A software testing estimation and process control model

Colin J. Archibald

A Software Testing Estimation and Process Control Model.

Colin J. Archibald

Ph.D Thesis

University Of Durham

Department of Computer Science

November 1998

Submitted for the degree of Doctor of Philosophy

Abstract

The control of the testing process and estimation of the resource required to perform testing is key to delivering a software product of target quality on budget.

This thesis explores the use of testing to remove errors, the part that metrics and models play in this process, and considers an original method for improving the quality of a software product. The thesis investigates the possibility of using software metrics to estimate the testing resource required to deliver a product of target quality into deployment and also determine during the testing phases the correct point in time to proceed to the next testing phase in the life-cycle. Along with the metrics Clear ratio, Churn, Error rate halving, Severity shift, and faults per week, a new metric 'Earliest Visibility' is defined and used to control the testing process. EV is constructed upon the link between the point at which an error is made within development and subsequently found during testing. To increase the effectiveness of testing and reduce costs, whilst maintaining quality the model operates by each test phase being targeted at the errors linked to that test phase and the ability for each test phase to build upon the previous phase. EV also provides a measure of testing effectiveness and fault introduction rate by development phase.

The resource estimation model is based on a gradual refinement of an estimate, which is updated following each development phase as more reliable data is available. Used in conjunction with the process control model, which will ensure the correct testing phase is in operation, the estimation model will have accurate data for each testing phase as input.
The proposed model and metrics have been developed and tested on a large scale (4 million LOC) industrial telecommunications product written in C and C++ running within a Unix environment. It should be possible to extend this work to suit other environments and other development life-cycles.

Acknowledgements

The author would like to acknowledge BT Laboratories for their financial support with University fees and visits to Durham. Special thanks to Stuart Birchall for his encouragement and support for this research and to Mark Norris for a constant supply of reference books.

For guidance, patience and encouragement a big thank you to my supervisor Dave Robson. Thanks also to Prof. Keith Bennett for advice and use of the facilities within the department, and to Malcolm Munro for advice following the review of the thesis proposal.

Final acknowledgements to Sandie, who maintained the house and garden during this period of research and to my family and friends who will no doubt be pleased to see me again.

November 1998 Colin J. Archibald

Statement of Copyright

Contents

1 Introduction

There is a growing problem with software, its size and complexity, the IT industries ability to deliver and the reliance we all have on software during our everyday lives. Norris [1] investigating industry trends says that, "system costs have become more and more dominated by the cost of software production over the last 20 years:

1985 hardware 60% software 40%

1995 hardware 20% software 80%

Most of the problems in the IT industry over the last 10 years (notably the spectacular failures) can be traced to software problems". The size of the software utilised has also grown over the last 10 years [1]:

Application	**1985 lines of code**	**1995 lines of code**
Car	less than 5K	over 30K
Telephone exchange	about 1M	from 5M to 25M
Aircraft	about 400K	about 5M
Television	less than 10K	up to 500K
Database	about 500K accessing 100MB data	about 4M accessing up to 5TB data

There is a problem with the delivery of software projects [1]:

Project size **million lines**	**Average delay**	**Cancellation** **probability**
0.1	2 months	5%
1	1 year	14%
2	1.4 years	18%
5	1.8 years	25%
10	2 years	36%
20	2+ years	40+%

A number of companies in different market segments of the software development industry have also discovered that between 40% and 70% of the money spent on designing, developing and supporting a software product will be spent on maintenance [2] [3] [4]. Some of the maintenance costs will cover the changes necessary to keep the software in step with the underlying hardware and operating system changes that occur during the life of a system. A large proportion of the 70%, 32% [5], will be spent on fixing problems that had not been found during the implementation phase before deployment. For telecommunications software 34% of the maintenance work is repair [2].

Even though techniques and automation, structured design methods (SSADM, LBMS) compilers and high level languages are available, the software development process is still basically a human process. Any human process is prone to error and therefore design reviews, code walk-throughs and a range of testing practices are needed to expose the errors that are introduced during software development.

In theory, formal methods should have removed the need for testing by now, with a formal proof of correctness. But the proof of correctness is often a more lengthy complex process than the initial program would have been, so that the errors are introduced during the specification phase, because this becomes the challenging part of the process. Formal methods may have a part to play on small safety critical problems, but they have not scaled up to industrial projects [6] [7]. Sommerville says [8] "The advantages of using formal software specifications are quite clear. However, there is one fundamental problem which so far, has militated against their use in the development of large software systems. The problem is that software specifications are often very difficult to construct and to understand. This is particularly true for high-level abstractions representing complex activities such as document formatting, aircraft navigation, etc.

The difficulties in constructing formal specifications mean that their practical usefulness is currently limited. It is not economic to spend the time required to develop and debug formal specifications for even a small part of a large software system except, perhaps, in very critical applications where it is

intended to verify the resulting program." Even so small developments have achieved successful deliveries using a rigorous approach to specification [9] and rigorous program design [10], plus the use of formal verification has been applied to industrial developments or 1 - 2K lines of source code [11].

A recent study [12] within BT found that across ten large scale development projects the spend on testing as a proportion of the end to end implementation cost (specification to deployment) averaged at 30% (+/- 5%). Dorothy Graham [4] believes that typically testing consumes 40% of software development effort and 75% of maintenance effort. Osterweil & Clarke [13] have a higher figure, "statistics gathered over the past ten or fifteen years show that the testing, analysis and debugging costs usually consume over 50% of the costs associated with the development of large software systems".

In common with other software developers BT has found that the cost to fix an error typically increases ten fold after each stage of the development life-cycle. It will cost about 1000 times more to fix an error found during operational use, than a error found at requirements stage. As an example, it was estimated that one US Air Force system cost $30 per instruction to develop and $4,000 per instruction to maintain over its lifetime [14]. Although errors are introduced at each stage of the development life-cycle it is imperative that they are found as soon as possible and preferably before moving on to the next development stage. Pressman [7] "Error rates for new programs cause customer dissatisfaction and lack of confidence."

It is possible, although not as common, to over-estimate the testing cost as well as under-estimate [12]. Software is not expected to be error free, the cost of removing all errors prohibitively high for most industries. But the testing does need to remove the errors that will have an impact on operational use, and therefore reduce the cost of maintenance.

The research and production of this thesis has been targeted at solving a particular problem encountered during software development and verification. When should a software build move from one phase of testing to the next and how can you estimate the cost of testing software. Industrial software

development projects will deploy different teams of engineers for each skilled task throughout the development lifecycle. The software development engineers will test their own code, possibly with the addition of peer reviews. A separate integration team will then be responsible for the integration and interface testing, and another team will cover the system validation activities. The system validation test team may also carry on to implement the reference model and customer acceptance testing, although for some organisations these will also be supported by separate teams. In practice the ownership and responsibility for the software, including overrun costs, is also passed along with the software between these teams. Each team will therefore wish to complete their testing to time and budget, passing the software on to the next stage as soon as possible. Each team will not wish to accept software that has not completed an earlier phase successfully as they will be inheriting the problems and cost to fix. This is sometimes described as the software being thrown over the wall between teams, if it is passed on without completing the phase successfully and without consideration for the next team.

A model is needed to show how faults are introduced, found and propagate around the parallel activities of the development and test lifecycle process. A set of metrics are required that can measure attributes of faults and provide control on the movement of software between the test phases. If the testing process is controlled and delivers a consistent quality product from each test phase the accurate estimation of testing resource will then be possible.

1.1 Overview of Research

This thesis explores the use of testing to remove errors, the part that metrics and models play in this process, and considers a new original method for improving the quality of a software product. The thesis investigates the possibility of using software metrics to estimate the testing resource required to deliver a product of target quality into deployment and also determine during the testing phases the correct point in time to proceed to the next testing phase in the life-cycle. The proposed model and metrics have been developed and tested on a large scale (4 million LOC) industrial telecommunications product written in C and C++ running within a Unix environment. It should be possible to

extend this work to suit other environments (e.g. main frames, pc's) and other development life-cycles, RAD.

1.2 Thesis Outline

The second chapter summarises software verification, definitions of testing and the product development process. A practical approach to testing is presented with special attention given to software validation.

The third chapter entitled 'process control and measurement' provides a survey of metrics and models which provide a foundation for the work in this thesis, including definitions, evaluation and implementation methods. The reasons for applying metrics to a project are considered along with the impact on the product quality/cost balance.

An overview of software development models is provided by the fourth chapter 'development models'. Included are the main model types; sizing, cost estimation, and reliability, with the theory and practice of each explained. Complexity metrics are also discussed as they have formed the basis for a number of models.

Original research is presented in the fifth chapter with a new fault propagation model and techniques to understand the parallel paths of faults from introduction to discovery. A new method of demonstrating the relationship between testing coverage and the chance of finding a fault is explained with the use of Venn diagrams. The possible uses for the fault propagation model, constructed for this thesis, are explained along with an experimental method to prove the model relationships.

The hypothesis for this unique work on a testing estimation and process control model plus the metrics employed are explained in the sixth chapter.

The seventh chapter outlines the case study, estimation of testing resource, and testing process improvements. A testing process model for data collection, along with the data collection forms and explanation for users, provides everything needed to start the metrics programme.

The eighth chapter contains the results from the experimentation along with an analysis of the results and discussion of the implications.

The last chapter provides a summary of the thesis, the conclusions and the contribution made by this thesis. Recommendations are included on future research that could be undertaken to develop the proposed model for wider use and also refine it for improved results within the initial industrial area where it has been applied.

The Appendix lists methods and techniques for software verification, providing a detailed reference of testing methods to support chapter two. The initial experimental model is refined and put forward as a practical estimation tool for industrial products in Appendix C. The detailed results tables can be found in Appendix D.

2 Software Verification

"Software engineering is hindered by the fact that no consistent terminology exists to describe software systems," Sommerville [8].
The meaning of verification and validation is often blurred and confused, and "used in a variety of contradictory and confusing ways" says Howden[15].
"Although verification and validation may appear synonymous, this is not in fact the case" Sommerville [8].

Within this thesis verification is the testing by analysis or execution, proving that the output from one stage of the development life cycle is consistent with the stage before; specification meets all the requirements, the system design implements the specification, module design consistent with system design, the code follows the module design. Quality reviews are used to look for the inconsistencies between each of these stages. The testing phases, module, integration, and validation, can also be considered as verification activities. Module testing verifies that the coding and module design have been implemented correctly, integration testing of the modules verifies the system design, and validation testing verifies the system specification. Validation can be split down into the system test, to prove that the development specification has been met and reference model testing. Reference model testing evaluates the system under test in the users environment to start the process of confirming that the customer actually asked for the product that he needs.

> Verification: 'Are we building the product right ?'
> Validation: 'Are we building the right product ?' [16]

The linkage between the development and verification phases is shown in Figure 1 'V' life-cycle on page 9. The 'V' life-cycle model, which is a variant of the Waterfall model (similar to a set of cascading waterfalls), is a sequential model of the development life-cycle. "Although by no means a perfect

representation of what really happens, this model has been in widespread use for some 15 years now," Norris [2]. "The model also helps to clarify the involvement of the purchaser and supplier; for example, whoever has produced the specification also defines the corresponding system tests and performs the testing at the same level" Daily [17]. Myers [18] uses a similar diagram to show the correspondence between the development and testing process, although he also includes the end user at the top of the 'V'.

Other life-cycle models include incremental and evolutionary, which are cyclic models applied to projects where fast development (RAD) techniques are used because the requirements are not complete or changing [17]. The testing process control research within this thesis can be applied to other life-cycle models, but the case study and measurements are based on the 'V' model.

2.1 Introduction

The definition of testing should be agreed and well understood, but this is not the case. The British Computer Society are unclear on the issue "software testing is at least the activities described in this book "[19].

Myers [18] says, "testing is the process of executing a program with the intent of finding errors" he then goes on to describe poor definitions of testing, "testing is the process of demonstrating that errors are not present; the purpose of testing is to show that a program performs its intended functions correctly".

Abbott agrees with the impossibility of the poor statements, "if software cannot be shown to be incorrect, it must be correct. But not practicable to test software by inputting all conceivable combinations of data. Testing therefore can never provide conclusive proof of correctness"[20].

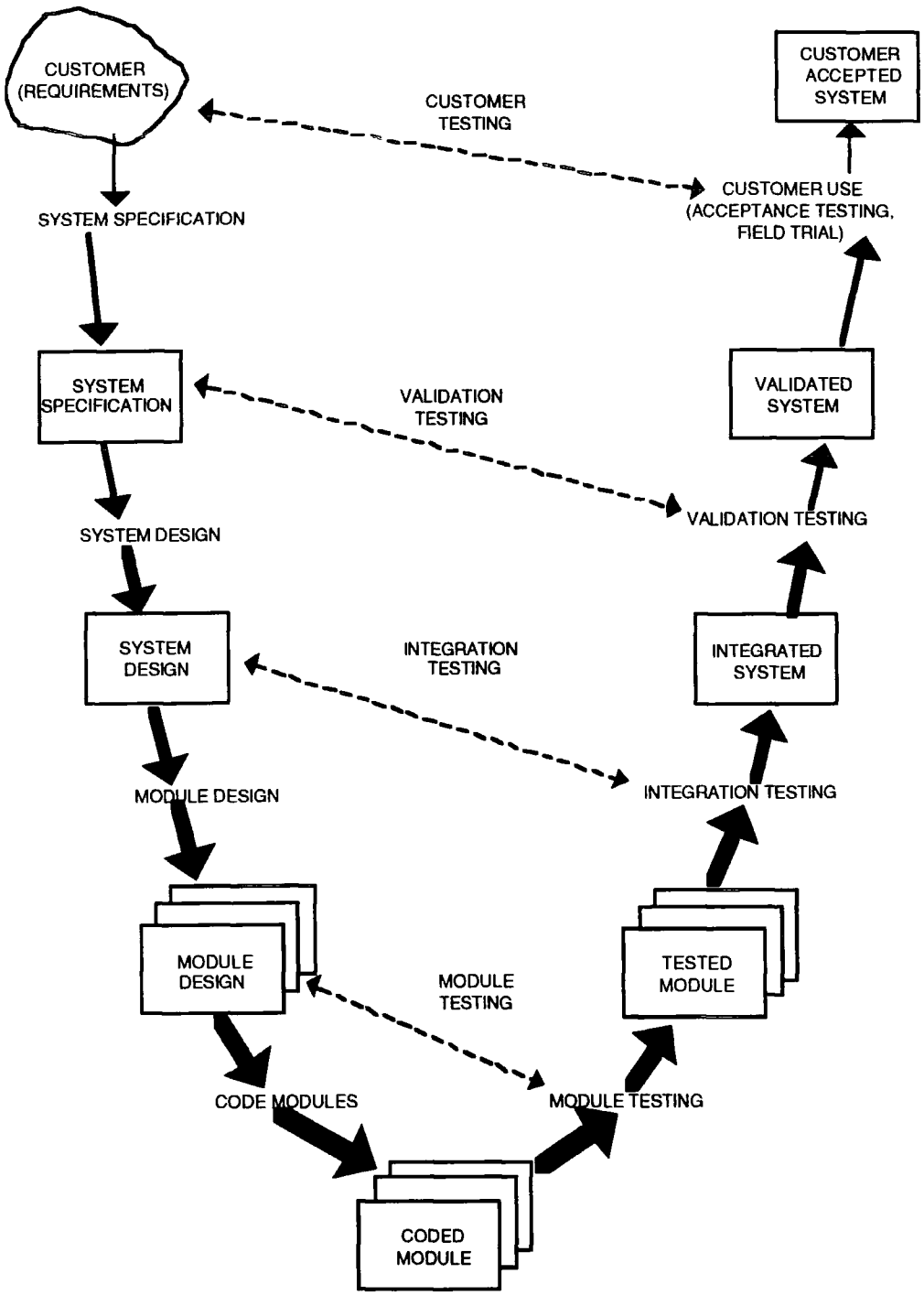


Figure 1 'V' life-cycle model

Deutsch does not agree, "testing is defined as the controlled exercise of the program code in order to expose errors. When, according to pre-established criteria, the number and severity of errors fall below a specified threshold, it is

normally concluded that proper operation of the software has been demonstrated"[21].

The IEEE explanation seems almost to fit one of Myers 'poor definitions', "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results"[22].

To complete this range of views on testing, DeMarco dismisses testing apart from the catastrophic faults, he says that "software testing is essential because it removes a particularly irksome kind of defect, the kind that brings the system to a halt. But, in terms of its effect on defect density, testing borders on the irrelevant. The only way to make a drastic improvement in the quality of the code that comes out of the testing process is to make a drastic improvement in the quality of the code that goes into the testing process"[23].

Roper [24] agrees with DeMarco on the impact of testing on the defect density of code, " testing is just sampling".

Before deciding on a definition for 'testing', an exploration of the key components from the definitions quoted so far, to check on some of the basic assumptions in use. The relationship between errors, faults, bugs and defects.

Error, two explanations from the IEEE , "first - a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. Second - human action that results in software containing a fault"[25].

Two from Myers, "one - a software error is present when the program does not do what its end user reasonably expects it to do. Second - an error is clearly present if a program does not do what it is supposed to do, but errors are also present if a program does what it is not supposed to do"[18].

DeMarco sees defects and errors as the same, "a deviation between desired result and observed result," he also has an unusual comment to make about bugs, "a bug is something that crawls of its own volition into your code and maliciously messes things up. It is certainly no reflection on you; it could happen to anyone. But a defect is your own damned fault" [23].

Another view of defects [26], "a defect is the evidence of the existence of a fault"; so what is a fault? "a fault is an error in software that causes the software to produce an incorrect result for valid input."

IEEE glossary entry for a fault [25], "an accidental condition that causes a functional unit to fail to perform its required function. A manifestation of an error in software. A fault if encountered, may cause failure."

For this thesis, Figure 2 Problem Terminology, page11 shows the relationships between errors, faults, and defects. An error in the design or coding process results in a fault within the design or code. This may be exposed as a defect, a failure or unexpected/unpredicted result.

Roper adds a possible multiplication factor to this relationship, which should not be overlooked, "one error may lead to several different faults, each of which in turn leads to several different failures" [24].

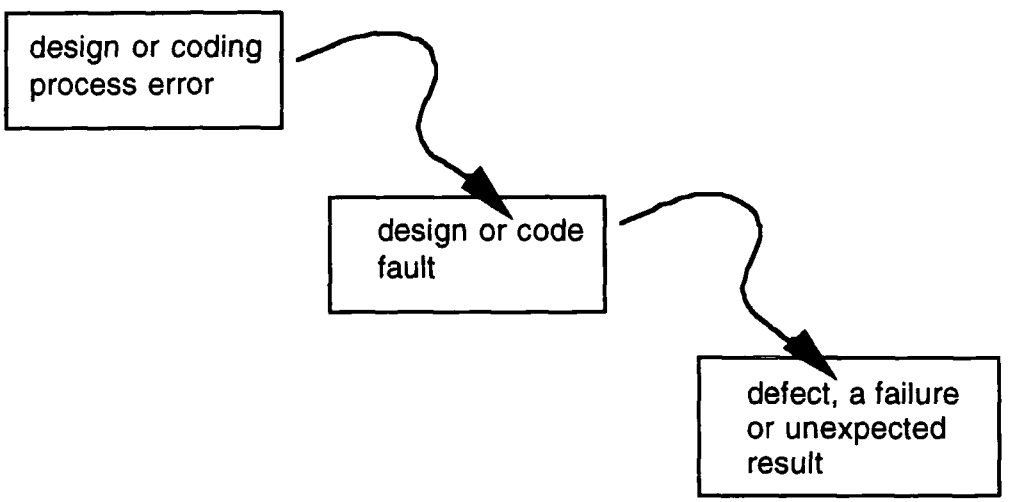


Figure 2 Problem Terminology

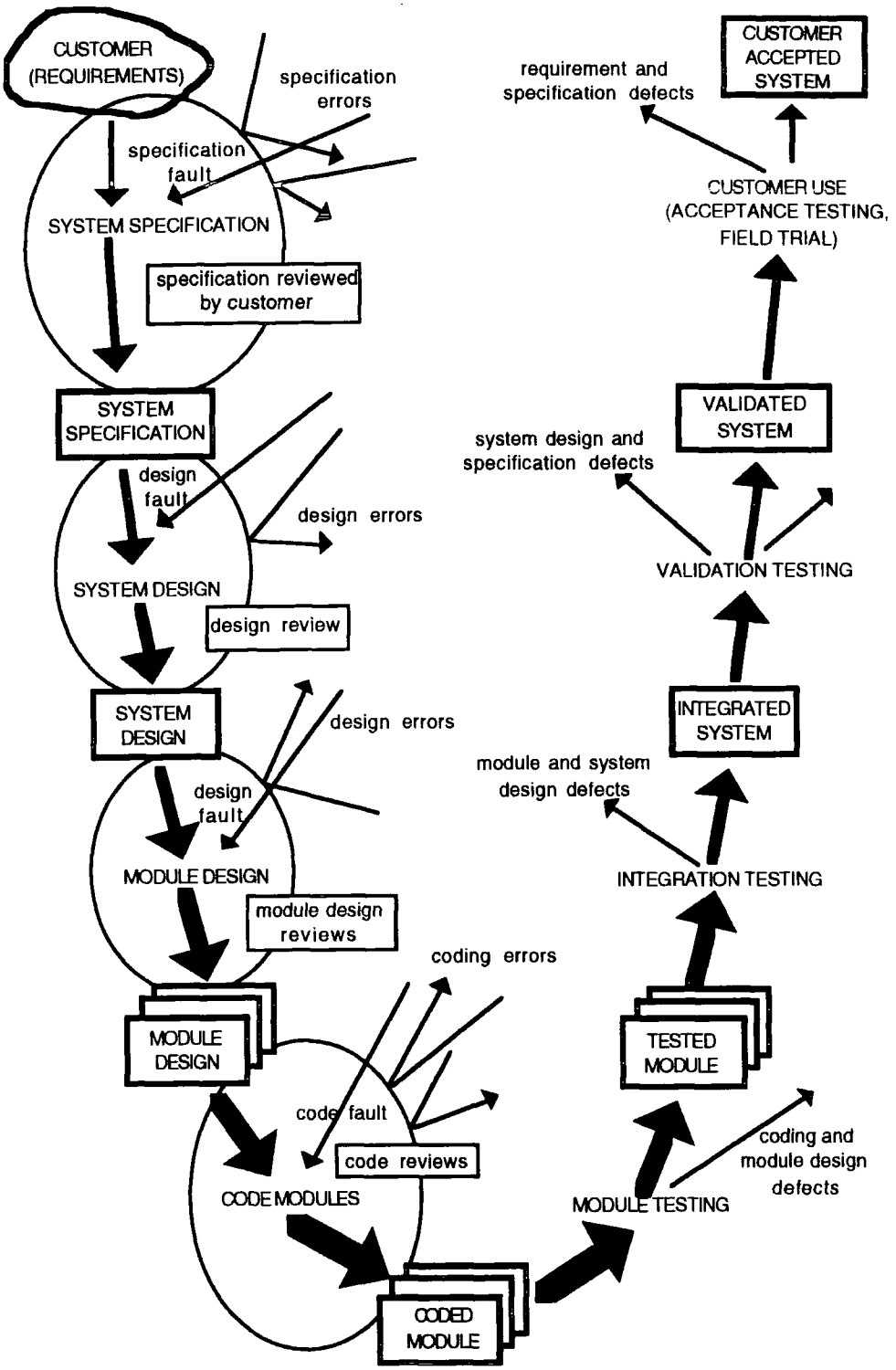


Figure 3 'V' model; error, fault & defect

The use of this terminology applied to the 'V' life-cycle model is shown by Figure 3 'V' model; error, fault & defect on page 12.

Specification errors will be captured by the review procedure or result in a specification fault. These defects in the system will not be exposed until validation testing or customer use. Down the left hand side of the diagram the thickening arrows represent the increasing number of faults in the system that have not been detected by the reviewers. The arrows rising up the right hand side, indicate the discovery of defects in the system during testing and there subsequent removal. There are faults within a system because of errors made during the first four phases of development, that where not exposed during the verification of these phases. The faults continue to exist because the testing phases did not have sufficient coverage to expose them, or a test that should have exposed a fault was itself flawed.

2.2 Testing defined

Testing has been defined a number of times in past, and those definitions have reflected the approach to testing at that time. Gelperin and Hetzel [27] list the approaches to testing that have been taken over time. "Since testing is as old as coding, most people involved with software have a mental model of testing. However, there are significant differences in the definition of a scope and objectives for testing, that have resulted in different definitions of testing success. Unrecognised differences in mental models have resulted in confusion between and among customers, managers, analysts, programmers and testers." They divided the past four decades into periods based on the most influential testing model at that time:

Testing phase models:

- 1956 The debugging oriented period
- 1957 - 1978 The demonstration oriented period
- 1979 - 1982 The destruction oriented period

Testing life cycle models:

- 1983 - 1987 The evaluation oriented period
- 1988 - The prevention oriented period

During the first three periods (until 1982) testing took place after implementation, during a phase of the project titled testing. The testing life cycle models (from 1983) considered testing as an activity that is spread across all the phases of development and not just the time spent with the product once it has been completed.

There has been an expectation that 'formal methods', proving a program mathematically, would provide the next step in software testing, but this has not been delivered for industrial projects at acceptable cost.

-

The Debugging period

The main concern at this time was the hardware and the reliability of the product. The terms 'testing' and 'debugging' where interchangeable and used by the programmer who 'checked out' his program.

The Demonstration period

Definitions started to appear:

testing - make sure that the program solves the problem

debugging - make sure that the program runs

The goal for testing at this time was to show that a program did not have any faults, a demonstration of it working correctly. This was not a very effective method of selecting test cases and data as there was no incentive to find situations where the program would fail.

The Destruction period

The definitions changed:

testing - fault detection

debugging - fault location, identification and correction

The goal now was to demonstrate that a program did have faults. This followed Myers [18] view that testing was the process of executing a program with the intent of finding faults.

The Evaluation period

Testing became a product evaluation process spread over the complete life cycle. This covered the analysis, reviews, walk-throughs and testing activities during all development phases and not just the testing of the completed product.

The Prevention period

Test planning, testability analysis and test design have side effects that improve the quality of software specifications. They reveal incompleteness, ambiguity, inconsistency and incorrectness of the specification by taking a different view of the specification. This also has the added advantage that it is cheaper to correct these mistakes at specification than to discover them in the final product. Prevention also includes the feedback of problems found with the development process to stop the same problems from occurring again in another product.

A definition of 'testing' for this thesis:

> The evaluation of a system or subset of a system throughout the development life cycle, to establish a level of confidence in the system and the development process.

This definition has components from the 'evaluation' model as the whole life cycle is covered during the testing, but mostly from the 'prevention' model as the development process is being considered along with the product.

By analysis of the reported defects an estimate can be made for the number of faults in a system and the specification, design or coding process errors responsible. Scaling up of defect numbers to estimate the total for the system is theoretically impossible because of determining an accurate size measure for the sample and a truly representative sample to test, but practically may be close enough to be of use.

The definition of testing given above adds a reason for projects to test the resultant products. When there is a reason for testing there must also be a reasoned approach as to when the testing should be stopped.

Criteria for halting testing range from:

running out of money and/or time, which must be the worst possible stopping point unless the reason for testing is to spend a fixed amount of money, to finding every fault in the system, requiring infinite resources.

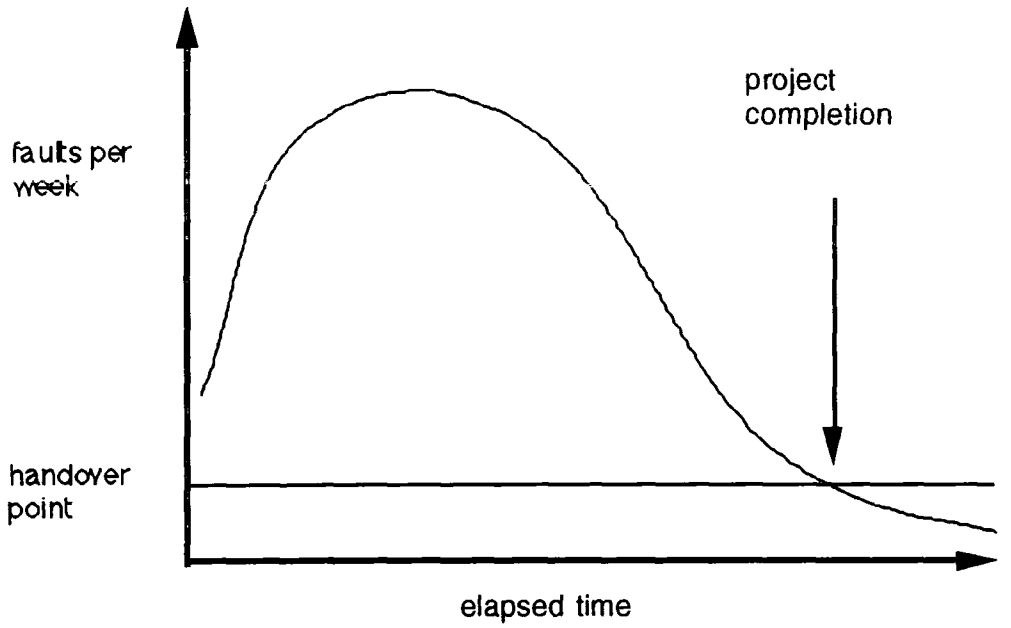


Figure 4 Defect rate

Measuring the faults found per week can give an indication as to the point of handover to the next stage or to the customer if it is validation testing that is being assessed. Figure 4 above, shows a typical plot of defects/per week with a level at which handover will be accepted [23]. Once over the hump of the graph the date at which this level will be reached can be predicted. A useful technique, but it does rely on the number of defects found being used to predict the number of defects uncovered, and the resultant quality. If a poor set of tests are used, inadequate testing will give a false prediction. The only way to be sure is to assess the testing as well as the product under test. Mutation testing [see Appendix A] [28] is a technique that can be applied during validation testing and coverage analysis during module testing to assess the level of testing achieved. Appendix A, covers a number of standard testing techniques and Appendix B IEEE standards for test documentation.

It is important to develop testing environments and test cases for products that can then be handed on to a support group responsible for the ongoing maintenance of a product. A regression testing environment and a full set of tests to run within that environment will save the support group a considerable amount of money and time [29].

For most products it has been shown that 2.3 times the development cost of a product will be spent on maintaining that product out in the field, 30% development, 70% maintenance [3]. It is unlikely that any product managers plan to spend such a large amount on maintenance, but that is what happens in practice. A sizeable proportion of this large amount, will be the cost of re-testing the product every time a change is made; scope for savings with a regression testing environment.

Decisions on regression test coverage are best made jointly, developer and customer, as the main objective of the testing is to maintain the customers level of confidence in the product.

2.3 The testing process

The testing process is an important part of the product development life cycle, and the quality assurance of products. Testing is a technique used to asses the quality of a product. A product is likely to consist of a software component running on some hardware.

Testing is carried out by the development organisation to ensure that the product performs as specified, which will include the product reliability. As maintenance costs can account for 70% [3] of the overall product life-cycle costs, attempts at reducing this overhead by ensuring the product is correct are worthwhile. The customer may also run his own acceptance tests or pay an independent testing organisation to checkout his proposed purchase.

Identifying the customer and his needs is critical if the right product is to be developed. The customer initially might not have been identified, so a profile put forward by a marketing department may be used. IT products might well be sold direct to customers, and products for a companies operational departments might well end up providing services to customers using an IT system.

2.4 The product development life cycle

There are a number of methods used to describe the product development life cycle, although the majority are variations of the waterfall model. The variant shown in life-cycle, page 9, emphasises the verification stages of the life cycle and the linkage to development. Each phase of the 'V' development life-cycle is explained below and includes the verification activities. "Software quality assurance is an 'umbrella activity' that is applied throughout the software engineering process" Pressman [7].

2.4.1 Requirements

This is an explanation of what the customer requires, in their terminology. In the case of a new service, there is likely to be two different customers. End customers who are requesting the service from a company; and a department within the company who will run the service and let a development contract for the development of a system to provide the service. The department becomes the customer for the development as well as the service provider for the external customers.

Ideally the requirements would come from the customer in a form that enables a direct translation into the specification. The other approach is to offer the customer a requirements proposal that can be modified as they clarify their ideas. This approach is taken with technology lead developments where the customer is not aware of new possibilities.

"The requirements specification always fulfils the following two roles:

It provides the primary input to the design phase.

It gives a baseline against which acceptance tests are carried out." Norris [2].

The BCS [19] describe the criteria for testing a requirements expression " The testing criteria used at this stage are intended to prove that the Requirements Expression covers as many requirements as are realistically within the scope of the system. We look at completeness, consistency, feasibility, testability and referability".

2.4.2 Specification

A document that will specify the product in an exact manner for use by the development team. All to often development projects fail because the specification was not up to standard. It is essential that the product is understood and can be specified before development starts [19].

It is sometimes argued that the software specification and the requirements definition are one in the same thing, but Sommerville says that there is a clear distinction, " The principal function of the requirements definition is to set out those services which the software must provide the user. The requirements definition should be a user oriented document and must be expressed in terms which are understandable to him. On the other hand the software specification is intended for the software designer rather than the user. It is made up of abstract definitions of software components, not user services"[8].

The effort required to produce a quality specification and the attention to detail needed is often underestimated. Specifications should not include details of how the system is going to be implemented (unless the customer has specific implementation requirements) but what the system will do.

Specification attributes [19]:

Consistency of meaning
Freedom from contradiction
Unambiguous
Testability
Referability
Feasibility

The document should also be controlled under the project configuration management system (CM). The specification will no doubt change and therefore require the CM control to link it to the other changed design documents. This will assist traceability and provide an up to date specification for use during validation and ongoing support [17].

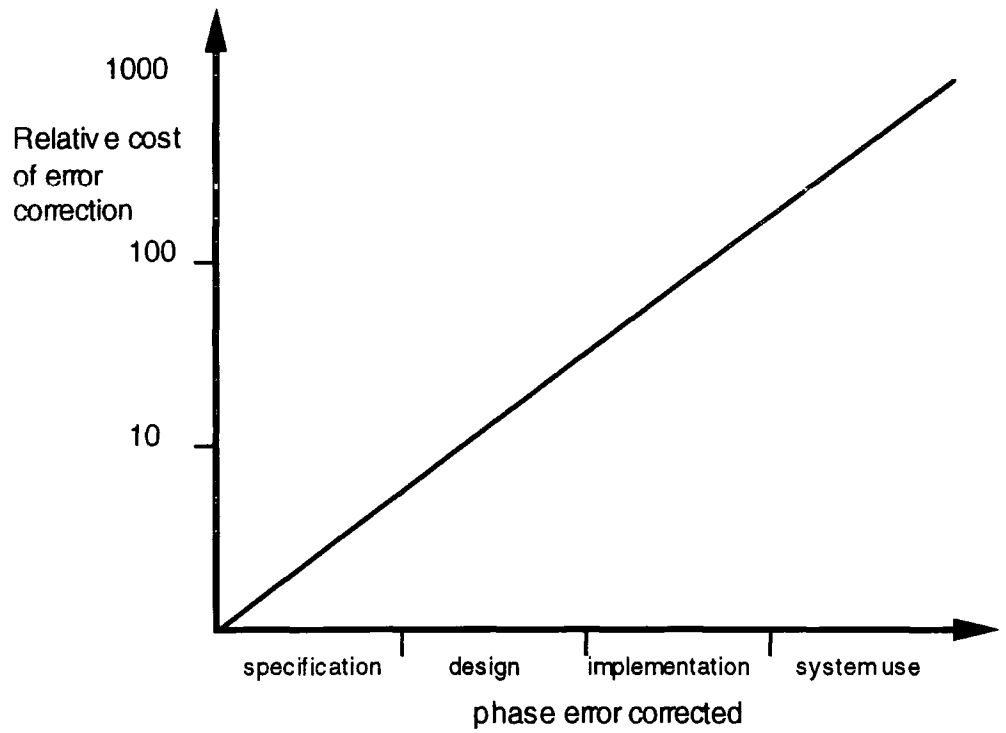


Figure 5 Cost to fix

One of the main trouble spots within development is the specification. If the specification is wrong a lot of money is wasted before the problem is discovered. The relationship between discovering a problem and the cost of fixing is shown by Figure 5, Cost to fix, on page 20 [12]. "A large data base of experience has verified that it is less costly to discover and rectify errors early in the development cycle" [21]. The earlier a mistake is made the later it is before that phase of testing is reached that will target finding the problems from the earlier implementation phase. Mistakes in coding are usually found at module test, mistakes in specification are not found until validation, because that is the first point in the project that the system comes together and can be verified against the specification, see Figure 6 Error linkage, page 21. Daily whilst discussing the 'V' model says "The approach to testing is based on the results of each integration and build phase being assessed against its corresponding design or specification phase at the same level" [17].

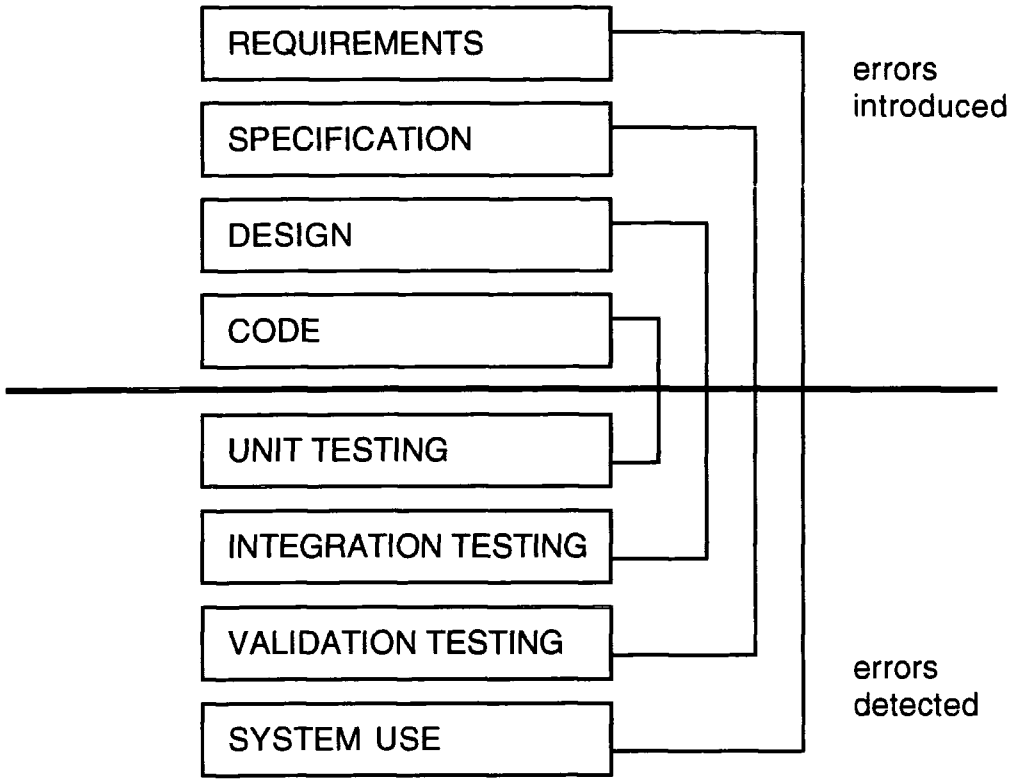


Figure 6 Error linkage

2.4.2.1 Verification activities during specification

The main input from verification engineers during specification is an analysis of the specification for testability [30]. This normally takes the form of technical reviews, and must be the most effective method for tightening the specification[17]. Testability analysis will force the clarification of any ambiguous statements, as it is not possible to test a vaguely specified function properly [19].

As the specification progresses, so should the top level test plan; see Appendix B for test plan contents to meet IEEE829 standard [22]. The reason for starting work on the test plan before the specification is complete, is due to practicalities and cost. There is not much point in specifying a product that can't be tested, or a product where the cost of testing would be so prohibitively high, that the business case for the product fails. The verification engineers provide this feedback through the quality review process.

2.4.3 System Design

The system design process defines how the product will be realised. The functionality split between hardware and software will be documented. Also the software structure down to the module level, the module interfaces specified, and a hardware overview produced. If the software is designed to run on standard computing platforms (pc, client/server, Sun UNIX etc. [31]) then a specification for the equipment is needed. " Design provides us with the representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirement into a finished software product or system", Pressman [7].

2.4.3.1 System design verification activities

The top level test documentation will be completed, and the validation test plan started. As validation testing is a black box testing activity, tightly coupled to the specification and not requiring visibility of the design, adequate information is available for starting the production of test cases. The specification will change as the design is refined; the validation test plan follows these changes as an item under CM control. The verification engineers will also take part in the technical review of the system design.

The validation of a software design is intended to achieve two objectives, Sommerville [8]:

" To show that the software design is 'correct'; that is, the design should correctly implement the intentions of the designer.

To show that the software is valid. That is, it should be demonstrated that the design meets the requirements in full".

2.4.4 Module design, low level design

This is the detailed design of modules down to the level of detail where the information for coding is complete. In parallel the hardware engineers will have

produced a detailed hardware layout. " During the low level design, the internal logic for each component identified in the top level design is developed. Detailed data structures are developed and the units that compose each component are identified and designed. The primary output of the low level design phase is the low level design document" [32].

2.4.4.1 Module design verification activities

The next level of verification plans, the integration test plan, proceeds as soon as the system design has been issued. The order in which modules are integrated together is carefully planned to maximise effective interface testing and minimise cost [7]. Feedback to development teams of the finalised order is important, as the first modules to be integrated should be developed early, to allow the earliest possible start date for integration. The development engineers will be producing module test cases as a parallel activity to designing the modules.

