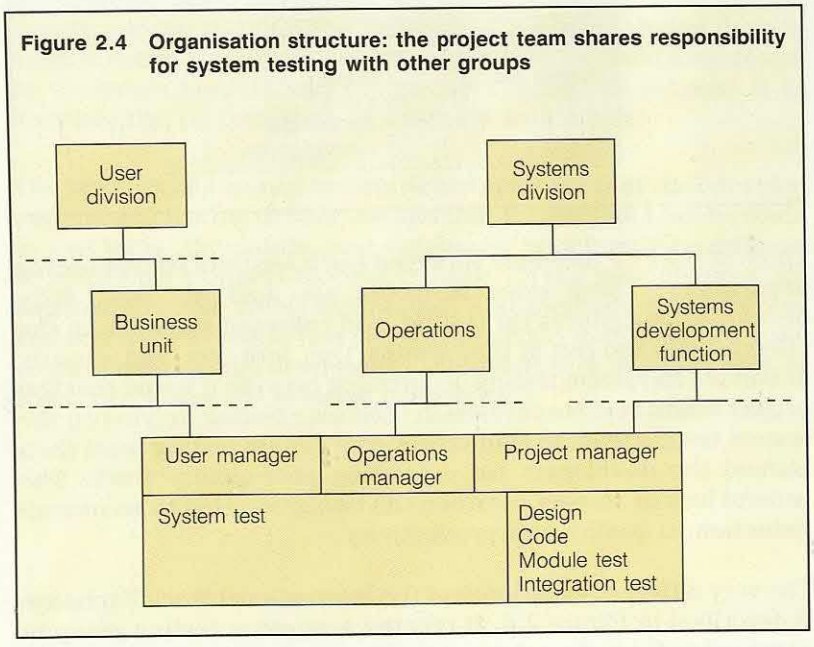
Making the project team responsible for system testing is appropriate where the applications are small and the requirement for reliability is average

This structure for system testing can be cost-effective in an organisation that develops relatively small applications, with a requirement for average reliability. We recommend that organisations choosing to adopt this structure should ensure that one person within the project team is given specific responsibility for system testing, and that this person has expertise in the design of system tests.

THE PROJECT TEAM SHARES RESPONSIBILITY FOR SYSTEM TESTING WITH OTHER GROUPS

In this organisation structure, illustrated in Figure 2.4, the project team provides the technical expertise in testing, but user groups and the operations department define and carry out their own tests. The user groups examine the functionality and usability of the system. The operations group considers such factors as whether the batch run can be completed within the scheduled time. The decision on whether to accept the system is made on the basis of these measurements. This structure has many of the same advantages and disadvantages as the first one. It does, however, allow at least one set of system tests to be carried out by a group that is separate from the development team.

Figure 2.4 Organisation structure: the project team shares responsibility for system testing with other groups



A common problem with systems development is that users are not sufficiently involved, particularly during the requirements

definition phase. This can lead to an excessive number of changes being requested throughout the development. By involving the users in the specification and execution of system tests, they are forced to examine the specifications critically, which should help to ensure that any faults in the specifications are corrected at an early stage, and to reduce the number of subsequent requests for changes.

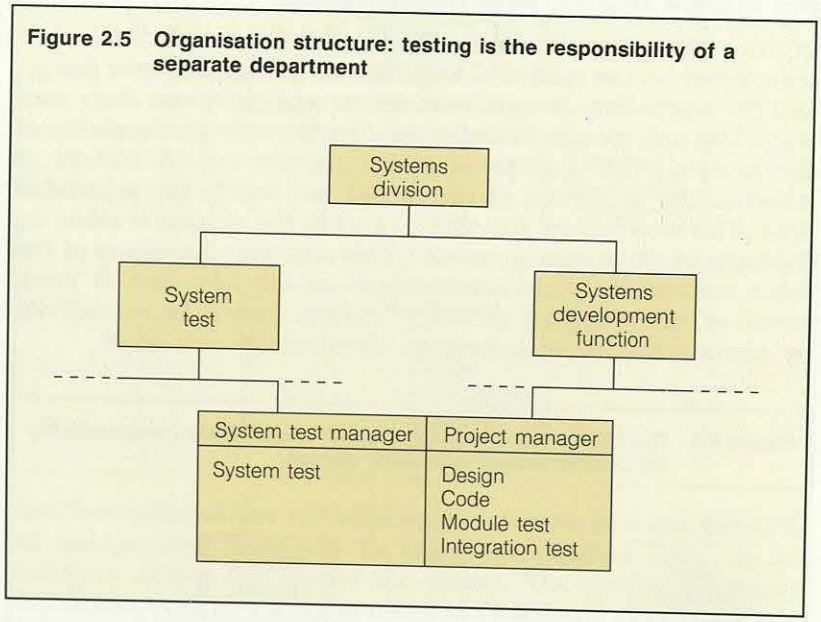
If users are involved in system tests, they will be forced to examine the specifications critically

There is no need for a system tester to know how to design or program software, but testing requires particular skills for which training and experience are necessary. User groups should therefore include at least one specialist adviser, not necessarily full-time, if they are to carry out effective tests.

TESTING IS THE RESPONSIBILITY OF A SEPARATE DEPARTMENT

In this organisation structure, illustrated in Figure 2.5, a separate department carries out the system tests on most of the systems developed by project teams.

Figure 2.5 Organisation structure: testing is the responsibility of a separate department



Three of the PEP members surveyed had a separate system testing department. These same members also had the most fully developed procedures for testing, and collected statistics on the effectiveness and cost of their testing. One, however, had recently disbanded its system testing department because it found that the project teams became careless in their own testing, relying on the system testing team to find errors. The system testing team then blamed the developers for delivering poor-quality work. The general lack of respect between the two groups led to an overall reduction in quality and productivity.

The very different experience of the International Stock Exchange is described in Figure 2.6. It created a separate testing group in preparation for testing the systems that were being developed for the 'Big Bang' in 1986. It was an expensive investment, but in this case, it did result in the development of very reliable systems.

A separate testing department is expensive but can result in very reliable systems

Figure 2.6 The International Stock Exchange set up a separate testing group and achieved highly reliable systems

In the period leading up to the 'Big Bang', the International Stock Exchange was involved in the development of some large systems that were highly visible to the public, and were essential to the future operation of the Stock Exchange. The systems department decided to set up a separate system testing group for the specific purpose of minimising the risk of implementing systems that might fail. Apart from some well publicised problems in the first hours of operational use, the systems have performed with a very high degree of reliability, and the investment in setting up a system testing group was considered to be justified.

Some of the factors considered in setting up a system testing group were:

- *Independence:* The testing group must be able to retain an objective view of the development, and should not be subject to pressure to cut short testing to bring the project in on time. The group should, however, act as advisers to the project manager, and should not have the final say on when a project is complete.
- *Terms of reference:* Terms of reference must limit the scope of the testing, because there can be a tendency for testing to expand to fill the time available.
- *Managerial support:* Senior systems management support is essential to resist pressures that may arise from the development team to limit system testing. To gain this support, management must be supplied with information on the progress of testing.
- *Marketing:* The role of the testing group should be marketed internally. The Stock Exchange produced a brochure describing the facilities offered by the group.
- *Cost:* An independent testing group is expensive. About 5 per cent of the Stock Exchange's systems development staff were in the testing group. The group also needed its own computer systems for building test environments.

THERE IS NO 'BEST' ORGANISATION FOR SYSTEM TESTING

The experiences of PEP members clearly illustrate that there is no obvious best organisation for system testing. It is worth considering setting up a separate system testing department if the cost of failing to achieve high reliability in systems is high, or if the systems being developed are large. It should, however, be clearly understood that the function of a system testing department is not to find errors that the project teams could have found by thorough module and integration testing. Its purpose is to measure the performance of a system as a whole.

The benefits of a system testing department are that staff develop expertise in testing techniques and that investments can be made in test tools, simulators, and databases, which may be difficult to justify on a project-by-project basis. In the next chapter, we explain how the tools, techniques, and methods that are now commercially available can play an important part in reducing the costs of testing, and describe the circumstances in which each of them can make the greatest contribution.

Chapter 3

Use tools, techniques, and methods to increase the cost-effectiveness of testing

In this chapter, we consider the formal tools, techniques, and methods that can be used to improve the effectiveness of software testing and to reduce its cost. There is a wide variety of aids to better testing on the market.

The greatest choice is available in testing *tools*, and the use of tools is quite widespread Among PEP members. The main *techniques* are the review processes of inspections and walkthroughs. These are applicable to testing (or verifying) the documents associated with software production (that is, program code, specifications, designs, user manuals, and so on). Inspections and walkthroughs are less widely used than testing tools, although the analysis of the PEP database, carried out for PEP Paper 12, *Trends in Systems Development Among PEP Members*, showed that the benefits to be gained are quite substantial. Formal *methods* for software development are widely used, but although testing is a component of them, it is poorly described, and is not based on the concept that testing and production are activities to be carried out concurrently. Software testing methods are, however, beginning to be produced, and significant developments can be expected in the future.

No testing aid will, of course, reduce the intellectual effort involved in designing the test environment and selecting test data. The use of testing aids will, in itself, do nothing to improve the quality of testing. Nor will any single aid to testing cover all aspects of the process. Most of those that are commercially available cover one of five main areas:

The use of testing aids will not, in itself, do anything to improve the quality of testing

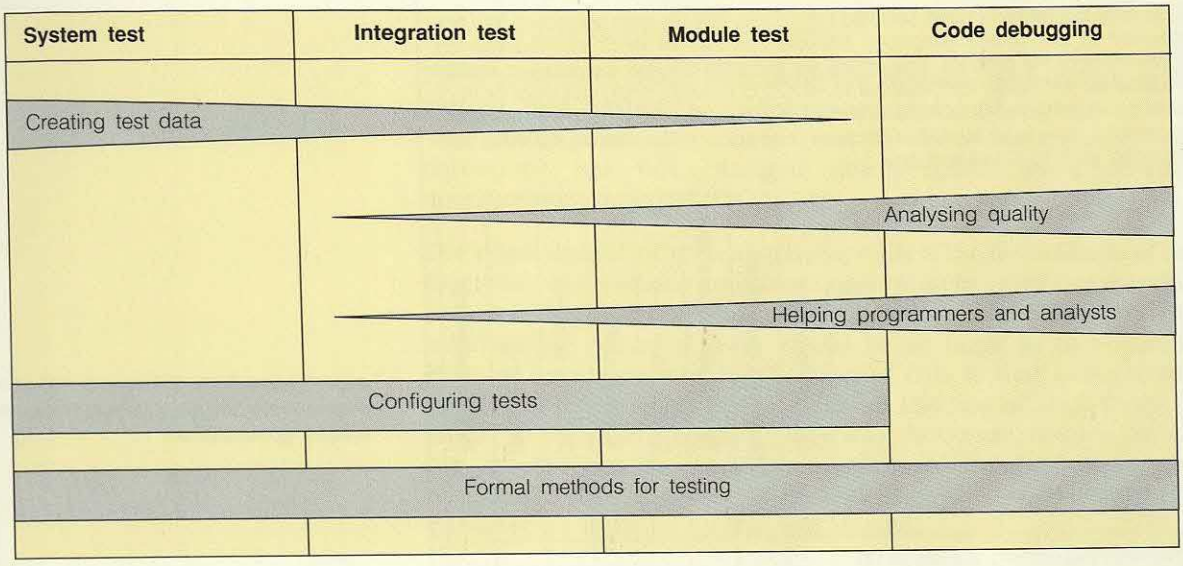
- Formal methods for testing over the whole development life cycle.
- Tools for creating and using test data.
- Tools and techniques for analysing the quality of the specifications, designs, and code.
- Tools to help the programmer or analyst to carry out testing.
- Tools to manage the configuration of test software and test data.

Figure 3.1 illustrates the types of testing for which each type of tool is most appropriate, and Figure 3.2, on page 18, gives a selection of the testing tools that are currently available in the United Kingdom. The categories of tools included in the figure are described in more detail later in this chapter.

FORMAL METHODS FOR TESTING OVER THE WHOLE DEVELOPMENT LIFE CYCLE

The only testing method available in the United Kingdom is a product from Lifecycle Management Systems Limited (LCMS),

Figure 3.1 Each type of testing tool covers different stages of the development life cycle



There is only one testing method available in the United Kingdom

called Structured Systems Test Methodology (SSTM). SSTM is primarily designed to be used in conjunction with the SSADM structured development method, but LCMS claims that it can be tailored to fit other structured development methods. SSTM was first released in November 1989, and there is consequently no experience yet of its use on real systems development projects.

The motivation for producing SSTM was that the quality of a systems development was often not known because the amount of testing had not been evaluated against any absolute scale. LCMS claims that SSTM provides a consistent environment in which project managers have a measure of the quality of the system; this enables them to assess the risk of implementing a system in the final stages of testing.

SSTM is driven largely from the outputs produced by SSADM. By using selected parts of the systems requirements, systems design, program specifications, and module specifications as a baseline, the number of tests needed to test the system thoroughly can be determined. Users and technical management can then decide on an acceptable level of testing.

SSTM incorporates a five-strand testing strategy that produces five self-contained test specifications related to SSADM deliverables. Each specification includes sections on test case identification, test inputs, expected results, supporting documentation, and levels of testing.