2.4.5 Implementation

The coding of the module software following the module design documents, takes place while the hardware is constructed. "Good programming - the production of reliable and maintainable programs - is a language independent process. Whilst high level languages such as Ada or Pascal simplify the process of converting a design into implementation, there is no reason why good programs may not be constructed in any languages whatsoever" [8].

The adding in of extra features by the development team or trying to sell side effects as extra features to the customer is a sign of problems to come. Not only will these late additions have not been thought through properly, and therefore will not work, but the customer has not asked for them; he may not want them.

Development of software is a complex process that will always lead to some human error. Syntax errors will be picked up by the compilers; the errors that compile and run but with subtle differences to the required code, are the tricky

ones to find. "When the code is complete and has compiled without error, the development engineers conduct a code walk-through or peer review with one or more engineers. The purpose of the walk-through is the detection of errors before the unit enters any part of the test phase" [32].

2.4.5.1 Implementation verification activities

Preparations for integration testing should now be complete, with the test environment in place and test cases written. Verification engineers move on to start producing the validation tests.

2.4.6 Module testing

This testing is normally carried out by the development engineers, and will confirm that the modules function as documented in the module design, plus conformance to the interface specifications. As modules pass the prescribed tests they are passed to integration, failed modules are returned for coding changes or design and re-coding for major problems.

Module testing is a white box testing procedure (testing with knowledge of the internal operations), carried out by the developers [33]. Although the developers are the best people to carry out this work from the point of view of understanding the code, there is a problem - cognitive dissonance. People can rarely see their own mistakes, and are therefore unlikely to find any problems in their own software. There is another reason why developers do not make good testers for their own code. They will have spent a fair amount of time developing their software, trying to show that it works and therefore trying to prove that it doesn't work (this is the principle behind testing) is very difficult, indeed impossible for some developers who get very attached to their code [8].

2.4.6.1 Module testing environment

The best method for testing software modules is to simulate the environment that they will be operating in. The software module is run within a computer that presents the correct interfaces to the module code, sometimes called a test harness. The interfaces can then be driven with sets of data to exercise the module code and check that the correct functions take place with the relevant stimuli. Another advantage of this method of testing is that conditions that would be difficult to produce in the real product environment can easily be presented to the software via the drivers in this simulation environment. The test data should be stored for use later in the products life, so that enhancements to the system can then be regression tested at module level [19].

2.4.7 Integration

Modules are linked together on the hardware and the interactions between them tested. The order that the modules are brought together needs to be carefully planned in advance of this stage, as the order of integration will determine the order that they will be developed in.

There are four types of integration [32] [18]:

1 Top down

Starting with the software nearest to the user (the top of the software) the modules are integrated moving down through the layers of software. Stubs will be needed to respond to software calls to the lower levels from the modules under test.

2 Bottom up

The modules furthest away from the user but closest to the hardware are tested first followed by further layers being built on top of the tested software. As the user interface or control software will be the last layer to be added, drivers are needed to drive and test the lower level software.

3 Thread

Any modules associated with a particular function are integrated and tested before moving on to the next function or thread. The capability of the system is gradually increased as more functions are added. Stubs or drivers will not be needed so long as the module design has been based on functions.

4 Big bang

No stubs or drivers are needed for this type of integration, all the software is loaded at the same time, and the machine switched-on to 'see what happens'.

The big bang approach to integration is not a very successful method of integration as it is difficult to isolate the problems that will be found, and a large number will be exposed in this first coming together of the software. This approach does sound attractive, no stubs or drivers to produce, just load all the software together. "The entire program is tested as a whole. And chaos usually results!" [7]. If a large number of problems are found and they can't be isolated then the only path to take is to re-plan the integration stage based on one of the other remaining methods of integration. This adds a large time and cost penalty to the project, project management will try to cut corners on this testing and a poor quality product finds its way out to the customers.

Hetzel [34] identifies two major problems with integration testing in practice:

"The first is integrating modules that have not been properly unit tested. Just as a single bad apple will quickly rot the entire barrel, it takes only several poorly tested modules to ruin a solid integration testing plan. Some form of quality check or acceptance must be made by the integration test team to ensure that modules are ready to be integrated. The second problem is failing to treat integration testing formally enough. Like all testing the integration tests must be designed and planned to ensure thorough coverage".

The integration stage will verify that the overall design has been implemented correctly and that the interaction between modules and the interfaces are as specified [19].

2.4.7.1 Integration testing environment

Test stubs or drivers will be required depending on whether top down or bottom up integration has been chosen. Also needed will be hardware test equipment to check that the software is controlling the hardware correctly [32].

Royer [32] considers the problem of new or special hardware, " it is here that the test engineer is most likely to encounter the 'chicken and egg' situation: the software is the best vehicle for verifying the hardware and the hardware makes it easy to verify the software. Which? There's no correct answer; the test engineer must select the most appropriate sequence".

One approach to system integration with new hardware is to integrate all the software modules first before loading them onto the target hardware. This requires an integration environment that will simulate the target hardware for the integration of the software modules. The advantages offered by this method include, early integration of software before the hardware has been completed, and the possibility of automating the integration testing as software will be used to simulate the hardware. The disadvantage is the effort to construct the simulated target hardware, and the availability of computing power to run the simulation and software under test in real time [8].

2.4.8 Validation

The complete product is tested to gain confidence that the customers requirements have been met, and the products 'fitness for purpose' [35]. This type of testing is often called 'black-box' testing or system testing. If the product specification is of good quality then validation testing is a straightforward but, perhaps, lengthy process [34]. A drop in quality at this stage is normally a problem with project finance. "The process of system validation is laborious and is often the most expensive stage of software development" Sommerville [8]. Validation being the last stage of the development most of the money, time, and contingency will have been used so

the project management are pressurised into reducing the validation testing. Norris [2] discusses the squeeze on the testing stage after overspending on the earlier stages, "This is borne out by recent surveys, which indicate that there are relatively few practitioners in the UK industry who consider that enough time is allowed for testing - typically 15% of time and effort is considered a generous allowance. To learn how the software industry, as a whole, is likely to develop in this country, one should look at the examples set by leading companies in the USA, where testing is treated with as much respect as any other part of the software engineering discipline and a 40% allowance for testing is widely accepted".

This being the last stage before customer trials, any problems that are not found in validation will no doubt be found by the customer. System testing is really split between integration and validation, as the only way to confirm the system design and its implementation during integration involves gradually building the system and testing until full system tests can be completed [34]. The integration system tests might well be repeated during validation, but as the software is likely to change during integration as problems are found, a rerun will be necessary on the new stable validation software. The validation testing environment is much closer to the users environment, and as such also warrants the rerunning of system tests to show that the product will perform as the user expects [34].

Pressman [7] defines validation testing; "validation testing can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when the software functions in a manner that can be reasonably expected by a customer. Reasonable expectations are defined in the software requirements specification".

The product can be considered as a black box, you can not see into it, and do not know what is happening inside the box. Tests are written from the details in the specification to show that a particular function is present and operates correctly, without knowing how it is functioning internally [34]. Due to the size and complexity of modern systems it is not possible to run all the tests that would be needed to gain 100% coverage of the system. A set of priority rules (listed below) can be applied to select the subset of tests that will provide the best possible coverage for the time and money available. AT&T [36] have

made use of similar priority rules to cut the testing on one project from 250,000 test cases down to 40,000 and still achieve effective testing results.

> Testing that something doesn't happen is just as important as checking that something does happen.

It is not only necessary to prove that the system operates as expected, but also to show that it has no unwanted extras. It is very easy to spend all the preparation time developing test cases to show the system functions in operation and not even consider checking that the system does not do something disastrous. Roughly equal time should be spent on these two approaches.

> Testing the system's capabilities is more important than testing its components.

Users of a system are much more interested in progressing their total job, and will worry less about a screen layout being wrong if they can complete the job in hand. The objective then is to test the basic job that the system is performing from start to finish.

> Testing old capabilities is more important than testing new capabilities.

Users of a system become annoyed if a facility that has been working well, fails when they are provided with a new release of software. This will cause them problems as they may have to change their working practices, whereas a problem with a new feature will not put them out as they did not have access to it before.

> Testing typical situations is more important than testing boundary value cases.

Validation testing involves understanding how the user will use the system, and making sure that the system will not fail under those conditions. Boundary value analysis is an important technique which should be applied during module testing, but is not applicable to validation testing as the values selected are unlikely to be typical user values.

Experienced verification engineers using error guessing techniques to provide provocative testing, will stretch the system further than the system tests [18]. But these tests still need to be planned in advance and not become an ad-hoc

activity on a user interface. In addition to functionality there are a number of abilities that are investigated during validation; maintainability, installability, extendability and usability. Performance testing and the product abilities are explained in more detail below, as they are just as important to a customer as the products functions [18].

2.4.8.1 Performance

The testing of response times and throughput rates over a range of system loading and configurations. There are two further performance testing areas, volume and stress testing.

Volume testing

A throughput rate at or near the maximum for a period long enough to give a high degree of confidence that the system could carry on at this rate. As volume testing is expensive, tying up valuable models, a balance of cost against level of confidence is required [18].

Stress testing

The subjecting of the system to heavy loads or stresses. Unlike volume testing this is a short burst of peak volume load. This technique is used to explore the systems response to loads greater than those specified as maximum. The system should gradually degrade its service or not offer process power to requests above the maximum rather than shut down or fall over if a peak above the normal limit is encountered[18].

This type of testing is most applicable to real-time, interactive, and process control systems. An example is the lifting of handsets by all the subscribers on a telephone exchange at the same time, an unlikely event, but not impossible. The telephone exchange should accept and route the maximum level of calls and reject any above that maximum limit.

2.4.8.2 Usability

A subjective form of testing unless you have an end user to sit at the system with the manuals. Otherwise you asses whether the Human Computer Interface HCI [37], the user interface (including user manuals), matches the expected user intelligence and background experience of such systems. A check is made to assess the implementation for a consistent user interface; format, style, syntax, semantics, abbreviations, short cuts, meaningful outputs and straightforward error messages [18].

A secondary effect of usability testing will be to validate the user documentation against the implemented system. If an end user or a suitable substitute is available, provide them with a configured system and a set of tasks that represent the expected use of the system. Monitor the exercise for difficulties and misunderstandings encountered with the system.

2.4.8.3 Reliability

Hardware reliability can be assessed by accelerated ageing using temperature cycling and other environmental stress conditions, software is a little more tricky. Musa [38] says that software failures are different to the 'wearing out' process of hardware, "The large number of possible states of a program and its inputs make perfect comprehension of the program requirements and implementation and complete testing of the program generally impossible. Thus, software reliability is essentially a measure of confidence we have in the design and its ability to function properly in all environments it is expected to be subjected to".

Weekend runs of systems under load is quite often the method used to gain confidence in a systems reliability, if a mean-time-to-failure (MTTF) in hours has been specified. Splitting the MTTF down into levels will assist the reliability measurement.

For example, a set of reliability requirements for a telephone exchange:

10 entries per hour in software log
1 minor restart (no calls lost) per 10 days
1 major restart (all calls lost) per 100 days
1 rollback (all calls lost, some software loaded) per 200 days
1 reload (all calls lost, major software reload) per 500 days

The first two of these will be tested during field trial, whereas a down time figure of two hours in twenty years can only be assessed by a sample period and a reliability model.

2.4.8.4 Maintainability

There are two main aspects to maintainability. The first is the support for the system, has it been developed in a manner which has produced maintainable code. This will cover design and code documentation, the structure and complexity of the code, plus regression testing coverage [19]. The best people to assess this aspect of maintainability will be the proposed maintainers. The second area to consider is the maintenance and diagnostic facilities provided on the system for the user and first line support engineer.

2.4.8.5 Installability and extendability

Installation procedures and time to install are important to the customer; what will be the down time and inconvenience caused when loading. It is also important to the selling organisation, trying to keep the installation time and therefore cost to a minimum. Installation tests that are used following customer installation to verify successful installation, will also require testing before use [18]. Once a system has been installed, its up and running, how easy will it then be to extend the system? These procedures are scrutinised and timed during field trials.

2.4.8.6 Recovery

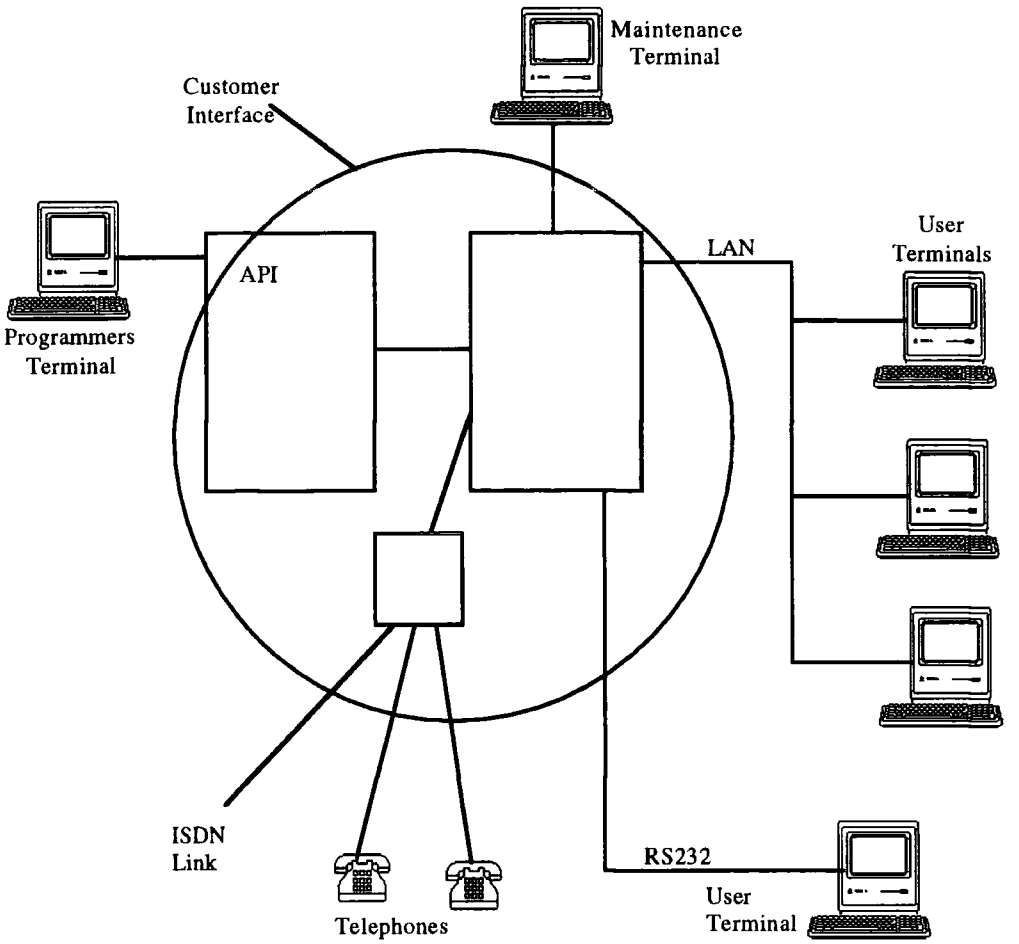
How does the system cope with hardware/software failures, and data errors. The creation of software faults, injecting data errors and simulation of hardware failures enables the systems recovery facilities to be exercised and timed if automatic recovery within a specified time is a requirement [7].

2.4.8.7 Security

Security has become much more of an issue over the last few years as more software is accessed remotely, via the WEB [31] or via LAN/WAN [39] connections. Attacks from hackers, disgruntled employees or individuals after illicit personal gain is a likely occurrence. "Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration" [7].

2.4.8.8 Validation testing environment

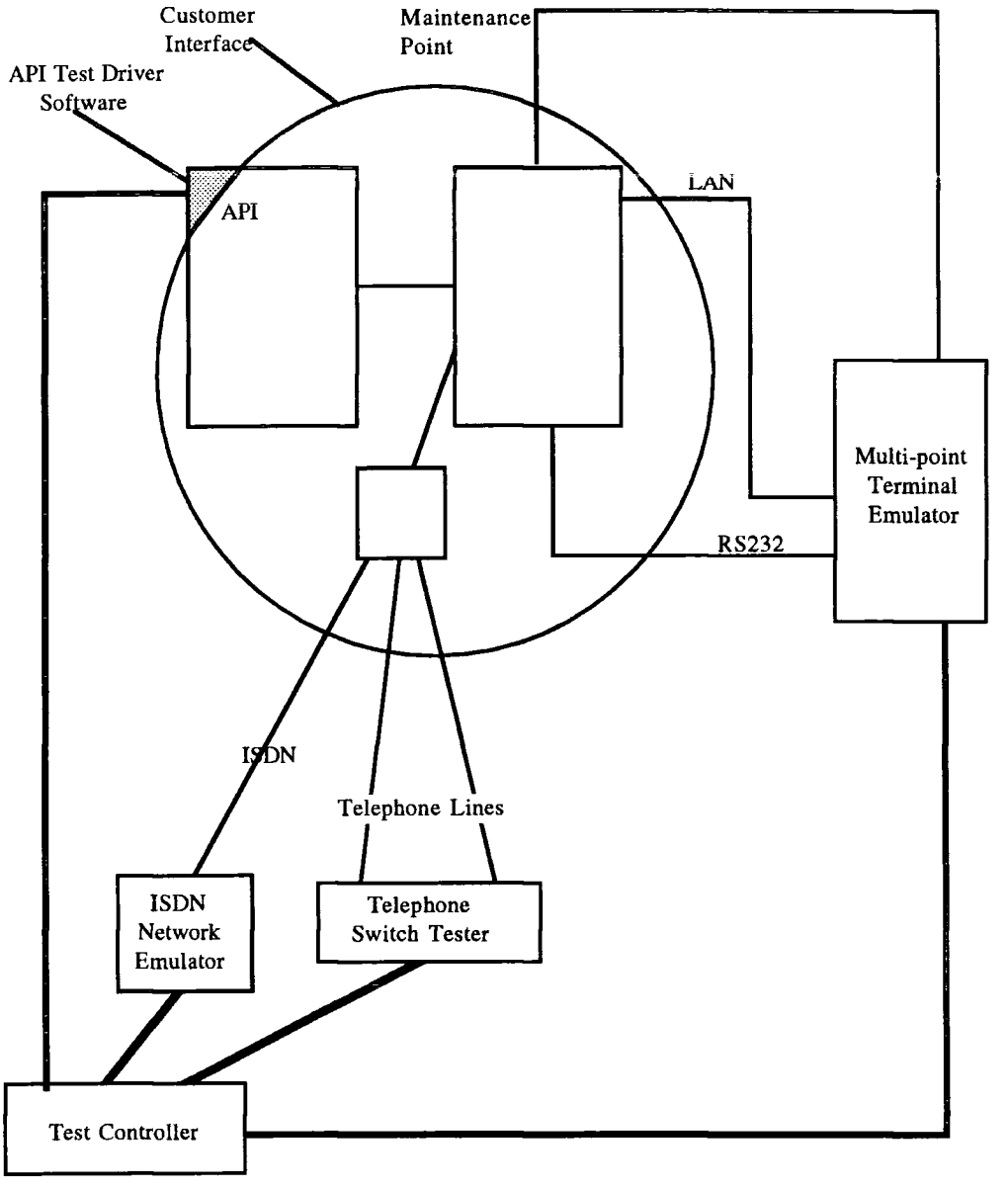
The testing of the product is from the customers view point, his interface into the system, so testers are likely to mimic a user of the system. The idea is to draw up the system and then decide where the boundary lies between the system and the user community [40]. As an example, an office system has been drawn showing the customer interface consisting of a maintenance terminal, user terminals via a LAN or RS232, telephones and an ISDN link [39]. Also shown is a programmers terminal, which has access to the high level application code via an Application Programmers Interface (API) and results in the testing boundary lying between software layers on that computer. During validation the components within the boundary are treated as a black box, only the actions and responses at the interfaces are of interest.



Once the boundary line has been drawn anything that cuts that line will require a tester or a human user, sometimes it is just not practical to provide automated testers for all of the interfaces, so a manual test is necessary. To fully automate the testing all the interfaces will need to be terminated with equipment that can analyse the actions taking place and respond in the expected way. These automated testers need to be connected together for synchronisation of testing and results, plus the ability to check that data has been passed across the system under test correctly. A common point is also needed for a test database, a method of linking the different test information for each of the automated testers to a common system function under test. This type of database would link the specification, functions under test, to the test cases for the automated testers involved and the results obtained [34]. Standards are now being developed for interconnecting test tools [41].

One purpose of validation testing is to simulate the working environment and emulate the users of the system. For example, a background load applied to the system while individual functions are exercised, rather than an idle system. The test system will require typical user data, configuration and usage patterns so that tests can be carried out with a representative user load as background. It is not always possible to load the system sufficiently by providing external stimuli via the hardware of the system, due to the cost of a large amount of loading equipment. In this case message generators and test data generators are used to load the system processor [8].

Automated test case generation is the next step and has been demonstrated using a test specification (mapping input to output) to derive test cases and expected results [42]. This process still requires the production of a formal test specification, but test case generation based on output from the design/development tools will remove this activity. As an example research has shown that developments using SSADM tools could generate test cases from the logical data model, the data flow model and the entity life history charts [43].



An automated testing environment makes an excellent regression testing system and should therefore be passed on to the support group for future use in a support role. The first use of a regression testing system should be validation, where the regression testing system is proved along with the system under test [29].

2.4.9 Customer acceptance

"Generally the acceptance test is just what it says: if the system passes the acceptance test the client will accept it: if it doesn't they won't. It is now that we realise the importance of having a system specification that is testable" [19]. A series of field trials or system demonstration are the normal methods of establishing if the customer requirements have been met. Alpha trials are normally sites within the organisation that developed the product, beta trials rely on 'friendly' customer sites outside the development organisation. The installation and support arrangements are also tested during the field trials [7].

This phase of the life-cycle should be a demonstration of the system working and not part of the suppliers testing, as Royer says "with the customer in attendance at a formal demonstration, and with acceptance of the software on the line, it should be clear that we're not looking for errors. That is, the last thing we want is a surprise from our system" [32].

If the product has been developed for a particular customer, he might impose a set of acceptance tests, require access to the results from the validation tests or have an independent assessment [44]. Acceptance that the development contract has been met, and payment is due, is likely to be tied to the results from the acceptance testing. Product acceptance testing, in this case, will form part of the contractual terms applied to the project [19].

This explanation of the verification life cycle may seem straightforward and in theory products should end up error free. Unfortunately many problems are still found in new products. "Not only are more errors made in the analysis and design stages as in coding (typically a ratio of 2 to 1), but while most coding errors are discovered during development (around 75%), most analysis and design errors (70%) escape to plague the user" [45].

2.6 Summary

This chapter has concentrated on defining some of the commonly used terminology for testing. The terminology is not universally agreed amongst testing practitioners and is one of the reasons why it has been necessary to define them for this document. The 'V' development model has been chosen to illustrate the verification phases as it is a well known and understood model. The testing aspects of each phase of the 'V' have been explained.

Other development models such as evolutionary delivery or prototyping will also have a similar set of verification activities associated with the development process. The main point to be made from this broad look at verification, is the emphasis on gaining a level of confidence in a module of software or a complete product, and not the impossible task of trying to prove that the software is 100% correct. This arises due to the impractical nature of trying to prove that something as complex as a computer program is correct and also the reality that no one would be prepared to finance such a costly task even if it were possible.

3 Process control and measurement

"You can't control what you can't measure" [23], a statement often used, but not so often acted upon.

The aim of this study is to improve the quality of a product through improved verification. The improvements must come from improved efficiency and effectiveness of verification, and not just an increase in testing resources and cost. Metrics are used to provide the improvements and also to measure present verification methods and show the impact of new techniques. Metrics on their own will not provide an understanding of the underlying process, just measurements of process or product attributes. The requirement is for a model constructed from metrics but also linking the metrics to each other. To take this concept further an appreciation of metrics and modelling is essential, and will therefore be covered over the next few pages.

3.1 Software metrics

"A software metric defines a standard way of measuring some attribute of the system development process" [46]. An attribute might be, the number of defects found, the cost of development or the code size. A system view from DeMarco, "a metric is the number you attach to an idea . More precisely, a metric is a measurable indication of some quantitative aspect of a system"[23].

Software metrics fall into three categories [47] [48]:

1. Process metrics which quantify attributes of the development process and are usually associated with some timescale.
2. Product metrics are measures of the software product and deliverables; complexity, size, development cost, field failures are all product metrics.
3. Resource metrics are used to measure the entities that are required by a process activity. Examples include personnel, tools, and methods.

This is not a particularly tight classification and some software metrics might apply to the process and the product. If this is the case then the major influence should be used as a guide to classification.

Metrics may consist of a single measurable attribute (primitive), or a mathematical combination of a number of measures (computed).

For classification purposes there are a number of groups of software metric types [26]:

- Size metrics. These are some of the most used metrics, as all programs have a size, it is easy to measure once the program has been produced and there is a high correlation between size and the cost of development. One of the popular size metrics is the number of lines of code, but this can also be very misleading for comparison purposes unless a line of code is actually defined.
- Data structure metrics. A measure of the amount of data input, output or processed by a program.
- Logic structure metrics. Included in this category would be reachability, number of paths and decision count metrics.
- Effort (cost) metrics. This may seem to be an easy metric to collect, but trying to determine the actual effort from the maze of time sheets, overtime, interruptions, long hours, and informal discussions is not an easy task.
- Reliability and defect metrics. Defect count, a measure of the number of errors that lead to design and code changes, is a standard component or a reliability prediction. The reliability is normally expressed as a mean time to failure (MTTF).

- Design metrics. A structured design is related to five design principles, each having associated metrics; coupling, cohesion, complexity, modularity and module size.
- Composite metrics. A vector of metrics used to describe complexity.

3.2 Modelling

Modelling when applied to software engineering provides a scaled down version of a product or process that enables an early or reduced cost view of the product, in advance of the main project cost. Problems found with the model can be solved before any large costs are incurred on the project.

"Modelling gives an inexpensive way to study essential aspects of a system long before the system is built" [23].

DeMarco goes on to explain the essential aspects as:

- system behaviour - (embodied in a specification) specification model
- system internal organisation - (embodied in a design) design model
- project organisation - (embodied in a project plan) project model

Models are used to provide the relationship between various system development attributes. There are a number of approaches to developing a model; theoretical, data driven, or a combination of both of these. For example, take a relationship between the cost of developing a product and the resultant code size of that product. A theoretical model, based on hypothesised relationship between cost and size, might be:

$$\text{cost} = size^2$$

This has been developed independent of any data, whereas a data driven model, the result of statistical analysis, might be:

$$\text{cost} = size^{0.756}$$

A compromise between these two approaches is to use intuition to determine the basic shape and then data analysis to provide a constant:

$$\text{cost} = 1.5 \times size^2$$

Although the model must be true to the data, to be considered useful it must also be simple to understand and use. The above model would be described as a static single variable model by Basili [49]. He also presents a static multi variable model as one with a number of parameters similar to Boehm's model [16].

A third type of model is the dynamic multi variable model. Dynamic because it produces a range of values (curve) rather than a single value, for example Rayleigh, Parr and Putnam curves. Examples can be found in the tutorial by Basili [49].

3.3 Application to software engineering

Process metrics provide the data source for a development life-cycle model. The model therefore provides the linkage between the different process metrics. Motivation for the use of a development model and software metrics comes from a need to improve the process, and the quality of the resultant products. But it is not possible to control what can't be measured, or to compare without a standard measure.

Improvements from the adoption of a metrics programme:

- better control of the project, earlier indications of overrun and problems
- better estimation of project cost and timescale
- improvements and fine-tuning of the software development process
- early prediction of product quality, especially reliability

These claims are supported by a number of software engineering 'process' books [23] [49] [16] which cover the use of metrics and the benefits found.

Grady & Caswell [46] aimed their metrics program at improving productivity and predictability. They explain productivity as a measure of output divided by input, or value divided by cost. It is interesting to note that productivity is made up not just of quantity, but also quality and reusability.

There are a great number of improvements that can come from a metrics programme, but it is necessary to decide on a reason for collecting data and

check that the cost of collecting the data is not greater than the benefits. The most efficient way of gathering data is automatically, by extending tools already in use. A size metric provided by the compiler used, does not involve any extra manpower cost or effort to collect.

There are some limitations to the use of metrics [26]:

- Models will need to be calibrated for a particular development environment, from past project data. If a model is later used in another environment it will need to be recalibrated.
- When comparing data from different models check the definitions of the metrics used. There are several definitions of 'lines of code' in use today that would give quite different values for the same piece of code. The problem is what to include as a line of code in the count; headers, comments, blank lines etc.
- Models should be used to assist managers and not to replace them. A model can produce a prediction way off target if the environment has changed, a manager can discount a prediction he does not believe.
- Programmers will always try to optimise the attribute of a project that will show them in the best possible light (lines of code produced, fastest time per module). Metrics should not be used for evaluating staff because in trying to improve particular metrics the overall model will be thrown off course [23].
- The accuracy of predictions from a model must be linked back to the accuracy of the collected data. There is always a danger that model predictions are looked at without considering the range of accuracy. To ensure the accountability of any decisions made from a model, control data collected at the same time will provide a means of reconstructing the conclusions from the data. Control data would include; date, observation method, and accuracy of data.
- "A software metrics program must not have a strategy unto itself. Collecting software metrics must not be an isolated goal. Software metrics can successfully be only a part of an overall strategy for software development improvement" [46].

This last point about a strategy requires further amplification to fully understand the reasons for launching into a metrics programme. Improving product quality is often a key goal for an improvement strategy, but quality is a difficult product attribute to measure.

3.4 Quality

"Quality is recognised as one of the prime drivers behind the acceptance of our goods and services by customers. It has a direct effect on the cost of production, delivery, support and customer satisfaction. Assuming there is a commodity that customers want, quality is often the final tie break of competitiveness" [50].

Quality, in relation to software engineering, covers two main areas. The quality of the development process and the quality of the product developed. Standards for software development and assessment include BS5750 [51], ISO9000 [52], SPICE [53] and TickIT [54].

The quality of a product, when measured by the number of problems found, can be linked to the level of verification. The standard quality/cost/precision curve does apply to verification, there is an optimum level of testing to maximise the level of quality for the testing expenditure. There is an optimum point for maximum contribution at minimum cost, beyond which you are paying a considerable price for finding those last few errors. Gilb provides a quality/cost graph in his book, Principles of software engineering management [55]. The graph (Figure 7 Quality/cost graph, page 45) shows the cost of improving a quality tending towards infinite cost, as 100% of that quality is approached. There is certainly not a linear relationship between quality and cost, as the quality level approaches state of the art.

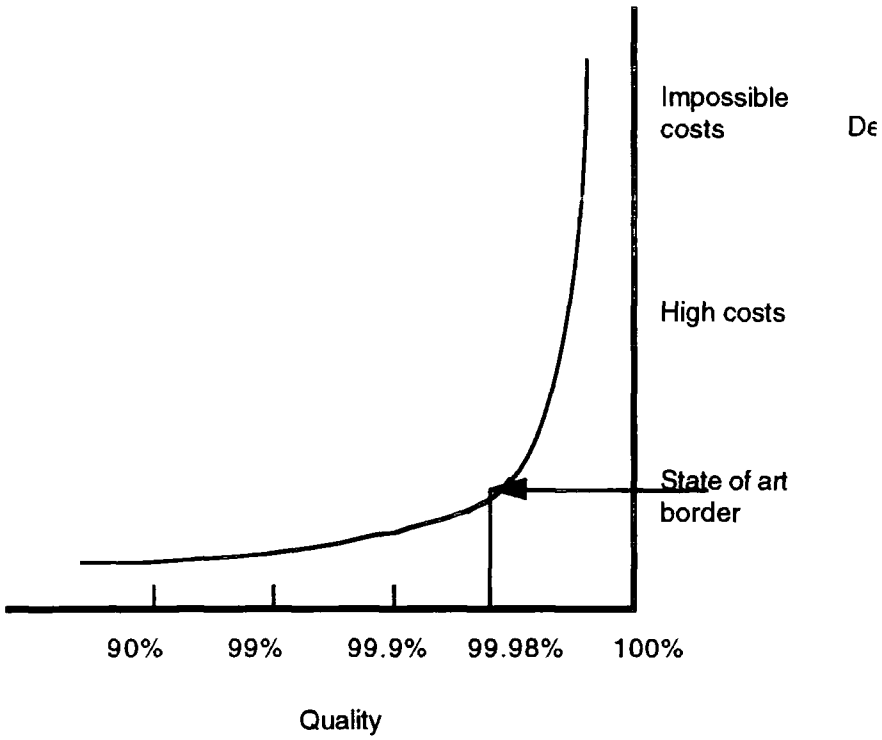


Figure 7 Quality/cost graph

The graph below (Figure 8, page 46) builds on Figure 7 Quality/cost graph, page 45, and shows the impact of the cost of quality on the contribution to the product and the product price. The contribution will fall as the state of the art border is passed and the cost of quality rises steeply. The product price will reflect the rising cost of quality until the product becomes so expensive that it becomes unsaleable. Where there is a risk to human life (software for aircraft, manned spacecraft, military etc..) a very high level of quality will be attained, but with a high product price.

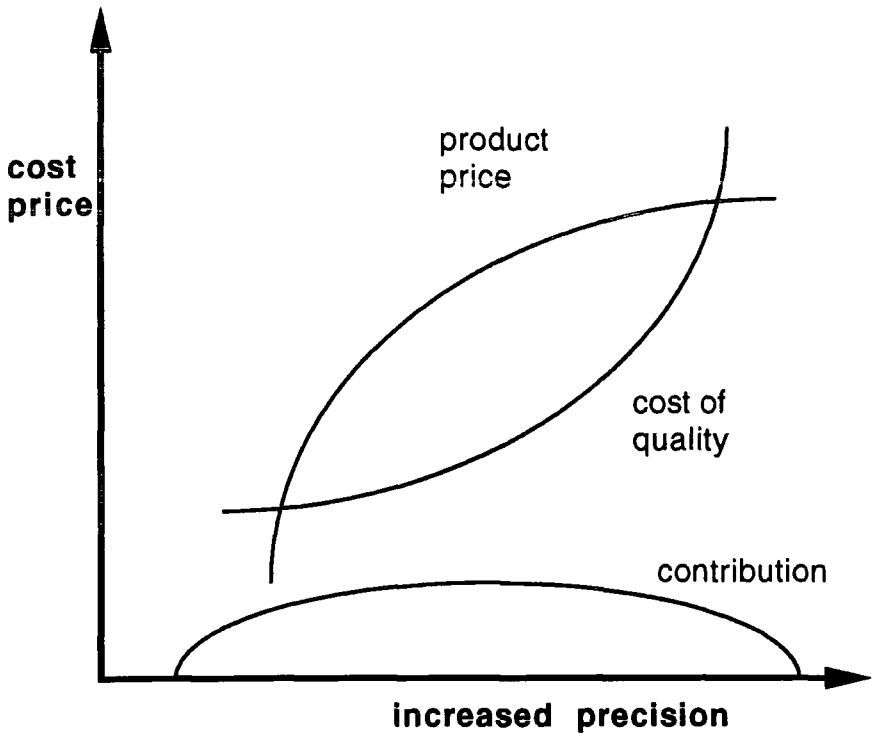


Figure 8 Cost curve

This measure of quality does not have the backing of all software practitioners. The problem is in linking the data from the graph to a real measure of quality, Conte explains " faced with the task of constructing an objective algorithm to measure an abstract concept such as software 'quality', one could measure the quality based on such items as the number of errors uncovered in testing, defects discovered per thousand lines of code, or time between the last two software failures. But does the number bear any relation to the abstract concept of quality"[26].

DeMarco has a completely different view of the relationship between testing and quality, "poor product quality is a sign of inadequate testing. Improving quality is as simple as increasing the investment in testing. While this idea is intuitively appealing, the facts don't bear it out at all. Incredibly, the testing investment is an inverse indicator of product quality. In one sample after another, we see that a heavy investment in testing is a symptom of poor quality, not a cure" [23]. But as Ferdand [50] says this is a complex relationship, "The problems associated with defect behaviour and quality are complex. They often involve a large number of elements and relationships. In software engineering,

for example, this may entail hundreds of thousands of lines of code and even more relationships. One has to deal with human nature as well as statistical laws of nature".

There are two main reasons for the use of metrics and models during software development and support - Quality and project costs.
Projects balance the quality of the product against the cost of development, as shown in Figure 9 Balance I, page 47. Throughout this thesis, diagrams will be used to illustrate the quality/cost balance and the different interpretations of quality in use today. If a poor quality product is released for customer use, then there will be a cost associated with this poor quality, "whenever a user speaks of 'poor quality,' this can almost always be translated into some kind of loss of money, time, or other resources" [56].

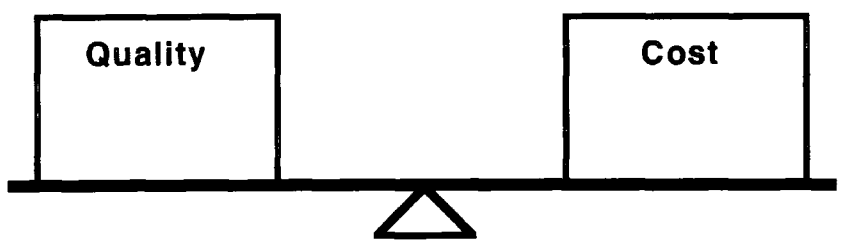


Figure 9 Balance I

Project costs cover the resource required to construct the software, equipment as well as manpower, and elapsed time. Why include elapsed time and not just man-months? Brooks [57] sums up the answer in his book, The Mythical Man-month;
"Adding manpower to a late software project makes it later. This then is the demythologizing of the man-month. The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using fewer men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calendar time than for all other causes combined."

"The quality determines cost principle:
You cannot accurately estimate the costs of anything when cost determining quality attributes are unclearly defined" [55].
Two interpretations of quality are discussed by Watts [58] " Firstly, quality could be taken to mean the degree of excellence. This would suppose that there is a general agreement on the characteristics which are considered desirable and necessary (and the reverse) and also on the extent to which specific characteristics have to be present in order to fulfil general quality standards. Since it seems to us that there can be no such general agreement, we reject this interpretation of quality.

According to the second interpretation, quality is to be understood as the degree of compliance (or non-compliance) with the specified requirements. Quality requirements are statements on characteristics of a software product (for example, on its maintainability) and on the desired extent of these characteristics for the type of use intended. The question is not 'Which characteristics must a good product possess?' but rather 'By which characteristics would a product be suitable for type of use X?' and 'To what extent is a given product suitable for that type of use?' "
Watts goes on to explain a conceptual framework for quality, which is similar to the quality attributes list referred to as FURPS by Grady and Caswell [46] given below:

Functionality	Feature set
	Capabilities
	Generality
	Security
Usability	Human factors
	Aesthetics
	Consistency
	Documentation

Reliability	Frequency/severity of failure
	Recoverability
	Predictability
	Accuracy
	Mean time to failure
Performance	Speed
	Efficiency
	Resource consumption
	Thruput
	Response time
Supportability	Testability
	Extensibility
	Adaptability
	Maintainability
	Compatibility
	Configurability
	Serviceability
	Installability
	Localizability

Hewlett-Packard [46] use FURPS to break down the product quality into its attributes. Each development project will have priorities set from the attributes, and a way of measuring them. The impact of changing one attribute can then be assessed against tradeoff's with other attributes. Improving functionality might reduce performance or reliability for example. The quality/cost balance can be redrawn as shown in Figure 10 Balance II, page 50.

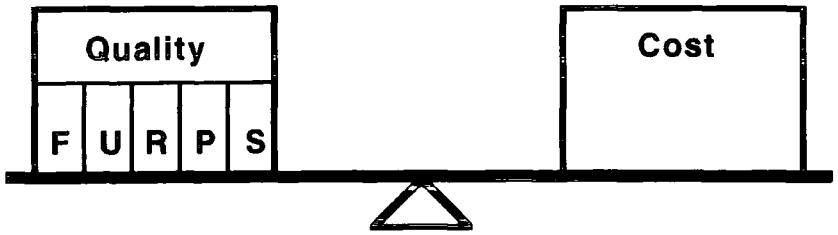


Figure 10 Balance II

Product metrics measure the quality attributes of a product and can therefore be used in the tradeoff's between the different attributes without upsetting the quality/cost balance. Explicit software quality objectives and priorities are needed, argues Boehm [59] "the degree of quality a person puts into a program correlates strongly with the software quality objectives and priorities he has been given. Thus, if a user wants portability and maintainability more than code efficiency, it is important to tell the developer this, preferably in a way which allows the user to determine to what extent these qualities are present in the final product."

Denhand [56] " The fact that users and developers evaluate the quality of software differently has been known for some time." This means that although quality attributes like maintainability and extensibility can be measured in technical terms that a developer will understand, when it comes to the quality of the human factors attributes this must be understandable by the users. Otherwise it will not be possible for the users to evaluate the software for usability.

Process metrics show how the quality/cost balance can change with improvements or degradation of the development process. "In manufacturing, the Japanese have demonstrated that productivity gains follow naturally from improvements in the process. Japanese companies are measuring software quality with the same interest that they showed in manufacturing quality" [46]. The measurement of product and process in relation to the quality/cost balance is shown in Figure 11 Balance III, page51.