TOOLS FOR CREATING AND USING TEST DATA

Tools that can ease the process of creating and using test data provide one or more of the following functions:

Figure 3.2 There is a wide range of testing tools available in the United Kingdom

The tools in this list have been selected from those that are obtainable and that are supported in the United Kingdom. Inclusion in this list does not indicate an endorsement of the product. The criteria for inclusion are that the product should be supported on Digital, IBM mainframe, or ICL computers and that Cobol or PL/1 should be supported on language-dependent products. Some static-analysis products, for example, have been excluded because they are aimed at military systems, and languages such as Coral and Ada.

	Capture/playback	Test data generation	Test database generation	Comparison	Static analysis	Dynamic analysis	Debugging	Test management
ABL Europe Ltd — TIP	✓						✓	
Advanced Programming Techniques Ltd — Oliver						✓	✓	
— Simon						✓	✓	
C A Computer Associates Ltd — CA-Datamacs/II		✓						
— CA-EZTest/CICS							✓	
— CA-Optimiser						✓		
Compuware — CICS Playback	✓			✓				
— File-aid			✓					
Digital Equipment Corporation — Dec Test Manager				✓				✓
Gerrard Software Ltd — Testgen		✓						
IPL Software Products Ltd — Softest		✓		✓				✓
John Bell Technical Systems — Pro-Quest					✓	✓		
— Testa	✓							
On-Line Software International — Datavantage		✓	✓					
— InterTest		✓					✓	
— ProEdit			✓					
— Verify	✓			✓				
Program Analysers Ltd — Testbed							✓	✓
QA Training Ltd — Evaluator	✓			✓				✓
Rand Information Systems Ltd — Testline						✓		✓
Sterling Software — Comparex				✓				
Verilog UK Ltd — Logiscope					✓	✓		
XA Systems UK — Pathvu				✓				

- Capture and playback of test scripts.
- Test data generation.
- Test database generation.
- File and output comparison.

Tools providing these functions are mainly used during system testing, and during the maintenance phase of a project, where 'regression' tests are carried out to check that the system's behaviour has not changed unexpectedly as a result of maintenance activity.

Tools for creating and using test data automate tedious and time-consuming tasks

The main benefit provided by these tools is the automation of tasks that would be tedious and time-consuming to carry out manually. In some cases, the amount of test data required to carry out a satisfactory range of tests would be so large as to preclude a manual approach; the implication of this is that some systems cannot be adequately tested without the use of such tools. The use of these tools does not in any way, however, reduce the need for careful test design. The function of the tool is simply to automate the process of generating test data within the parameters defined by the test design.

An important consequence of automating a tedious manual task is the increase in accuracy that is achieved. If each piece of test data is designed to test a particular function, any inaccuracy in the creation of test data is likely to mean that some functions are not tested as the designer intended.

CAPTURE/PLAYBACK

Capture/playback tools are particularly useful for testing online systems with significant amounts of data entered by users via screen-based systems. They greatly simplify the creation of test scripts, and can save the cost of employing large numbers of unskilled staff to type in the data. They have their main value during system testing, but could also be used very effectively during the module and integration testing of those parts of the system that handle the user interface. The tools run either on the host computer, or on a PC that emulates a terminal on the host computer.

The tools contain some or all of the following components:

- Capture and recording of all the user's inputs, including mouse movements, where these are used. This input is stored as a script, which can be edited if required.
- Recording of responses generated by the system.
- Editing capability on the captured input data.
- Replay of the captured (and edited) script at varying speeds.
- The ability to run multiple copies of the script or scripts on 'virtual' visual display units.
- Comparison of the system-generated responses between different runs of the script, and documentation of the results.

In using a capture/playback tool, each script must be designed to test particular features of the system. If the tool is used merely

to capture a large amount of unplanned user input, very little benefit will be gained.

Capture/playback tools can facilitate tests that would otherwise be very difficult to carry out. A good example is stress testing — subjecting the system to large volumes of test data, or to high transaction rates. This particularly applies to systems, such as ticket-reservation systems, which have large numbers of user terminals. In the test environment, a large number of terminals will almost certainly not be available, and even if they were, organising large numbers of staff to simulate the expected volume of user inputs would be difficult and expensive.

Capture/playback tools facilitate tests that would otherwise be very difficult to carry out

These tools also have particular benefits during regression testing because they allow a script of input commands (including deliberate user errors) to be repeated precisely. To compare the results of the original test and the test of the modified system, differences have to be sought on possibly hundreds of screens. Most capture/playback tools can do this rapidly and without error. In addition, much less effort is required to carry out regression tests since the reruns of the script can be carried out in batch mode without supervision.

TEST DATA GENERATION

Data-generation tools facilitate the automatic creation of large files of data. They are particularly useful for testing systems that process large volumes of data in sequential files. A comprehensive test of such systems generally requires each record in the test data files to be different from all other records. To generate such files manually would be very time-consuming and prone to error. It would, of course, be possible to write a separate file-generation program for each system that is developed, but it is likely to be more cost-effective to purchase a data-generation tool if systems developed by the organisation often use sequential input data files.

Data-generation tools are useful for testing systems that process large volumes of data in sequential files

Data-generation tools use at least two methods to generate the files. One is to define the file off-line using a special programming language. The other is to generate files from within the program itself, by embedding control statements in the program. These statements either generate new records, or select and modify records from existing files.

The generated files contain records in user-defined formats. The tools allow the values of fields in successive records to be generated in various ways — random numbers within a specified range, values clustered about a specified point, sequential values, dates in various formats, and so on. These features allow the test designer to include data to test for particular conditions such as data on, or either side of, boundary values, and also to generate large numbers of different records which may be used for volume testing.

TEST DATABASE GENERATION

Many systems need to access a database. It is not usually advisable to use the live database for system testing, but even where it is

possible, it may be more convenient to use a smaller and more easily monitored subset of the live database.

The test database generation tools available at present are limited to specific products

The test database generation tools currently available in the United Kingdom are limited to the creation of IBM IMS and DL/1 test databases. They work by selecting and modifying records from an existing similar database. Some tools provide an interactive editing facility, while others have a command language that defines the file-conversion rules. There are many similarities between these tools and data-generation tools. Some products, such as CA-Datamacs/II, include both facilities within the same tool.

COMPARISON

Comparison tools can be particularly useful in the maintenance phase

Comparison tools are the only ones that operate on the outputs from the system under test. Systems that generate significant amounts of output in the form of files can benefit from the use of these tools. They can save time and improve accuracy when the tester is looking for small differences between runs of a test program, or when he is comparing expected results with actual results. They can be particularly valuable in the maintenance phase of a system's life cycle, where it is essential to verify that a correction or a change to a program does not have unexpected side effects. The tools produce printed reports that management can use as an objective measure that the system has not been degraded by the change.

File-comparison tools identify records within a file that have been inserted, deleted, or modified. The ability to make comparisons in this way is usually included as a feature of capture/playback tools. In these instances, comparisons are made of outputs sent to a display screen by the system. Some file-comparison tools are included in manufacturers' operating systems — for example, the 'Difference' command in Digital's VMS operating system.

TOOLS AND TECHNIQUES FOR ANALYSING THE QUALITY OF THE SPECIFICATIONS, DESIGNS, AND CODE

Tools and techniques for analysing the quality of the specifications, designs, and code are essential where applications need to be highly reliable

The quality of every deliverable produced during the development of a system, including requirements specifications, designs, code, and test specifications, should be analysed as part of the normal development process. Various techniques and tools are available for this purpose. The two most commonly used techniques are inspections and walkthroughs.

In addition, two types of tools — static analysers and dynamic analysers — can be used to analyse the quality of the code itself. The appropriateness of these tools depends, however, on the programming language used. They are suitable for use with commercial programming languages such as Cobol. Code written in manufacturer-specific languages, such as Tandem's TAL, or in fourth-generation languages, cannot be analysed by these tools.

The use of these techniques and tools is almost essential for applications with high reliability requirements, such as those where human life depends on the successful operation of the

software. These applications should therefore be written only in a language that can be analysed by these tools.

INSPECTIONS

The inspection technique was first developed by Michael Fagan while he was working at IBM in the early 1970s. An inspection is carried out by a team, typically of four people, whose roles are precisely specified. A key to successful inspections is that the team must identify errors only; it must not be side-tracked into discussions of solutions, or alternative design strategies. It is important that the results of the inspection are recorded, and that all errors are corrected by the original designer or programmer.

Inspections are time-consuming (typically, between 4 and 8 per cent of total development effort), and need to be scheduled in the project plans. The total time (including preparation time) for an inspection of a design that produces 1,000 lines of code is 10 to 20 man-hours, and for an inspection of the 1,000 lines of code produced from this design, 20 to 60 man-hours.

In carrying out inspections, each person needs to have a clear understanding of his individual role, and of the purpose of the inspection procedure. The techniques are not easy to learn, and an organisation that intends to introduce inspections should train its staff on formal courses.

Studies of the effectiveness of performing inspections on source code indicate that an inspection typically detects up to 60 per cent of the errors in the code. The reduction in development cost, after allowing for the additional cost of the inspections, is estimated at 10 per cent. This is consistent with the results reported in PEP Paper 12, where PEP members using inspections or walkthroughs had a Productivity Index about half a point higher than members who did not use them. If the subsequent maintenance phase is also included, the savings may be considerably larger.

An inspection reduces development costs by about 10 per cent

WALKTHROUGHS

Walkthroughs are a less formal type of inspection and may have few, if any, of the formal characteristics of inspections. There are very few rules on how to carry out a walkthrough. At a minimum, it involves one person checking another's work. Because of the lack of formality, walkthroughs are cheaper to carry out than inspections. They are almost certainly less effective, although quantitative data are lacking.

STATIC ANALYSERS

Static-analysis tools examine the structure of code without running it, and can typically find between 10 and 20 per cent of all errors in a program. They are cost-effective tools in the development of reliable systems, but do not seem to be widely used in the commercial environment. Much of their use in the United Kingdom has been in avionics and military systems. We believe that there are considerable benefits to be gained from the use of these tools in commercial applications and that their use should be carefully considered by all PEP members.

Static-analysis tools typically find between 10 and 20 per cent of all errors in a program

The tools provide managers with objective measurements of characteristics that are directly related to quality. These help to identify areas of poorly structured or excessively complex code. It is advisable to redesign such code before proceeding further with testing. If part of the code is unavoidably complex (a complex logical algorithm, for example), extra attention should be paid to its module testing since it is likely to contain an above-average number of errors.

Static analysers typically assess the following characteristics of the code:

- Conformance to user-specified standards (for example, no more than a predefined number of lines in a module, or no use of 'go to' statements).
- The paths (or different sequences of instructions) through a program.
- Complexity analysis. Two of the most widely used measures of complexity are McCabe's measure, and the number of knots, explained in detail in Figure 3.3, overleaf.
- Data-flow analysis, showing procedure calls, usage of procedure parameters, and unreferenced or unused data items.
- Cross-reference of all data items.

Some training is required in the interpretation of these measures, but carefully used, they can identify many coding errors before any attempt is made to run the code. This obviously saves effort and machine time.

DYNAMIC ANALYSERS

Dynamic-analysis tools monitor the code while it is being executed

Dynamic-analysis tools provide objective measures of the testing procedure, commonly known as 'white-box' testing. This procedure is carried out as part of the module-testing phase, and may also be done during integration testing. The tools monitor the code while it is being executed, and produce a report at the end of the execution, giving various statistics. These statistics can be used to assess the effectiveness of the test cases.

It is essential that white-box testing is carried out, since in a typical program, over 50 per cent of the code is not directly related to end-user functionality, but to the manipulation of internal pointers, flags, and intermediate results. 'Black-box' testing, which views the system externally in terms of inputs and outputs, cannot be designed to guarantee complete coverage of this 'hidden' code.

One tool, Testbed, provides three measurements, or test effectiveness ratios (TERs), resulting from a dynamic analysis of the code. Other dynamic-analysis tools, which are sometimes also known as coverage analysers, provide at least the first measurement. These measurements are:

- TER1 — statement coverage analysis: the percentage of the lines of code that have been exercised at least once. No operational system should be released where this measure is less than 100 per cent.