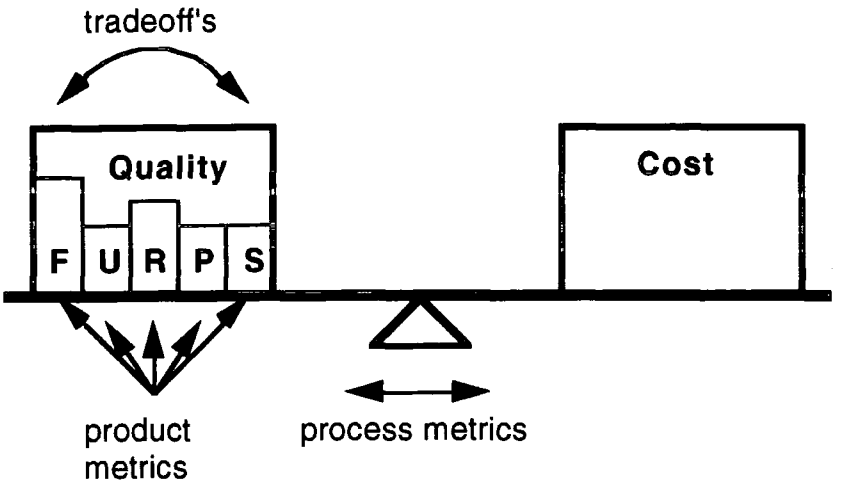


Figure 11 Balance III

The process metrics effectively show the position of the balance point, and the leverage that can be applied by changes to the process. Improve the development process and the result is an increase in quality for the same level of project cost. The result of process improvement is highlighted by Figure 12 Balance IV, page 51. This effect works both ways and increased project cost would result from a change to a less effective process.

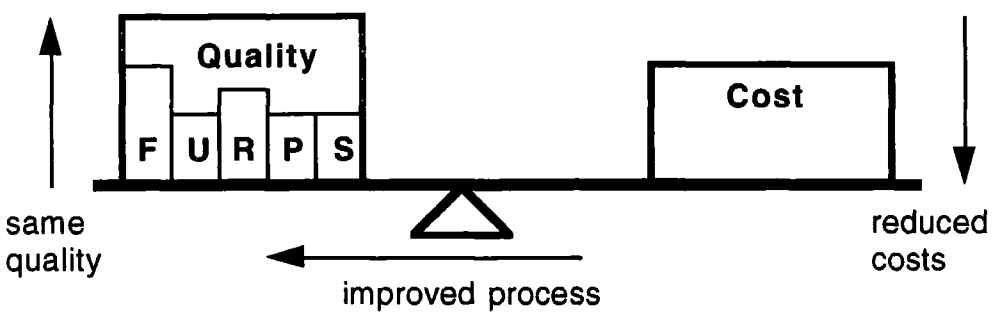


Figure 12 Balance IV

Adding project management, feedback paths and control flow completes the picture of software development and the use of metrics. Figure 13 Balance V, page 52, shows the information flow from the metrics applied and the control flow from the project management decisions.

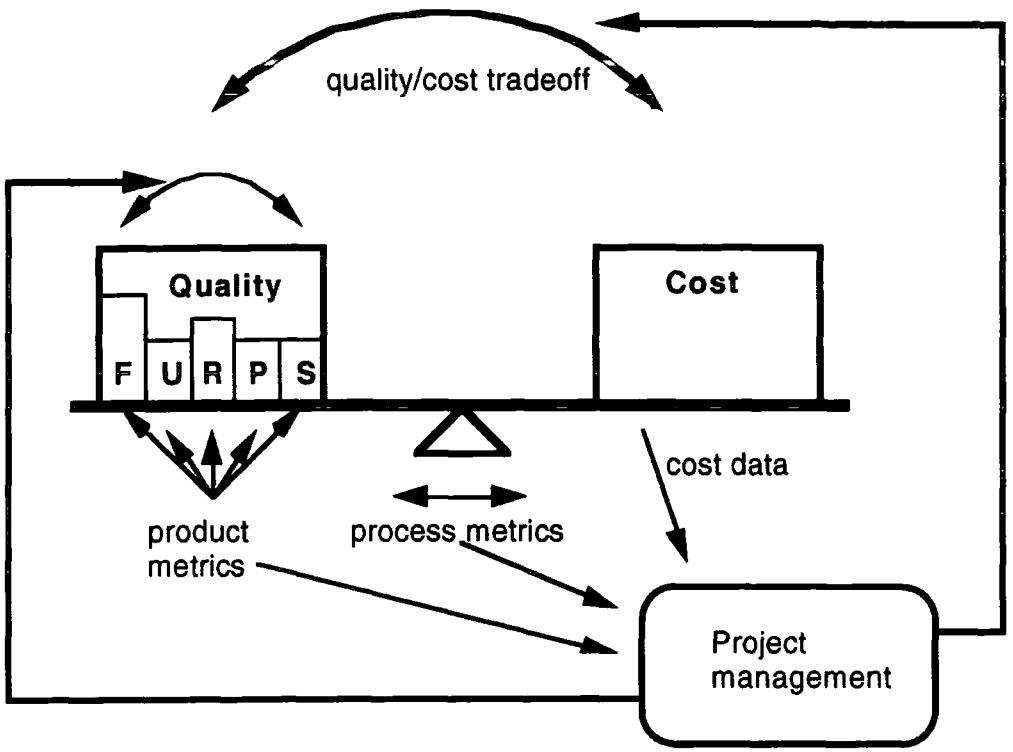


Figure 13 Balance V

3.5 Methods of implementation

The most important first step when starting a metrics programme is decide on the objective. In most cases the objective will be linked to the quality/cost balance. Improved product quality, reduced development cost or both, are the likely top level objectives. At a lower level the target might be improvements to a particular quality attribute or reduced cost of a particular process. Once there is an objective it becomes clearer which metrics will be important, and how much can be spent on the work.

An approach to implementing software reliability metrics used by Siefert [60] involved determining the 'best of class' metrics. This was achieved by a survey of 100 organisations (26 replied) asking for:

there most frequently used measures

the measures considered most important

the measures that are the easiest to use

measures that are the easiest to implement

Weighted and average ratings produced for each of the above questions were combined to give a 'best of class' list, reproduced below:

Rank	Measure	Normalised Value
1	Fault density	16
2	Failure rate	21
3	Error distribution	26
4	Defect density	29
5	Cumulative failure profile	30
5	Failure analysis	30
6	Test coverage	44
7	Fault days number	49
8	Cyclomatic complexity	51
9	Entries and exits	52
10	Functional test coverage	53
11	Mean time to failure	56
12	Software science difficulty	62
13	Graph theoretic complexity	64
14	Software science listings & documentation	65
15	Combined HW/SW operational availability	71

The normalised value to the right of the table is determined by adding together the ranking a measure has in each of the seven ratings. Top of a rating is one point for that measure, second two points and so on. The best possible normalised value is therefore seven, if the measure came top with all seven

ratings. The best measure in the list is 'Fault density with a normalised value of 16, which is effectively the result of coming second in most of the ratings.

The paper also includes a four-step implementation guideline:

1. select measures from the table, based on data available, objective for the metrics programme, and culture of the organisation
2. identify measurement standards
3. performance is then measured against those standards
4. evaluate the variance and fine tune the development process

In this particular method the model is constructed from the development process used being superimposed onto a development life-cycle. The linkage between the model and real life is determined by experimentation with the model. The model is gradually built as the data from experimentation with the process provides the relationships between the different metrics.

The metrics that David Siefert included in his survey, came from the IEEE Standard dictionary of measures to produce reliable software [61]. The standard lists 39 metrics, splitting them into product and process measures. The product measures consider both projected reliability prior to operation and operational reliability. To cover the different dimensions of reliability the measure has six subcategories:

1. Errors, faults, failures - Count of defects with respect to human cause, program bugs, observed system malfunctions.
2. Mean-time-to-failure, failure rate - Derivative measures of defect occurrence and time.
3. Reliability growth and projection - The assessment of change, in how unlikely a failure will be during product testing and operation.
4. Remaining product faults - The assessment of how unlikely a failure will be of the product, in development, test, or maintenance.
5. Completeness and consistency - The assessment of the presence and agreement of all necessary software system parts.
6. Complexity - The assessment of complicating factors in a system.

The process measures are divided into three subcategories:

1. Management control - The assessment of guidance of the development and maintenance processes.
2. Coverage - The assessment of the presence of all necessary activities to develop or maintain the software product.
3. Risk, benefit, cost evaluation - The assessment of the process tradeoffs of cost, schedule and performance.

The measure classification index below is a cross index of the 39 measures against the classifications given above. The sequence of the 39 measures is as presented in the IEEE standards [61] [62]:

Measure	Product						Process		
	1	2	3	4	5	6	1	2	3
Fault density	X								
Defect density	X								
Cumulative failure profile	X								
Fault-days No.	X						X		
Functional/modular test coverage					X			X	X
Cause effect graphing					X			X	
Requirements traceability	X				X			X	
Defect indices	X						X		
Error distribution							X		
Software maturity index			X						X
Man hrs per major defect detected							X		X
No. of conflicting requirements	X				X			X	
No. of entries/exits per module					X	X			
Software science measures				X		X			
Graph-theoretic complexity architecture						X			
Cyclomatic complexity					X	X			
Minimal unit test case determination					X	X			
Run reliability			X						
Design structure						X			
Mean time to discover next K faults									X

Software purity level			X							
Estimated No. faults remaining, seeding					X					
Requirements compliance	X				X			X		
Test coverage					X			X		
Data or information flow complexity						X				
Reliability growth function			X							
Residual fault count				X						
Failure analysis using elapsed time			X	X						
Testing sufficiency			X					X		
Mean-time-to-failure		X	X							
Failure rate		X								
Software doc and source listings					X					
Required software reliability RELY								X	X	
Completeness					X					
Test accuracy				X	X			X		
System performance reliability			X							
Independent process reliability			X							
Combined HW/SW system operational availability		X								

3.6 Evaluating metrics, meta-metrics

A meta-metric is a measure of a metric, that can be used to evaluate a metric. It is necessary to compare metrics when trying to find and evaluate suitable metrics for a particular purpose; meta-metrics provide the measures for comparing and selecting metrics. To monitor the performance of metrics over a period of time will also require the use of meta-metrics. A common approach taken to validate new measures is to show that they correlate with some well known existing measures. But this approach should not be taken, "if there is no hypothesis about the reason for a relationship, there can be no real confidence that it is not spurious" [48].

Five meta-metrics are described by Conte [26] which map reasonably well on to the seven criteria of goodness for software measures described by Watts

[58]. The information from these two sources has have been merged to produce the list of meta-metrics below.

1. Simplicity. Does the metric lead to a simple result that is easily interpreted? A single, intuitive value, like the number of reported errors as a software quality metric, is simple.

2. Reliability. Reliability indicates the degree of accuracy with which a characteristic can be measured. It is therefore a measure of the reproducibility of the measuring results; it depends on stability and precision of measuring, as well as on the constancy of measuring conditions.

3. Validity. Does the metric measure what it purports to measure? It is easy to demonstrate that lines of code is a valid measure for program size. A measure is only valid if it is justified that one should infer the real characteristics from the indicators (the observable properties). There are a number of validation methods; the measure is compared with an external criterion (external validity), with other measures already proven valid (internal validity), with expert estimates (concurrent validity), or with estimated values (predictive validity). If there is reasonable similarity between the measure and the comparison criterion, it is likely that the measure is valid.

4. Robustness. Is the metric sensitive to the artificial manipulation of some factors that do not affect the performance of the software?

5. Prescriptiveness. Can the metric be used to guide the management of software development or maintenance. A metric used to guide development, will be assessed during the development and not at the end. Watts uses the term usefulness in a similar way to indicate the extent to which a particular measure satisfies a practical need when it is used to quantify a particular characteristic.

6. Analysability. Can the value of the metric be analysed using standard statistical tools? The lines of code metric is easily analysable, while a yes/no binary type of metric is not. The other approach is to decide if the metric is economical. A measure is economical if the determination and evaluation of measured values involves low cost in comparison with the benefit gained.

7. Objectivity. A measure can only be considered objective if it is free from any subjective influences of the measurer. As well as measurement, evaluation and interpretation will also alter the objectivity of a metric.

3.7 Evaluating models

The only way to validate a model is the collection of evidence (data) to show that the model does actually work. The data can be collected from a series of similar projects or experimentally in parallel using teams set up to produce the same software under controlled conditions.

Experimentation is the best method for proving models and metrics. The environment can be closely controlled and individual factors changed one at a time to see the effect on the model. The results unfortunately do not always lead to a straightforward mathematical relationship. For example, an experiment to find the best link between complexity metrics, code failures and time to fix, came to this conclusion:

"Based on this experiment we conclude that, for similar programming environments and assuming a stable programming personnel situation, structure would have a significant effect on the number of errors and labour time required to find and correct errors. This relationship is not expressible as a mathematical function; rather, complexity measures partition structures into high or low error occurrence according to whether the complexity measure values are high or low respectively" [63].

The cost of setting up parallel teams and development environments is normally prohibitively expensive. It is seldom possible, outside university environments, to fund or collect together teams for this scale of experiment; so data, that can be compared from similar projects and environments is used. A validity check will therefore be needed to examine the data for consistency and accuracy. Fenton & Pfleeger [47] are concerned that exploratory research in software engineering is often conducted on artificial problems in artificial situations because of the high cost of running large scale studies, "practitioners refer to this body of research as 'toy' projects in 'toy' situations. The number of

research studies using experienced practitioners (rather than students or novice programmers) is minuscule".

Then the next step is to see if the data matches the model. Basili, from a paper titled Data collection, Validation, and Analysis [49], "If the data supports the model, then it reinforces our understanding of the software development process and product. If the data does not support the model, then we must further analyse the model and the appropriateness of its application to the data and the data collection environment. It is possible that the data collection environment did not satisfy some of the assumptions of the model, explicit or otherwise. We can use this data either to refine or refute the model or to gain new insights into our software development environment. In any case, the application of the model to the data often generates more questions than it answers and sets the stage for new analysis and collection of new data."

Confirmatory studies might well change just one parameter on a project to validate the model predictions for that change. It would obviously take a considerable number of projects to confirm the predictions of a complex model. The only alternative is the high cost option of experimentation.

3.8 Summary

Metrics are important for continued improvements to software verification. But they must be validated (use of meta-metrics) and linked together within the framework of a model. The model will also require validation and is best achieved with real project data or experimentation. A stated objective is needed before embarking on a metrics programme, to focus the need for metrics and to stop the collection of data for its own sake. A cost/benefit analysis of the proposed programme will no doubt be asked for, so a good starting point is to consider just a minimum set of metrics that are easy to collect and use.

The flow chart on page 60 summarises the procedure for applying metrics and models into a number of steps.

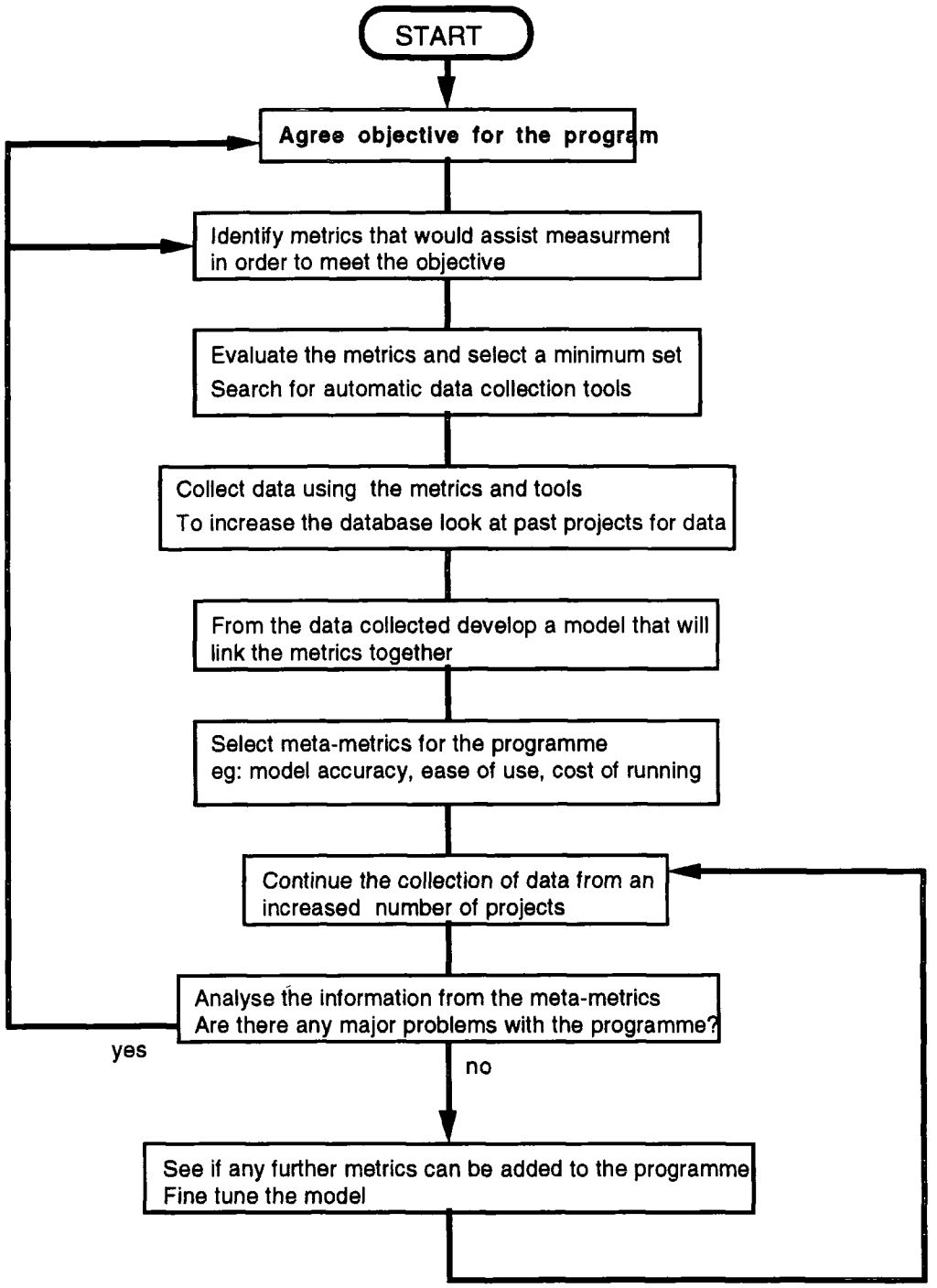


Figure 14 Flow chart for a metrics programme

4 Software Development Models

"The current status is:

- there is now a fairly complete solution to a (very restricted) part of the problem
- there is active research in areas where advances are likely
- parts of the problem look near impossible"

Littlewood [64]

To provide an understanding of the progress to date on the use of models, this chapter reviews a number of popular models which have been used for sizing, cost estimation and reliability. The underlying theory for some of these models has been included, along with studies and experiments based on the use of these models. Complexity metrics have also been included as they have influenced the thinking behind a number of models.

4.1 Sizing

Sizing models have been around for a number of years and are supported by a number of tools, but are not as fashionable at present as the cost estimation models. Even so, a prerequisite for a number of the cost estimation models is a value for the size of the software. So sizing models are still important even though there is a range of sophisticated cost models available today.

There are five different approaches to sizing in use at present; sizing by analogy, size in size out, function point analysis, linguistic approach, comparison of

project attributes. An explanation of each is given below along with a list of commercially available tools that are based on each of the methods described. For further information about the tools, including suppliers, see USAF report [65].

Sizing by analogy

This method tries to relate the proposed project to a set of previously developed modules or system, of similar function and hardware base. Accuracy depends on the correctness of the size data held already and the validity of the comparison between the systems.

Tools: ESD software sizing package

SSA (Software Sizing Analyser)

QSM Size planner, fuzzy logic

Size in size out

An approximate size is input and then refined to give a closer estimate of the proposed system. This refinement might use expert consensus (Delphi technique[i]) [66] or in the case of SSM (Software Sizing Model) an automated question and answer session to build up module sizes. The Delphi technique has been extended by Boehm and Farquahar [16] to include a group meeting of the experts to discuss the estimates, although the individuals estimates remain anonymous. This has been named the Wideband Delphi technique and has been shown by Boehm [16] to offer improved results over the original method.

Tools: SSM(Software Sizing Model)

Function point analysis

Developed in 1979 by Albert Albrecht and refined in 1983 [67] as an alternative to the lines of code metric. Function points relate development effort to the

[i] Developed by RAND corp in 1948. Set of anonymous estimates from experts summarised by a co-ordinator and returned to the experts; process is iterated a number of times until agreement.

amount of user function delivered. The information processing of the system is first split into five categories; external inputs, external outputs, external inquiries, logical internal files, and interfaces. The data in the five categories is counted, and then modified by a processing complexity value made up from 14 program characteristics to give a function point count. If the program size is required in the 'lines of code' measure, a conversion table based on language options is needed to convert from function points.

Albrecht and Gaffney [67] suggest that as function points can be estimated early from the requirements, that this value is then converted into lines of code, which is much more difficult to estimate early, but does give a good measure for development effort. Their paper covers this two step effort estimation method and also compares the theory of function points to Halstead's software science metrics [68].

Tools: BYL(Before You Leap)
SPQR Sizer
QSM Size planner, function points

Linguistic approach

This approach is based on Halstead's [68] software science work. It provides a relationship between the number of operators and operands to program size. Pseudo code will be needed before the count can take place.

Tools: ASSET-R (Analytical Software Size Estimation Tool - Real time)

Comparison of project attributes

Like sizing by analogy this method requires previous project data, but rather than comparing system functions, project attributes are compared. The typical attributes to be compared might include: reliability, complexity, number of screens, reports etc..

Tools: CEIS (CEI Sizer)
QSM Size planner, standard components

A report for the USAF-air force cost centre [65], evaluating the models against a known project of 9,177 LOC produced the table of results below:

Model	LOC
ESD	37,600
SPQR	35,910
BYL	22,402
PRICE SZ	21,410
ASSET-R	11,943
SSM	11,700
ASSET-R	6,622

The two estimates for ASSET-R reflect two different approaches used to derive the model input. ASSET-R along with SSM provided the most accurate size estimates for this experiment. ESD, CEIS, SSA, QSM fuzzy logic and SSM can be applied the earliest in the life cycle, during requirements analysis. BYL and ASSET-R score for providing a user friendly interface and user support. The function point approaches (ASSET-R, BYL, QSM function point, SPQR), QSM fuzzy logic and PRICE SZ require the least knowledge or experience in the application area to be effective.

Application areas for the models:

- ASSET-R, CEIS, PRICE SZ, QSM, SSM virtually all user environments
- ESD limited to applications comparable to those held in the database
- BYL, QSM function point, SPQR not validated for scientific or real time systems

4.2 Estimation

The estimation of software costs started with the sizing models that predicted a size for the proposed software. The cost estimation came from comparing the predicted size with past projects of a similar size and noting the cost of these comparable projects. Jones [69] in 1978 provided a different method based on multiplying costs per line of code by a size estimate.

In 1981 Barry Boehm published 'software engineering economics'[16] which included the cost estimating model COCOMO (COnstructive COst MOdel) developed at TRW. The table below, from Boehm [16], shows the strengths and weaknesses of software cost estimating methods taken from his book.

Algorithmic models

These methods provide one or more algorithms which produce a software cost estimate as a function of a number of variables which are considered to be the major cost drivers. COCOMO is an example of an algorithmic model.

Strengths:	Objective, repeatable, analysable formula
	Efficient, good for sensitivity analysis
	Objectively calibrated to experience
Weaknesses:	Subjective inputs
	Assessment of exceptional circumstances
	Calibrated to past, not future

Expert judgement

This method involves consulting one or more experts, perhaps with the aid of an expert consensus mechanism such as the Delphi technique.

Strengths:	Assessment of representativeness, interactions, exceptional circumstances
Weaknesses:	No better than participants
	Biases, incomplete recall

Analogy

This method involves reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project.

Strengths:	Based on representative experience
Weaknesses:	Representativeness of experience

Parkinson

A Parkinson principle ("Work expands to fill the available volume") is envoked to equate the cost estimate to the available resources.

Strengths:	Correlates with some experience
Weaknesses:	Reinforces poor practice

Price to win

The cost estimate developed by this method is equated to the price believed necessary to win the job.

Strengths:	Often gets the contract
Weaknesses:	Generally produces large overruns

Top down

An overall cost estimate for the project is derived from global properties of the software product. The total cost is then split up among the various components.

Strengths:	System level focus
	Efficient
Weaknesses:	Less detailed basis
	Less stable

Bottom up

Each component of the software job is separately estimated, and the results aggregated to produce an estimate for the overall job.

Strengths:	More detailed basis
	More stable
	Fosters individual commitment
Weaknesses:	May overlook system level costs
	Requires more effort

Boehm describes COCOMO as a composite algorithmic model [16] using both top down and bottom up estimating. The data for the model came from 63 projects developed during the period 1964 - 1979, which includes a wide range of sizes, productivity rates and seven different languages. The model is split into three levels; basic, intermediate and detailed. The projects are also split into three, covering three different modes of software development:

"Organic mode

In the organic mode, relatively small software teams develop software in a highly familiar, in-house environment. Most people connected with the project have extensive experience in working with related systems within the organisation, and have a through understanding of how the system under development will contribute to the organisations objectives.

Semidetached mode

The semidetached mode of software development represents an intermediate stage between the organic and embedded modes. "Intermediate" may mean either of two things:

An intermediate level of the project characteristic

A mixture of the organic and embedded mode characteristics

Embedded mode

The major distinguishing factor of an embedded mode software project is a need to operate within tight constraints. The product must operate within (is embedded in) a strongly coupled complex of hardware, software, regulations, and operational procedures, such as electronic funds transfer system or an air traffic control system" [16].

Basic COCOMO provides an effort equation for each mode of development:

Organic	$MM = 2.4(KDSI)^{1.05}$
Semidetached	$MM = 3.0(KDSI)^{1.12}$
Embedded	$MM = 3.6(KDSI)^{1.30}$

MM = Man Months

KDSI = a count of source statements

Basic COCOMO is within a factor of 2 of actuals 60% of the time.

Intermediate COCOMO is an extension of basic COCOMO to improve accuracy (within 20% of actuals 68% of the time) and level of detail. A nominal estimate (MM)nom provided by the equations below is then modified by fifteen cost driver multipliers.

Organic	$(MM)_{nom} = 3.2(KDSI)^{1.05}$
Semidetached	$(MM)_{nom} = 3.0(KDSI)^{1.12}$
Embedded	$(MM)_{nom} = 2.8(KDSI)^{1.20}$

The table below shows the cost drivers, the ratings that can be selected for each of them and the multiplier value that is applied to the nominal effort estimation equations above. More information is provided by Boehm [16] on how to judge which rating to use for each of the cost drivers.

The detailed COCOMO model has development phase sensitive, effort multipliers, for each of the cost driver attributes. This is to cater for the real world situation where attributes affect some development phases more than others. Detailed COCOMO also avoids the problem of having to provide separate cost driver ratings for different product components by using a three level product hierarchy, Boehm:

"Some effects, which tend to vary with each bottom level module, are treated at the module level.

Some effects, which vary less frequently, are treated at the system level.

Some effects, such as the effect of total product size, are treated at the system level" [16].

Cost drivers	very low	low	nom	high	very high	extra high
Product attributes						
RELY Required software reliability	0.75	0.88	1.00	1.15	1.4	
DATA Data base size		0.94	1.00	1.08	1.16	
CPLX Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility		0.87	1.00	1.15	1.30	
TURN Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel attributes						
ACAP Analyst capability		1.46	1.19	1.00	0.86	0.71
AEXP Applications experience	1.29	1.13	1.00	0.91	0.82	
PCAP Programmer capability	1.42	1.17	1.00	0.86	0.70	
VEXP Virtual machine experience		1.14	1.07	1.00	0.95	
LEXP Programming language exp						
Project attributes						
MONP Use modern prog practices		1.24	1.10	1.00	0.91	0.82
TOOL Use of software tools	1.24	1.10	1.00	0.91	0.83	
SCED Required development sched		1.23	1.08	1.00	1.04	1.10

The same fifteen cost driver attributes are used in the detailed model as found in the intermediate, but they are spread over the hierarchy:

The module level is described by the cost drivers which tend to vary at the lowest level, module complexity, programmers capability and experience; CPLX, PCAP,VEXP, LEXP. A table is constructed which consists of values for each cost driver above, for each module and for each development phase. Subsystem level uses the drivers which tend to vary from subsystem to subsystem;

RELY, DATA, TIME, STOR, VIRT, TURN, ACAP, AEXP, MODP, TOOL, SCED. A similar table is used for each cost driver within each subsystem, and then repeated for each phase.

The system level includes the nominal effort and schedule equations and the breakdown between phases.

COCOMO is evaluated by Conte et al [26] and these observations made: Basic COCOMO does not perform well on its own database, the intermediate does much better. The effort multipliers have a great effect on the results; the maximum estimated effort is some 800 times the minimum. This provides great flexibility and range but also great volatility.

The 16 parameters present a weakness, as large amounts of data needs collecting and maintaining. It is not considered necessary to use multipliers accurate to two significant digits when estimating the size is subject to errors of 50% or more. A number of the multipliers could be combined, so that the parameters could be reduced to 3 or 4 and still stay within the accuracy of the model.

The data from the 63 projects was used to develop the model and its multipliers empirically. The model might not work so well on projects that are not similar to the to the 63 that the initial database was created from. Although a wide size range of projects was used (2K - 1,000K lines, business & scientific), development methodologies and languages have come a long way since 1964 - 79. Conte [26] puts forward a set of modified equations for intermediate COCOMO based on a process of calibration, using a large number of projects:

Organic	$(MM)_{nom} = 2.6(KDSI)^{1.08}$
Semidetached	$(MM)_{nom} = 2.9(KDSI)^{1.12}$
Embedded	$(MM)_{nom} = 2.9(KDSI)^{1.20}$

DeMarco [23] raises an objection to any model that uses lines of code as an indication of volume of work, "You can't count lines of code at the beginning of a project, you have to estimate it. Most people are not much better at estimating line count than they are at estimating resultant development costs."

But he does agree that this type of model can be very useful after the point in a project at which an accurate count of lines of code can be made. The model can then be used for prediction over the remainder of the development work; for example. unit test and integration costs.

A criticism often made of the COCOMO model is the lack of consideration given to the effect of team size on productivity. There is not a linear relationship between the number of programmers used and the productivity of that team, which should be reflected somehow by estimation models. "Adding manpower to a late software project makes it later" [57]. Partitioning work between programmers increases the co-ordination effort and the communication needed between all the programmers. The model COPMO, developed by Thebaut [70], includes a metric for team size as well as the project size. But just like project size, team size is likely to be unknown at the start of a project.

Rather than using team size as an input to the model, what is really required is for the model to determine the most useful team size (optimal staffing pattern) that would result in the earliest delivery date. This type of model is referred to as a time sensitive model.

Norden [71] of IBM plotted manpower curves from projects during the sixties. He found a strongly recurrent staffing pattern, one that matched the Rayleigh distribution. There was no evident reason why staffing rates should fit a Rayleigh curve, but the data showed that they did.

This work was followed by Putnam [72] in the seventies, who developed a resource allocation model. Based on the Rayleigh distribution the model was validated using a number of Army projects in the early seventies and then tuned with the data from several hundred additional projects. The model is used to forecast effort and manpower loading as a function of time.

Effort forecast is based on:

Volume of work
Difficulty gradient (complexity measure)
Project technology factor (staff experience, programming environment, hardware constraints, program complexity)

Delivery time

Manpower (staff profile) is based on:

Total cumulative manpower

Project acceleration factor (how quickly the project can absorb staff)

Project month (the month of interest)

Putnam [73] has also incorporated his model into a software product called SLIM (Software Life cycle Methodology).

Conte et al [26] evaluated the Putnam model and found that it worked well on large systems, but overestimated on medium and small size systems. The model relies heavily on the size and development schedule attributes. It is also sensitive to the value of the Project technology Factor; a change in value of one to this factor and the effort may change by as much as 100%.

The model tends to exaggerate the effect of schedule compression on development effort. Putnam uses a fourth power relationship between delivery time and effort. Cut the delivery time in half and sixteen times more effort is needed to complete the project. The reverse is also true up to a point; extended delivery time allows a reduction in the team and improves the productivity. If the delivery time is extended too far then there will be a decreasing sense of urgency about the project, the more chance there is of a specification changes due to changes in the target environment, and inflation will increase costs.

Some managers consider the Rayleigh curve as inappropriate for the start and finish of projects as the Rayleigh curve has a zero start and finish value. Most projects start from a non-zero level and reduce to a constant level during maintenance. A model based on the Rayleigh curve but with a non-zero start and finish is the model proposed by Parr [74] whilst working at Imperial college. He concluded; "the model is shown to be sufficiently detailed for it to be possible to show how the use of different methodologies could affect the natural work profile associated with the project; state-of- the-art projects requiring a trial and error approach will have effort curves skewed to the left, while more standard tasks admitting a highly structured approach should display curves skewed more to the right."

Another model very similar to Putnam's model which performs better under schedule compression is the Jensen model. Evaluation by Conte et al [26] showed that overall the model performed slightly better than the Putnam model.

DeMarco's cost model described in 'controlling software projects' [23] has three basic predictors which are available at different points in the development life cycle:

- 'bang' a measure of true function to be delivered, as perceived by the user (spec)
- design weight (design)
- implementation weight (code)

The cost model is driven by the three predictors so that as a stronger predictor becomes available, as the project progresses, it can be used in place of an earlier one. The model uses the best data available at the time to gradually increase the accuracy of the estimations as the project progresses. Some predictors are better suited to providing forecasts for particular component costs than others, as shown in the table below.

Predictor	Used to forecast
Bang	design effort
	conversion effort
	acceptance test generation
	documentation
	user training
Design weight	total effort
	implementation effort
	debugging effort
	defects
	residual defects
	maintenance characteristic
	machine time

Implementation weight	total effort
	unit testing effort
	integration effort
	acceptance testing
	defects
	residual defects
	maintenance characteristic
	machine time for testing

An attempt to combine the best features of the most widely used models resulted in SOFTCOST, a composite model developed at the Jet Propulsion Laboratories by Tausworthe in 1981 [75]. It is based on the GRC[i] model [76], Walston-Felix study [77] and Rayleigh Putnam model [72].

It uses 68 parameters deduced interactively by asking 47 questions. Little confidence can be gained from the model as it has not been tested on a significant database. It is unnecessarily complex with a total of 68 parameters and yet assumes a simple linear relationship between effort and size.

4.3 Complexity metrics

This particular group of metrics have been applied to many of the different models (cost, reliability, defect, quality, productivity) and have received a lot of research attention. The importance of complexity metrics as the bases for a number of models warrants a section to consider past history and the use of complexity metrics.

There are two aspects to complexity:

1. Complexity from the human point of view, psychological complexity. This is concerned with the difficulty encountered during the development and

[i] GRC - General Research Corporation

support of software; why is it that some programs are much harder to develop, understand and support than others?

2. Complexity as the software is applied to a machine. The computational complexity and how to make the best use of the underlying computers capabilities.

The majority of the complexity metric work has concentrated on the analysis of the code to obtain some measure of psychological complexity. Halstead's 'Software science' [68] developed at Purdue university during the seventies is one of the most well known approaches to complexity.

A program is considered by Halstead, to be a collection of tokens (basic syntax units) which can be either operators or operands. Operators are command names IF, WHILE, FOR, and arithmetic symbols + , - , /. Operands are the data, variables, and constants of a program.

Software Science metrics:

η_1 = number of unique operators
η_2 = number of unique operands
N_1 = total occurrences of operators
N_2 = total occurrences of operands

Size of program (token count) $N = N_1 + N_2$

Vocabulary $\eta = \eta_1 + \eta_2$

Volume $V = N \times \log_2 \eta$

Volume can be interpreted as the number of mental comparisons needed to write a program of length N.

Potential Volume $V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$

There are a number of ways of implementing an algorithm, the one that has the minimum size has the potential volume $V*$.

Program level $L = \frac{V^*}{V}$

The program level for a module has a maximum value of one, when the minimum size program has been used.

Difficulty $D = \frac{1}{L}$

Effort $E = \frac{\eta_1 N_2 N \log_2 \eta}{2\eta_2}$

Effort is measured in elementary mental discriminations.

John Stroud proposed that the human mind is capable of making a limited number of elementary discriminations per second. This has been called the Stroud number, b and usually varies between 5 and 20.

Programming time $T = \frac{E}{\beta}$ $(\beta = 18)$

Language level $\lambda = L \times V^* = L^2 V$

The language level is used to characterise a programming language.

Halstead's metrics (λ, V^*, E) have not all found support or validation from project data [26], while others (η_1, η_2) continue to be used by researchers. The most important impact of this work and its publicity, has been to encourage further research effort into software metrics.

The other well quoted complexity measure is McCabe's [78] Cyclomatic complexity number v(G). Designed to measure the number of distinct basic paths through a program, it utilises a directed graph or flowgraph[i] of a program to define v(G).

v(G) = e - n + 2

e =number of edges (or arcs)

n =number of nodes in control flow

[i] flowgraph - the structure of an algorithm represented graphically

The example below of a complexity graph, Figure 15 page 77, has eight nodes and nine edges, giving a Cyclomatic complexity of three.

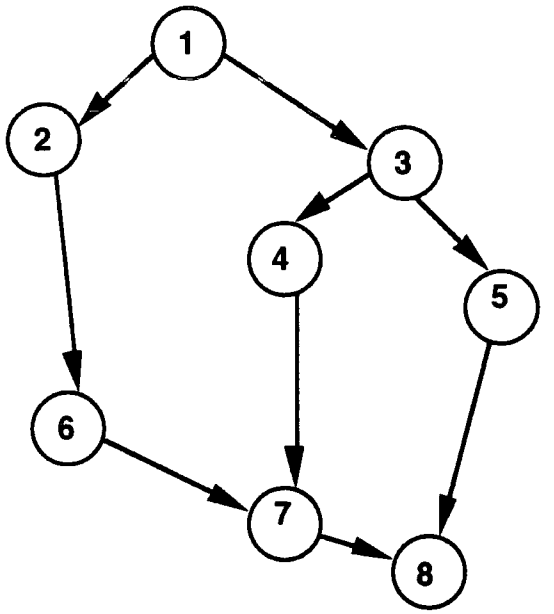


Figure 15 Complexity graph

McCabe suggested an upper limit of ten for Cyclomatic complexity, to allow for the proper testing of the code. Modules which have a value of ten or greater should be examined to see if the number of decisions can be reduced. If that is not possible then system design should be returned to so that the module can be re-partioned. The metric can also be arrived at using code analysers to count the decision points in the code.

$$v(G) = DE + 1$$

DE = decision count. This is the IF, DO, WHILE, CASE and other conditional and loop control statements. Compound conditions in conditional and loop controlled statements are counted individually as decisions.

McCabe [79] presents a methodology for structured testing, from design through coding to unit test, using the V(g) complexity metric. The paper also identifies a metric called 'essential complexity' that is used to evaluate proposed changes to a program during maintenance. McCabe and Schulmeyer [80] extend the ideas of essential complexity to the direction of regression testing and the use of Data Flow Diagrams (DFD's) to assist functional testing.

A number of studies [81] [82] [83] have used McCabe's and Halstead's metrics and compared them with the number of errors found, time to fix, program size and time to comprehend a program.

Curtis et al [84] produced empirical evidence from two experiments to show that software complexity metrics are related to the difficulty programmers experience in understanding and modifying software. Although, the correlation's observed were not as high as those reported by Halstead [68] and the number of statements in the program proved to be as strongly related to performance on the experimental tasks as the Halstead and McCabe metrics.

Walsh [85] produced a set of recommendations for utilising McCabe's complexity measure:

- "The complexity measure should be viewed as a structured programming technique and employed with other structured programming techniques to enhance software reliability.
- The complexity measure should be used to create a more testable and maintainable system be warning designers when a program has become to complex.
- The complexity measure should be used to evaluate alternative designs with the goal of finding the simplest possible solution to the problem specifications.
- The complexity measure should be used as a more through and methodical testing process which quantifies the amount of work necessary for reliable testing.
- The complexity measure should be viewed as an aid to the maintenance process via its strengthening of testing and the limiting of the complexity of the program to be fixed.
- The complexity measure should be used on existing software to identify programs that will be difficult to maintain and extend. These programs are prime candidates for redesign."

A conclusion from a paper by Schneidewind and Hoffmann [63] states, "It would be worthwhile to use complexity measures as a program design control

to discourage complex programs and as a guide for allocating testing resources. The use of complexity measures in this fashion should be tested on large programming projects so that the hypothesis can be tested under more typical conditions."

Ottenstein [86] used the data from a number of studies to validate her model (based on complexity metrics) but also considers that a large carefully run experiment is still required; "although the model fits the available data surprisingly well, many factors which could have an effect on the number of bugs present in a program were not considered. More carefully gathered data is needed to determine if the model is indeed useful. Studies to determine what other factors are important in predicting numbers of bugs should result in improvements to the model or perhaps other, more accurate models. Experiments to test the hypotheses on which the model was based might also produce interesting results."

The STARTS guide [87] concludes, "in isolation , not a great deal of weight can be attached to such measures. Their use in identifying modules or sections of code with a gross deviation from the norm, thus indicating where attention and possible remedial action by a more experienced programmer than that currently assigned to the task could be worthwhile." The majority of these independent experiments and studies have shown that complexity metrics can assist in keeping design complexity within safe limits and can direct testing towards modules most likely to have the highest error rate. It has not been shown conclusively that the complexity metrics offer any more than this, nor has it been shown how cost effective it would be to employ these metrics on a project.

The Harrison and Cook MMC metric [88] attempts to capture the complexity due to the interrelatedness of the various parts of the software system and the individual complexities within the smaller components making up the system. The former is referred to as "macro-complexity" an the latter as "micro-complexity". The metric is referred to as the macro-micro complexity (MMC) measure.