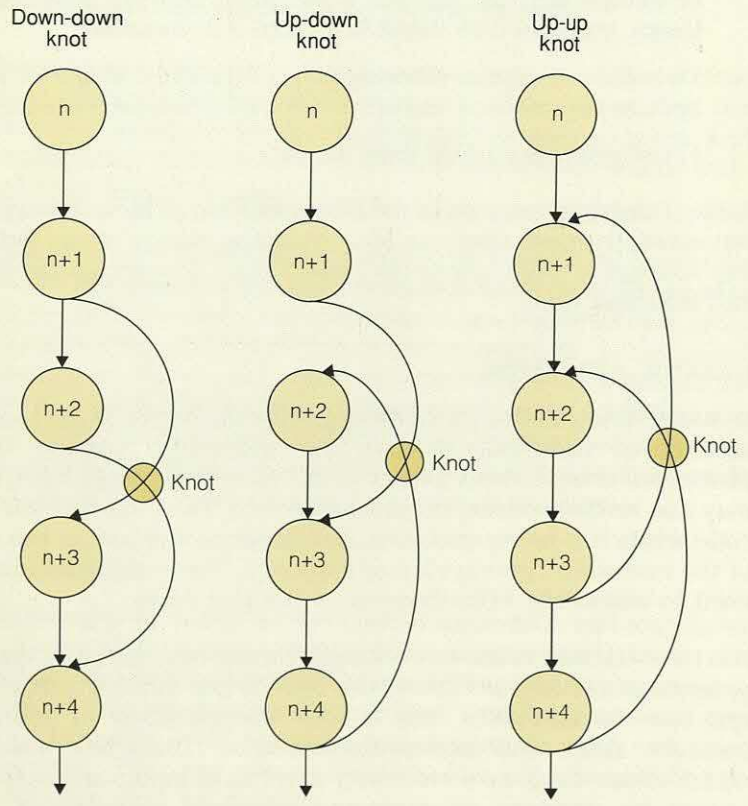
Figure 3.3 Two of the most widely used measures of the complexity of a program are McCabe's measure and the number of knots

McCabe's measure

McCabe's measure is defined as one more than the number of decision statements in a program. The metric is very simple, but experience shows a significant correlation between McCabe's measure and the number of bugs, or debugging effort applied to a program. Programs with a McCabe value in excess of 10 seem to have disproportionately more bugs than those with values of less than 10.

Knots

The purpose of looking for 'knots' is to identify unstructured code, which tends to contain more errors than properly structured code. A control-flow knot is defined as occurring when two control jumps cross, as illustrated in the diagram. Three types of knots are depicted. A down-down knot is relatively harmless, and represents an 'if ... then ... else' construct. Up-down knots are more likely to represent unstructured code, but may arise from 'do' or 'while' loop constructs. Up-up knots always represent unstructured code.

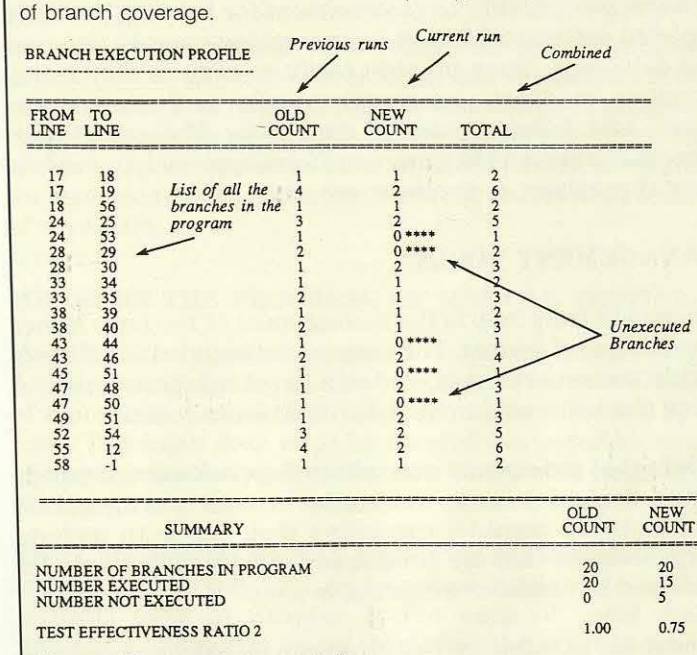


- TER2 — branch coverage analysis: the percentage of all outcomes of branch instructions that have been exercised at least once. The goal of testing should also be 100 per cent, although this is not as easy to achieve as 100 per cent on TER1.
- TER3 — path coverage analysis: there are several ways of measuring paths through a system, all of them quite complex. It is difficult, in practice, to achieve 100 per cent path coverage during testing, and except in ultra-high-reliability systems, it is probably not worth attempting it.

An example of output from Testbed for TER2 is shown in Figure 3.4.

Figure 3.4 The testing tool, Testbed, provides three measurements resulting from a dynamic analysis of the code

The diagram is an example of output from Testbed for the second of the measurements, or test effectiveness ratios (TERs), that it provides. TER2 is an analysis of branch coverage.



(Source: Program Analysers Ltd)

An experiment carried out on a large military system in the United States showed that, when static analysis was combined with dynamic analysis, 70 per cent of all errors in the system were discovered. The remaining errors were caused largely by errors in the specifications or misunderstandings of the written requirements.

Dynamic-analysis tools produce reports containing objective measures of testing progress

Project managers will find the use of this type of tool particularly helpful since they can provide reports containing objective measures of progress such as, *"tests covering 78 per cent of statements and 64 per cent of branches have been successfully completed"*. This gives them much better control over a project than having to rely on a programmer's typical estimate that *"testing is 95 per cent complete"*.

TOOLS TO HELP PROGRAMMERS AND ANALYSTS TO CARRY OUT TESTING

Two types of tool — debuggers and test harnesses — are used during module testing to help in the process of debugging and testing:

— *Debugging* is a distinct activity from testing, as described in Chapter 1, and has been excluded from the scope of this paper. However, several debugging tools include features that enable them to be used for formal testing. The list in Figure 3.2

(on page 18) is not exhaustive; we have included only those for which the manufacturer also supplies another type of testing tool that can be used in conjunction with it. All PEP members use some debugging tools, since it would be expensive to develop programs without them. If additional debugging tools are required, it would be worth considering tools that are also useful for formal testing.

— *Test harnesses* provide an environment for running partially completed software when it is undergoing module tests, or being debugged. They provide facilities such as simulating incomplete modules, intercepting calls to external procedures, and defining external data areas. The use of such a harness could provide a more uniform approach to module testing throughout a development team.

TEST-MANAGEMENT TOOLS

Test-management tools are useful if there are many test cases

Test-management tools help in the management of the tests rather than in the process of testing. They are particularly useful if there are many test cases to manage, and as a long-term investment in maintaining the test environment for regression testing.

Organisations that are already controlling their software developments using tools for code management and configuration management should consider extending their scope to include testing. Organisations that are not already using such tools would be well advised to consider investing in them.

While investment in software tools that can be used to carry out a planned level of testing within an organisation will undoubtedly help to ensure that the cost of software testing is reduced, they do not in any way absolve managers of their responsibilities for project control. Managers need to monitor the progress of software testing and to draw on the results that the process provides as a basis for making decisions to ensure that projects are delivered on schedule. How the progress of software testing should be measured is the subject of Chapter 4.

Chapter 4

Measure the progress of testing to improve project control

One of the main tasks of development project managers is to decide how to allocate resources so that the project is completed on schedule. This requires regular monitoring of progress, so that resources can be re-allocated if the project deviates from plan. In this chapter, we show how progress can be measured during the module-testing, integration-testing, and system-testing phases of a project.