MMC = macro-complexity * average(micro-complexity)

where micro-complexity is the Cyclomatic complexity of each subprogram within the module, and macro-complexity is calculated as follows:

$$\sum_{i=1}^{subprograms} [glob(i)^{*}(subprograms-1)]+[param(i)^{*}(1-D(i))]$$

glob(i) number of times global variables are used in subprogram i

param(i) number of times parameters are used in subprogram i

DI(i) is the documentation index, the quality of the internal documentation of a subprogram. The ratio of comment lines to total lines for the subprogram was used by Harrison.

4.4 Reliability and Defect models

Software 'reliability' and 'defect count' are two different views of the same type of model. How many defects are there in a program is another way of deciding how unreliable the program will be, the inverse of reliability. Defect models are concerned with the total number of errors in a program at the different stages of development and when released to the field. Reliability models are more useful for predicting when the software will fail, and the time between failures. It would seem plausible that once the number of defects within a program are known it should be possible to predict the reliability of the program. But the exact relationship between errors present and reliability is not yet known, and it is not possible to go straight from the number of defects present to predict reliability accurately [89].

Software reliability is different to hardware reliability, because software does not become unreliable due to ageing as hardware does. All the possible errors are present in the software from the start, it is the use of the software that will uncover them in time. But it is possible for the number of errors in software to increase via maintenance activities. Beizer [90] "with maintenance included in a theory of software reliability, the theory should predict that software does wear out in the sense that it is no longer economically modifiable".

There are two main fields of use for these models:

Prediction of the total number of errors remaining in a software product. This enables a decision to be made on the extent of testing, should testing continue or should the product be released? The number of errors remaining prediction provides the data for a 'cost of testing' v 'cost of failure in the field' calculation. The improvement in reliability from continued testing can be estimated. " The ultimate objective of studying software reliability is to predict future failure behaviour and to provide information for decision making" [91].

The second use of defect models is to increase the effectiveness of testing, which can result in reduced costs or a better quality product. The prior knowledge of which software modules are prime candidates for high fault levels, is used for the allocation of testing resources. The modules that are likely to have errors are well tested due to the extra effort that has been directed at them. A policy of equal testing effort for every module might waste a lot of time on modules that don't have any errors.

Dunn [89] says that at the start of testing the number of defects are related to six items:

1. Size
2. Complexity
3. Development environment, methodology, languages
4. Programmer competence
5. Passive defect removal procedures previously performed
6. Stability of requirements and top level design

"Assuming items 3,4 and 5 are fairly consistent from project to project, in theory we should be able to perform regression analysis on items 1, 2, and 6 - by using some modelling technique for each - to estimate the number of bugs that will have to be removed. However, item 6 is not readily quantifiable by any theoretic technique. Perhaps the most realistic approach to item 6, given still vivid memories of whatever turmoil attached to the early development stages, is to upgrade the number of bugs estimated from 1 and 2 by some intuitive ratio.

What we are left with, then, is a formula in which

Number of bugs = F(complexity, size)

As a practical matter, it is unlikely that a formula for the number of bugs as a function of complexity and size can be formed except by averaging the normalised history of a very few relevant projects, with size and complexity considered independently. Then one can go with whichever of the two seems the more consistent" [89].

The models in use at present are still limited, there are areas that are not covered very well by today's models. Dale [92] listed the important issues that are hardly covered by these methods and models:

- "What are the consequences to the user of the faults remaining in this software?
- How should we go about developing this software in order to meet this target of reliability?
- How should we apportion the system reliability target to software and hardware?
- How do we predict the reliability of this system, which contains a great deal of software?"

There are also two different types of defect models:

Static model - metrics are used to estimate the number of defects within the software. This model can target a particular phase of development or provide a total for the complete development process.

Dynamic model - this model relies on the past defect discovery rate of the project to estimate the future level.

4.4.1 Static models

Akiyama's study in 1971 [93] was based on data obtained from Fujitsu relating to a single development project. Although the study was based on this early data it has been used by a number of researchers.

Metrics used:

S	Lines of code
DE	Decision count
J	Number of subroutine calls
C	Complexity of debugging (DE + J)

The four metrics above are utilised in separate equations for the calculation of the total number of defects.

Equations:

$$d_{tot} = 4.86 + 0.018\ S$$
$$d_{tot} = -1.14 + 0.2\ DE$$
$$d_{tot} = 6.9 + 0.27\ J$$
$$d_{tot} = -0.88 + 0.12\ C$$

d_{tot} Total number of defects (during testing and within 2 months of release).

An analysis [26] of the results from the nine modules of the development project provided PRED(.25), % of predicted values that fall within 25% of actuals, as:

S = .44

DE = .78

J = .67

C = .78

This shows that DE and C provided reasonable estimates for the total number of defects. But it might not be so appropriate, to continue to use it on today's programming environments as considerable improvements have been made to these environments since 1971. Use of this out of date data would result in incorrect predictions.

Motley & Brooks study [94], 1976 (for the Rome Air development centre). Data from two large command and control projects was collected, although different metrics were used for the two projects. As well as error data, code analysers collected 53 metrics from one project and 15 from the other. A multi linear model was used, which was reduced by factor analysis to remove some

of the metrics that were linearly related to one another. Application of a stepwise multilinear regression technique selected independent variables from the metrics that remained to determine the coefficients for the equation which represents the model. Conte et al [26] suggest that the brute force application of multilinear regression often leads to results that cannot be reasonably interpreted. An example of a 10 variable model from the study is given below.

$$d_1 = -0.465X_4 + 0.762X_5 + 0.544X_8 - 0.176X_{17} - 0.386X_{23}$$

$$+0.6X_{28} + 0.436X_{37} - 0.386X_{42} - 0.374X_{51} + 0.602X_{53}$$

Variable	Definition
X_4	Number of USING instructions which establish data structure interface
X_5	Number of comment statements
X_8	Number of unconditional branch instructions
X_{17}	Number of instructions performing scale/round operations
X_{23}	Number of times address variables are referenced
X_{28}	Number of times fixed-point variables are referenced
X_{37}	Number of variables referenced but not defined within the program
X_{42}	Number of non-nested DO loops
X_{51}	Number of source instructions in all 4th level DO loops
X_{53}	Number of source instructions in all 6th level (and beyond) DO loops

The model does have some strange variables; a positive coefficient for X_5 suggests that the use of comments leads to defects. Although this could mean that the developers only comment against the difficult to understand or complex

area's of code. Also the 4th and 6th level DO loops (X_{51}, X_{53}) are specifically included in the model without the other DO loop levels.

"The results are typical of those studies using regression analysis - the 'goodness of fit' may be reasonable, but the potential use as a predictive model is very limited"[26].

Extensions of Software Science by both Halstead [68] and Ottenstein [86] have led to defect models. The Software Science V metric (Volume) is a measure of the number of mental comparisons needed to write a program of length N. Halstead, Ottenstein and later Lipow [95], Gaffney [96], all base the calculation for the number of defects on the equation below, although they all use different methods to determine V. The constant 3000 was defined as the 'mean number of elementary mental discriminations between potential errors in programming'. The model has been shown to work well with some published data, such as the Akijama study, but has not worked well with other studies.

$$d_{tot} = \frac{V}{3000}$$

Potier's study [97] used error data from compiler developments to identify metrics that could direct testing by predicting the most likely error prone modules. This is a different approach to the prediction of total defects which is much more fraught with problems. One of the problems with total defect estimation is the accuracy of the data to validate the model. A low defect count may be due to good programming or poor testing. If the overall defect count is considered, then poor testing will result in more field faults, but only if the user usage pattern uncovers the errors. In all these cases it is difficult to decide if the model is wrong or that the defects have just not been found to match the model prediction. At this point it is possible to argue that a model that predicts errors that are not found by testing or by the user is perhaps too much of a theoretical model, the interest lies in the defects that are going to be found. Models that can direct testing to the most likely error prone modules appears to be much more useful.

Potier's model is based on some of the software science metrics, Cyclomatic complexity, path and reachability metrics. As some of the metrics are related to length, a set of normalised metrics are used by dividing the metrics by the program length.

The mean values of a metric for a set of procedures that had errors and a set that did not have errors are then calculated. The discriminent effect of a metric is defined as the ratio of these mean values. If a metric M has a mean value of 25 for the modules that did not have errors and a mean value of 52 for the modules that do have errors, then somewhere between these two is a threshold value that is used to decide if a module is likely to have errors or not. Non-parametric discrimination analysis is used to select a set of metrics and their threshold values, so that a decision tree can be constructed.

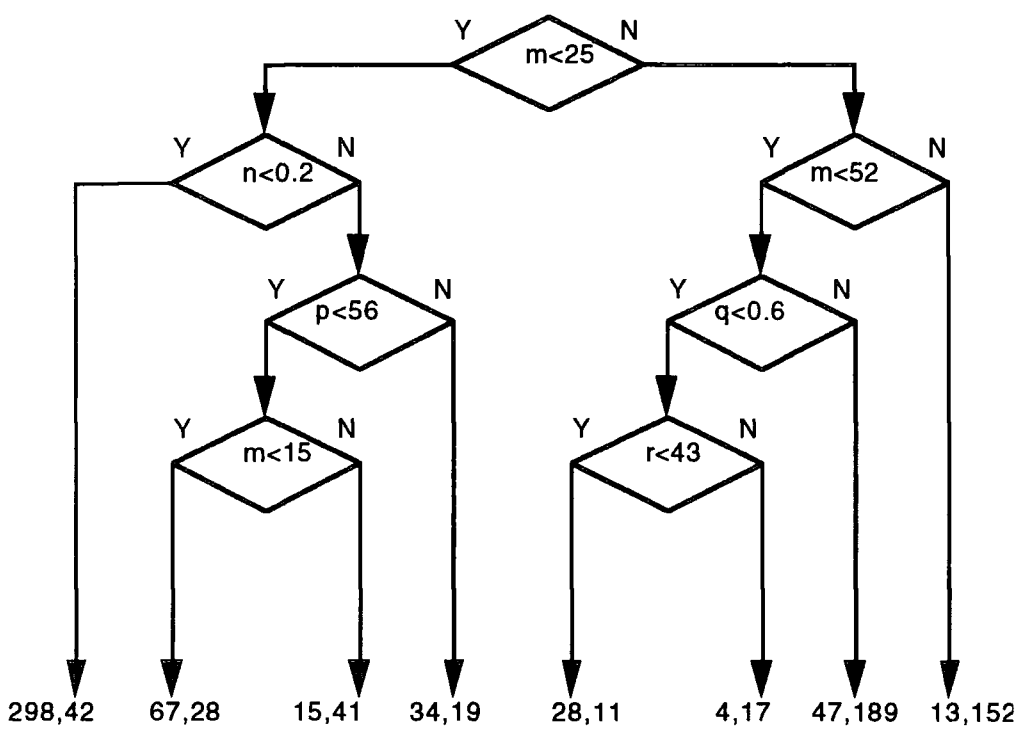


Figure 16 Decision Tree

To use the decision tree the metrics data recorded for a module under investigation is compared with the decision points on the tree, following through the tree until a leaf node is reached. That leaf node will have a pair of numbers, indicating the number of modules with errors and the number without, from past data. This result shows that it is difficult to arrive at a

threshold value that will give a clear cut result. Figure 16 Decision Tree on page 86 is an example of a three level decision tree, each level is based on a separate metric and its threshold level.

A study by Harrison [88] considered a range of metrics that could be used for directed testing[i]. Data from a compiler development project (35K lines of C, 20 modules) was used to compare these metrics:

Module size - lines of code
McCabe's V(g) metric
Halstead's effort measure
Harrison & Cook's MMC metric

Although none of the metrics gave a perfect match of testing effort to errors found, the MMC metric provided a considerable improvement over lines of code. McCabe's V(g) metric did not result in a significant improvement over lines of code, Halstead's effort measure performed somewhat better.

This study cannot be considered completely impartial as only one set of data was used and the conclusion reached supported Harrison's own metric as the best performer. In this case the MMC metric did fair the best, but how it would perform over the data from a number of different projects cannot be surmised from one experiment.

The objective of Shen's study [98] in 1983, was to find the best predictor, at three points in the development process, for the total number of faults present in a program. Based on Software Science metrics, an analysis of the metrics took place at the end of design, coding and testing phases to determine which metric was the most accurate predictor of defects. The unique operands metric η_2 was found to be the best predictor for all phases, decision count DE was a close second for prediction after design and coding, which supports Akijama's results. It is very surprising that at the end of testing when the testing defect count (d_1) is available that η_2 provides a better prediction of the total defects (d_{tot}) than the initial testing (d_1).

[i] directed testing - the allocation of testing effort to modules to maximise error detection.

$d_{tot} = d_1 + d_2$

d_1 = defects found during testing

d_2 = defects found during field use

The success of these models is dependent upon the relevance of the data and the continued relationship between the collected metrics. It has been found that the models are tied to the development environment in which the model was developed and are not generally applicable to other environments.

4.4.2 Defect density

All studies have shown that larger modules have more defects [26], which ties in perfectly with Halstead's V metric. It is interesting to consider a metric that is independent of the modules size and that can be used as a measure of the modules quality. By dividing the defect count of a module by its size a normalised metric called 'defect density' is arrived at.

$$\text{defect density} = \frac{\text{number of defects in module}}{\text{module size}}$$

Unexpectedly it has been found [99] [94] [98] that larger modules have a lower defect density than smaller modules. The reverse would be expected as larger modules tend to be more complex. Shen et al [98] found a minimum size (500 lines) above which error density can be considered unrelated to module size. Modules below the minimum size have increasingly higher defect densities. Gaffney [96] also comes to the conclusion that there is an optimum size for a module, from the point of view of error density.

Explanations put forward for larger modules having lower defect densities:

Longer programs less thoroughly tested. Motley & Brooks [94].

Proportionally less interface code in larger modules. Basili & Perricone[99].

Larger modules coded with more care. Basili & Perricone [99].

The result of learning; programmer more familiar with the operators and operands in duplicated code of larger modules [87].

To counter these studies Lipow [95] using Halstead's estimation for the 'bugs per line' shows that this defect density metric does increase with the size of the program.

Perhaps the answer to this stalemate situation, with experiments and 'proof' for both sides, is the conclusion that the experiments are not accurate enough. Using estimates for the number of defects in a program or counting the ones found through testing will not provide the total number of defects for a large program. If the exact total is not known then the exact density cannot be calculated either.

4.4.3 Dynamic models

A dynamic model includes the component time; this is the interval of time between failures. It is a random interval, with a cumulative distribution F®, and probability density function f(r).

Reliability R(r) = 1 - F(r)

$$MTTF = \int_0^\infty rf(r)dr = \int_0^\infty R(r)dr$$

Hazard rate h(r) is the probability that the software fails during the interval (r,r+dr) given that it has not failed before r.

Some models assume that all errors found during development and testing are fixed without introducing any further errors. The reliability of the software therefore increases, and this type of model is referred to as a 'reliability growth model'. The cost of testing and fixing can be balanced against the increased reliability of the software. Other models do not consider fixing, and assume that the software is being tested to asses the reliability at the time of testing.

An assumption is made that the testing environment and pattern of usage is representative of the users environment and that the reliability measured during testing will be the same as that experienced by the customers. Unfortunately it is difficult to predict the users environment and pattern of usage before the software is released to the customer. More research is needed to establish how usage patterns affect software failure rate; that is the link between testing and user. Lack of reliable data for dynamic models has also made the validation of the models difficult.

The Musa reliability growth model [100] 1975, 1980, can cope with execution time if the software is running continuously or on calendar time if it is not.

Assumptions:

Test input sets are selected randomly

All software failures are observed

Failure intervals are independent of each other

The hazard rate is fixed during an interval between failures

The hazard rate is proportional to the number of defects remaining in the system, and is a decreasing function of the execution time

The rate of defect detection is proportional to the hazard rate

$$MTTF(t) = \frac{1}{h(t)} = \frac{1}{cd_{tot}} e^{bct}$$

d_{tot} requires estimation from a static model, parameters b & c are determined from the defect history of a similar program.

As testing may be halted while problems are fixed, and the program is not therefore running continuously, it is necessary to convert the execution time (t) into calendar time. The model has been shown to predict a slower rate of defect detection early in the testing process and a faster rate than actual during the later stages of testing.

Musa and Ackerman [101] describe the use of the model by AT&T for predicting the failure rate of a switching system and returning a value for the failure intensity overestimated by only 5% over a period of one year, "How do

you validate that a piece of software loaded into a processor functions correctly? The answer is that you take advantage of both the fact that you are dealing with a vast number of possible input states and the fact that for commercial grade software only a small percentage of these states will result in failure. These conditions make a rigorous statistical approach possible - this is the essence of the technology of software reliability measurement."

Jelinski-Moranda model [102], Shooman model [103], and the Schick-Wolverton model [104] can all be described as 'General Poisson Models'; they are all deterministic reliability growth models and share the first four assumptions of the Musa model. An assumption made by these models is that no new errors are introduced as faults are fixed. In general this assumption is not true and can therefore lead to invalid estimates. The Musa model partly overcomes this problem by using an estimate of the total number of errors that will be found (dtot), and therefore takes into account errors introduced by fixing, so long as the estimate is accurate.

Ramamoorthy and Bastani [105] suggest that there are three ways of modelling failure rates:

1. deterministic - the general Poisson models are deterministic reliability growth models as the residual failure rate decreases by a known amount after each error correction.
2. stochastic - different errors have different failure rates, so a better approach is to model a random failure rate. The stochastic reliability growth model does model a random failure rate, but it still has a constant failure rate between error conditions.
3. bayesian - a bayesian model shows a varying failure rate between the error conditions. The failure rate is viewed from the testers point of view; the longer the period of time from a failure the more confidence a tester will have in the program.

The three approaches are shown below in Figure 17 Failure rate models, page 92.

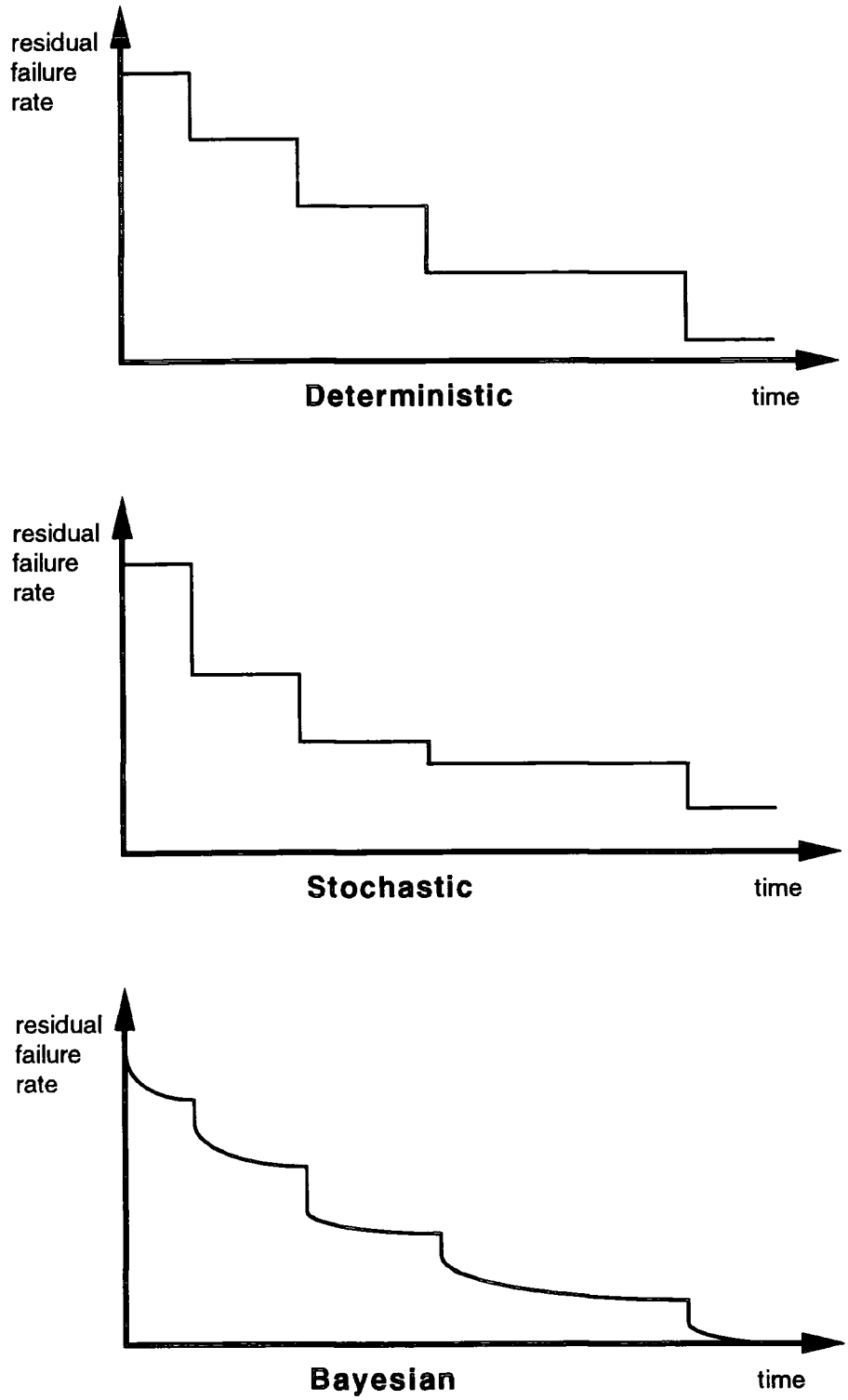


Figure 17 Failure rate models

Littlewood & Verrall [106] consider the software failure process to be the result of two sources of uncertainty; the selection of input and the state of the program. A set of inputs will cause a program to fail and these inputs will be arrived at randomly. The state of the program will change with fault fixing. If

all faults are corrected first time, the reliability growth is deterministic. This model assumes that the faults are probably fixed and that the reliability growth is probabilistic. Littlewood and Verrall have used the Bayesian reliability growth approach as the basis for their model.

The Remus and Zilles [107] defect removal model for predicting the number of errors remaining at various stages of development is outlined below, Figure 18 Defect removal model, page 94. This model also does not expect perfect fault fixing. It has been developed from work by Jones and Turk [108] on a model of the defect removal process. The model makes use of easily made measurements, such as the number of major problems discovered during reviews, inspections and testing to estimate the number of defects that will be discovered during the products lifetime. Also estimated is the cumulative efficiency of testing (CRE), and the rate at which defects propagate from one defect removal step to the next (1/m).

MP = major defects found during inspections and reviews
PTM = number of defects found during testing
TD = total defects over life of product
Q2 = remaining errors over life of product
CRE = cumulative removal efficiency

$$\mu = \frac{MP}{PTM}$$

$$TD = MP^{*} \frac{\mu}{(\mu - 1)}$$

$$Q_2 = \frac{MP}{\mu(\mu - 1)}$$

$$CRE = \frac{(\mu^2 - 1)}{\mu^2}$$

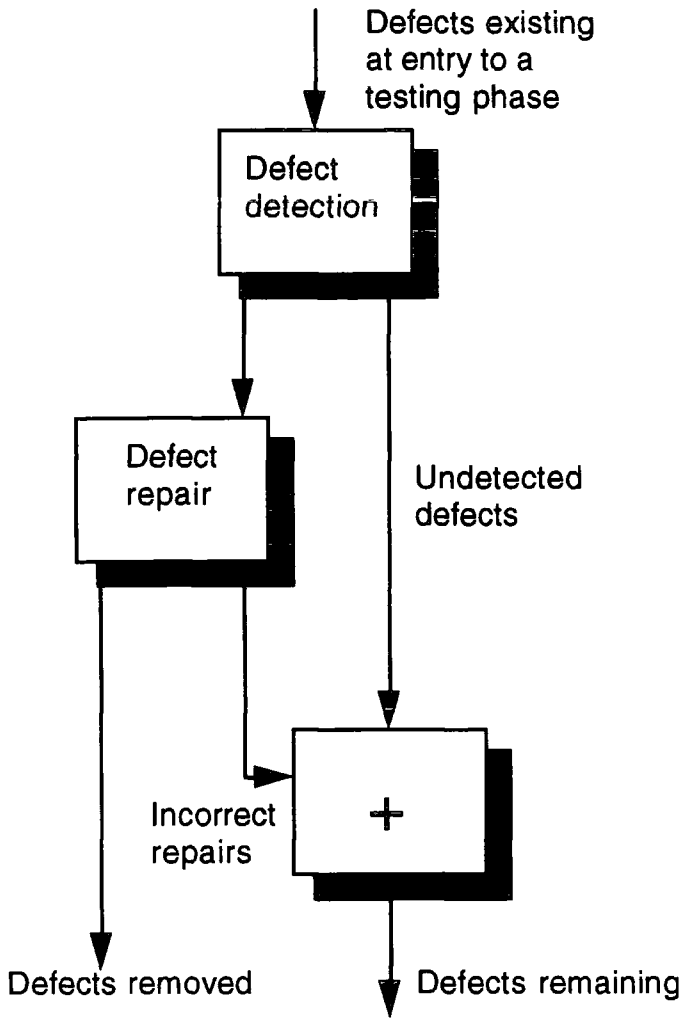


Figure 18 Defect removal model

Studies to find a universally 'best' model have not uncovered a single, simple mathematical model that can cover the great variety of software. Littlewood [109] argues that comparison of models should be based on the quality of the software predictions and not their underlying assumptions and plausibility in relation to software engineering practices, "Mathematical modelling to predict/forecast software reliability is intrinsically difficult

- software is pure design
- software fails because of intellectual faults
- understanding the problem is a social process

Sociology and economics may be better metaphors than physics: this does not bode well..." [64].

Beizer's [90] view is that "no one model of software reliability appears to have a distinct superiority to any other, and none seems ready for practical application. No person in their right mind today should agree to contractually binding software reliability requirements or predictions. That is, making a proven minimum mean-time-between-failure, say, a contractually binding component of system acceptance". Beizer lists six criticisms of software reliability models:

1. There is no notion of bug severity
2. There is an implicit assumption that testing is done at its worst instead of its best. Good testing is not random nor in any sense uniform across the set of program paths. Good testing focuses on improbable paths. No model offered takes into account the tester's integrity, cunning and dedication to make the system crash.
3. High-load crashes are more frequent than low-load crashes. Bug discovery's not only a function of the number of transactions executed under test, but also of the rate at which the transactions were presented to the system.
4. The efficiency of debugging is assumed to be independent of the systems reliability. The longer the systems been in operation, the subtler the remaining bugs are and the dumber the debugger. Both of these lead to a constant number of bugs in the system over time.
5. The impact of maintenance is ignored.
6. Bug discover rate is assumed proportional to test intensity, or constant with time, or decreasing with time. In reality it goes up and down.

4.5 *Summary*

Sizing models are best suited to particular development environments, and some of them require a database of past project information before they can be used effectively. The USAF [65] evaluation shows the wide range of results from the sizing models, one of them providing an estimate four times larger than the target. The estimating models like COCOMO provide much more information, in the form of costs, effort and manpower profiles but they do tend to be much more complex than the sizing models and therefore require more effort to use them. Some of the estimating models also need as input, an estimate for the size of the software to be produced, before they can provide the cost and manpower predictions. If this is the case, then the inaccuracies of estimating the size in the first place will be passed on into the further predictions. In general, the use of these two types of models on real projects, should be treated with caution. Validation of the model to be used against past project data from the proposed development environment is advisable before a commitment to use the model is made.

The complexity metrics are best employed in keeping module complexity within certain limits and for the direction of testing resources to modules with high complexity and associated high defect counts. In the same way, the defect and reliability models are best used to identify the modules that will have the high defect counts rather than trying to determine the exact number of defects in the software at any one time. The use of any of these models appears to be wide open[i]; because of the need to tailor them to the intended environment (or build a database of past project data) it is very difficult to prove in advance that they would be cost effective for a particular company or project that has not used one before.

[i] There are no restrictions to the use of these models, but also no well defined and demonstrated advantageous.

5 A Fault Propagation Model

Fault finding is often considered as a single activity during 'testing', unfortunately the real life situation is complex, due to the interactions between parallel activities within the development life-cycle. Therefore before it is possible to start measuring the effectiveness of testing, an understanding of faults is needed. Where are they introduced and discovered, in a life-cycle context, and how do the measures of testing coverage relate to the chance of finding the faults.

This chapter introduces original research which has resulted in a new fault propagation model and techniques to describe the flow of faults and the relationship between faults and test coverage. An explanation on how these techniques and model, constructed for this thesis, represent the parallel flow of faults at each stage of the life-cycle is provided.

The fault propagation model is required to demonstrate in a simple diagrammatic form the possible paths of a fault through the development lifecycle from introduction to discovery and on to fixing, plus the parallel nature of fault propagation. In addition to the opportunity of creating further faults whilst trying to fix one, there is also the possibility that a fault will not be fixed and will loop around the fault propagation model again until discovered. To understand and control the testing process measurements are required for each path in the model and the effectiveness of the process applied at each phase of the lifecycle. The control metrics explained in chapter 6 & 7 of this thesis are based on the mapping between the creation and discovery of faults and rely on the ability to distinguish between initial development errors and errors made whilst fixing a fault.

A software error introduction/removal model (Figure 19 Error intro/removal model, page 98) was developed by Boehm [16] as a conceptual model to aid in the understanding of software reliability during software production. It was used to define a 'required reliability' factor for the COCOMO model. The model shows that a percentage of errors are introduced at each stage of the life-cycle and a percentage eliminated during other stages. The model also suggests that the percentage eliminated at a stage is dependent upon the amount of resource (cost) applied during the error avoidance or error removal activities. In a section entitled 'difficulties in completing the picture' Boehm admits that each error removal activity will be highly effective against some classes of errors, and much less effective against others.

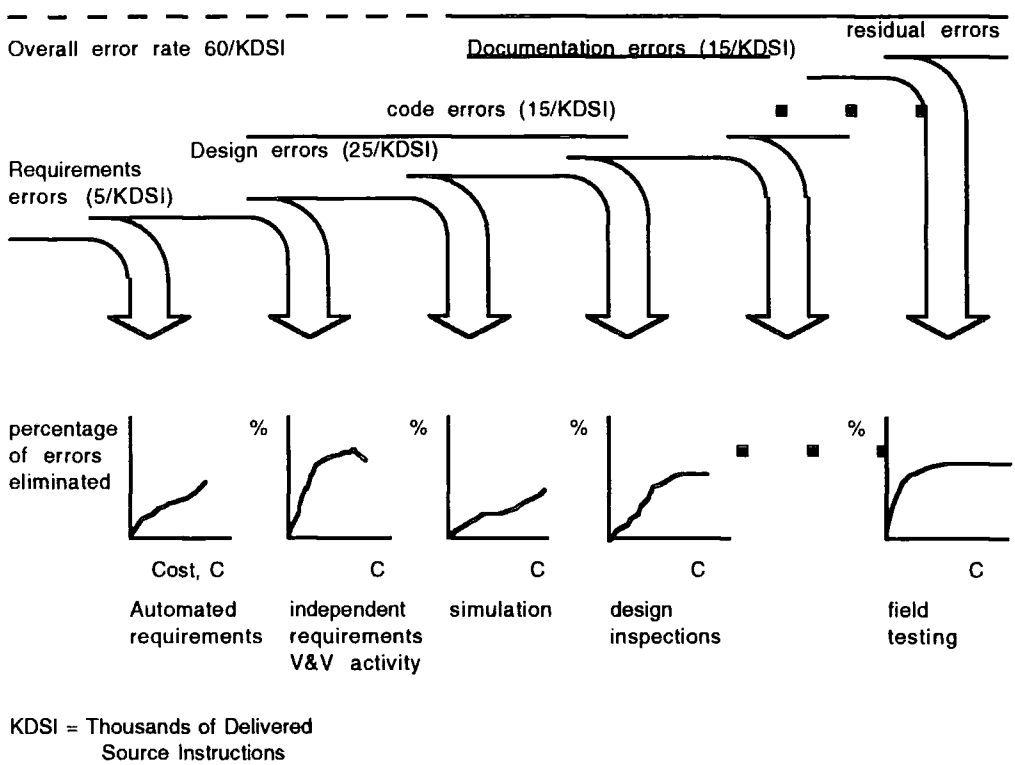


Figure 19 Error intro/removal model

Although there is not a simple relationship between faults removed and cost, the principle of adding cost drivers into the model for the different removal processes would be valid and useful if the capacity for finding faults by using a particular method is known in relation to the total number of faults present.

As each removal method will only be effective against certain types of faults the percentage of each type of fault in the system is needed before this principle can be put into practice.

Norris et al [110] analysed a set of projects to determine where the faults were actually caused, as most of the data available today, like the Boehm model, is either old or based on projects in the USA. The Norris data set, they admit, is based on a small set of projects, but it does cover the maintenance phase:

Requirements	38%
Design	23%
Installation	7%
Maintenance	32%

The after acceptance figure of 39% to cover deployment, is typical of development projects as a large proportion of the spend on a project will be during maintenance, but the 38% on requirements does not align with the Boehm model. The requirements problems where split between functional and non functional requirements (performance, reliability, usability etc.) and this could point to greater expectations from customers and more emphasis today on the quality aspects of a product.

In addition to not covering the maintenance phase the Boehm [16] error introduction/removal model does not include the possibility of fault removal activities introducing further faults or not fully correcting the ones being worked on. To be of use during software testing a model that considers the propagation of faults through the life-cycle, including those introduced during fault removal activities is required. A fault propagation model has been constructed for this thesis that supports all the fault propagation paths through the life-cycle, with the use of set theory notation and Venn diagrams to describe relationships and results.

5.1 Propagation of faults through the life cycle

The ripple effect of changes and the parallelism of the modification phases, due to the discovery of faults, results in a complex environment in which faults are found and corrected. To understand how faults enter a product, are found and resolved is best explained by looking at the development and testing phases with the paths that faults take between the phases highlighted.

An overview of the proposed fault propagation model is shown in Figure 20, page 102, which provides a diagrammatic view of the phases and the top level fault paths. Each of the development phases is shown as consisting of a main activity, like system design, and also the process of fault correction for that phase; e.g. modify system design. The main fault 'highway' is shown as a thicker line in the diagram, this is the route for the faults between the development phases. This representation follows the life-cycle presented earlier and also the expectation that the earlier the fault is introduced the later in the life-cycle it can be detected. This is shown by the testing phases removing faults from the main highway and passing them back to the relevant development phase for correction. Validation testing is shown passing faults across to the modify specification phase.

It is only with good quality development methodologies and procedures that the data required to see this overlay of fault propagation paths on top of the life-cycle can be found. With an undisciplined approach to software development it is not possible to filter out these paths and therefore expose the fault generating culprits. Without adequate control it is indeed very difficult to predict the state of the product with all the phases operating at once.

To simplify the diagram text and equations representing the flow of faults, a short hand form of identification is given below using set theory notation.

software development phases:

r	requirements
s	specification
sd	system design
md	module design
c	code
mt	module test
it	integration testing
vt	validation testing
ct	customer testing and use
a	all phases

a set of measures:

F	faults found
I	faults introduced

examples:

$I_s \cap F_{vt}$ = the faults that are introduced during specification and found in validation

$(I_s \cup I_{sd}) \cap F_{ct}$ = the specification and system design faults found by the customer

Each of the development life-cycle phases is taken in turn, with an explanation of fault activity and a detailed diagram which is an expansion from the basic building blocks shown in the overview Figure 20 Fault propagation model, page 102.

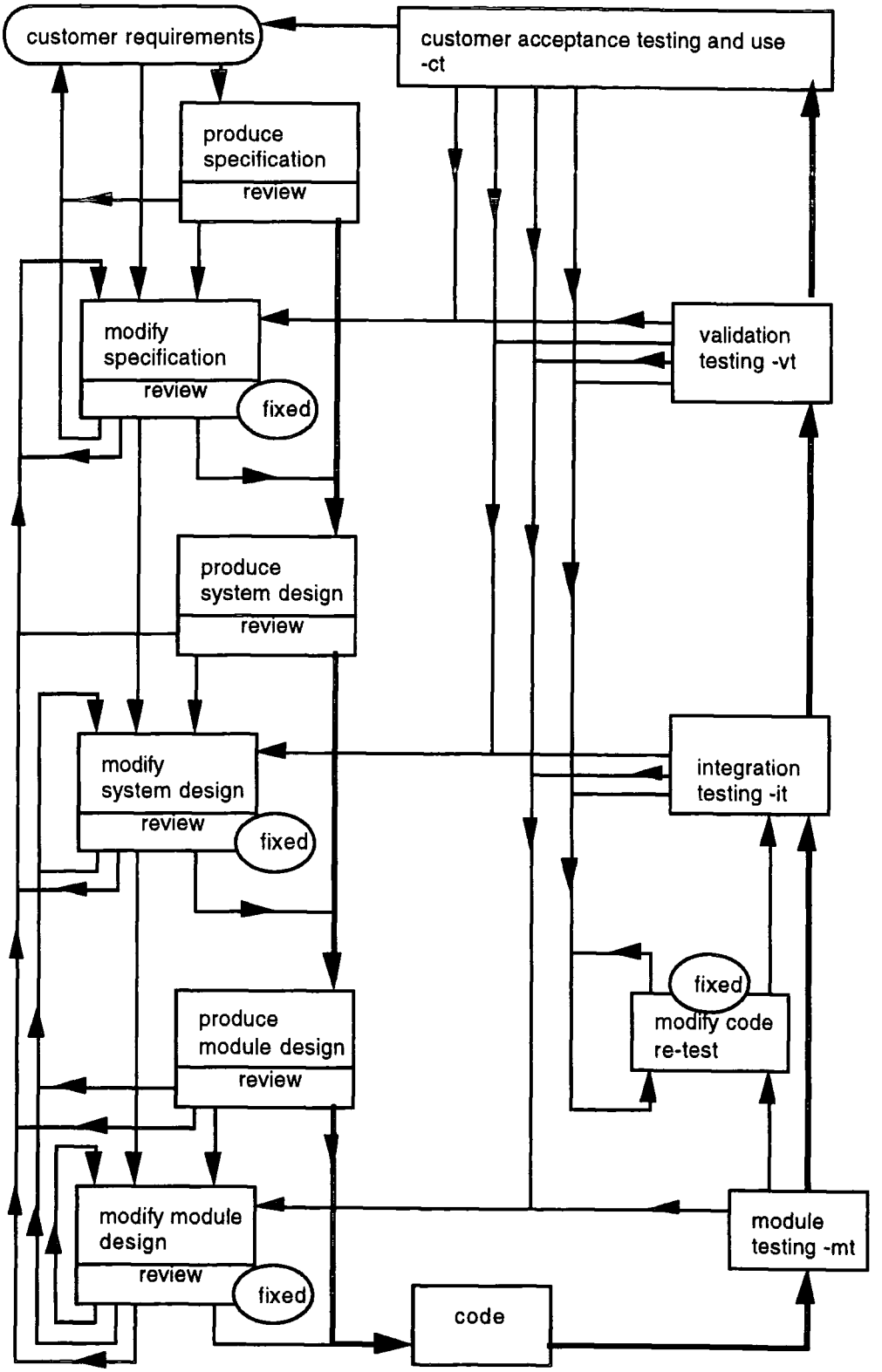


Figure 20 Fault propagation model

5.1.1 Specification

The specification is produced from the requirements, but there is likely to be faults with the requirements which will be reflected in the specification. Faults will also be introduced by the specification process itself; as shown by the 'process errors' arrow in Figure 21, page 104. On completion of the specification there is a review procedure to capture faults introduced by the errors in the specification process. Not all of the faults will be discovered and the development process will continue with a requirements and specification that contain undiscovered faults. The faults that are uncovered by the review are dealt with by the 'modify specification' process. Again not all the these faults will be removed correctly, in fact the modification process is likely to introduce more new faults (per hours effort) than the original specification work. Some will be found by review, others that slip through the review will remain in the specification.

The modification of the specification process has a number of other input routes. Change requests for the specification might well occur as the customer requirements change. Customer requirements do tend to be dynamic and will change with the market opportunities and the movements of the competitors. As the development proceeds there will also be feedback, firstly from system design and later from module design due to technical or commercial changes that become apparent when that point of development is reached. These design changes might force changes in the specification to keep the two in step.

Later still in the project, specification faults will be exposed by the validation testing(vt) or even the customers (ct) use of the system. Some specification faults might also show-up towards the latter stages of integration testing (it) when defects with functionality across the system become apparent. It is very unlikely that any specification faults will be found during module testing, due to the low level of this type of testing and the lack of an overall system view. No feedback line has therefore been shown from 'module test' to 'modify specification'.

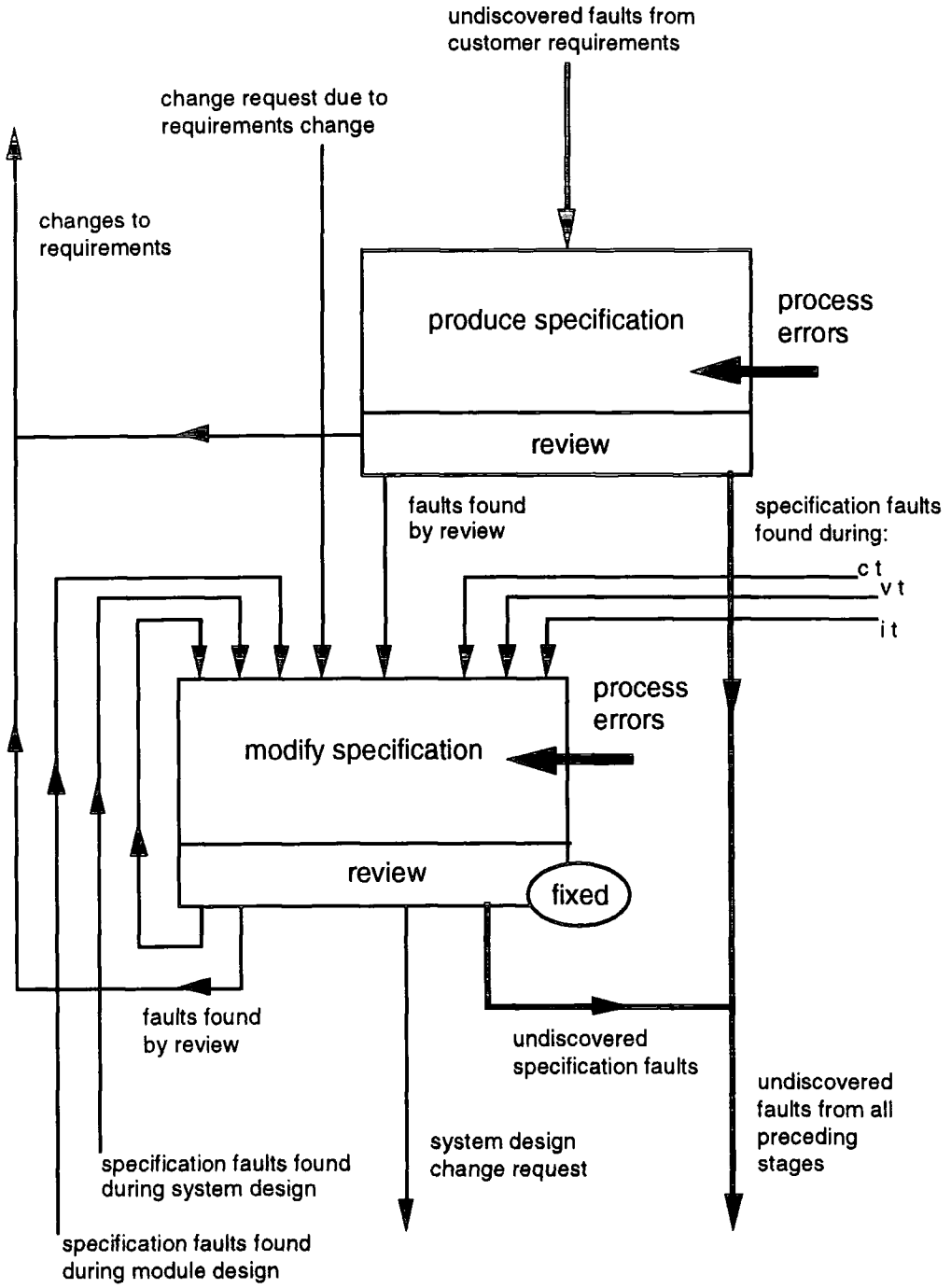


Figure 21 Specification

5.1.2 System design

System design inherits a requirements and specification that may contain faults. If they are not found during the system design activity, or by subsequent review, then they will be contained in the system design as faults. The system design process is also subject to error, and the resultant increase in system design faults. System design faults not found at review will be passed on to module design; those detected will be corrected by the 'modify system design' process. Problems relating to system design that are found during module design, integration testing, validation testing or customer use are all directed to this process for correction.

Modifications to the system design might not correct all the faults returned and may also introduce some new ones in the process. Change requests for the system design caused by modifications to the specification will also arrive at the modification process. These later changes are more likely to introduce new faults than the original design work. The review stage is there to capture errors with this process, but some faults will find their way down to the module design.

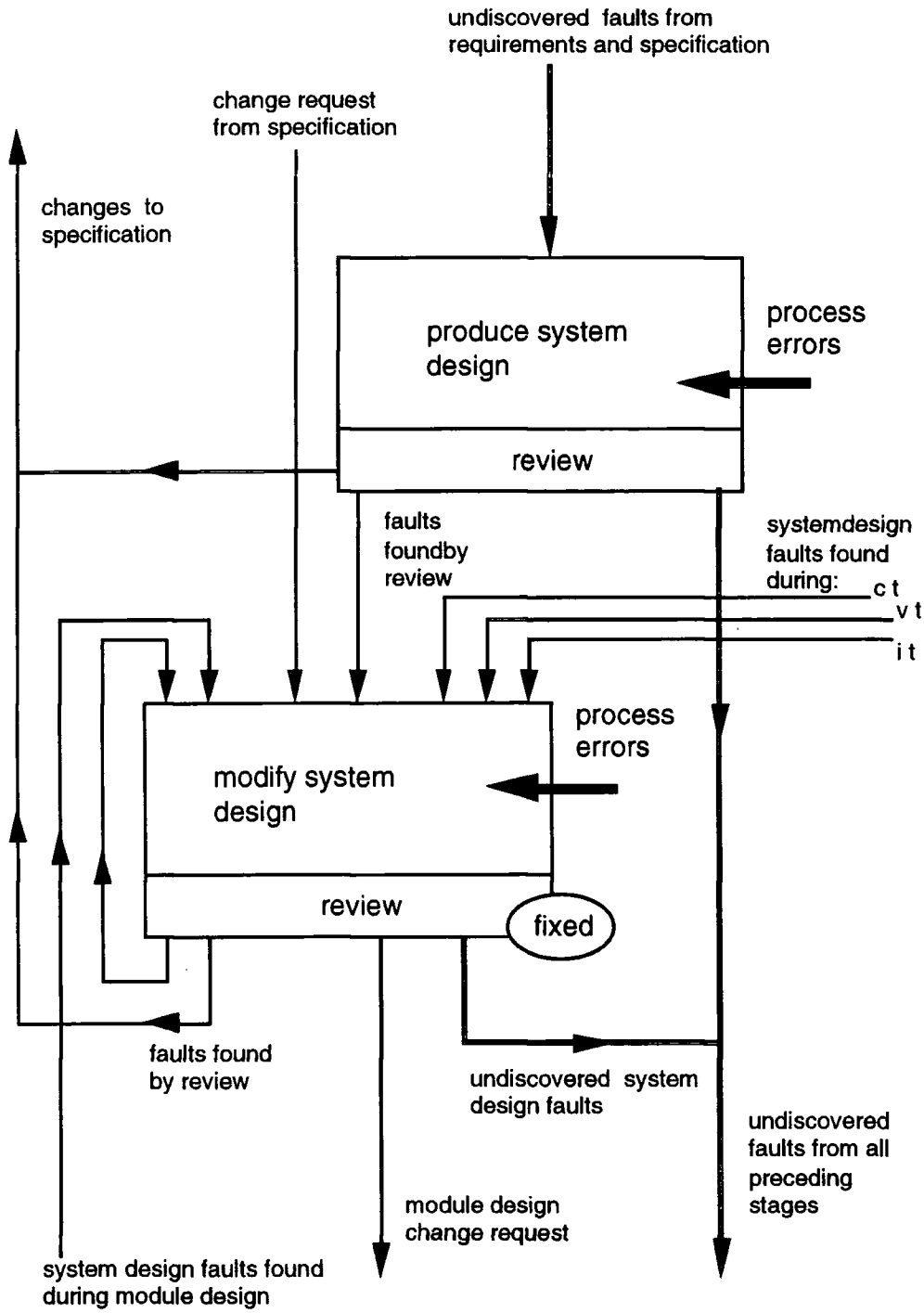


Figure 22 System design

5.1.3 Module design

Undiscovered faults in the proceeding stages ripple down to cause problems with the module design. Faults with the specification and system design unearthed during module design are fed back to be corrected. As all the testing stages have the potential for locating module design problems, there are feedback paths from module, integration, validation and customer use to the 'modify module design' process. Also change requests from system design will be acted upon by making modifications to the module design. This is the last stage before coding, where any undiscovered design faults will then be implemented in code.

5.1.4 Coding and Module testing

The design is implemented by the coding process (Figure 24 Code and test, page 109), which may flush out some of the module design problems. But it is unlikely to find system design or specification faults due to the level of detail entailed in coding; 'cant see the wood for the trees' fits the coding position very well. The same applies to module testing; coding and module design faults are found, but no impression is made on the specification and system design faults. A view of the paths inside the process blocks of module code and test is given by Figure 25, page 111 which shows how the higher level design faults pass straight through the module test process. The diagram also shows that a percentage of the module design and coding faults will be found.

The path for coding and module design faults is split, with the ones not found feeding into the main highway of undiscovered faults and the ones that have been found passed back for modification. The actual value for this percentage can be calculated from the equation below, but only as a post-mortem exercise when the product has been retired. It is only then that the total number of faults would have been found. While the product is still in use there is always the possibility of another fault being found by the users.

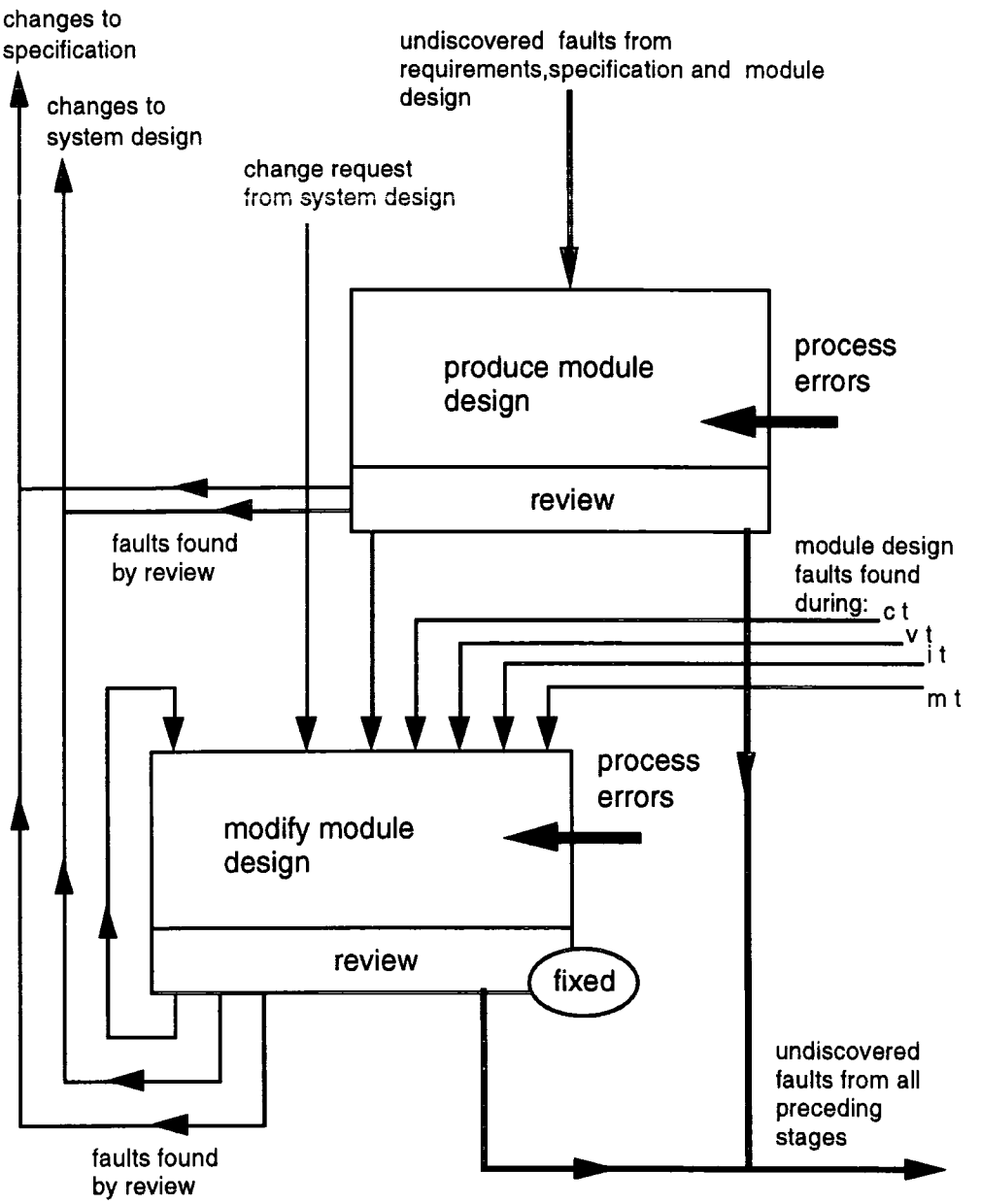


Figure 23 Module design

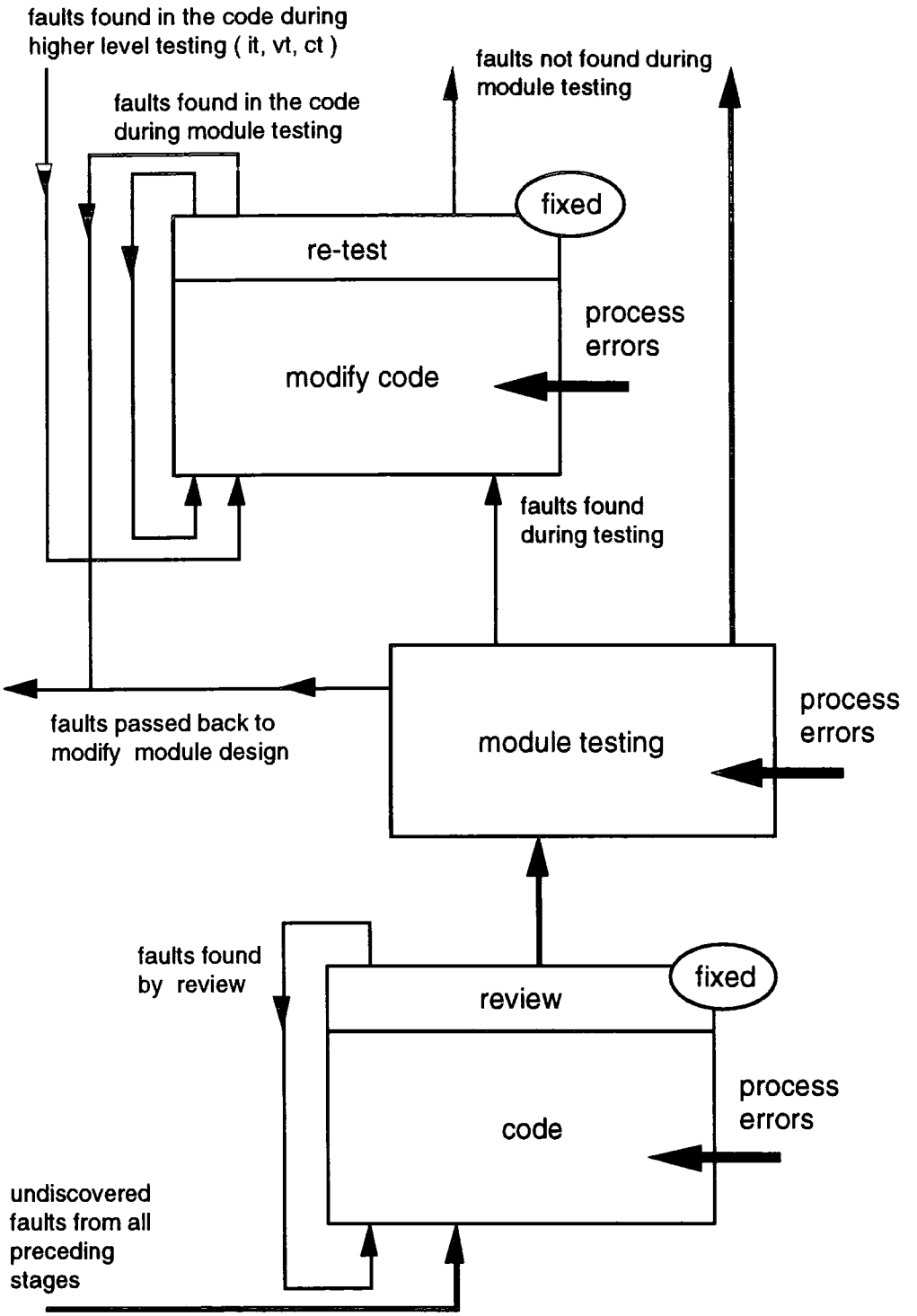


Figure 24 Code and test

The number of module design and coding faults found during module test,

$$\left|(I_{md} \cup I_c) \cap F_{mt}\right|$$

The total number of module design and coding faults found during module, integration, validation testing and customer use,

$$\left|(I_{md} \cup I_c) \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})\right|$$

The percentage of module design and code faults found by module testing: =

$$\left|\frac{(I_{md} \cup I_c) \cap F_{mt}}{(I_{md} \cup I_c) \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})}\right| \times 100$$

The top line is affected by three factors:

A - The set of the test cases. This is difficult to calculate for large modules as the number of possible test cases increases dramatically with size.

B - The number of test cases that are incorrect and do not therefore uncover faults that they should have.

C - The number of test cases that do locate faults but are not recognised as such due to poor analysis of the test results.

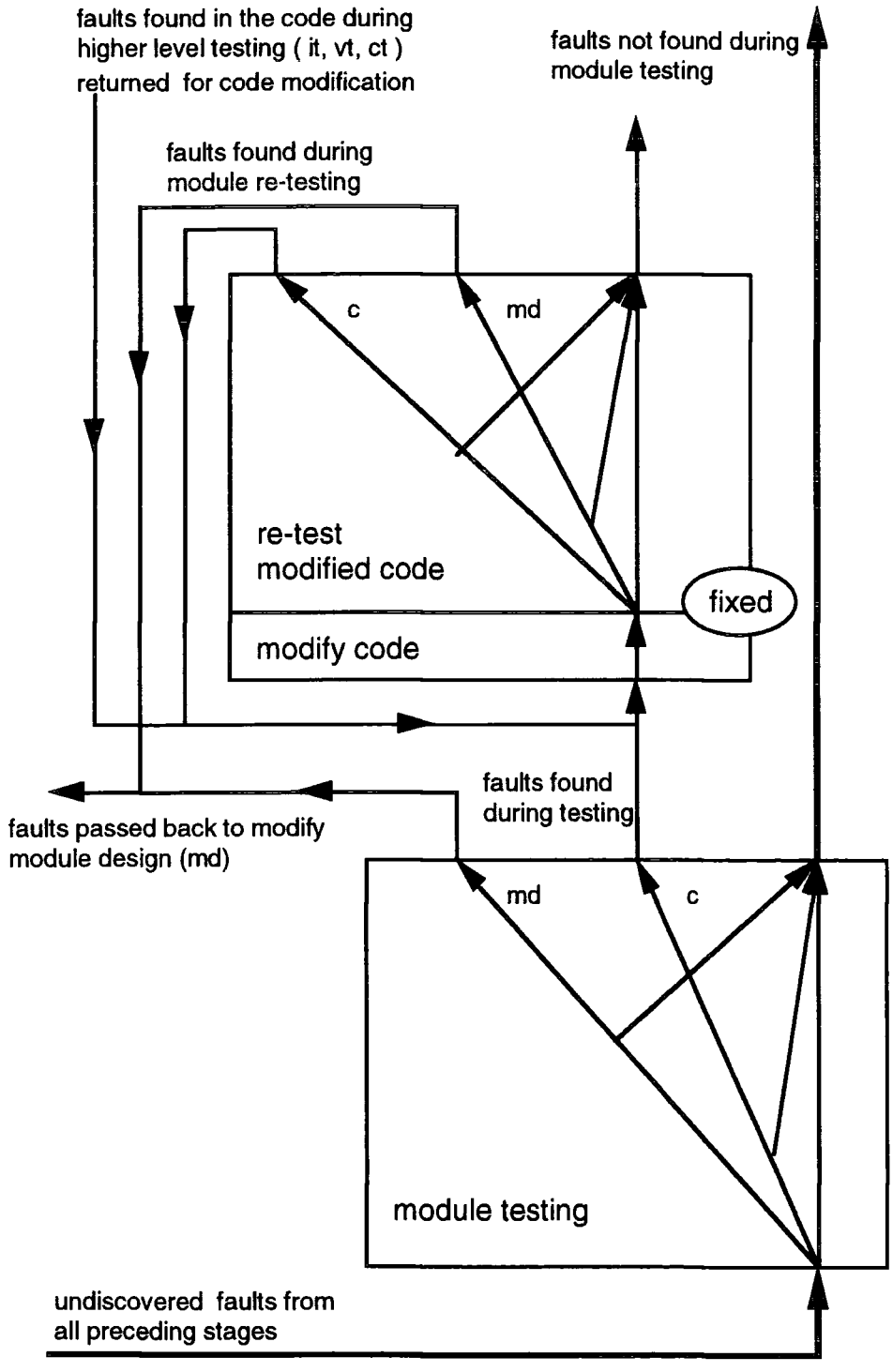


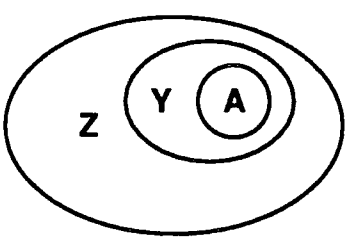
Figure 25 Internal, code and test

Due to the limitations imposed by the three factors above there is a restricted possibility of finding module design and coding faults. The maximum percentage possible is calculated in the equation below.

$$possible\%ModuleFaults = \left| \frac{(I_{md} \cup I_c) \cap (A \cap \overline{(B \cup C)})}{(I_{md} \cup I_c)} \right| \times 100$$

An explanation of coverage is needed as it can be interpreted a number of ways. It is not the tests that will be run from the total of all possible combinations, but the tests to be run from the representative combinations. It is not humanly possible to test every single combination of data and use, for even a fairly small module. The starting point is to decide what would be a sensible set of tests, to give a high confidence rating for the testing. Boundary value analysis, equivalence partitioning and typical values are used to restrict the test cases to a sensible level. The coverage is then calculated as the percentage of these tests that it is proposed to run. Obviously different test engineers will take the boundary value analysis and equivalence partitioning to different levels, they will have different views on what represents typical values, so calculating and interpreting sensible test coverage is not an exact science.

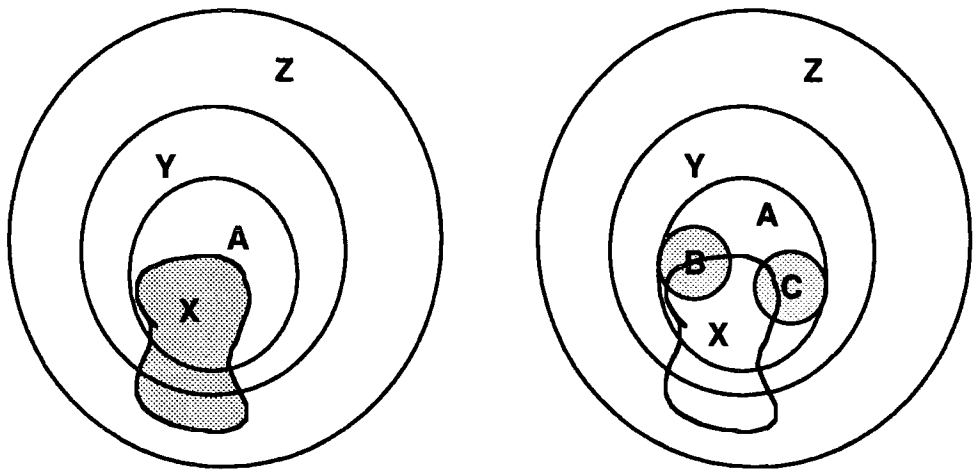
Set theory is an ideal medium in which to explain the relationship between the faults and the test coverage. The set of all possible failures with a system and also the set of tests needed to find all those possible failures is represented by the reference set 'Z' in the Venn diagram below.



Venn diagram 1 Reference set

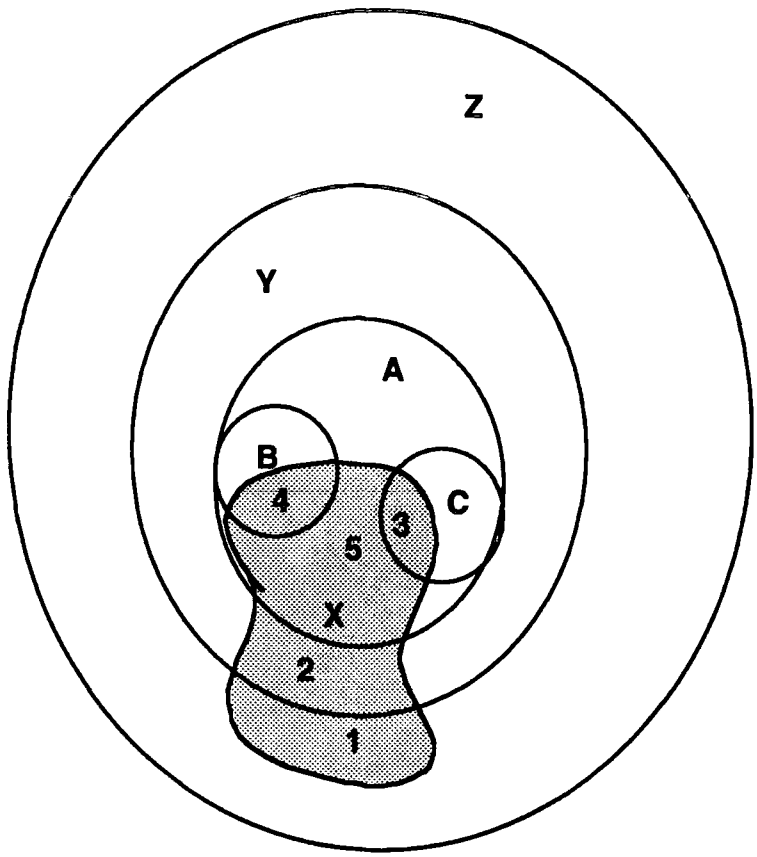
This total set of all possible failures is very large, it could even be considered as tending to infinity for most commercial software packages. The representative combinations 'Y' are a subset of 'Z' ($Z \supseteq Y$) while still a large number, they are less than infinity. In addition to showing the set of tests that make up the representative combinations, set 'Y' is also the set of failures that the tests

would uncover if they where used. The actual test coverage applied during testing, and the possible faults that could have been found are shown by 'A'. The set 'A' is a subset of 'Y' and also therefore of 'Z' ($Z \supseteq Y \supseteq A$). A set of actual faults 'X', which will be a subset of 'Z', can be superimposed onto the Venn diagram, as an example to show which faults will be found.



Venn diagram 2 Fault & coverage set

By adding in the sets 'B' (incorrect tests) and 'C' (poor analysis of test results) see Venn diagram 2 Fault & coverage set, page 113, where 'B' and 'C' are both subsets of 'A' and splitting set 'X' down into a number of areas the result of this coverage set and fault set can be described, see Venn diagram 3 Set X, page 114.



Venn diagram 3 Set X

Set 'X' (1-5), actual faults within a system:

1) $X \cap (Z \cap \overline{Y})$ Faults are not found, as this area is not even covered by the representative combinations. Nobody has considered this part of the system with regard to testing.

2) $X \cap (Y \cap \overline{A})$ This area has been considered and is documented as part of the representative combinations; but no tests have been included in the actual test coverage and the faults will not therefore be found.

3) $X \cap C$ The intersection between the actual faults 'X' and poor analysis of test results 'C' leads to the faults not being recognised although the tests have been run. Note that 'B' and 'C' are subsets only of 'A', as it is not possible to have incorrect tests or poor analysis of tests that do not exist as part of the test coverage to be applied.

4) $X \cap B$ This intersection between the actual faults 'X' and incorrect test cases also results in the faults not being found.

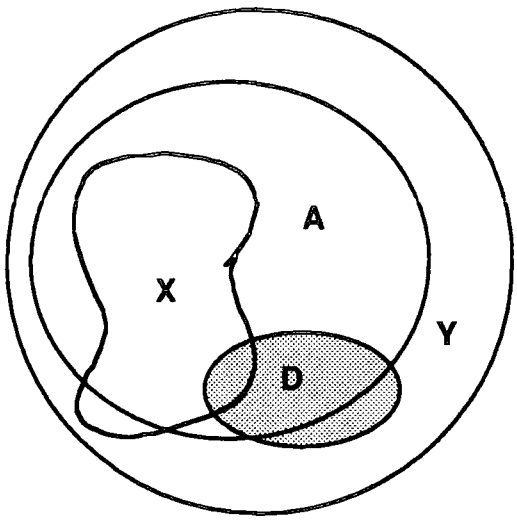
5) The faults are found in the area shown as the overlap between the test coverage 'A' and the actual faults 'X' minus the areas 'B' and 'C'.

$$\text{faults found} = \left| X \cap (A \cap \overline{(B \cup C)}) \right|$$

The aim in test selection is to achieve the best possible overlap between the tests to be run and the faults. The ideal is a test set which completely encompasses the set of faults but with a minimum of tests over and above the ones needed to find all the faults. The aim during test preparation, running and analysis is to reduce the 'B' and 'C' areas to the empty set, by not producing incorrect test cases or missing faults during the analysis of results.

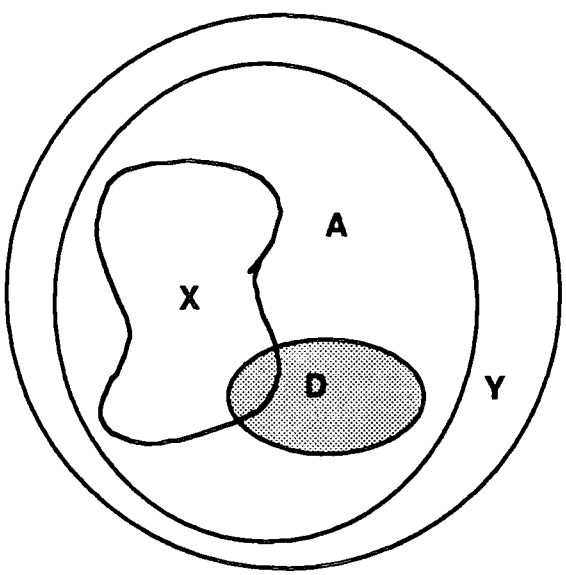
The faults that are found are solved by modifications to the module design or re-coding. The tests that initially found the problem are then repeated, plus some extra regression tests to gain confidence in the changes and to show that no new faults have been introduced.

The results of initial testing and bug fixing is shown in Venn diagram 4 Initial testing, page 116. Set 'D' represents the outstanding faults after bug fixing. The intersection of 'X' and 'D' are the original faults that have not been cleared correctly. The rest of 'D' covers the new faults introduced during the modification process. Some of these new faults lie outside 'A', the test coverage attained with the test set, which means that these faults will not be found if the test coverage remains the same. The area of 'X' outside of set 'A' are the original faults not found during the initial testing and will remain hidden if the test coverage does not change. The rest of 'X' within 'A', but not overlapped with 'D' are the faults successfully found and cleared.



Venn diagram 4 Initial testing

To regression test set 'X', increase the tests to capture all of 'D' and the area of 'X' not covered originally, the test coverage 'A' will need to be increased as shown in Venn diagram 5 Regression test set, page 116.



Venn diagram 5 Regression test set

Those faults correctly solved are represented in the fault propagation diagrams (Figure 24 Code and test, page 109; Figure 25 Internal, code and test, page 111) by the oval with the word 'fixed' inside it. The faults that have not been correctly solved and the faults added due to the modifications, that then slip through the module testing, will pass on to the integration testing phase.

5.1.5 Integration testing

The tested modules come together, with the interfaces between them being exercised and functionality across the complete system being tested when the majority of the modules have been integrated. It will be mostly system design faults and module interface faults that are found at integration time, although some specification faults might be found when the functionality across the system is exercised. Any faults found at this stage are passed back to the relevant area (specification through to code) for correction. This phase is also susceptible to process errors as well as lack of coverage, with both of these leading to faults that were introduced earlier not being captured at this point.

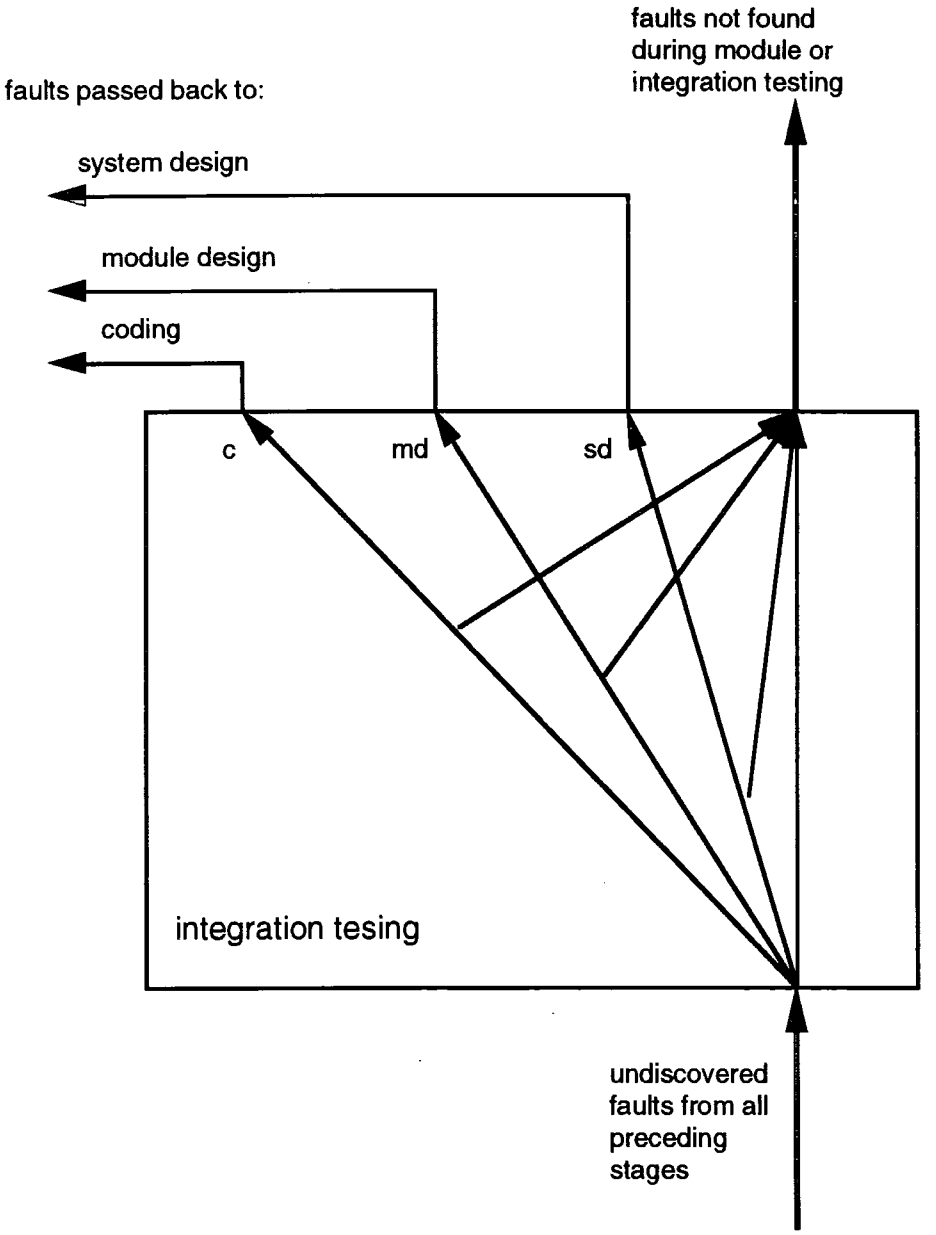


Figure 26 Integration test

5.1.6 Validation testing

The aim of validation testing is to show that the developed product is the one that the customer actually wanted. In addition it must be possible to install, maintain and extend the product, along with it being usable. Performance and reliability should be in line with the customer requirements. Therefore faults with the specification are usually found at this stage because the testing

environment is so close to the customers. Other lower level faults found at this stage indicate missing tests at the integration or module testing stage. This type of testing is the last chance to exercise the product before it goes to field trial.

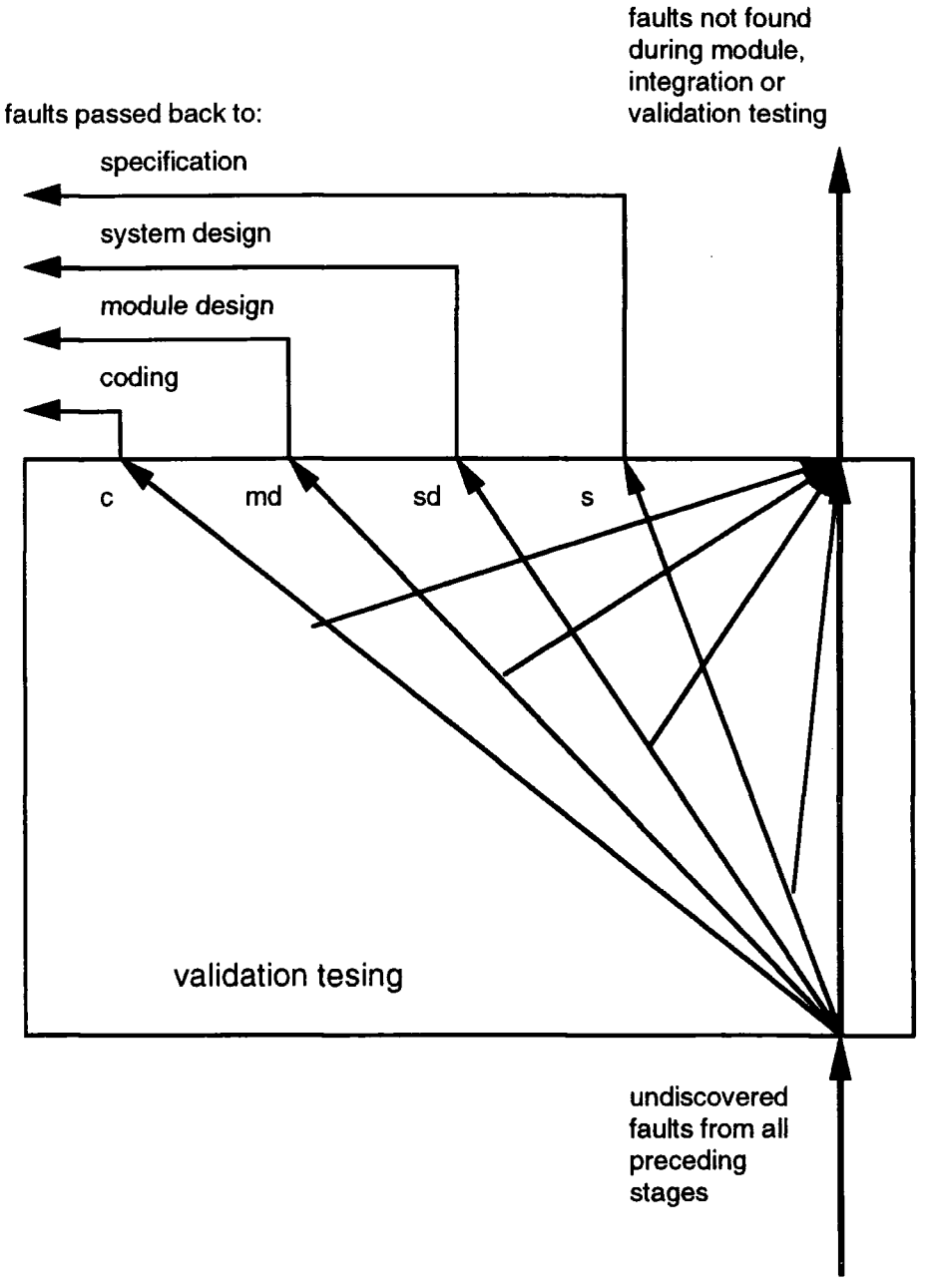


Figure 27 Validation test

Coverage is not so much of a problem for validation testing, as it would be impossible to achieve a high coverage at this level of testing. Instead customer data and usage patterns are used to flush out the faults that a customer is likely

to experience. The main reason for faults not being found will be due to not selecting representative data and usage patterns for the testing.

Over the life of the product the customer will find a number of faults which have been missed by the various testing phases. Once a fault has been identified by the customer, and passed to the support group, the reason for the fault is investigated and then passed to the relevant phase for modification. There will be a residual level of faults in the product that are not found by the customer. These are the type of faults that require a specific usage pattern and data that has not been used during testing or by the customer, or an unusual set of circumstances to trigger the fault. If they have not been found by the time the product is retired then they do not give any cause for concern, in fact if they have not been found by this point in time no one is aware that they exist.

The expansion of the testing boxes (see Figure 25, Figure 26, Figure 27) shows the split between the faults being found at that stage and those that continue on through to the next level of testing. By collecting data on where the faults are introduced and where they are found it would be possible to evaluate if improvements to the development or testing process really are providing improved products with less faults, or just shifting the problems to another part of the life-cycle.

5.2 Parallel processes

The phases have been stepped through one at a time for explanation purposes, but in real life once the product has reached validation all the modification phases will be in operation at the same time. The faults found during the different testing phases will be passed back to the relevant modification phases which in turn will produce changes which have knock on effects elsewhere in the overall process. To appreciate what is taking place within these parallel processes, measurements at each of the phases are needed to build up a composite picture of the activities.

A set of metrics are required for the collection of this data, but there will be a large amount of data to collect because of the low level within the overall system that this data becomes apparent. The answer is automated collection, the

question then becomes 'what automated system?'. All the fault data identified so far will be entered onto the problem reporting scheme for the development project, so a link into that system could provide the capture and recording point for the data. This possibility will be given further consideration in section 9.2 of this thesis looking at the operational use of the proposed model.