MEASURE THE PROGRESS OF MODULE TESTING

Module tests should be designed to fulfil two requirements — each function provided by the module should be tested with a range of input data, and all of the code should be exercised by the input data. The input data must be carefully selected to achieve both of these goals; testing each function with a wide range of randomly selected input data is unlikely to achieve the second goal.

There are several ways of measuring the extent to which code is exercised by the tests, as described in the section on dynamic-analysis tools in Chapter 3. For each of these measures, a numerical value can be set, to define the point at which testing of the module should stop. Such values could, for example, be 100 per cent of code coverage and 90 per cent of branch coverage. This value should be specified according to the level of confidence that is required in the reliability of the system.

It is not practical to define module tests that will achieve preset targets

It is difficult to select, in advance, a series of tests that will achieve the target values, and it is not worth attempting to do so. The most practical way of approaching module testing is to begin by specifying a series of tests that test all the functions with a good range of input values. A dynamic-analysis tool can then be used to indicate which parts of the code have not been exercised. Additional test cases can then be defined to fill in the gaps until the code has been exercised to the required extent on the different measures of coverage.

It should be noted that the completion of testing, defined in this way, does not guarantee a module free of errors. It does, however, provide a certain level of confidence in the reliability of the module. The higher the percentage coverage on the three measures, the greater the confidence in the module's reliability.

MEASURE THE PROGRESS OF INTEGRATION TESTING

Integration tests should be designed to measure that the flow of control between modules is correct, and that data definitions between modules are consistent.

The control flow between modules can be measured using the same dynamic-analysis tools that were used for module testing,

but operating on the module, rather than the line of code, as the basic unit. Test data should be selected so that each module in the system is called at least once, and a predefined percentage of the control paths through the modules is exercised. On a small system, it is relatively easy to check manually that all control paths have been exercised, possibly using the checkpointing facility of debugging tools. On systems with more than about 20 modules, a tool that can analyse control paths through programs and measure the extent of the program coverage is essential if integration testing is to be carried out thoroughly. The measurement of progress of the control-flow part of integration testing is then the measurement of the percentage of module coverage and control-path coverage that has been achieved.

Integration tests should ensure that each module is called at least once and that a predetermined percentage of control paths is exercised

Ensuring that data is consistently defined between modules requires a series of tests to be defined that explicitly check that each data item is handled consistently by all modules that create, read, modify, or delete it.

Measuring the progress of a series of predefined integration tests is very similar to measuring the progress of a series of predefined system tests. The techniques for doing this are described in the next section.

MEASURE THE PROGRESS OF SYSTEM TESTING

System tests are carried out on the complete system at the end of the development cycle, and should be started only when module and integration testing have been completed. The tests measure the performance of the system against the original requirements specification, and are normally used as the basis on which a decision is made to introduce the system into live operation.

System tests measure the performance of the system against the original requirements specification

DEVISE A SYSTEM-TEST PLAN

The essential prerequisite for monitoring the progress of system testing is a test plan. This should be produced soon after the requirements specification is completed, as indicated in Figure 1.2, and should include a specification and reference number for each test, the expected result of each test, and a timetable for the execution of the tests.

A test plan is a prerequisite for monitoring the progress of system testing

USE FORMS TO COLLECT MEASUREMENTS

The progress of testing can be documented using two forms — the test log, and the incident report. Examples of these forms are shown in Figures 4.1 and 4.2. A test log is filled out when each test is run, whether it is successful or not. Incident reports are filled out for each fault discovered during the running of a test. The incident reports are given to the development team, who should diagnose the problem and carry out any necessary rework. Project managers should record the effort spent on testing and on rework, using either the organisation's standard timesheet, or the incident report itself.

Test logs and incident reports should be used to document the progress of system testing

INTERPRET THE RESULTS

There are two main styles of system testing. In one style, the tests take place over several weeks, and the development team corrects

Figure 4.1 A test log is filled out when each test is run, whether it is successful or not

The text below shows an example of test log entries.

System: Services marketing.

Test log identifier: TL-21.

Description: Tests Version 2 of the Customer Index subsystem against Version 4 of the System Test Plan for test procedures TP21-1 to TP21-10. The tests are run using the file TP21.TST (dated 1 November 1989), running under the installation standard capture/playback tool.

Activities and event entries			*Incidents*
10 December 1989			
09.45	TP21-1 run.	No discrepancies from expected results.	
09.55	TP21-2 run.	No discrepancies from expected results.	
10.15	TP21-3 run.	System crashed.	IR21-1
11.15	TP21-4 run.	Missing update to screen.	IR21-2
11.20	TP21-5 run.	No discrepancies from expected results.	
11.30	TP21.6 to TP21-10 not run since they depended on the successful execution of TP21-3.		

Figure 4.2 An incident report is filled out for each fault discovered during the running of a test

The text below is an example of an incident report.

Incident identifier: IR21-2, 10 December 1989.

Summary: Test procedure TP21-4 did not update the average balance field on the screen.

References: Test log TL-21.

Incident description: The average balance field for customer John Abel did not appear on the screen. All other fields updated correctly. All other customers showed correct average balance values. Possibly relevant is the fact that John Abel is first in the alphabetical list of customers.

Impact: No impact on the execution of other tests within TP21.

errors concurrently with continuing system testing. In the other style, the tests are completed in a few days, and there are then several distinct test cycles, each one testing a corrected version of the system. The following paragraphs apply specifically to the first style of testing. With slight modifications, however, the techniques could equally well be applied to the second style.

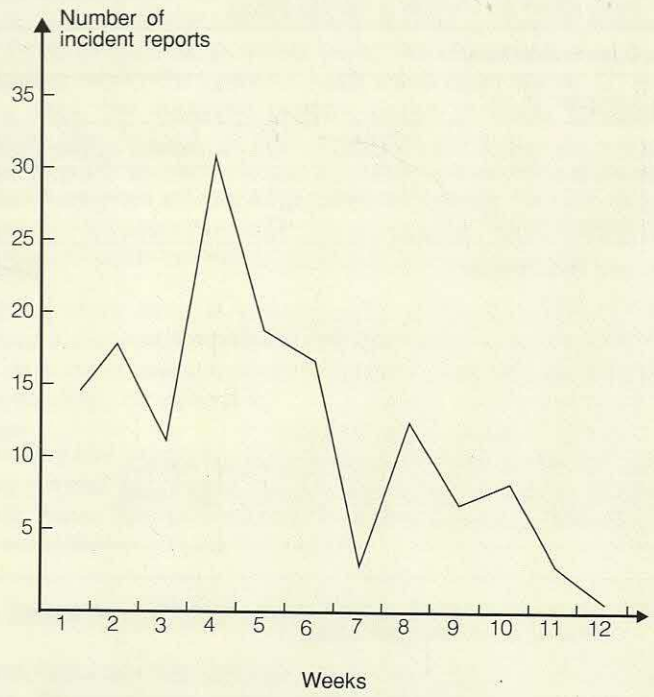
The progress of system testing can be plotted on graphs

The technique for interpreting the results is to display the progress of the system testing on graphs, which are regularly updated. This enables potential problems to be identified quickly. It is possible to extrapolate trends to estimate completion dates, but these extrapolations should be treated with caution. If they appear to indicate that the completion of the tests will take longer than expected, the test logs and incident reports should be examined in more detail to identify the cause of the slippage.