5.3 Summary

This chapter introduces the results from original research into fault propagation, with a new fault propagation model proposed. It has also been demonstrated how this model can represent the detailed flow of faults at each stage of the development life-cycle, using set theory notation to capture the flow between phases. The detailed fault flow present at each life-cycle phase has been explained, phase by phase, and it has also been pointed out that in a real life project, all of the phases are running in parallel and therefore creating a complex fault propagation situation. Venn diagrams have also been used to explain the relationship between faults and test coverage.

6 Hypothesis

This chapter describes a new metric Earliest Visibility EV, and the hypothesis that there is a relationship between the phase that a fault is introduced and its EV. This leads on to provide a method for determining the true stage of a testing activity in the testing life cycle, based on the EV of faults found during the testing activity.

6.1 A testing process control model

There are contrasting views on how useful defect prediction can be to industrial software development projects. Ferdinand's view [50], "the goal of a well-planned software development project should not be to predict fewer defects during development. The goal should be to correctly predict the defects that are inherent in that statistical process, and then extract the defects that were predicted. Furthermore, quality of the resulting software has little to do with the number of expected defects. The quality of a delivered product is a function of the residual defects that exist in the software when it is put into productive use, and it is also reflective of whether or not the functional content of the software meets the requirements of its users".

Beizer [90] takes a different view. " The question of when the system will be ready for cutover seems at first glance to be intimately related to the number of bugs remaining - but it doesn't have to be. Suppose we've designed a set of tests which we believe will constitute an adequate check-out of the system. The builder and the buyer have agreed that when that set of tests has been passed, the system is ready for operation, no matter how many bugs remain be it one or a thousand. The pragmatic issue, then, is not how many bugs remain but when

is the system considered to be useful - or at least sufficiently useful to permit the risk of failure. I never saw a system not go into operation because we couldn't predict how many bugs it had".

Predicting the number of faults in software does not therefore have a practical application unless the criticality of the problems and the likelihood of them being found by the user can also be predicted. The testing process control model proposed in this thesis will concentrate on improving the control of the process to remove the maximum faults, but will not predict the number of faults to be removed.

Earlier in this thesis (see section 5.1) it has been described how faults introduced in the different phases of the development life-cycle are tested for during specific testing phases; e.g. system design problems are found during integration testing. The hypothesis constructed for this research proposes that there are certain categories of faults that are associated with specific development phases, and that the faults created are exposed in a linked testing phase. If this link between types of faults, there inception and detection, exists, then it would be possible to predict the position in the development life-cycle, from the type of the majority of faults being found at that time.

By collecting data on the faults being found, analysing the data to determine which category of fault is being found, and then comparing the results with the fault propagation model it will be possible to determine which phase of testing is actually taking place. If predominately coding faults are being found in what the project manager believes to be validation testing there will be some resource and elapsed time problems to face before the product is ready for release. This is because the testing taking place has only progressed as far as module testing.

At present projects might move from one phase of testing to another due to a variety of reasons:

- the project plan is followed exactly without consideration to the real state of the project
- the budget/time for testing has been reduced and therefore moving on to the next phase of testing early will save time/money

- the general opinion is that the next phase has been reached
- all the tests have been run for the current phase of testing, therefore the next phase is due
- not many faults have been found, so it must be time to move on

The importance of understanding which testing phase is taking place can be explained in terms of the quality of the testing. Module testing verifies that the coding process has not introduced any faults and that the coded module performs as specified in the module design. In addition the coded module must respond to illegal or error conditions in a sensible manner. Defensive coding of the module is needed to trap possible error conditions that might arise in use and recover in a predetermined way when something unexpected does take place. This type of testing is only sensible at module level when harnesses surrounding the module code can control completely the environment encapsulating the module.

As testing progresses through the testing life-cycle phases, this tight control over the interface to the module is not possible. Black box testing takes place during the validation phase so the test engineers are not aware of the break down into modules, let alone module interface definitions. Therefore the task of making sure that a module is error free must be tackled during module testing as it will not be possible during the later testing phases. Each testing phase attacks a different aspect of product verification and builds upon the preceding phase. If one phase of testing is not completed satisfactorily then it will only be by chance that a latter stage of testing finds an outstanding problem from an earlier stage.

If module testing faults are found during validation this is indicative of poor module testing. Validation testing is unlikely to uncover many module design or coding faults because of the black box approach to testing, so a return to module testing is required in this case. Hetzel [34] "the rush to get systems testing started makes us fall prey to accepting poor unit testing and to integrate with little planning or formal interface testing. Just as a single bad apple will quickly rot the entire barrel, it takes only several poorly tested modules to ruin a solid integration testing plan."

Any system that can indicate when to move on to the next testing phase or flag a potential problem in moving on too fast could not only help the project manager understand the extent of the testing that has been completed, but also in improving the quality of the end product. Sroka & Gosling [111] "reasoned and timely decision making is the cornerstone of effective project management. To make good decisions, the project manager must be able to understand why actual progress differs from expectations." Deciding on the optimum point to move to the next testing phase is a widespread problem, as stated at a USA direction setting workshop for research. "Humans are relied upon to use their intuition and judgement to create testcases, evaluate the results, and decide when sufficient testing has been done. Further, there are no widely accepted formal guidelines or standards for determining when adequate testing and analysis have been done" [13].

The standard life-cycle diagrams and product development management tools used today imply that testing moves through distinct testing phases in discrete steps. In reality the testing phases overlap in time. It would be wasteful of elapsed time to programme in testing as a serial activity. Integration testing will actually start as soon as the first two modules have been tested, so long as the module development plan produces the modules in the correct order for integration. Validation testing might well start with an early release from integration as soon as the initial functionality has been proven. This overlap of the testing phases is shown in Figure 28 Testing overlap, page 125.

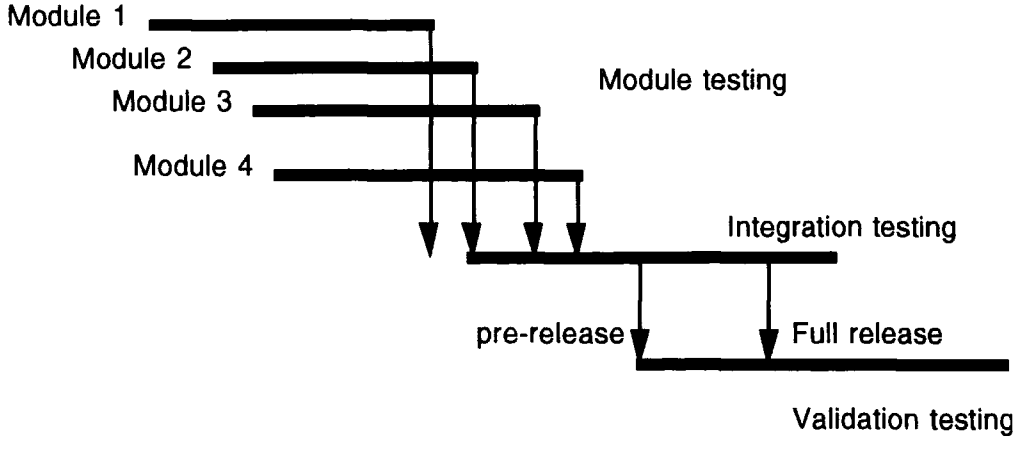


Figure 28 Testing overlap

Hetzel [34] suggests that plotting the number of faults found against time can be used to determine the testing time required to complete testing, " in larger projects useful status information can be obtained from cumulative plots of faults found versus testing effort or time. It can be observed empirically that the plots take on a characteristic exponential shape and that as the shape develops it becomes possible to use it to roughly gauge the testing time or effort remaining or the total number of errors remaining in the system".

An ideal method then, for deciding when and how to move between testing phases could be built into a model using fault category data. A graph (see Figure 29 Fault category graph, page 126) showing the fault categories against time for past projects, overlaid with data plotted from a project under development, would highlight the best time to make a controlled move to the next phase of testing for each module, integration build and release. But for these graphs to indicate the time at which a move to the next phase of testing is required, the fault category data must be relevant to the life-cycle phase which is being observed. For example the 'system design faults' graph must contain data on system design faults that have been found during integration testing, and not data on any faults, that happen to have been found during integration testing.

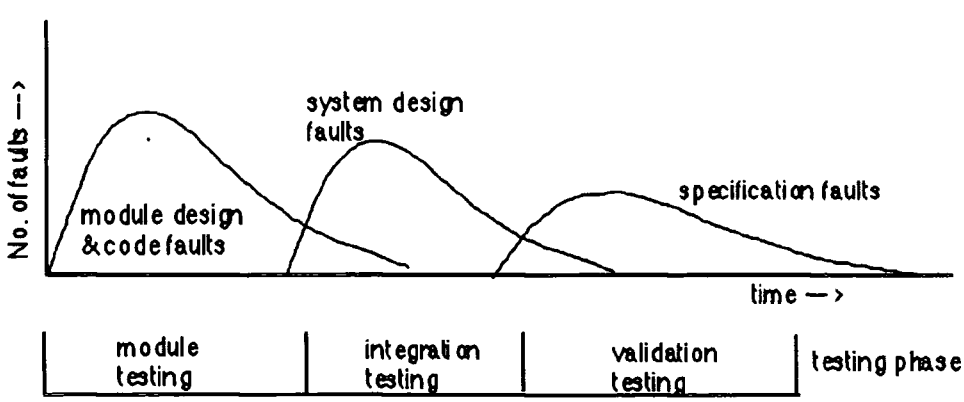


Figure 29 Fault category graph

The fault category data therefore needs to provide, for each fault analysed, the point in the testing life-cycle at which it should have been found. A new original metric that identifies fault category, in relation to life-cycle phase, is proposed and called Earliest Visibility, EV.

For each fault there should be a window of opportunity for discovery which will start with the EV and continue on through the rest of the testing phases. This is shown in Figure 30 Earliest Visibility, page 127 , where the first fault has an EV starting at validation and the second could be found from module testing onwards.

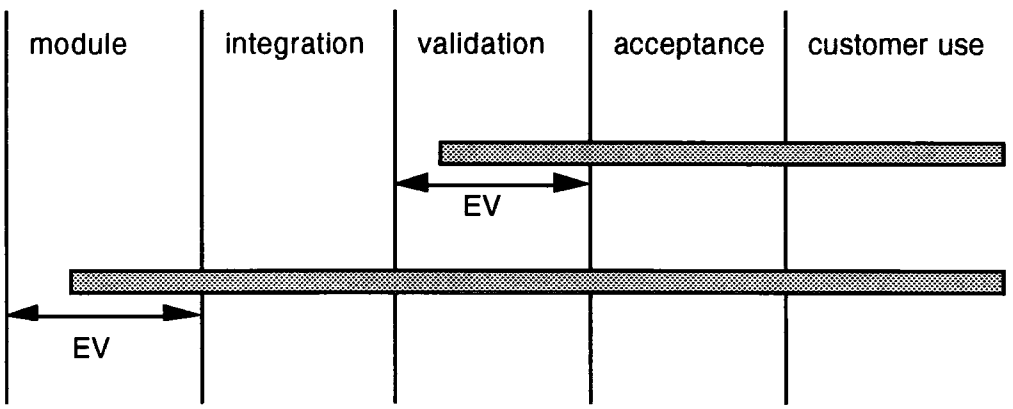


Figure 30 Earliest Visibility

The expectation is that faults should be found during their EV test phase, because that particular test phase will have a set of tests designed to find that type of fault. The objective of the experimentation for this thesis, is to show that there is a relationship between the phase that a fault is introduced and its EV. Having established the EV for each fault it will also be possible to look for the relative levels of faults associated with each of the testing phases.

For example faults introduced during the specification phase will be exposed only by the validation and customer testing. The EV will therefore be validation as this phase comes before customer testing in the overall life-cycle, and the faults should be found during validation, any later and this indicates poor validation testing.

$I_s \cap (F_{vt} \cup F_{ct})$ EV = vt

Key:

ct - customer testing

vt - validation testing

it - integration testing

mt - module testing

r - requirements

s - specification

sd - system design

md - module design

c - code

The expected relationships are:

$I_r \cap F_{ct}$ EV = ct

$I_s \cap (F_{vt} \cup F_{ct})$ EV = vt

$I_{sd} \cap (F_{it} \cup F_{vt} \cup F_{ct})$ EV = it

$I_{md} \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})$ EV = mt

$I_c \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})$ EV = mt

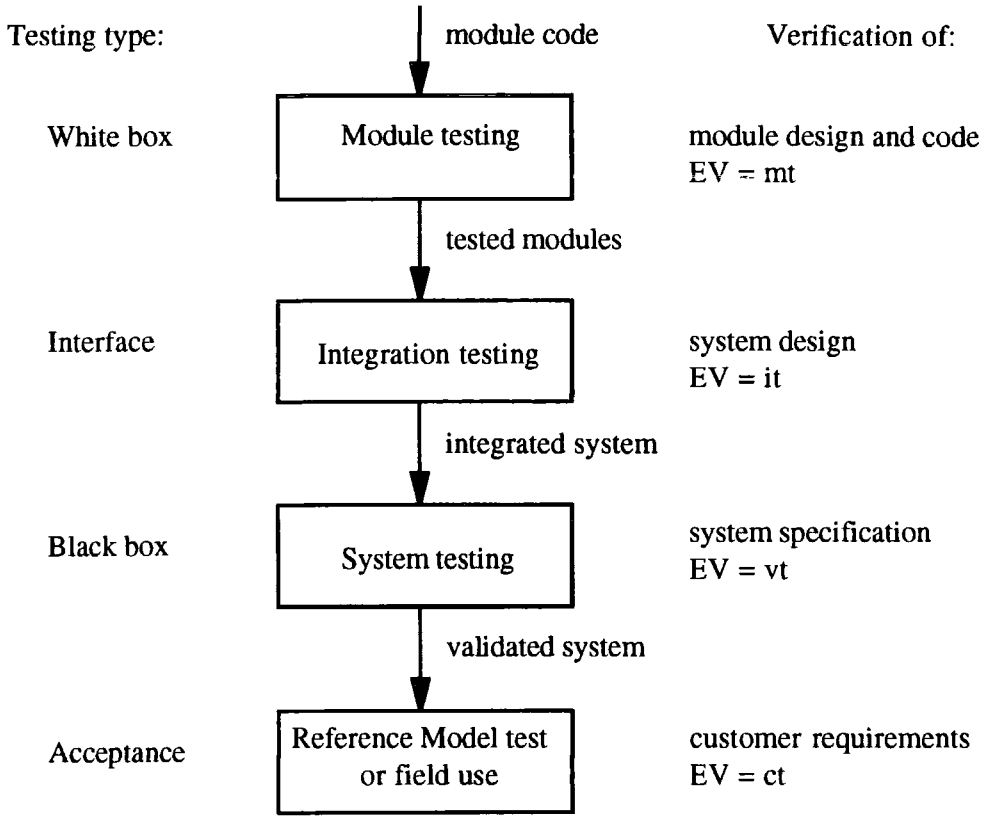


Figure 31 Life-cycle EV

Each fault therefore has EV as an attribute, it is the testing phase that should discover the fault based on the reverse linkage to the development phase that introduced fault. Having established the EV attribute for each fault reported during a testing phase and measured the number of faults that have been discovered post EV, the metric (post EV) can then be used to control the movement of the software through the test phases. The post EV metric is defined as the percentage of faults found during a test phase that should have been found earlier, these faults have been found post their predicted EV. If a high percentage of faults are being found post EV during a test phase, this is an indication that the previous test phase had not been effective and the software should therefore be returned to the previous phase for further testing.
Although EV is fundamental to the proposed model a number of other metrics will be explored during the case study to determine the best metrics for life-cycle control.

6.2 Estimation of testing resources

The estimation of testing resources and elapsed time for testing, suffers the same problems as development estimation at the start of the project. There is nothing tangible to work on; most of the widely used estimation models are based on a comparison with past projects within a particular organisation, development methodology and personnel. If any one of the variables change then the comparison and predictions from the information stored in the past project database is not truly valid. By keeping any changes to a minimum the closer the estimate will be to the actual resource and elapsed time. This will provide a model similar in concept to DeMarco's cost model [23], in that the model uses the best data available at the time to gradually increase the accuracy during the project.

The testing estimates can be drastically improved as the start of test execution approaches, because the size of the code to be tested will be known. This has been shown to have a very close link to actual development effort [16]. It is quite often used as an input to estimation models for development, even though the code size used is an estimate. Unfortunately the development engineer would have to deduce by comparison with other past projects the expected lines of code. Whether estimating the actual code size is any more precise than estimating the development effort is debatable.

Fenton & Pfleeger " we have noted that a collection of related models each based on measurable data available at a given stage of development, is likely to be more accurate than a single, generic model. Early estimates necessarily require use of incomplete information. Thus, estimation is likely to be improved by taking two related steps: basing preliminary estimates on measures of available products, and re-estimating as more information becomes available and more products are produced" [47].

IPL, Information Processing Ltd [112], found that "the division of effort between activities (design, review, code, test, rework) could be used to estimate the effort required for activities subsequent to module design". Although IPL found that this estimation technique can only be expected to be accurate on a large scale.

To provide an estimate for the total testing effort required (across all testing phases) a two pass system is therefore proposed. The initial estimate based on a straight forward percentage of the estimated development effort, or a separate analysis of the specification into function points or equivalent sizing measure. The second pass estimation would be based on a revised estimate produced when the actual code size is known. Re-estimation can then be applied following each testing phase to refine and confirm the estimate. A database of past projects will be essential if the mapping from the measurement metrics (LOC, function points, development effort) to the actual effort required, is to increase in accuracy with time.

Testing is made up of a number of activities which will have a set of relationships between them that will reflect the effort that each activity requires. These relationships once understood could provide a method for refining the total resource estimates as the testing work progresses. For example; the more effort that is spent on testability analysis of a specification might give an indication of the size of the overall project and therefore influence the estimate for the overall testing effort.

Rombach [114] found a set of effort estimates that he applied on NASA SEL projects:

Preliminary design	15%
Detailed design	17%
Code/test	26%
System test	23%
Acceptance test	19%

The database of testing data should therefore cover the effort spent on each of the activities so that the links between the activities can be found and useful metrics developed. Before deciding therefore on what measurements to take and why, an understanding of the testing process is needed, based on a case study, which identifies the individual activities and the measurements that can be taken.

6.3 Summary

If the hypothesis put forward in this chapter can be proved correct, there will be a number of opportunities for improving the effectiveness of testing using EV measurement:

1. a method for deciding the optimum point at which to move onto the next phase of testing, and therefore managing the cost/quality balance for the project.
2. an indication that the testing has moved on too quickly and that a return to a previous phase is required to find and remove remaining faults.
3. by controlling the movement of testing between phases for a number of projects a fault database with the relationship between the number of faults per stage can be collected. This data can be used to produce a prediction of fault level for later stages of testing.
4. providing a measurement of the testing effectiveness by test phase and fault introduction rate by development phase.

A model based on re-estimation is proposed for testing resource estimation.

7 Case study

There are two objectives for the testing estimation and process control model:

improvements to the testing process - section 7.2, 7.3

better estimation of testing resource - section 7.4

Improvements to the testing process results in improved efficiency, reduction in cost of the system tested and a higher quality product. By measuring where testing effort was spent on various projects an improved model for testing effort estimation can be built, based on re-estimation.

The two objectives are taken in turn and expanded to cover the metrics to be collected and the methods to be used during the case study, following some information on the case study development project.

7.1 Development project background

The case study follows the development and testing of a telecomms network and service management system, IMS - Integrated Management System. The system consists of an X-windows user interface, database, core processes, interfaces to collect real time network data, an interface to feed processed data onto higher order management systems and additional bolt on modules for reporting, archiving and data loading. The core software for release 3C6, contained 4,002,424 lines of code (C and C++) within 4,724 files, running in a UNIX environment.

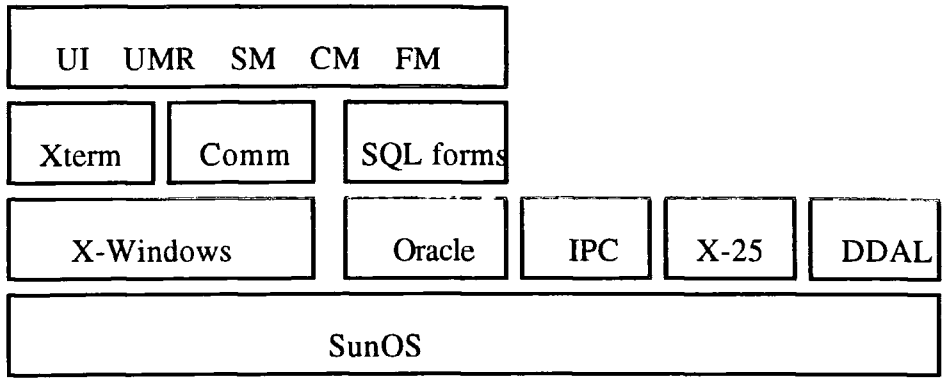


Figure 32 IMS internal structure

The IMS internal structure diagram on page 134, provides a view of the bought in platform components SunOS, Oracle database, SQL forms, X-windows, Xterm and X.25 communication.

The processes developed for the IMS include:

UI	User Interface
UMR	Unsolicited Message request
SM	System Manager
CM	Configuration Manager
FM	Fault Manager
Comm	CMIP interface
IPC	Inter Process Communication
DDAL	Direct database Access

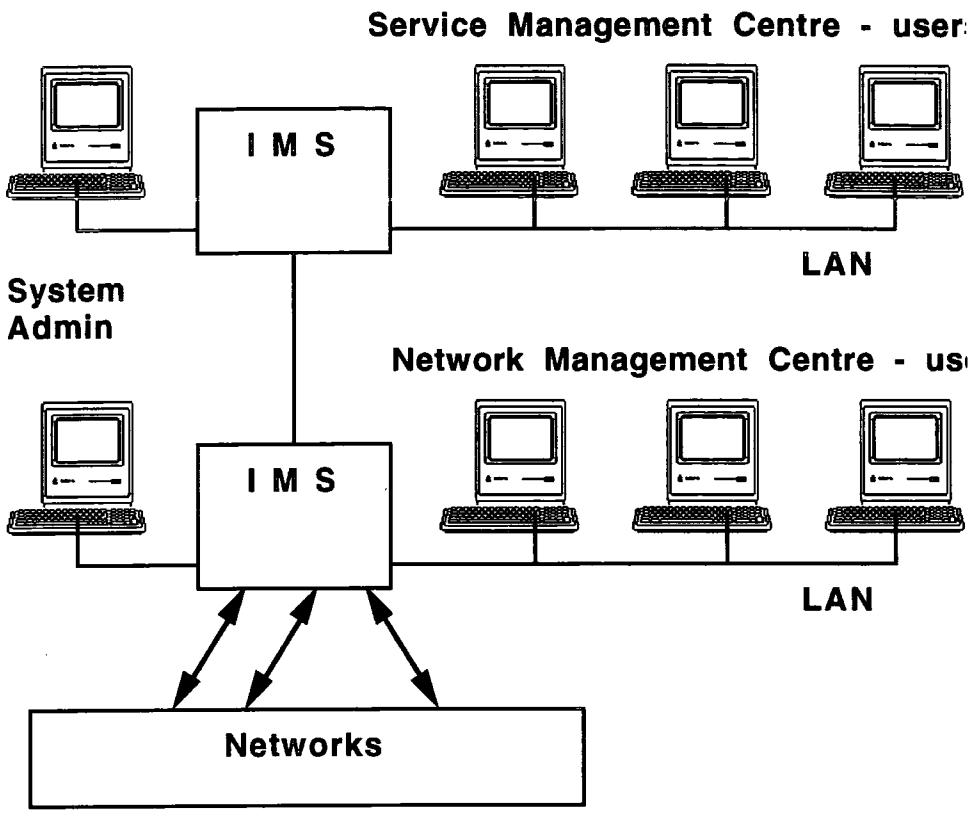


Figure 33 IMS deployment

The system can be configured for network management operations where a number of networks provide real time events into the system that are processed and then displayed graphically. With a backward channel through the network interface, the system can send back control messages to re-route services around network failures. Information on customer services that have been impacted by a network failure is passed to a second variant of the system that is used by the customer service management team so that they can inform a customer of service failures. Customer services are depicted as logical routes between geographical locations, following the mapping from the physical route which is managed by the network management system.

7.2 Experimentation - EV

The model of fault propagation described in chapter 5 follows similar concepts to the Remus & Zilles model [107] of defect removal. Both models support a number of defect removal steps, and also the likelihood that some new faults

will be added and some old faults will not be fixed correctly when trying to fix the original set of faults. The models do differ over the level of detail for the fault removal steps. The Remus & Zilles model provides a generic fault removal step which can be applied at any inspection, review or testing stage, and provides a global fault level for the complete development process when all the individual steps are added together. In addition to the number of faults at each step the fault propagation model proposed in this thesis, chapter 5, also considers the type of fault, which leads to a better understanding of the fault removal process at each individual stage of the development.

A common metric used to measure testing progress is the percentage of test cases completed [3] [113].This provides a measure of the progress in running test cases, but this is not a measure of the quality of the software. Just because all the test cases have been run it does not mean that testing has been completed satisfactorily. Other measures are required to measure the quality of the software so that more test cases can be produced if the quality level required has not been reached. Also there is an opportunity to stop testing even if all the test cases have not been used, if the measures indicate satisfactory quality. Unfortunately the measures in use today are not precise, and can not guarantee a quality level, but the objective is to provide a better method than the number of test cases completed as a measure of quality.

The possibility of using Earliest Visibility (EV) analysis to determine when to move into the next phase of testing, when the required quality level has been reached or alternatively to use it as a check to decide if the project management view of the project phase aligns with the reality from the testing results, has been identified in chapter 6. Experimentation is required to show that EV can be used to determine the optimum point at which to move through the test phases accurately and to evaluate the validity of this method for industrial scale projects. The data required for analysis is obtained from project fault reports. The terminology may vary between projects, so that the fault reports might also be called problem reports, incident reports or trouble log. But it is the first documentation of problems found during testing that is of interest, no matter what it is called.

The experiment will concentrate on the analysis of all the problem reports and will be focused on answering these three questions for each of the problems reported:

- at what stage in the development process was the problem introduced?
- at what stage in the testing process was the problem found?
- with improved testing would it have been possible to find the problem earlier?

Along with each individual fault report, the testing phase in which the fault was discovered and the development phase in which it was introduced are also required. An engineer with a good understanding of the particular problem then has to decide the earliest point in the overall testing that the problem could have been identified.

The relationships to be considered are:

- customer requirements errors have an EV during reference model testing
- specification errors have an EV during validation testing
- system design errors have an EV during integration testing
- module design errors have an EV during module testing
- coding errors have an EV during module testing

A second use for the EV metric, is to provide a measure of 'Testing Effectiveness' by test phase. This can only be applied at the completion of all the test phases for a system build and a period of field use if the build is released. The measure compares the number of faults found during a specific phase that have associated EV, with the total number of faults with that EV. The number of faults found when they should have been against those that have been found later then they should have been. This therefore provides a measure of how effective a particular test phase has been at removing the faults that should be unidentified during the test phase, those with an EV equal to the phase. Comparing these values across the testing phases will identify the test phases that require investigation as they are not delivering the testing needed for

a quality product. The Testing Effectiveness equations for each test phase are listed below:

$$\text{module testing} = \left| \frac{(I_{md} \cup I_c) \cap F_{mt}}{(I_{md} \cup I_c) \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})} \right| \times 100$$

$$\text{integration testing} = \left| \frac{I_{sd} \cap (F_{it} \cup F_{mt})}{I_{sd} \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})} \right| \times 100$$

$$\text{validation testing} = \left| \frac{I_s \cap (F_{vt} \cup F_{it} \cup F_{mt})}{I_s \cap (F_{mt} \cup F_{it} \cup F_{vt} \cup F_{ct})} \right| \times 100$$

The equations provide a comparative measure of testing effectiveness, not an absolute measure, due to the limitations on the scope of testing. The limitations, the three variables A, B, C that reduce the maximum test coverage are explained in chapter 5 with the use of Venn diagram 2 Fault & coverage set, page 113.

7.3 Testing process control improvements

In addition to EV a number of other metrics are included during the case study to determine the best combination of metrics that can be built into a testing process control model.

Bellcore [113] use a combination of metrics as criteria for completing the system test phase. Metrics employed include; number of system tests executed, and passed, number of open TT's, TT rate, number of open critical TT's and stability (performance and reliability measure).

7.3.1 Clear ratio

If the number of faults corrected is plotted on the same graph as faults found, this is often used to predict the testing effort required to complete testing and the status of the testing. Hetzel [34] says, "it will show if debugging is becoming a

serious obstacle to testing progress and provide a further indication of true status".

ITT programming, Dunn [89] (see Figure 34 Worrisome trend of TT's. Page 139) apply trend analysis to open and closed trouble tickets, to provide 'leading indicators' to warn of impeding problems.

If the error rate is increasing, or if the fixes are introducing more errors than are being cleared, the project will fail unless action is taken to improve the quality of the work, including re-coding the troublesome modules. A ratio will be used in the proposed testing process model, faults found/faults cleared.

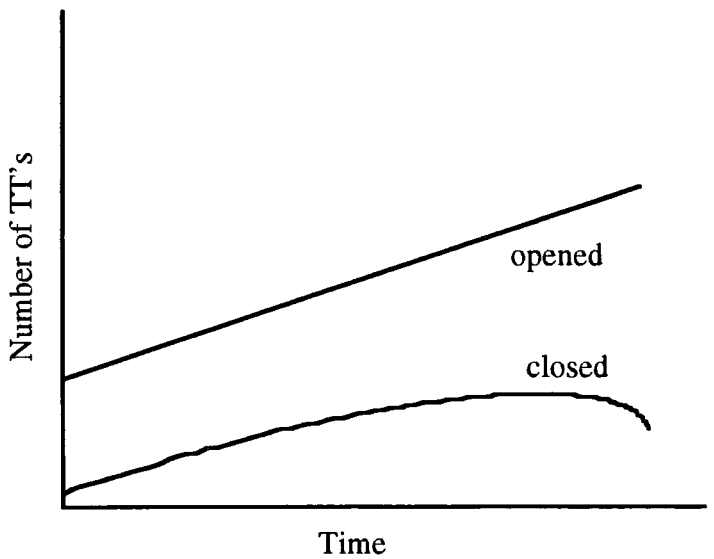


Figure 34 Worrisome trend of TT's.

7.3.2 Churn

Two types of churn are covered within this thesis, requirements and bug fix churn. Churn can be used as a measure of the change in perceived requirements. If the requirements change, then the further through the development activities the larger the impact on the re-work that will be needed. Increased functionality, enhancements or scope creep are also considered to be requirements changes. The requirements might change because the customer changes his mind or because the developers realise that the requirements have been interpreted incorrectly. Tice [3] calls this requirements definition stability, "how well do the users understand what they want, how well does the

developer understand what the user wants". Tice suggests this stability can be measured by comparing the number of units (modules) affected by change compared to the total system size in units. Dunn includes a stability measure as part of a defect estimation algorithm [89] and admits that it is not quantifiable by any theoretical technique, but a simple count of change requests applied to requirements or top level design, can be used as a measure.

The second type of churn is caused by bug fix activities. Ince [6] talks about how software will 'rust', with more and more changes carried out on the system its structure degrades. "The reason is twofold: first, staff unfamiliar with a software system are usually assigned to its maintenance; second, relatively low grade staff are also employed in this activity. What happens then is that a software system, when released, starts off with a clean architecture. During maintenance, errors are discovered and new requirements arise, and random hacks are made to the system. This results in interfaces between modules becoming more complex and the logic inside modules becoming more tortuous. What happens is a progressive rusting of the software system which adversely affects further maintenance effort: as the system rusts, further changes become more difficult and consumes more resource".

In both cases, requirements and bug fix, the code and possibly the design will be changed by engineers that did not create the code initially or to a greater extent than initially envisaged. There are two impacts of churn:

1. Over time and a number of builds, changes will have been made upon changes and it will become increasingly difficult to make further changes without introducing faults.
2. The greater the number of changes taking place in parallel for the same build, the greater the risk of faults being introduced because the design and code become unstable due to the large amount of change.

The measure that will be used within the proposed testing process control model to asses churn will be the actual number of changes made (enhancements and bug fix) compared to the initial scope for the build.

7.3.3 Error rate halving.

A study carried out by Rombach [114] at NASA's software engineering laboratory, found that the error rate halved with each subsequent testing phase. This phenomenon can be seen in Figure 35 Error rate halving, page 141.

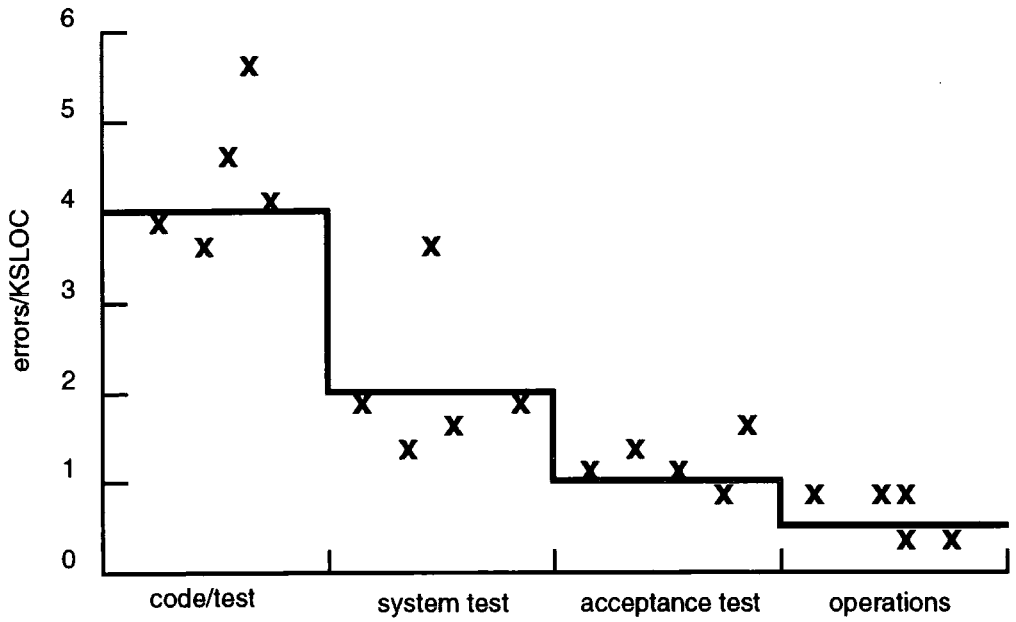


Figure 35 Error rate halving

In testing communications software Siemens [115] have found that the percentage of faults found in each testing phase is similar for most development projects and can therefore be used to predict the number of faults that should be found during each testing phase. The results from Siemens show that the percentage of faults found at each phase:

60 - 70% during development

15 - 20% during system test

2 - 5% during field trial

3 - 10% during operational use

This reduction in error rate following each phase of testing is expected, although it has been shown that problems can be introduced at each phase of testing. So a poor specification, but with good coding, will result in a lot of failures at customer acceptance, but not many errors during module and integration

testing. This measurement can still be used as a guide and any anomalies can be picked up early. The reduction in error rate should also be measured during a life-cycle test phase for each subsequent build. Some modules may require several compile and test phases before they are ready for integration with other modules. During integration testing several build and test phases may be needed before system validation can start, and during validation several builds may be needed before the system is ready for operational use. Again it is expected that the error rate will fall following each subsequent build and test cycle within the testing life-cycle phase.

7.3.4 Severity shift

Another measure that can be used during builds within a test life-cycle phase is severity shift. There are likely to be critical problems found during each phase of the testing life-cycle, but within a test phase the builds should move from identifying critical and highs to mediums and lows, as the important problems are fixed. A shift is therefore expected from high criticality faults to low faults over a number of builds.
This will be less obvious if fixes introduce further critical problems or gateway problems are found. The gateway problems hide an area of functionality or code, that can not be accessed until the gateway problem has been fixed. When it has been fixed another set of problems are found in the code that can now be accessed, and it may contain further gateway problems and critical faults. The effect can be similar to peeling layers from an onion, but with an unknown depth and therefore time to fix all the problems hidden.

7.3.5 Faults per week

The rate at which faults are being found, assuming a fixed size test team, provides an indication on how easy it is to find faults and therefore when the testing should be completed, or how effective the test team are at finding faults. A common feature during testing and maintenance is for the fault rate to increase for a while following a build/release until the most frequently

encountered faults are recorded and then for the rate to fall and stabilise at a low level. Within HP [46] this is called "the hump phenomenon".

7.4 A process model for data collection

The testing process model developed and used during the analysis of testing resource usage is illustrated in Figure 36 Testing process model, page 144. The large circles represent the major testing phases, with the smaller circles showing the activities associated with each phase. The squares are the output's expected from each phase. Each phase is described below with the activities, the rows in the data collection form, highlighted in bold text.

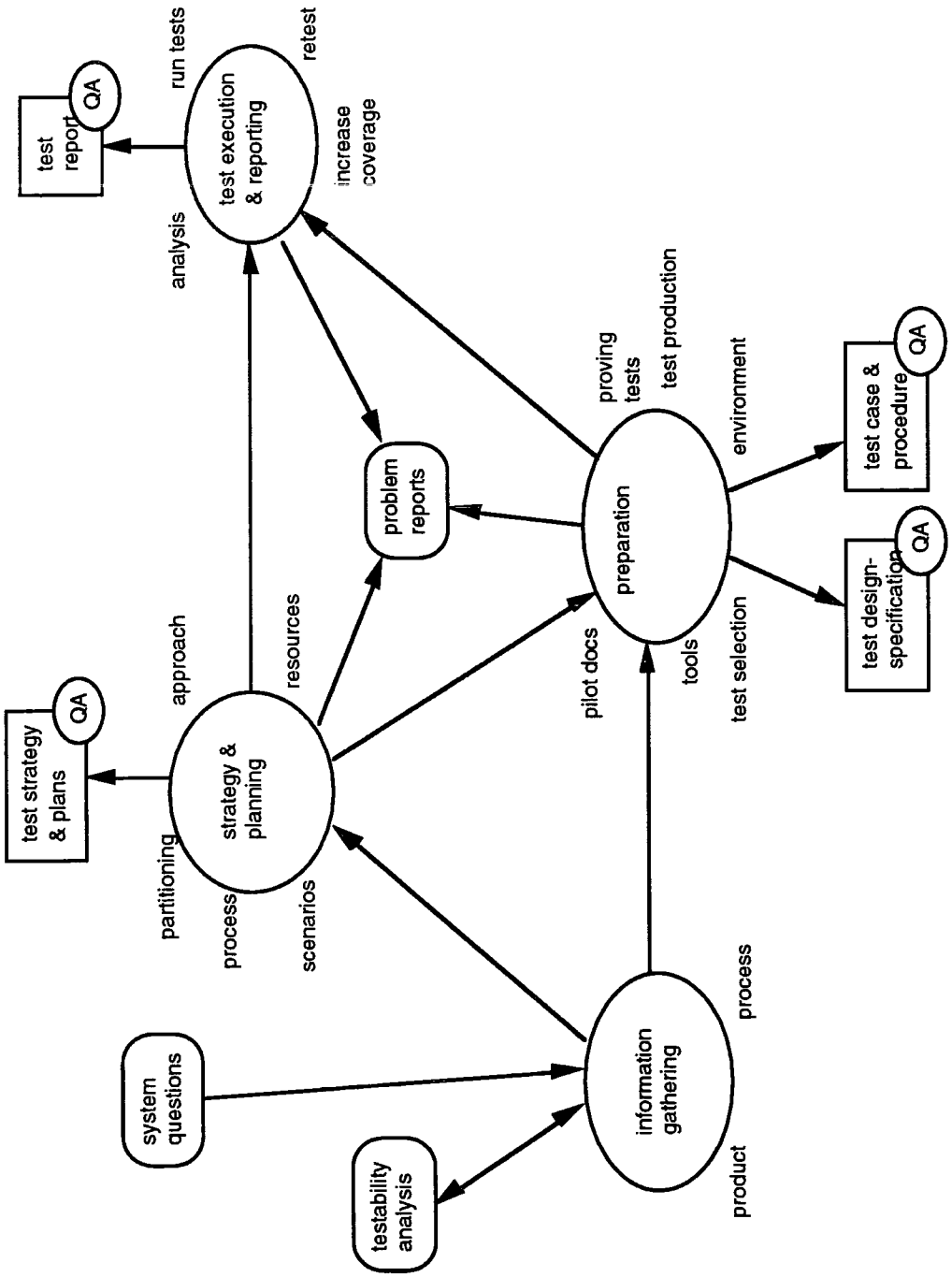


Figure 36 Testing process model

7.4.1 Information gathering

A data collection form has been produced (Figure 37 Metrics collection form - A and B, page 147,148) which maps onto the major testing phases and the associated activities identified on the process diagram. Each of these major

testing phases is now taken in turn and expanded with a description of the activities that will be measured.

It is interesting to note that the first major phase was seen as information gathering and not just about the system under test, but also the process being used for development and interaction with the test team. A fair amount of effort goes into understanding the product and development process which is often overlooked when estimating testing effort. Information gathering is likely to be at its highest at project start up, but will also take place during the strategy and planning activity, and the test preparation activities.

7.4.2 Testability analysis

At this early stage in the identification of metrics it will be worth collecting the number of problems raised during the testability analysis as well as the hours spent on this activity. System questions should also be noted as this might also lead to a good indication of the quality of the specification and the resulting development. Testability analysis may not be significant in the overall testing activity but it is worth measuring and removing if not significant.

7.4.3 Strategy and planning

After the normal project start up process of information gathering, the first major activity is a strategy document to outline the proposed testing strategy and a set of top level plans providing the base resource data for the testing project. Time spent deciding on the process that will be used during the testing work is recorded under process. This includes start and finish criteria for the testing, re-testing criteria, support arrangements and the quality assurance procedures that will be used on the testing project. Approach covers the technical aspects of the way in which the testing will take place and will lead onto the testing environment and test tools necessary to support the chosen approach.

A first cut is needed in the strategy document to show how the testing of a system is going to be grouped. This work is referred to as partitioning. Test design (TD) documents follow on in the preparation phase for each area identified in the strategy. Deciding on what configurations to test, the data that should be used and the likely usage patterns that might be applied by a customer, develops into a set of scenarios. A lot of information gathering is associated with this activity as the base data has to come from marketing or potential customers.

Project plans detailing the resources (manpower and non-manpower) are produced and calendarised over the project life. The dates for the completion of the milestones are also planned at this stage. The strategy and planning activities have a test strategy document as a deliverable. So the time spent in producing the documentation and reviewing the deliverable are two activities that can be recorded on the data collection sheet.

7.4.4 Preparation - test team

The preparation activities are split between test preparation and test environment preparation, as the work is different and separate teams are likely to be used so that the activities proceed in parallel.

The TD's follow on from the direction set in the strategy, but for each area they go down to the level of detail where the individual test cases (TC) are identified and named or numbered. The TD's are reviewed by test engineers from the same project but different TD work area, and a representative from development. A documentation pilot trial is the concentration of effort into one TD work area, working all the way down through the documentation to completed TC's.

Metrics data collection

Name:

Project

Date

Information gathering

	activity	area of interest: product *hours:*	area of interest: process *hours:*
	start up		
	strategy & planning		
	test preparation		

background, features, user environment - use

management,change control, project interface

Testability analysis

hours:

review of functional spec for testability

Strategy & planning

hours:

	process	*I/P, O/P & retest crteria, support, QA*
	approach	*techniques, test tools & environmen*
	partitioning	*split between TD's*
	scenarios	*customer use, patterns & data*
	resources	*manpower, nonmanpower, dates*
	documentation	*time spent writing strategy*
	review	*QA review of strategy*

Number of system questions raised

Number of problem reports raised

1 - 10 , 10 high

Job satisfaction

Figure 37 Metrics collection form - A

Preparation - test team

		hours:
	test design production	
	test design review	
	test documentation - plot trial	
	test selection	
	test case production	
	test case review	
	test procedure spec - test days	
	test procedure production	
	test procedure review	
	proving tests	
	audit of integration and system testing	
	acceptance of deliverables into validation	

Preparation - environment team

		hours:
	test tool specification	
	test tool development	
	test tool proving	
	test environment (inc model) specification	
	test environment development	
	test environment proving	
	test environment support & maintenance	

Test execution & reporting

		hours:
	run tests	
	results analysis	
	retest	
	increased coverage - new tests	
	test report production	
	test report review	

Figure 38 Metrics collection form - B

The output is reviewed to see if the resource expended is reasonable and that the level of detail achieved is correct for the particular project and staff involved.

This is a method for establishing early on in the project if the correct level of preparation and documentation has been specified. See Appendix B for further information on this technique.

Test selection can take place several times during a testing project. The first time is on the completion of the TD's when a complete list of all the TC's to be produced is available. A review at this point might decide to increase coverage and therefore the number of TC's or reduce the number of TC's if there is an overlap in the coverage or just not the budget to go ahead with all of them. As the project progresses and more information becomes available on the areas that might need enhanced testing the coverage and test cases selected will be reviewed again.

Test case production is the pulling together of all the information needed for a test case and then producing it. The TC's have been identified in the TD's, they have been through a selection process and the information needed is available from the system specification, design or user guide. A TC review is not always needed, but if it takes place it tends to be a peer review, looking at the details of a TC for accuracy and usefulness as a test.

The test procedure (TP) is the working arrangement for running a series of TC's whilst utilising the same testing environment configuration. The instructions provide the order of the tests and a description of the test environment set-up. Test days are another way of providing a TP except that the tests will require a days worth of test time to run. They will all be in a related area with a specific run order. Test procedures may be used to supplement test days if a further breakdown of detail is required. The data collection form shows a column for the effort involved in deciding and specifying what the test days or procedures will consist of, and a row allocated for the man hours spent in producing them. This may be followed by a peer review to obtain agreement on the decisions or details.

'Proving tests' is an activity to try out the prepared tests using the test environment and perhaps an early release of the system under test to determine if the tests are correct and will function as specified. The idea is to pre-empt any hold up in the testing programme due to tests failing because of a misunderstanding by a test engineer in producing the tests or the test

environment. Test engineers can make mistakes the same as development engineers.

An audit of module, integration or system testing is used to understand the level of testing that has taken place during other testing phases and therefore help to determine the level of testing required during validation. It can also be used to check that the input criteria to validation has been met.

Acceptance of deliverables into validation can just be a decision taken on the basis of the above audit, or it could be a set of tests that will be applied before the system is acceptable for validation.

7.4.5 Preparation - environment team

The test environment preparation has two main components to the work. The first area shown in the data collection form is the development of specific test tools. This is split into the specification, development and proving activities. The test tools will need testing, but to a lesser degree than the real system under test otherwise there will be a continual development of test tools to test each other to product quality.

The second area, the specification, development and proving of the test environment includes the hardware/software model that will be used to test the target system, proprietary testers, communications and computing equipment. In fact anything that will be needed to run or support the test engineers apart from the tests themselves.

The last row in this table is for the effort spent in maintaining and supporting the test environment and tools once they have been provided for use.

7.4.6 Test execution and reporting

The data collection form is completed with a table that covers the test execution and reporting. The number of man-hours spent in the execution of the tests is covered in the first row. The analysis of the test results data is catered for in the second row. Re-test is the running and analysis of tests that are run for a

second time due to a failure of the system under test or the test (or its environment) the first time around.

As testing progress, certain areas of the system under test will be identified as having a higher proportion of faults than others. The more faults that are found in one area the higher the probability that there are further problems to be found. Extra tests will be needed to increase the testing of these 'hot spots'. The effort put into the development of additional tests during the testing phase is contained in the row called new tests.

At some point in time a test report will be asked for. This could be on a fixed date when the report would give the testing to date and outstanding problems, or the test report might drift until the system under test is approved and the test report documents the history of the test period. The time spent producing the report and reviewing it goes into the last two rows respectively.

Other items to be recorded:

the number of system questions
the size of the test strategy - pages
the number of test designs & size - pages
the number of test cases
the number of test procedures
the number of problem reports raised
the number of new tests developed during actual testing
the size of the test report - pages

7.5 Data collection

Having agreed the objectives for this case study, and decided on a testing process plus the data to be collected, the next activity concerned the actual data collection. For the estimation of testing resources it was decided to collect the number of hours spent on each activity on a weekly basis. This did not take the form of a time sheet, which might have resulted in some erroneous data due to the testing engineers spreading their 42 hours per week over the activities rather than recording what actually happened on the project. So only the hours spent

on the project testing activities are documented, the rest of their week will have been spent on non project activities. To be able to use the results to calculate back to the number of engineers required for an activity, a test team utilisation figure is needed. This is calculated on a weekly basis and provides the proportion of time that a test team would be available for project work; the equation for utilisation is shown below.

$$\% \text{ utilisation} = \frac{\text{total hours booked to project that week}}{42 \text{ hours} * \text{number of staff on project}} * 100$$

The raw fault data that has been analysed for this thesis came from a number of sources. To validate the data and ensure that it can be used, data integrity checks have been made and any inconsistencies have been resolved. Because this data has been provided from different sources, using different data capture methods and specifications for data items, there has been a large amount of data mapping required.

The fault data has been provided from:

1. Change Request, CR, list covering faults and changes required. The 3,750 entries had been identified during an 18 month period across integration, validation, reference model and field testing activities. CR's can be raised against the core system, other systems that interconnect, data, documentation and the installation process. Although there was no differentiation between changes to functionality and faults initially, they were held separately on the database towards the end of the observation period.
2. Change review weekly notes. These provide information on the action taken to investigate and resolve CR's and other items raised at the weekly CR forum. 2,700 CR's covered.
3. Field Incident Reports, FIR. 5,760 FIR's raised on a helpdesk by users of a number of supported systems. The incident reports do cover problems that are fixed by field engineers without the need to raise a CR against the system and also provide the background to a CR that has been raised due to a field problem.

4. Weekly validation testing reports. This report provides details on the number of problems raised, cleared or awaiting test with the validation test team.
5. Release review documentation. For each version that is targeted at release for use in the field, a review takes place to confirm that the version is fit for purpose and that it will be supported by the maintenance team. The review includes information on new functionality, problems that have been cleared, outstanding problems, work-arounds and performance data.
6. Twice weekly development team status report. This provides the status on each outstanding CR, the analysis of the problem, fix proposal and target release. Each report tracks the progress of about 500 CR's.
7. Scope statements. Each release starts life with a scope statement that provides a list of CR's to be fixed and new functionality that is planned for a release.
8. Validation and reference model test reports providing detailed information on the test environment, build, test cases and the problems found.

The information from all these sources has been pulled together, cross-referenced and reconciled to provide the data required for the analysis of faults through the life-cycle. 1061 faults have been analysed in detail, covering version introduced, date, severity rating, life-cycle phase found, EV, fixed release, and type of fault - core software, data, system, installation. The only area where sufficient data has not been available, is module testing. Due to the geographical and departmental split of the development teams there is no structured way to record module testing failures. Also some teams may not be recording them if they can fix the problem on the fly.

The problem can be visualised by looking at Figure 24 Code and test, page 109 and realising that the feedback paths between the different life-cycle stages are following different operational processes and using alternative systems to record CR's. Also, that this situation applies within the module test box due to the large number of development teams working on the project, 200+ engineers within 12 development teams split across five separate development centres in the UK.

A large proportion of the time spent on this thesis has been dedicated to validating the data and reconciliation of differences, because the data has come from a diverse set of teams across the product life-cycle. This has been an iterative process, because quite often the data analysis has shown inconsistencies in the data and investigation has been needed to rectify the problem and improve the data integrity, before restarting the analysis.

A typical problem encountered is one where the faults in different categories did not equal the total number of faults targeted for a build. The total faults targeted for a build (the build scope) should equal; the faults fixed and included in the build (the release statement), the faults that will be fixed in a later build (scope statement), and the faults not fixed (outstanding faults list). But as the data is held in different documents/spreadsheets created at different times to the original scope statement, they did not always equate. Typical reasons for faults disappearing include finding duplicate faults and faults removed as it is later decided that they do not cause a problem. Faults appear in the build due to a problem being found after the scope statement had been produced, fixed and included in the build.

When a check on the data exposes an inconsistency the history of each fault associated with a build has to be tracked across the data sources until the problem has been resolved.

7.6 Summary

The case study has started with two objectives, this has led on to the identification of the data to be collected and the data collection methods. The background to the case study project the IMS, its internal structure and code size are explained. A testing process model has been proposed so that the testing estimation data can be linked back to the overall testing process. The next activity is the collection of the data from suitable software builds and the analysis to expose any problems with the metrics selected.

8 Results - Analysis and Discussion

Chapter seven introduced the case study for this thesis, the IMS project, and the metrics identified for data collection. Chapter eight considers the results from the data collection and analysis, taking each of the metrics in turn to determine if they should be a component part of the software testing estimation and process control model.

8.1 Data Collection

Data has been collected over an eighteen month period following the progress of the software through module, integration system and deployment testing plus the feedback from operational use. The system had been in use for about two years with one major re-work following release, before the monitoring started on the next release. The development consisted of new functionality, fixing point problems in the earlier version and re-writing modules that where beyond point fixes.

Testing starts with the testing of the modules that make up the core processes. This included the user interface driver and database management. Integration tests the interfaces between all of these components by building them into a system that will run. System testing exercises the end-to-end operational workstrings which are supported by the functionality provided by the system. This includes the outer interfaces of the system and the external systems that interwork with it over the interfaces. The deployment or reference model test exercises the system in a configuration that replicates its intended use.

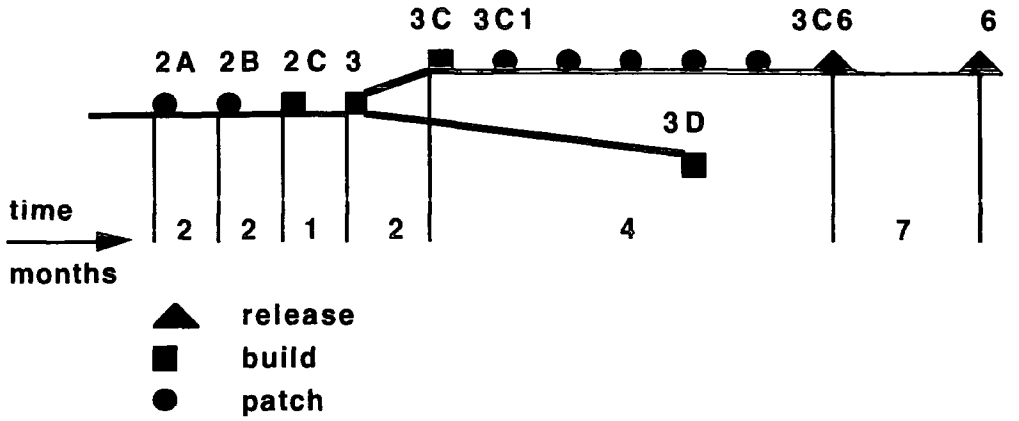


Figure 39 Build & release schedule

Data captured from early builds 2A, 2B, 2C and 3 is provided in table 1 showing severity shift and in table 6, faults per week plus clear ratio.
The full data capture started during the testing for release 3C. Poor integration testing led to a number of releases which where not good enough to go beyond system test. Table 6 below, demonstrates how during this period (2A) faults had been introduced into the code faster than they had been corrected. The engineers trying to fix the faults were creating more then they where fixing. At this stage the development is out of control and will never be completed successfully unless a different approach is taken.
A decision to de-scope the build's to provide a simpler set of functionality did succeeded in providing build's (2B,3) where the faults introduced rose slower that the faults being fixed. The functionality then had to be increased via a set of smaller incremental builds 3A, 3B, 3C, 3D. 3C and 3D occurred in parallel, building on 3B, but with the 3C work being fed directly into 3D to reduce the time scale down from having to do all the work in series.

The scope of each of these builds did increase as faults where found during the testing phases. Build 3D was a failure due to the quadruple increase in scope (see table 5) and although not many faults had been reported against it (see table 1) very little testing was completed as the build process proved unstable. Development continued to build upon 3C, providing a number of bug fix patch releases until 3C6, when following a full test it was released for operational use. One of the minor patch releases (3C2) included a new reporting module. The test data for this patch has been pulled out as a demonstration of poor module testing (see table 4).

Field problems from 3C6 and some re-work of the code instigated a new build (number 6) some 18 months later by a team put in place to maintain this system. This team (varying between 10 -20 engineers) produced maintenance builds every 4 - 6 months and are still supporting the system today (November 98). Although the system is in use today for the network management operations it has been replaced by a new system for service management. The IMS software is undergoing maintenance work during 1998 to ensure that it will be millennium compliant and can continue to be used beyond 1999.

8.2 Fault Analysis

For each of the software builds 3C, C1 to C6, 3D and 6 the detail results tables (Appendix D) show for each phase of testing the number of core faults found and the EV for each fault. The tables are converted to show the percentage figures. The tables also contain the number of system failures (beyond core software failures), problems caused by faulty data and installation problems.

Key:

Fault severity

C Critical

H High

M Medium

L Low

Test Phase

F **F**ield

M Reference **M**odel - deployment test

R **R**elease test, system test

P **P**oint fix test

I/T **I**ntegration **T**est and build

V Development **V**erification - module test

W Code/design **W**alk-through

Problem type

C Core software

D Data

S System, including gateways

I Installation script

+E Enhancement, improvement

R Specification change, new Requirement

Development phase

md&c module design and code

sd system design

s specification

r requirements

The test phases listed above map onto the fault propagation model (chapter 5) test phases as:

Customer testing and use (ct) = F field use, M reference model/acceptance test

Validation test (vt) = P Point fix test, R release test/system test

Integration test (it) = I/T integration test and build

Module test (mt) = V module test, W code walk through

Some of the test phases identified are combined, carried out at the same time and it has not been possible to identify the data split between the two test phases. In this cases a combined result is shown (e.g. P/R).

Summary Tables

Table 1. Severity percentage for each build, excluding field faults

	2A	2B	2C	3	3D	3C all	6
C	6	17	22	32	14	15	8
H	10	33	37	36	32	45	63
M	37	40	33	20	32	30	29
L	47	10	8	12	21	10	0
Total faults	157	126	144	164	28	423	24
field faults	-	-	-	-	-	145	22

Table 2. Detail results for build 3C to 3C6

	3C	3C1	3C2	3C3,4,5	3C6
C	22	17	1	16	13
H	40	41	47	44	57
M	31	39	20	34	25
L	7	3	31	6	8
Total faults	147	70	70	64	72

Table 3. Percentage post EV

	3C	3C1	3C2	3C3,4,5	3C6	3D	3C all	6
I/T	8	-	0	47	40	42	29	17
P/R	17	33	57	38	35	19	35	11
M	64	72	-	80	29	-	65	0
F	-	-	-	-	-	-	55	50

Table 4. Discovery, percentage by test phase

	3C	3C1	3C2	3C3,4,5	3C6	3D	3C all	6
V/I/T	13	-	3	23	7	43	10	50
P/R	56	64	97	61	83	57	70	38
M	30	36	-	16	10	-	20	12

Table 5. Increase in target scope

	3C all	3D	6
target fix	35	47	24
actual fix	115	191	55
% increase	329%	406%	229%

Table 6. Faults per week/clear ratio

	2A	2B	3	3C all	6
average faults per week	35.5	17.7	21.2	-	-
clear ratio	[1.97]	[1.20]	[1.22]	3.68	0.44

Table 7. Testing Effectiveness

	3C	3C2	3C3,4,5	3C6	3C all	6
V	39	-	-	-	6	-
I/T	69	0	55	38	36	80
P/R	69	100	88	93	63	61
M	-	-	-	-	77	100

Table 8. Fault introduction

	3C	3C1	3C2	3C345	3C6	3C all	3D	6
md&c	12	16	50	34	25	25	25	8
sd	9	6	6	17	11	9	11	21
s	61	66	31	39	40	50	57	58
r	11	10	13	9	24	13	7	13

Table 9. One phase PRE EV%, (actual number of faults).

	3C	3C1	3C2	3C345	3C6	3D	6
V	13	-	-	-	-	-	-
I/T	25 (3)	-	100 (2)	13 (2)	0	33 (4)	50 (6)
P/R	12	4	13	10	20	6	-
M	23	8	-	0	0	-	-

Table 10. Two phases PRE EV%, (actual number of faults).

	3C	3C1	3C2	3C345	3C6	3D	6
V	0	-	-	-	-	-	-
I/T	0	-	0	0	0	8 (1)	0
P/R	1 (1)	0	0	0	0	0	0

F Field problems.

These are user specific problems within user environment, which result in a change to the requirements. The problems will be related to how the users actually use the system, operational process, user specific data, plus the environment/configuration (performance and interfaces).

M Reference Model, deployment or customer acceptance test.

This validation testing will expose typical user environment problems. This includes user data loaded for testing, operational workstrings, specific applications and interfaces developed to other systems. The main difference compared to field problems will be that the reference model may not have the full hardware configuration to replicate the user configuration, so that the non functionals like performance will not be exercised as in the live user system. Also work practices that are not documented as part of the workstring but are actually followed by users, will not be exercised.

R Release system test.

This will exercise a generic configuration, data and workstrings (end to end) across the core system, with standard application code, and systems that interface to the core. The system test should find the problems where the delivered system does not meet the system specification, by exercising the system in a typical user fashion. This will be the first time that all components of the system are exercised together. System test will not find user specific configuration, data or process problems.

P Point fix test.

Individual fixes are tested within the system test environment. This will ensure that problems have been fixed and operate within the complete system. This will not prove that the system meets the requirements, just that a problem has been fixed.

I/T Integration test.

This will find the problems with the interfaces between components within the core software. The technical specification of the interface and data supported will be used to prove that the interfaces work and that the data is passed correctly. The complete system will gradually be built as the components are added and tested. Proving that the system does something useful is part of the system test not integration test.

V Component test.

The internal code of a module is tested up to the interface. Selected data is used plus boundary values and off scale values to ensure that for a given input the correct processed output is provided or error message.

W Code Walk through.

Similar to component test, to check code logic, data names and object definition, during the early stages of module design and code.

8.2.1 Earliest Visibility EV.

Before looking in detail at the fault tables, the principal of EV and how it can be used is shown by extracting the percentage fault results into a separate summary table. Table 3 shows the percentage of faults post EV per phase. This is the percentage of faults found during that phase of testing, that should have been found during an earlier phase. A dash '-' indicates no measurement, and 0 means that only faults relevant to the testing phase had been found, as none had been identified that should have been found earlier. Therefore the lower the figure the better the testing during the earlier phases. As discussed earlier (chapter 6), each testing phase will expose the problems with an EV linked to

that phase. When considering the EV results for a test phase a simple way to view them is to split the results into pre and post EV failures:

- post EV - failures that should have been found during an earlier phase of testing than the one in progress. The fault has been found later than expected, later than its EV.

- pre EV - failures that you expect to find during the test phase in progress or in a later phase of testing. The fault has been found during the expected phase or earlier than expected, before its EV.

Module coding problems should be found during module test $I_c \cap F_{mt}$, and system specification failures should be found during system test $I_s \cap F_{vt}$. If a problem is not found during the related EV test phase the testing is not good enough. The only way to deliver a quality product is for each testing phase to remove the faults that should be found in that phase, as each testing phase builds upon the one before it and will not necessarily find faults from an earlier phase (faults with an earlier EV).

As you move through the testing phases there is a tendency to find more faults that have missed their EV, due to the number of testing phases that have been passed and the additive effect of the faults in each phase.
The example of poor module testing of the patch 3C3 release shows that at integration testing almost 47% of the faults found are module coding faults. Not only is it unlikely that the integration tests will have found all the module coding faults, because the tests are targeted at finding different types of faults, but also the module coding failures are likely to be masking integration faults. Also build 3C2 has a post EV of 57% during system testing, which means that 57% of the faults found should have been identified in earlier testing phases. The more detailed tables will show that this patch release was almost a failure, not because of critical faults, but due to a large number of medium severity module coding errors. The system test team effectively had to produce and run the module tests. By contrast build 6 has a very low post EV failure rate, and 3C falls between the two sets of results.

Because this analysis is based on the type of faults found at different life cycle testing phases it can be applied to different software products, software languages and size of developments. As long as the testing phase life-cycle is defined along with the type of fault that will be tested for at each phase, this technique can be re-used. The target pre/post EV split will vary depending on the quality of the end product required. But following a couple of releases the target pre/post EV % figures can be calibrated to match the quality required.

The metric EV is original work for this thesis and therefore requires evaluation to determine if it is suitable for use during software development. The meta-metrics described in section 3.6 will be applied to EV and the three applications for the use of EV, process control, testing effectiveness and fault introduction rate.

1. Simplicity. *Does the metric lead to a simple result that is easily interpreted?* Pre and post percentage measurements for EV by test phase provide a simple way to decide if the correct phase of testing is taking place. Testing effectiveness is a percentage figure for each test phase showing the percentage of faults that have been found during the correct phase of testing. The fault introduction rate provides a split across all development phases. In all cases EV provides a simple result that can be easily interpreted.
2. Reliability. *Reliability indicates the degree of accuracy with which a characteristic can be measured.* EV does rely on the interpretation of each fault, which development phase introduced the fault and therefore where it should be found. In most cases this will be obvious to an engineer associated with the project being measured and the results would be reproducible.
3. Validity. *Does the metric measure what it purports to measure?* There is no implied association between EV and another measure, although there is a link with its use in testing process control. But the testing process control model proposed in this thesis does not rely solely on EV, a combination of metrics are used.

4. Robustness. *Is the metric sensitive to the artificial manipulation of some factors that do not affect the performance of the software?* The only factors that could impact EV when they change, is the life-cycle model being used. The development and test phases, the fault types associated with each phase should be defined at the start of the project and not altered throughout the measurement period.

5. Prescriptiveness. *Can the metric be used to guide the management of software development or maintenance? A metric used to guide development, will be assessed during the development and not at the end.* EV provides a way to measure and control the testing process during testing, plus the testing effectiveness and fault introduction rate results at the end of a testing life-cycle, for quality improvement.

6. Analysability. *Can the value of the metric be analysed using standard statistical tools?* A simple spreadsheet can be used to calculate and present the EV results. The data required for EV can be captured as part of a fault report. EV is therefore economical as there is a low cost for collection and analysis of the data, with a high pay back in control of the testing process.

7. Objectivity. *A measure can only be considered objective if it is free from any subjective influences of the measurer.* In the same way that Reliability can be influenced, so can objectivity by the interpretation of the faults into the development phase that generated the fault.

The two meta-metrics where EV warrants closer scrutiny are Reliability and Objectivity, due to the interpretation of the faults by the measurer. For this case study the author interpreted the faults and the results show that this has proved accurate. One test that has been applied to the EV results is to check if any of the pre EV figures show faults being found two or more test phases earlier that expected. It would not be unusual for a small percentage of faults to be found one phase earlier than EV, see table 9. But if a large proportion is found earlier than EV, or by more than one phase earlier than EV, see table 10, then this would indicate a flaw in the concept of EV or misjudgement by the measurer in interpreting the fault data. The results from Appendix D, summarised in table 10, show that across all builds in the case study, there is only two occurrence of faults being found more than one phase ahead of EV. Both of these relate to a single fault being identified two phases early, one in build C and another in

build D. The actual faults found one phase pre EV during I/T are shown in brackets in table 9 for clarification, as the percentage figures seem high but there is a low number of faults reported during I/T and the integration testing has included some system (P/R) testing.

8.2.2 Testing Effectiveness

EV can also be used to provide a measure of testing effectiveness by test phase; how effective has the testing been at removing the faults associated with that test phase. Table 7 presents the testing effectiveness measures as a percentage value, the higher the number the more faults that should have been found, have been found. A value of 100% equates to all problems with an EV equivalent to the test phase having been found. The values for this measure are only valid up to the last but one phase that data exists, for each build. The last phase will always be 100% because there are no further phases to show that faults have been missed. In this case a dash '-' is entered in the table. For some of the case study builds there is no value for the module testing phase effectiveness, as no data has been collected. But there has been module testing taking place and more importantly a number of faults with an EV of module test have been found during later test phases. Although it is not possible to calculate the testing effectiveness of module testing in these cases, it is certainly not 100% effective because of the number of faults that have been missed and found later.

Build 3C1 and 3D have not been included in the table of results as there is only a result for one test phase in each case. An improvement over the earlier builds can be seen in the effectiveness of the integration testing I/T at 80% and reference model M testing at 100% for build 6. System testing effectiveness at 61% for build 6 is similar to 3C all, but not as good as 3C345 at 88% and 3C6 at 93%. The change in testing strategy for build 6 has therefore improved the effectiveness of integration and reference model testing, but system testing has lost the improvements seen growing through the C builds.

8.2.3 Fault introduction

In addition to the EV metric supplying a measure of how effectively the faults have been removed, it can also provide a view of when the faults have been introduced. In order to establish the EV of a fault the stage at which it was introduced is required and is recorded as part of the data collection process. This therefore delivers a measure of development effectiveness (or non effectiveness as the fault introduction rate is measured) and provides a way to target, for improvements, the development phase that is creating the most faults. As this is expressed as a percentage of the total faults introduced (split across the test phases) it is used to highlight the phase introducing the most faults, but for a comparison between builds the actual number of faults needs to be compared to see if a reduction has been achieved.

Table 8 above provides a view across the case study builds, where the values represent the percentage of the total faults that have been introduced at each development stage. From table 8 it can be seen that 3C2 had a problem with module design and code as 50% of the faults for this build originated from this development phase. Also build 3C6 has double the customer requirement faults compared to the other builds.

8.2.4 Churn

The original scope for each of the three build's 3C, 3D, 6 is shown in table 5. The total number of enhancements and problems to be fixed is used as a crude measure of the amount of changes being made for each release. There is a limit to the amount of churn that can take place on a piece of software before the structure and flow of the code is lost due to simultaneous changes by engineers that did not produce the code in the first place.

It is difficult to determine the number of changes that can be made before the churn rate is too high and problems occur. It depends on the size of the initial build, how well spread the changes are and the interactions between the changes. Self contained changes spread over the software will have less

problems than a number of changes in the same area of code with a high level of interaction.

A measure based on a percentage change of the initial build size or function point analysis may prove possible, but for this project a trial and error approach found the limit to the size of change possible. This limit to the amount of changes possible within a fix, build and test cycle became the maximum number of changes for the scope of a build. Therefore any scope creep will provide an indication of unacceptable churn. Whatever set of changes are planed (scope of build) other changes/fixes will be added (scope creep) as problems are found during the testing of the preceding build. The percentage increase from the original scope to the actual build, is shown in table 5. Build 3D failed due to an unstable build, which in turn was due to the large number of changes (191 fixes) four times the initial scope for this build.

High churn combined with poor module testing leads to the initial problems found with the 3C development, of faults being introduced faster than they can be fixed. If the initial scope for a release is sensible then the ratio of actual scope to initial scope is an important indication of software build success/failure. It can be seen from table 5 that release 3D with a 400% increase in actual scope is a pointer to the failure of this build.

8.2.5 Severity shift

It was expected (see section 7.3.4) that early builds would have a large number of critical and high severity faults, and that these levels would decrease with further fix and build cycles. But it can be seen from table 1 that the critical faults start at a low level, 6% for build 2A, increase to 32% for build 3, and then drop to 8% for build 6. This set of builds took place over an 18 month period, with the first release at the end of the 3C patch builds. During this same period of time the high severity faults increased from 10% to 63%, while the medium's dropped from around 40% to 30%. The results for '3C all' in table 1 are the combined results from all the 3C patch builds shown in table 2. The same

pattern can be seen in table 2, which spans only six months, with a drop in criticals from 22% to 13% and a rise in high's from 40% to 57%. Build 3C2 is an exception, as this patch just added a reports package to the system and the functionality and importance is low compared to the full system. It is therefore unlikely that any critical problems will appear for build 3C2 unless the reports package impacts key functionality elsewhere in the system.

It is difficult to explain the low level of criticals found in the early builds, but the shift from critical to high can be seen from build 3 onwards. The drop in criticals shows the software maturing as the original faults in the software are found over the extended testing period. But for each fault that is fixed or new piece of functionality that is added, there is the opportunity to introduce further critical faults. If a graph is used to plot the number of critical faults for each subsequent build, the release point can be predicted based on the quality level required for a release. The graph below (Figure 40 Critical faults per build, page 169) shows both measures plotted for builds 3C to 6. It is more likely that the actual number of criticals (series 2) would be used rather than a percentage figure (series 1). Severity shift would therefore be used to track the maturity of the software, but to determine the likely quality of a build the actual number of outstanding faults and there severity will be used.

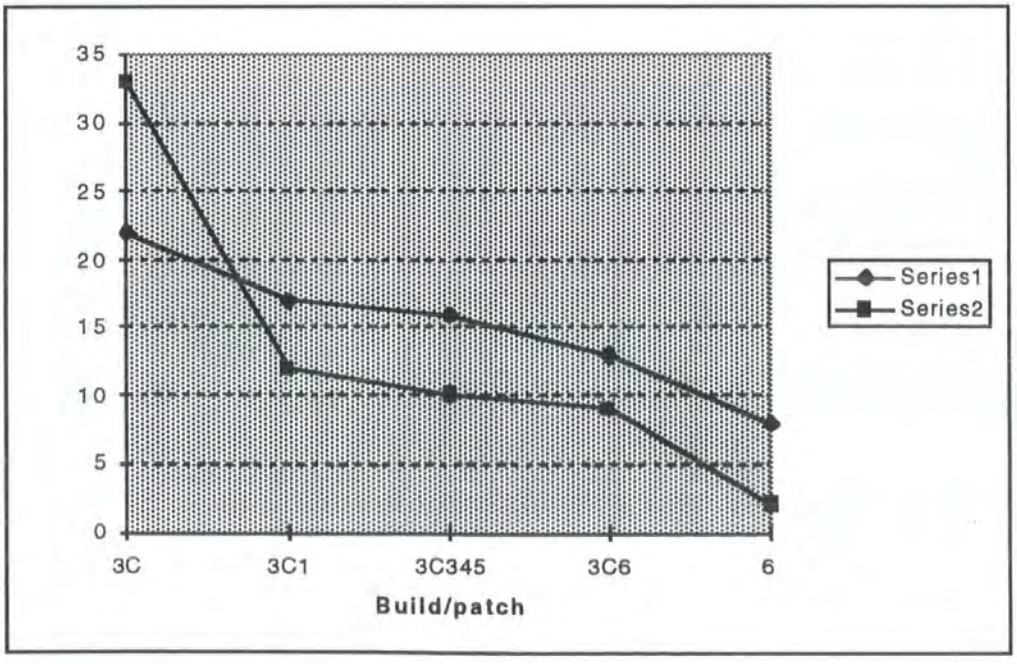


Figure 40 Critical faults per build

Further research is needed to establish if the shape of the graph below (Figure 41 Percentage critical faults, page 170, build 2A to 6) is a common feature for the discovery of critical faults over a number of builds, or if for this project the reporting of low level faults dropped in importance as the project progressed and therefore skewed the early measurements. Because the low severity faults recorded for the first couple of builds will lower the overall percentage of criticals.

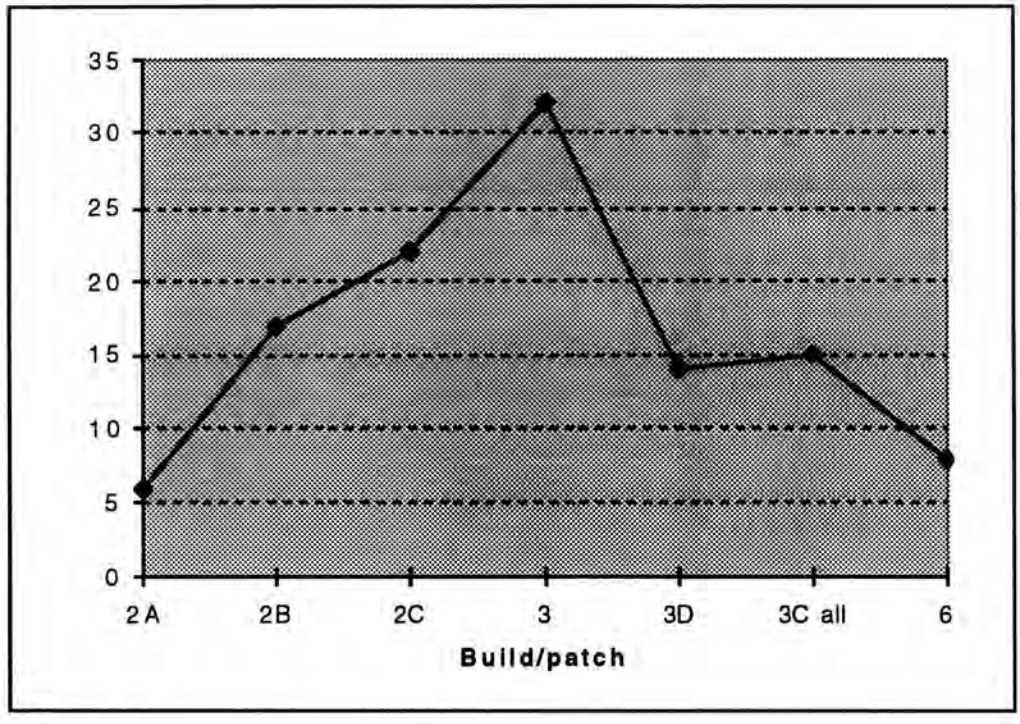


Figure 41 Percentage critical faults

The other question that needs answering is whether the high severity faults would during future builds for this product, increase to a peak and then start to fall as the medium and low severity faults increase. For the testing process control model the fall in critical faults will be used as one of the indicators of software maturity and improved quality.

8.2.6 Error rate halving

As explained in section 7.3.3, Rombach [114] found that the error rate halves with each phase of testing. Due to the dispersed development teams and different ways in which the results of module testing has been captured for this project, there is only a minimal set of data associated with module testing. Module testing fault count would be much higher than shown in the results tables and a drop from module to integration testing would be expected. Table 4 therefore shows the results from integration, validation and reference model testing. Builds C, C1, C345 and C6 show that the majority of problems have been found during validation testing. This is the result of an ineffective integration testing strategy. During this period the integration team would produce a build from all the modules that made up the system and then run a simple system test to demonstrate that the system worked. They did not test the interfaces between the modules and the interface faults were not uncovered until validation testing or later.

The strategy had changed by build 6, as one team of developers had been given the job of making changes to modules, testing them, integrating the modules, and testing the interfaces before running a simple system test. The results in table 4 for build 6 show a reduction in faults for each phase and suggest that for a well managed project with a test strategy that builds upon the preceding phase, error rate halving is correct. The testing process control model will use error rate halving as one of a set of measures to identify the correct progression through the testing life-cycle phases.

8.2.7 Faults per week

The average faults per week data is only available for the testing of build's 2A, 2B and 3, see table 6. The size of the test team may not be constant throughout the project so a better measure, on a weekly basis, is the average faults found per man day effort. Over a number of build and test phases this metric can provide a view on how easy it is to find faults with the build under test. A low level could be due to the majority of the faults being found or poor testing. If

the majority of faults have been found then depending on the target quality required, the next phase of testing can be started. If the low measure is a result of poor module testing, then module testing must continue and further expertise applied to the testing to improve the results. When this metric is used in conjunction with the other metrics identified for the testing process control model, the reason for a low figure can be identified and the measure can be used to trigger the movement to the next build or test phase.

8.2.8 Clear ratio

The clear ratio metric for builds 2A, 2B, 3, 3C all, and 6 have been measured using different standards. The first three are based on the number of faults found during the testing of a build, divided by the number of faults cleared (for whatever reason they have been cleared). The last two builds use confirmed and filtered faults divided by the number of fixes applied to the build. The data for these later two builds is therefore data that has been 'cleaned' to remove any inconsistencies or bad data. The data for the early build's is provided as it arrives from the data collection systems, before it has been cleaned. The ratio's can not therefore be compared directly between these two sets of results, but the general trend can. The ratio should start with a value greater than one, as the fault clearance process is front loaded, you have to find some faults before you can fix them. Once the initial faults within the first build have been discovered and a build and fix cycle is under way, the ratio should drop below one. Above one the ratio indicates that faults are being found faster than they are being fixed. If this occurs during a build and fix cycle after the initial build faults have been found the project is out of control as the fixes being added to a build are introducing a greater number of faults into the build.

The data in table 6 shows that this case study project did not stabilise until build 6, where the clear ratio drops to 0.44. Further detailed analysis of the faults found from build 3C through to build 6 could be used to determine if the faults are from the initial build or introduced by the fixes applied to later builds. To ensure that the testing process control model can determine if initial build

problems are still being found or if the project is running out of control, fault data must be linked to its source, initial build or later fix.

8.2.9 Assessment of combined measures

Each of the metrics explained above provides a measure of the testing process and the resulting quality of the end product. Each metric can only provide one measurement, one view of the testing process, but there are a number of factors that can influence the end quality. To provide a model that will take into account the main factors that impact the testing process a number of metrics are needed and are used in combinations to obtain the view required. Each of the builds from 3Call onwards are discussed and how the measurements taken are related to the builds.

3Call

There is a large number of faults found during the testing (423), with a large amount being critical (15%) and a poor clear ratio. The testing is therefore finding the faults, but from table 4 it can be seen that 70% of the faults have been found during system test and from table 3, that 35% of the faults are post EV. These 35% should have been found in earlier testing phases and points to poor testing during module and integration testing. Testing effectiveness measures of 6% for module testing and 36% for integration testing support this view. The detail results tables show that the 35% is split with a figure of 28% for faults that should have been found during module testing, and 7% for integration testing. The next phase of testing, reference testing, accounts for 20% of the faults found, but now the post EV figure has risen to 65%. The reference model testing has been started too early as most of the 65% are faults with an EV of system test. This testing process is very inefficient because each testing phase is finding faults that should have been found earlier, time is therefore not spent on trying to find that faults that the phase is equipped to find, faults are missed, and the next phase is started but now with faults to be found from several testing phases that did not complete. Even during reference model testing of 3C all, 6% of the faults have an EV of module test. The 329% churn figure suggests that too much is being changed compared to the plan for this

release, which has resulted in the high level of faults within the code. It is possible to look in more detail at 3C all, as the information on the patch builds has been captured:

- Build 3C. 147 faults found, 22% critical (see table 2), 56% of the faults found during system test and 17% post EV. The reference model testing found 30% (see table 4) of all the faults, but 64% post EV, all of them had an EV of system test (see table 3). Testing effectiveness is low at 39% for module test (see table 7), although this is probably due to a lack of data for module testing. 69% for integration and system testing, which is low compared to the target of 80%. Most of the errors are system specification errors at 61% (see table 8). This covers faults with the specification, incorrect interpretation or the system under test not meeting the specification. This build should have been returned to system test, and highlights problems with the specification or delivering to the specification.

- Build 3C1. The number of faults dropped to 70, 66% introduced during system specification with a total of 17% critical, an improvement over 3C, but this might just be that the faults are not being found because the system test has a post EV figure of 33%, and reference model test has 72% with an EV of system test. 66% of the errors are specification faults, the highest percentage for any of the builds monitored. Another release that should have been returned to system test. There was not enough data across all the phases to provide a testing effectiveness result.