Rate of generation of incident reports

The sample graph in Figure 4.3, overleaf, shows the number of incident reports produced each week. The number of errors

Figure 4.3 The number of incident reports produced each week should be registered and monitored



should diminish as errors are corrected after the first run-through of the complete system tests. If, in any week, there are marked peaks or troughs, the reasons for them should be examined. A peak could be caused by a particularly error-prone part of the system, or a trough by a problem holding up the progress of the testing.

Cumulative error rates

Representations of the cumulative error-detection and error-correction rates are illustrated in Figure 4.4.

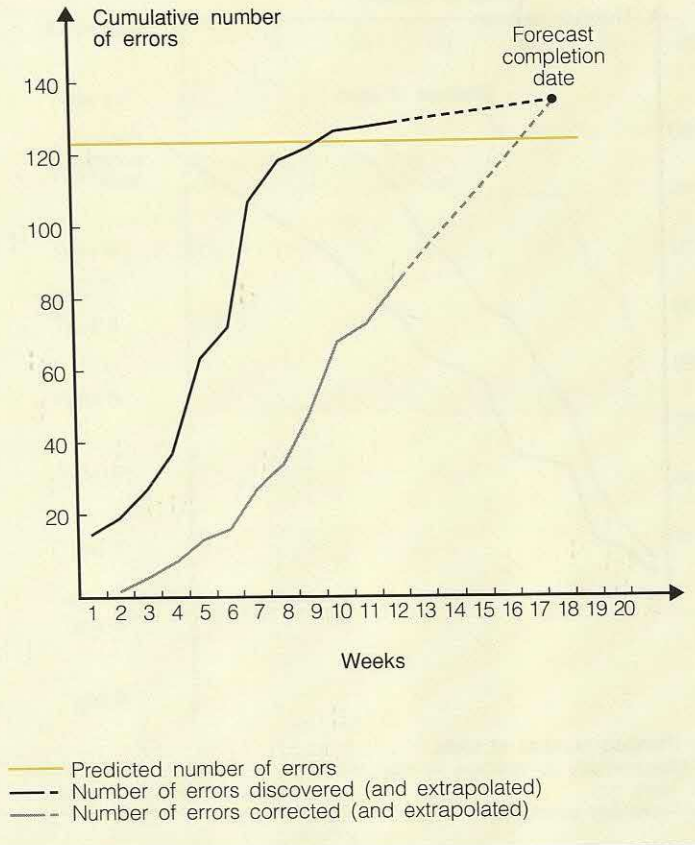
The yellow horizontal line is the predicted number of errors in the system, based on the number of lines of code. PEP Paper 12 showed that the number of errors at the start of integration testing was in the range of 0.5 to 2 per thousand lines of code, with small programs containing proportionately more errors than large ones. If records are kept for each project, each organisation could derive its own relationships between program size and total number of errors.

The two graphs shown in Figure 4.4 are the cumulative totals of the number of errors discovered, and the number of errors corrected and successfully retested. These are derived by analysing the incident reports.

The rate of discovery of faults should diminish as the testing proceeds and faults are corrected. The rate of correcting faults gives a useful indication of how long the rework is taking. Once the trend line for the number of errors discovered has flattened out, simple extrapolation of both trends will provide an estimated completion date.

The rate of correcting faults gives a useful indication of how long the rework is taking

Figure 4.4 Records should be kept of cumulative error-detection and error-correction rates



A steep gradient for the cumulative errors-discovered graph, particularly if the cumulative number of errors exceeds the predicted number of errors, indicates that modules contain errors that should have been detected at the module-testing phase. It will almost certainly be more productive in these circumstances to suspend system testing, and return to module testing. This is a difficult decision for a project manager to make, but the evidence provided by these graphs should make the decision easier to justify.

Test completion rate

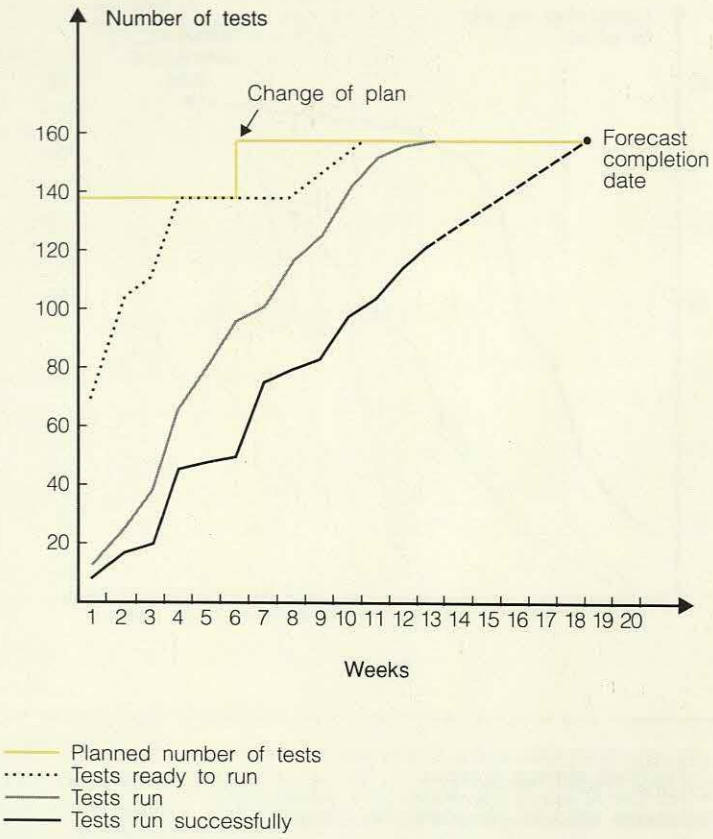
The rate of completion of the planned tests gives a useful forecast of project completion

The graphs shown in Figure 4.4 suffer from the disadvantage that the total number of errors is not known, and the end point of the testing is not clearly defined. A more suitable measure for predicting project completion is the rate of completion of the planned tests. An example of graphs based on this parameter is shown in Figure 4.5, overleaf.

The yellow line shows the planned number of tests, which should not change during the testing phase. An increase such as the one shown in the figure should be a cause for concern, because it probably indicates that the original test plan was unrealistic, or that changes are being made to the requirements.

The dotted black line is a plot of the cumulative number of tests that are ready to run. On some systems, all the test data and test

Figure 4.5 Project completion can be predicted from the rate of completion of the planned tests



programs may be developed by the start of system testing, in which case the graph can be omitted. More probably, particularly on larger systems, some tests will be developed concurrently with others being executed. The trend of this line should be monitored to ensure that the preparation of test data does not become a bottleneck.

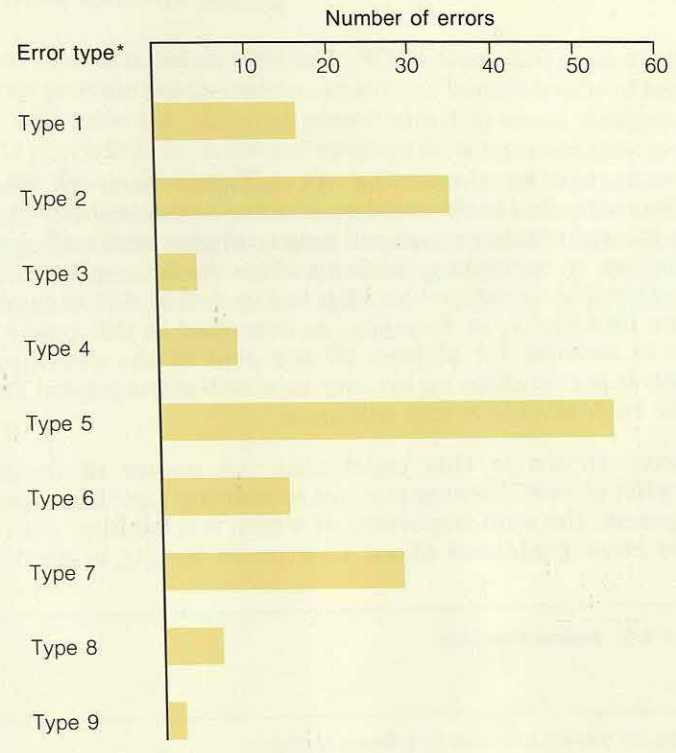
The number of tests that have been run is plotted on the grey line, and the solid black line is the number that have been run successfully. A low rate of success indicates that the earlier module testing and integration testing were inadequate. The solid black line also measures the progress of the rework. The slope of this curve gives an indication of progress, and a simple extrapolation can be used to forecast the completion date.

Frequencies of error types

A useful analysis for helping to improve the future productivity of the systems development department is a histogram showing the types of system-testing errors and their frequencies. An example is shown in Figure 4.6. This histogram could be constructed both for individual projects, and as a cumulative total over all projects. It could also be extended to include errors discovered during integration and module testing. It indicates what areas of software design or programming techniques cause

Analysis of the types of system-testing errors and their frequencies can help to improve future productivity

Figure 4.6 Analysis of the types of errors and of their frequencies should help to improve productivity



* Each number refers to a particular type of error, such as 'incorrect branch condition', or 'requirements error.'

the most problems, and should help management to decide where training needs are greatest.

Chapter 5

Review the software-testing policy

Software testing should be an integral part of systems development, and in this report, we have recommended changes in traditional practices that will help to ensure that testing is not treated as a secondary activity. The recommendations are summarised as an action checklist in Figure 5.1. Whether testing is done informally, or formally, as described in this paper, it is likely to account for at least 25 per cent of the development budget. It is therefore an activity to which management should devote both attention and resources.

Testing is likely to account for at least 25 per cent of the development budget

We have shown in this paper that the nature of testing is somewhat elusive. Testing is about measuring various properties of a system, the most important of which is reliability, yet there are no clear guidelines about how much testing is needed to

Figure 5.1 Action checklist

Analyse the current approach to software testing: — What is its objective? — How much does it cost? — What benefit is gained from it?
Assess whether the current approach to software testing is satisfactory: — Do systems have unexpected operational problems? — Are there problems with cost overruns or late delivery, which could be attributed to difficulties in testing? — Are there any guidelines about when to stop testing?
Define an overall approach to software testing: — What measurements are needed from the testing process? — How will the measurements be used to improve the quality of the systems and the efficiency of the developments? — What measures are taken to ensure that staff understand the objective of testing? — How will the cost of testing be measured?
Ensure that the structure of the systems department and of the project teams is the most effective one for software testing: — Should there be a separate group responsible for system testing? — Do project teams have clearly identified responsibilities for testing?
Provide adequate staff training for software testing: — Are there any reference books on software testing in the organisation's library? — Has consideration been given to the value of training courses?
Consider the use of walkthroughs and inspections.
Consider investing in a standard set of software testing tools for use by all projects in the systems department: — Which are the most effective tools for the organisation to acquire? — Is testing efficiently supported in all phases of the systems development life cycle?

produce a given level of reliability; it is not necessarily the case that more is better. In many systems development projects, there are, nevertheless, substantial benefits to be derived from carrying out formal software testing.

Formal testing is not an easy task. While there are tools available that can provide precise measurements of some aspects of testing, none will take the place of experienced systems designers or testing specialists in an area of systems development that is more of an art than a science. They do, however, need to practise their skills within the framework of a clearly defined policy for software testing. The first step for many organisations in improving the effectiveness of their software testing will be to review the checklist in Figure 5.1 to see that such a policy is in place.

Bibliography

Beizer, B. *Software system testing and quality assurance.* New York: Van Nostrand Rheinhold, 1984.

Hetzel, W C. *The complete guide to software testing.* 2nd edition. Wesley, MA: QED Information Sciences, 1988.

Myers, G J. *The art of software testing.* Chichester: Wiley, 1979.

Parrington, N & Roper, M. *Understanding software testing.* Chichester: Ellis Horwood, 1989.

BUTLER COX

P,E,P

Butler Cox

Butler Cox is an independent international consulting group specialising in the application of information technology within commerce, industry and government.

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.

Public Reports
Analysing trends and experience in specific areas of widespread concern.

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

Forthcoming PEP Papers
Software Quality Measurement
Selecting Application Packages
Project Estimating and Control

Butler Cox plc
Butler Cox House, 12 Bloomsbury Square,
London WC1A 2LL, England
☎ (01) 831 0101, Telex 8813717 BUTCOX G
Fax (01) 831 6250

Belgium and the Netherlands
Butler Cox BV
Burg Hogguerstraat 791,
1064 EB Amsterdam, The Netherlands
☎ (020) 139955, Fax (020) 131157

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