- Build 3C2. This is the addition of a new reports module, 70 faults although only 1% are critical which is expected as this is not a major piece of functionality that will cause a lot of problems if it should fail. Most of the faults (97%) have been found during system testing, with a 57% post EV rate, 51% have an EV of module test. Looking at these figures together, with almost all the problems found during system test, but half of them should have been found during module test the analysis points to poor module testing. Unfortunately there is insufficient data to provide a measure of testing effectiveness to substantiate this claim but the fault introduction level for module design/code is 50% (the highest for any build) and most of

these did not get found until system test. The maximum 100% testing effectiveness is recorded for system testing, but 0% for integration testing effectiveness, showing that there are problems with integration testing as well as module testing. The impact of this module on the complete IMS system is low as the module is self contained with a minimum of integration with the system, but the cost of implementation for this module could have been reduced if all the faults with an EV of module test had been found during module test and fixed, before the module reached system test.

- Build 3C3,4,5. 64 faults, 16% critical, a slight improvement over 3C1. There is an increase in the percentage of faults found during module testing, now 23% of total, 61% are still found during system test. The post EV figures show that a large number of faults are not being found as soon as they should, 47% post EV at integration test, 38% at system test (26% have an EV of module test) and 80% at reference model test (50% have an EV of module test, 30% an EV of system test). A fault introduction figure of 34% for module design/code and 39% for specification show that although there is an increase in the number of faults found during module testing, there are a higher percentage of module faults that have been introduced. The testing effectiveness of the system testing is measured at 88%, but only 55% for integration testing. Increase module testing or better quality module testing should have been applied to this build before the latter testing phases.

- Build 3C6. 72 faults, 13% critical, 83% found during system test with 35% post EV. Most of the faults are being found during system test, and a third of them should have been found in earlier testing phases. The testing effectiveness at system test is high at 93%, but integration testing is low at 38%. The fault introduction metric identifies a figure of 24% for customer requirements, which is the highest for any build. This could have been due to the fact that this build did make it to field release and therefore more effort would have been applied during customer acceptance test when requirements faults would have been found. In summary, this build is still suffering from poor module testing, which results in wasting money on fixing problems at a higher cost later in the life cycle.

Build 3D

Only 28 faults, 14% critical. The testing on this build was stopped during system test due to the build being unstable, so the discovery by phase table shows a high percentage for integration testing as the system testing was not completed. A high 42% post EV figure for integration testing identifies poor module testing, and is similar to builds 3C3 and 3C6. The main warning signal for this build is the high 406% scope creep on an already high target of 47 changes.

Build 6

24 faults, 6% critical, discovery by phase approaching error rate halving, with low post EV rates of 17%, 11%, and 0%. The churn has been kept to 229% and on a small target of 24 changes. Clear ratio at 0.44 shows that the development phase is under control, with very few faults being introduced due to the target changes. Of the faults introduced, 58% are system specification faults and with only 61% effectiveness at removing them during system test this could still be improved. The 80% testing effectiveness for integration testing and 100% for reference model testing shows a good improvement over earlier builds. The improvements in the indicators for build 6 over the 3C builds can also be seen in the resultant field problems. Although 3C3 and 3C6 had short field lives (a few months) compared to build 6 (several years) 145 field reported faults for 3Call is much higher than the 22 for build 6. Both have similar post EV figures for the field problems, 55% post EV for 3Call and 50% for build 6. So 50% of the field problems found should have been identified during testing before release.

8.2.10 Target measurements

The early builds did not perform well and the metrics applied to them have shown up problems in over ambitious developments, poor quality of development, lack of integration, and poor testing. The last build measured,

build 6, performed much better with a successful development, little scope creep and a low level of faults introduced, restructured integration and improved testing. These builds therefore provide a range of results for the metrics being used and some target measures that can be applied to manage the testing process and determine the quality of the end result. More analysis is needed from other projects to calibrate the proposed testing process model, but based on the results so far the minimum expected before release is:

- less than 200% scope creep
- clear ratio below 1, after initial build faults have been found
- error rate halving per test phase
- post EV below 20% for each phase
- critical faults less than 10% of total
- faults per week. Plot over time, look for fall in rate
- testing effectiveness of 80% minimum
- development effectiveness (fault introduction). Target highest error introduction rate.

8.2.11 Detail results tables

Appendix D, holds the detailed results tables. For each build of software there are two pages of tables. The first page holds the fault counts and second page the percentage calculation results. Taking each of the two pages in turn, the first page has five tables, one for each phase of testing. Each of these tables shows the number of core software faults found during the test phase, broken down by EV (columns) and criticality (rows). Core software faults are the software faults found within the IMS system under test, extra columns at the end of the tables also include a count of data problems, inter system faults (other systems that connect to the IMS), and installation faults, by criticality. These extra columns are not used any further during the analysis of the faults but do provide an indication of the number of these type of faults in relation to the core software faults.

Below these first five tables on the first page are another two. The first collects the data required to calculate testing effectiveness. The columns represent a test phase and the associated faults of EV that should be found during the test

phase. The first row of the table holds the count of the number of faults with the correct EV for the test phase, that have been found during that phase and faults that have been found in earlier test phases but with this EV (pre EV faults). The next row holds, again by EV, the faults found that are post EV. In other words, the faults that should have been found during the phase under analysis, but in fact were found in a later phase. The third row provides the total number of faults found by EV (including field faults) by the addition of the first two rows. The last table on the first page collects together, by EV, the faults sorted by criticality for fault introduction by development phase. This table does not include faults found during field use, so that the different software builds can be compared, as field failures are only available for two builds.

Where tables have no data this means for the early test phases it was not available, for the later test phases and field use this means that the software build did not progress this far through the testing life-cycle. The tables of results on the first page map across onto the tables on the second page, which provide the results of the calculations required for the metrics identified in this chapter of the thesis.

The second page of results therefore, contains the five tables of results, which show for each test phase the percentage of faults found that are EV or earlier (in bold), and post EV (the faults found later than they should have been). The results are presented in rows by criticality and an overall post EV/ EV figure is presented at the top of each testing phase table. At the right hand end of each table, the total of the faults within each row is shown as a percentage of the total number of faults found during all phases of testing (excluding any field faults). The next table after the five EV tables, provides the results of the testing effectiveness calculation. The higher the percentage figure the greater the effectiveness. The results are split by testing phase per column. The last table presents the percentage of the total faults by EV and criticality, excluding field faults, split across the development phases. This provides a measure of the development effectiveness, or which phases are introducing the faults. The first five tables therefore provide a breakdown to EV per phase, and the last table the breakdown by EV across all phases.

8.3 Resource usage analysis

The table below (Table 11, Resource usage analysis) presents the data collected over a one year period across three projects. The first project (1A & 1B) follows the second main release of the IMS system plus a follow-on build and test phase to fix outstanding problems, so although there is some re-work of the test cases within the test preparation category most of the effort is spent on running the tests.

% effort	1A	1B	2	3
process info gathering	0	0	4	0
product info gathering	7	0	10	12
strategy & planning	1	1	7	17
testability analysis	1	0	1	0
integration preparation	0	0	14	0
integration execution	0	0	18	0
system preparation	38	16	17	51
system execution	31	78	20	20
CM & build	7	1	1	0
data development	14	1	1	0
environment	1	3	7	0
Total man hours	1980	1617	1714	173

Table 11. Resource usage analysis.

There is little work needed (10%) to understand the product/process, define a test strategy, provide testability analysis or build a test environment as all these areas have been covered by the first release of the system. As this system has a data driven configuration there is a fair amount of work (14%) to ensure that the data is built and validated before system testing can start. The data build is re-used for the phase 1B testing so no effort needed for the re-test.
Project 2 is a new system to be tested so 29% of the total effort is needed to prepare for the testing. Both integration (32%) and system validation testing (37%) are covered, with in both cases the test execution taking slightly more effort than the test preparation.
Project 3, as can be seen from the total effort, is a small development to be tested, delivered built and ready to test. In this case 29% of the effort is spent understanding the project and developing a test strategy and plan. A high percentage (51%) is spent on preparing the test cases and only 20% on test execution, this is probably due to the inefficiency of test case preparation for such a small project.

8.4 Summary

Details of the software build and release programme of the IMS system that has provided the data for the case study has been explained. The fault data has been analysed, by looking at the results from applying the metrics EV, testing and development effectiveness, Churn, Severity shift, error rate halving, faults per week and clear ratio to the data. By applying combined metrics to the IMS software builds it can be seen how the quality of the software is improved over time and how the metrics can identify problems in development and testing. The way in which the metrics can be applied, to control the testing process and movement through testing phases, plus a measure of quality that is used to decide if a build should be released and the effectiveness of the testing, are all explored. Target values are proposed for the metrics. An explanation is given on how to read the detailed results tables in Appendix D. The analysis of the resource usage data provides the measures for a testing resource estimation model based on an evolving set of metrics that provide an estimate that increases in accuracy over time.

9 Conclusions

This chapter concludes the thesis, with a summary, a review of the original objectives for this research, achievements, limitations and future work that could be undertaken to advance the proposed model.

9.1 Thesis summary

The first chapter of this thesis identified the increasing reliance on software as the use of computers and the size of programs continue to grow. Also the cost and time overruns, and the large proportion of effort spent in maintenance are discussed as some of the problems facing industrial development projects today. A review of the testing process, following the 'V' life-cycle model through each of the phases from requirements to customer acceptance is then covered.

The main point to be made from this broad look at verification, is the emphasis on gaining a level of confidence in a module of software or a complete product, and not the impossible task of trying to prove that the software is 100% correct, or that all the faults have been removed. This arises due to the impractical nature of trying to prove that something as complex as a computer program is correct and also the reality that no one would be prepared to finance such a costly task even if it were possible.

The third and fourth chapters review the use of metrics and models within software engineering, provide an understanding of quality and form the basic reference on past work in this field. There is always a balance between quality and cost, with the attributes of quality (FURPS) providing one method of

measuring quality. An explanation of sizing, estimation and complexity metrics along with meta-metrics for validating metrics is included. Models provide the framework for metrics but like metrics also need validating with real data to prove that the results are accurate and practical.

Sizing models are best suited to particular development environments, and some of them require a database of past project information before they can be used effectively. The estimating models like COCOMO provide much more information, in the form of costs, effort and manpower profiles but they do tend to be much more complex than the sizing models and therefore require more effort to use them. Some of the estimating models also need as input, an estimate for the size of the software to be produced, before they can provide the cost and manpower predictions. If this is the case, then the inaccuracies of estimating the size in the first place will be passed on into the further predictions.

Complexity metrics are best employed in keeping module complexity within certain limits and for the direction of testing resources to modules with high complexity and associated high defect counts. In the same way, the defect and reliability models are best used to identify the modules that will have the high defect counts rather than trying to determine the exact number of defects in the software at any one time.

Original work starts in chapter five with the presentation of a new fault propagation model which captures the parallel activities across the development life-cycle as faults are introduced and discovered. Set theory notation is used to describe the flow of faults between phases and Venn diagrams are used to explain the relationship between faults and test coverage.

Predicting the number of faults in software does not have a practical application unless the criticality of the problems and the likelihood of them being found by the user can also be predicted. The testing process control model proposed in this thesis concentrates on improving the control of the process to remove the maximum faults, but will not predict the number of faults to be removed.

Earliest Visibility is explained in chapter six along with the hypothesis that there is a relationship between the development phase that a fault is introduced and its EV. There are a number of opportunities for improving the effectiveness of testing using the EV measurement:

1. a method for deciding the optimum point at which to move onto the next phase of testing, and therefore managing the cost/quality balance for the project.
2. an indication that the testing has moved on too quickly and that a return to a previous phase is required to find and remove remaining faults.
3. by controlling the movement of testing between phases for a number of projects a fault database with the relationship between the number of faults per stage can be collected. This data can be used to produce a prediction of fault level for later stages of testing.
4. providing a measurement of the testing effectiveness by test phase and fault introduction rate by development phase.

A model based on re-estimation is proposed for testing resource estimation.

The proposed model and metrics have been developed and tested on a large scale (4 million LOC) industrial telecommunications product written in C and C++ running within a Unix environment. The case study is explained in chapter seven.

Chapter eight holds the results from the case study and explains how a number of measures can be combined to determine the status of a software build and the expected quality level.

9.2 Review of Objectives

The two key objectives for this thesis:

1. improvements to the testing process
2. better estimation of testing resource

delivered via a software testing resource estimation and process control model.

This thesis explores the use of testing to remove faults, the part that metrics and models play in this process, and considers a new method for improving the quality of a software product. Improvements to the testing process results in improved efficiency, reduction in cost of the system tested and a higher quality product. By measuring where testing effort was spent on various projects an improved system for testing effort estimation can also be built. The thesis investigates the possibility of using software metrics to estimate the testing resource required to deliver a product of target quality into deployment and also determine during the testing phases the correct point in time to proceed to the next testing phase in the life-cycle.

9.3 Achievements

The thesis provides a full analysis of the results from the case study (Appendix D) and demonstrates how the metrics proposed can be used to measure aspects of the software testing process and estimate testing resource. The measures EV, churn, severity shift, error rate halving, faults per week, and clear ratio have been used across a number of builds during the development of a telecomms product and demonstrate how each measure can be used to piece together the status of a build, to determine how close it is to the target quality. The metrics also determine if the correct phase of testing is taking place and can highlight if a return to an earlier phase of testing is required. This in turn improves the resource estimate as improved accuracy of the resource measurement on preceding testing phases will result from better control of the testing phases.

Practical guidance is provided on using the metrics, including the data measurements required and an explanation on how a combination of metrics are used to construct a process control model. The use of a set of metrics to 'control' the testing process is original work and the definition and use of EV for process control and testing/development effectiveness, is also original.

Section 8.2.1 applies the meta-metrics described in section 3.6 to EV and the three applications for EV, process control, testing effectiveness and fault introduction rate. This verification of EV as a metric and its applications, for operational use by the analysis of the case study results and the use of meta-metrics forms the basis of the thesis proposal and experimentation for software testing process control.

9.4 Limitations

The case study used to prove the hypothesis for EV and test the measures used in the process control model follows a standard 'V' development life-cycle. While in theory the measures and model should be life-cycle independent, it has not been proven and any use with other types of development life-cycle should be evaluated first.

In order to use EV for process control, each fault reported must be analysed to determine in which development phase it had been introduced and therefore its EV. This requires commitment from a development team to undertake this analysis and backing from the project manager to accept the cost of this activity and to act on the results.

The testing estimation model requires calibration before it can be applied, so data from a number of past projects using the same skill base, development/testing tools or development methodology is needed. Any major changes to the skill base, tools or development methodology once the model is in use, will necessitate a re-calibration of the model.

9.5 Future work

In order to resolve some of the limitations listed above, more analysis is needed from other projects to calibrate the proposed testing process control model described in chapter 8 and Appendix C. Also use of the model on further projects to demonstrate its potential, following the standard waterfall development process and other techniques like RAD, that are fast becoming the standard methods for development today. Because the EV analysis is based on the type of problems found at different life cycle testing phases it can be applied to different software products, software languages and scale of developments. As long as the testing phase life-cycle is defined along with the type of problem that will be tested for at each phase (process entry and exit point), this technique can be re-used. The target pre/post EV split will vary depending on the quality of the end product required. But following a couple of releases the target pre/post EV percentage figures and the other proposed metrics can be calibrated to match the quality required.

It is difficult to determine the improvements that the process control model can bring to development projects unless parallel teams can test a developed product, one using the model, one relying on standard project management techniques. But this is not practical with industrial projects due to cost and the scarcity of testing resource. The only method that can be applied is to monitor the cost, delivery and quality of projects within a development organisation before applying the testing process model and then afterwards. If no other changes, (new tools, development process, different products etc.) have been made during the observation period, any improvements are likely to be the result of applying the model.

Further research is needed to establish if the shape of the percentage critical faults graph (Figure 41, page 170) is a common feature for the discovery of critical faults over a number of builds, or if for this project the reporting of low level faults dropped in importance as the project progressed and therefore

skewed the early measurements. Because the low severity faults recorded for the first couple of builds will lower the overall percentage of criticals. The other question that needs answering is whether the high severity faults would during future builds for this product, increase to a peak and then start to fall as the medium and low severity faults increase. It will be important to establish a typical 'critical faults by build' graph as this is used as one of the indicators of software maturity and improved quality within the testing process control model.

It has been difficult to collect data from the early testing phases of the projects monitored, but with a model and metrics defined it would now be possible on a new project to put the appropriate measures in place from the start and collect this early data.

9.6 Summary

This chapter contains a summary of the thesis, review of the objectives set out at the start of this thesis and demonstrates the extent to which they have been met, via the achievements. The limitations are considered and a section on future work (section 9.5) provides recommendations for improving this work where the objectives have not been met completely. This includes work that will expand the scope of the Software Testing Resource Estimation and Process Control Model to cover other development methodologies.

Appendix A Verification methods

Error seeding (mutation testing)

A predictive model for determining the number of remaining software errors or the effectiveness of the test cases. By injecting errors into the software and then counting the seeded versus existing errors found, an estimate of the outstanding errors can be made. This model is sometimes called the fishpond model. To save later embarrassment, good records of the seeded errors should be kept.
e.g. 20 errors injected, 5 of them found by the test cases and another 3 existing also found. Therefore as 25% of the seeded errors found, 3 represents 25% of the existing errors; estimate for remaining errors = 9.

Equivalence partitioning and Boundary-value analysis

These two techniques are used to reduce the large possible range of tests down to a small subset that are most likely to find a problem. Equivalence partitioning is based on the principle of partitioning the input domain of a unit under test into a number of equivalence classes. The assumption follows that any value within that equivalence class is just as likely to find an error as any value within the class. Therefore it is not necessary to try all the possible input values within that class, one will suffice. If the value selected does not uncover a problem then it is expected that none of the other values would either. To identify the equivalence classes for a system take each input in turn and follow the procedure below. An input can be considered as the lowest level of identifiable data.

Input examples:

a user entered field within a screen as part of a menu interface a data byte representing a temperature from a temperature sensor the number of entries that a user can make to a particular prompt (enter three dates)

Procedure:

Form two equivalence classes initially, valid and invalid inputs. Split these classes into further classes if the program treats some input values in a

different manner (age, under 18 and over 65). For each of the equivalence classes produce a test case with an input value from that class.

Boundary-value analysis complements equivalence partitioning and builds on the equivalence classes defined. The boundaries between equivalence classes are quite often error prone; therefore test cases that explore these areas have a high payoff rate. In addition to the boundaries between equivalence classes extra data dependent boundaries will be worth examining.

Examples:

If positive and negative values accepted, try zero and either side of zero.

The maximum and minimum of input values, plus max+-1, min +-1.

For the number of entries allowed, try the maximum, plus max+-1. If the data structures are known for the program, tables, arrays, lists etc.. try the first and last elements.

The counting system used, check magic numbers like 64,128,256. The boundaries for the output data should be treated in the same way that the input boundaries are exercised.

Error guessing

This is not a well defined technique but an intuitive process based on experience. The idea is to establish the types of errors that are likely to have been made during the design and implementation process, producing test cases to expose these errors. Error guessing should be used in addition to the other techniques identified and not as the only method of test case generation.

Cause effect graphing

This technique covers combinations of inputs, an area not covered by boundary value analysis or equivalence partitioning. As cause effect graphing very quickly becomes unworkable with larger specifications, a small part of the specification is worked on at a time.

All the causes(inputs or classes of input) are identified, along with the effects (outputs or a changed system state). A Boolean graph linking the causes and effects is drawn following the specification semantics. The graph is then annotated with the constraints that describe the impossible combinations of

causes and effects. State conditions from the graph are converted into a limited-entry decision table. Each column in the table can then be converted into a test case.

Statement coverage

Every statement in a program under test to be executed at least once. This is not a very satisfactory criteria , as some paths may skip a number of lines of code and yet still form the main route that the program will normally take. In this case the most useful path to test will have been missed by this technique.

Decision coverage

Each decision is forced through its true and false paths by the test cases. For multiway decisions (case statements) each possible outcome must be exercised. Another way of specifying decision coverage would be the coverage of all arcs on a flow graph.

Condition coverage

Each condition in a decision takes on all possible outcomes at least once.

Testing principles

To complete the background information on testing, statements from Myers 'The art of software testing' have been included from his chapter on the psychology and economics of program testing - testing principles.

- A necessary part of a test case is a definition of the expected output or result.
- A programmer should avoid attempting to test his or her own program.
- A programming organisation should not test its own programs.
- Thoroughly inspect the results of each test.
- Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions.
- Examining a program to see if it does not do what it is supposed to do is only half of the battle. The other half is seeing whether the program does what it is not supposed to do.

- Avoid throw-away test cases unless the program is truly a throw-away program.
- Do not plan a testing effort under the tacit assumption that no errors will be found.
- The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
- Testing is an extremely creative and intellectually challenging task.
- Testing is the process of executing a program with the intent of finding errors.
- A good test case is one that has a high probability of detecting an as-yet undiscovered error.
- A successful test case is one that detects an as-yet undiscovered error

Appendix B Test Documentation

IEEE 829 [22] test documents:

Test Plan
Test-design specification
Test-case specification
Test-procedure specification
Test log
Test-incident report
Test-summary report

The diagram below shows the hierarchical relationship between the documents.

The purpose and contents of the documents are then described following the diagram.

TEST PLAN

Purpose. To prescribe the scope, approach, resources, and schedule of the testing activities. To identify the items being tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risks associated with this plan.

Outline. A test plan shall have the following structure:

1. Test-plan identifier
2. Introduction
3. Test items
4. Features to be tested
5. Features not to be tested
6. Approach
7. Item pass/fail criteria
8. Suspension criteria and resumption requirements

9. Test deliverables
10. Testing tasks
11. Environmental needs
12. Responsibilities
13. Staffing and training needs
14. Schedule
15. Risks and contingencies
16. Approvals

TEST-DESIGN SPECIFICATION

Purpose. To specify refinements of the test approach and to identify the features to be tested by this design and its associated tests.

Outline. A test-design specification shall have the following structure:

1. Test-design-specification identifier
2. Features to be tested
3. Approach refinements
4. Test identification
5. Feature pass/fail criteria

TEST-CASE SPECIFICATION

Purpose. To define a test case identified by a test-design specification.

Outline. A test-case specification shall have the following structure:

1. Test-case-specification identifier
2. Test items
3. Input specifications
4. Output specifications
5. Environmental needs
6. Special procedural requirements
7. Intercase dependencies

TEST-PROCEDURE SPECIFICATION

Purpose. To specify the steps for executing a set of test cases or, more generally, the steps used to analyse a software item in order to evaluate a set of features.

Outline. A test-procedure specification shall have the following structure:

1. Test-procedure specification identifier
2. Purpose
3. Special requirements
4. Procedure steps

TEST LOG

Purpose. To provide a chronological record of relevant details about the execution of tests.

Outline. A test log shall have the following structure:

1. Test log identifier
2. Description
3. Activity and event entries

TEST-INCIDENT REPORT

Purpose. To document any event that occurs during the testing process which requires investigation.

Outline. A test-incident report shall have the following structure:

1. Test-incident-report identifier
2. Summary
3. Incident description
4. Impact

TEST-SUMMARY REPORT

Purpose. To summarise the results of the designated testing activities and to provide evaluations based on these results.

Outline. A test-summary report shall have the following structure:

1. Test-summary-report identifier
2. Summary
3. Variances
4. Comprehensive assessment
5. Summary of results
6. Evaluation
7. Summary of activities
8. Approvals

Original improvements to the test documentation process.

Pilot scheme.

As soon as the test strategy has been completed and the preparation has started, documentation starts to become a major problem. The main culprit for eating up documentation effort are the test cases. This is due to the sheer number of test cases needed; there is a one to one relationship between test cases and tests that are run. The test case documentation effort can be out of proportion time wise when compared to the time required to run the tests. A test that may take seconds to run might well take an hour to document. So deciding on the level of detail for the tests and the amount of information to be held within the testcase is one of the first considerations.

The normal way to approach the test preparation phase and the production of the documentation is top down, strategy first followed by the test design and then the test cases and procedures. By the time that it is apparent that an incorrect split has been made between the test designs or that the level of detail is wrong for the test cases, too much time will have been taken out of the project to put things right. This is because the activities are going on in parallel and by the time that the test cases are well under way the testing preparation time is almost exhausted. Although a standard set of documentation procedures and templates can be agreed across many projects the level of detail and the split of testing areas will change with every development project. Each project is different and it is these differences which create the unique approaches to testing and therefore the test documentation structure.

The idea behind a pilot scheme is to take the test documentation right the way down to completing some test cases but in a very narrow field. This could be considered as taking a slice through the test documentation. The level of detail required for the test cases can be established very early on in the test preparation phase, and a standard can then be applied to the rest of the test cases. If this approach is used for each of the test designs the split of the testing work between the test designs can be assessed, again before too much effort has been put into the test preparation.

Appendix C Estimation and process control model

The metrics identified in the thesis and evaluated during the case study are combined to form the software testing estimation and process control model. Knowledge gained during the case study has been used to refine the model and set the ranges/limits for the measures. Direction is also provided on how to use the model operationally.

Metrics - components of the model

Some of the metrics defined in chapter 8 are used to determine if the correct life-cycle phase (phase) of testing is being undertaken, others are used to monitor progress within a phase and to control the repeat build process (repeat build). The effectiveness measures are used to monitor the procedures, methods and skills of the development and test teams (methods & skills).

Metric	Application	Description (section No.)
post EV %	phase	8.2.1
Testing effectiveness	methods & skills	8.2.2
Development effectiveness	methods & skills	8.2.3
Churn	repeat build	8.2.4
Severity shift	phase & repeat build	8.2.5
Error rate halving	phase	8.2.6
Faults per week	phase & repeat build	8.2.7
Clear ratio	repeat build	8.2.8

All the above metrics can be constructed from the data listed below.

1) Individual fault reports containing:

- Fault identification number (CR No.)
- Date found
- Severity
- Software build
- Testing phase discovered
- Earliest testing phase for possible discovery (EV)
- Root cause - initial build or previous fix

2) Scope of build

3) Fix list at build

4) Testing resource applied

The diagram below contains the process flow and metrics that make up the testing process control model. For each of the testing life-cycle phases there is a build, test, build review and fix cycle. This applies to module, integration and system builds. The build review will measure the improvements in the builds within a testing phase or identify failure if the clear ratio is too high, severity shift is missing, faults per week still too high or churn is too high.
The monitor and review process runs alongside the build review and will monitor the results from each build within a testing phase, but will also use the EV and error rate halving metrics to determine if the correct phase of testing is being applied. The process control monitor will provide advice to the project management team on whether to continue testing in the current phase, return to a previous phase because the post EV metric is too high, or move onto the next phase if the build review, EV and error rate halving are within set bounds.
On satisfactory completion of all testing phases the monitor review process will provide the data for a release review.

In addition, as explained in chapter seven, EV is also used to provide a measure of the effectiveness of the testing by test phase and the development process (which development phases are making the least errors). This information is used to highlight any weak points in the development and test process and to target quality improvements. The measures can only be used on the completion of a development and test lifecycle, as EV results from each phase are required

before the results can be calculated. The development figures are given by development phase as a percentage of the total errors introduced across all phases. This will highlight the development phase introducing the highest percentage of errors, but to see if improvements are being made over several builds then actual numbers rather than a percentage should be used. As the testing effectiveness measures provide a percentage of the expected faults found per test phase these results can be compared between builds to see if improvements have made a positive impact.

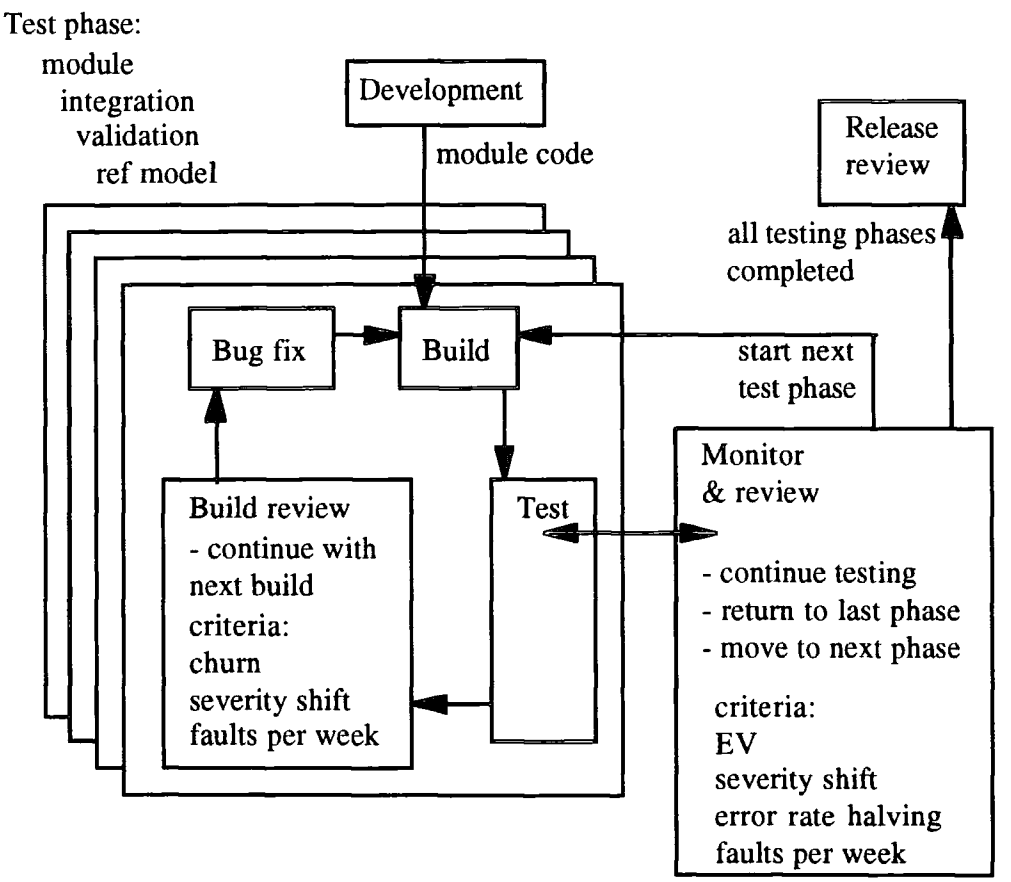


Figure 42 Testing process control model

Operational use

Data capture with a minimum time and cost penalty to a development project is key to a successful metrics programme. Grady & Caswell, "the reason project managers collect data is that it is impossible to make responsible, informed decisions without it. However, there is a cost to data collection. It is an investment whose return depends on the implementation of the data collection and analysis process" [46]. The majority of the data required for the metrics proposed in this thesis can be collected via the trouble ticketing system and engineering timesheets. These are common data capture measures that will be in use in most software development organisations. But some enhancements will be needed to the data capture systems to increase the scope of the data collected to cover measurements such as EV.

It is important to insure that the data is recorded accurately and that the data captured is not influenced by the measurement process. There is always the possibility that the data may be recorded in a way that presents an engineers work in the best possible way, or that it may influence an engineer to concentrate on certain aspects of work because they are being measured. Grady & Caswell, "software engineers can and will successfully work to maximise whatever elements you choose to give priority to" [46].

A simple spreadsheet (similar to Appendix D) can be used to calculate and display the proposed metrics. Data from the trouble ticketing and timesheet collection systems can be totalled on a weekly basis and entered into the spreadsheet. The metrics can then be displayed as a simple table showing the results of the spreadsheet calculations for each life-cycle phase and for each build/test iteration. The trends through the testing phases and builds can then be viewed directly from the tables, or as a set of graphs and used to provide management control to the testing process. The target measures provided in

section 8.2.10 can be used to identify a project that is progressing satisfactorily and one that is running into problems. A three tier scale can be used to provide a top level view of progress, with the scale for each metric calibrated to show:

- project within acceptable limits
- indication of possible problem, investigation required
- outside acceptable limits, action required.

Progress through the testing life-cycle can be influenced or controlled by the use of the metrics identified in this thesis. A requirement on moving between test phases or the release of a software build can be a set of metric results that are within specified limits. This control can be built into quality gates and the release review process. "A quality gate is, as its name suggests, a gateway that clearly delineates phases in a development life-cycle and provides the opportunity to audit and maintain appropriate quality levels at those points" [12].

Resource estimation

Based on a combination of, BT testing experience [12] across ten projects (see section 1) and the results (section 8.3) of the measurements taken for the case study, plus the figures on cost estimation provided by Rombach [114] for NASA SEL projects, the proposed estimation model is:

$$Total_{est} = D_p + D_d + C + T_p + T_i + T_s + A$$

Key:

$Total_{est}$ = estimation of total cost

D_p = preliminary design	$D_p = 0.15 \times Total_{est}$
D_d = detail design	$D_d = 0.17 \times Total_{est}$
C = code and module test	$C = 0.26 \times Total_{est}$
T_p = test preparation	$T_p = 0.07 \times Total_{est}$
T_i = integration test (I/T)	$T_i = 0.075 \times Total_{est}$
T_s = system test (P/R)	$T_s = 0.085 \times Total_{est}$
A = acceptance test (M)	$A = 0.19 \times Total_{est}$

Estimation steps:

1) $$Total_{est} = \frac{estimate(D_p + D_d + C)}{0.58}$$

2) $$Total_{est} = \frac{D_p}{0.15}$$

3) $$Total_{est} = \frac{D_p + D_d}{0.32}$$

4) $$Total_{est} = \frac{D_p + D_d + C}{0.58}$$

5) $$Total_{est} = \frac{D_p + D_d + C + T_p}{0.65}$$

6) $$Total_{est} = \frac{D_p + D_d + C + T_p + T_i}{0.725}$$

7) $$Total_{est} = \frac{D_p + D_d + C + T_p + T_i + T_s}{0.81}$$

This resource estimation model does not include the cost of requirements capture and specification, it starts with the preliminary design and continues to the end of acceptance test. Deployment and maintenance has not been included. As described earlier in this thesis (section 6.1) the objective is to gradually refine the estimate as more data becomes available, so that by the time integration testing starts more than 50% of the estimate is based on actual data. $Total_{est}$ is the estimate for the total resource cost for the project and at each stage of the life-cycle $Total_{est}$ should be estimated using the best data available.

The estimate for the phase required should then be calculated from the equations in the key above, all based on $Total_{est}$. This process is started with an estimate from the development team on the cost of development, step 1. As soon as some actual measurements are available, completion of preliminary design D_p, the original estimate is replaced by the results of the calculation in step 2. It can be seen from the estimation steps that the value of $Total_{est}$ is gradually refined as the development progresses and the accuracy of the estimates for the remaining phases will also increase. All the measurements and estimates will use the same units of measurement (man days, months, or years) although any one can be selected for use at the start. The model will require calibration for different development/testing environments (process, tools, skills), but because of the experience from a wide range of projects that have provided the data for this model only minor adjustments should be required.

To ensure the accuracy of the estimates from this model, as it relies on accurate measurements of the phases completed, it should be used in conjunction with the testing process control model. Movement through the test phases will then be controlled and movement will be on the basis of completed testing and not completion of the testing budget for that phase. This will provide accurate resource measurements as input to the estimation model and the model can then produce accurate estimates as output.

This is the key improvement over similar resource estimation models. Because the movement between test phases is being controlled, based on measurements to determine if a phase is completed, the data can be compared between projects using the same control mechanism. In the past the movement between the test phases has been an arbitrary judgement and the resource usage data is not then accurate for use in estimating further test phases.

Summary

The metrics defined and analysed following the case study are used to construct a software testing estimation and process control model. A diagrammatic representation of the testing process control component of the model and the metrics used within the model is presented. The metrics are applied in four areas, resource estimation, testing phase control, software build release control and effectiveness of testing and development. The mapping from measurements to metrics is provided, so that the data required during a development project has been identified and can be applied from the start.

Appendix D Detail Results

Results table 3C										
Phase=V										
EV CORE	V	I/T	P/R	M	F	total	system	install	data	
Critical	2	1				3				
High	5					5				
Medium						0				
Low						0				
Total	7	1	0	0	0	8	0	0	0	
Phase= I/T										
EV CORE	V	I/T	P/R	M	F	total	system	install	data	
Critical		1				1				
High		1				1				
Medium	1	4	2			7				
Low		2	1			3				
Total	1	8	3	0	0	12	0	0	0	
Phase=P/R										
EV CORE	V	I/T	P/R	M	F	total	system	install	data	
Critical	3	1	12	3		19				
High	5	3	23	3		34				
Medium	2		19	4	1	26				
Low			4			4				
Total	10	4	58	10	1	83	0	0	0	
Phase=M										
EV CORE	V	I/T	P/R	M	F	total	system	install	data	
Critical			6	4		10				
High			14	2	3	19				
Medium			8		4	12				
Low					3	3				
Total	0	0	28	6	10	44	0	0	0	
Phase=F										
EV CORE	V	I/T	P/R	M	F	total	system	install	data	
Critical						0				
High						0				
Medium						0				
Low						0				
Total	0	0	0	0	0	0	0	0	0	
Efficiency										
EV CORE	V	I/T	P/R	M						
EV	7	9	61	16						
post EV	11	4	28	0						
total	18	13	89	16						
Totals										
EV CORE	V	I/T	P/R	M	F	total				
Critical	5	3	18	7	0	33				
High	10	4	37	5	3	59				
Medium	3	4	29	4	5	45				
Low	0	2	5	0	3	10				
Total	18	13	89	16	11	147				

	Results table 3C%							
	Phase=V							
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	25%	13%	0%	0%	0%	38%	2%
	High	63%	0%	0%	0%	0%	63%	3%
	Medium	0%	0%	0%	0%	0%	0%	0%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	88%	13%	0%	0%	0%	1	5%
	Phase= I/	8%	92%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	8%	0%	0%	0%	8%	1%
	High	0%	8%	0%	0%	0%	8%	1%
	Medium	8%	33%	17%	0%	0%	58%	5%
	Low	0%	17%	8%	0%	0%	25%	2%
	Total	8%	67%	25%	0%	0%	1	8%
	Phase=P/R		17%	83%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	4%	1%	14%	4%	0%	23%	13%
	High	6%	4%	28%	4%	0%	41%	23%
	Medium	2%	0%	23%	5%	1%	31%	18%
	Low	0%	0%	5%	0%	0%	5%	3%
	Total	12%	5%	70%	12%	1%	1	56%
	Phase=M			64%	36%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	14%	9%	0%	23%	7%
	High	0%	0%	32%	5%	7%	43%	13%
	Medium	0%	0%	18%	0%	9%	27%	8%
	Low	0%	0%	0%	0%	7%	7%	2%
	Total	0%	0%	64%	14%	23%	1	30%
	Phase=F				0%	0%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical						0%	
	High						0%	
	Medium						0%	
	Low						0%	
	Total						0	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	39%	69%	69%	100%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	3%	2%	12%	5%	0%	22%	
	High	7%	3%	25%	3%	2%	40%	
	Medium	2%	3%	20%	3%	3%	31%	
	Low	0%	1%	3%	0%	2%	7%	
	Total	12%	9%	61%	11%	7%	1	

			Results table C1						
Phase=V									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase= I/T									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase=P/R									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	1	2	2			5			
High	4	2	12	1		19			
Medium	6		13			19			
Low			1	1		2			
Total	11	4	28	2	0	45	0	0	0
Phase=M									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical			5	1	1	7			
High			9	1		10			
Medium			4	3	1	8			
Low						0			
Total	0	0	18	5	2	25	0	0	0
Phase=F									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Efficiency									
EV CORE	V	I/T	P/R	M					
EV	0	0	28	7					
post EV	11	4	18	0					
total	11	4	46	7					
Totals									
EV CORE	V	I/T	P/R	M	F	total			
Critical	1	2	7	1	1	12			
High	4	2	21	2	0	29			
Medium	6	0	17	3	1	27			
Low	0	0	1	1	0	2			
Total	11	4	46	7	2	70			

	Results table C1%							
	Phase=V							
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase= I/	0%	0%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase=P/R		33%	67%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	2%	4%	4%	0%	0%	11%	7%
	High	9%	4%	27%	2%	0%	42%	27%
	Medium	13%	0%	29%	0%	0%	42%	27%
	Low	0%	0%	2%	2%	0%	4%	3%
	Total	24%	9%	62%	4%	0%	1	64%
	Phase=M			72%	28%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	20%	4%	4%	28%	10%
	High	0%	0%	36%	4%	0%	40%	14%
	Medium	0%	0%	16%	12%	4%	32%	11%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	0%	0%	72%	20%	8%	1	36%
	Phase=F				0%	0%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical						0%	
	High						0%	
	Medium						0%	
	Low						0%	
	Total						0	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	0%	0%	61%	100%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	1%	3%	10%	1%	1%	17%	
	High	6%	3%	30%	3%	0%	41%	
	Medium	9%	0%	24%	4%	1%	39%	
	Low	0%	0%	1%	1%	0%	3%	
	Total	16%	6%	66%	10%	3%	1	

Results table C2									
Phase=V									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase= I/T									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High			1			1			
Medium						0			
Low			1			1			
Total	0	0	2	0	0	2	0	0	0
Phase=P/R									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	1					1			
High	12	4	11	5		32			
Medium	6		6	2		14			
Low	16		3	2		21			
Total	35	4	20	9	0	68	0	0	0
Phase=M									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase=F									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Efficiency									
EV CORE	V	I/T	P/R	M					
EV	0	0	22	9					
post EV	35	4	0	0					
total	35	4	22	9					
Totals									
EV CORE	V	I/T	P/R	M	F	total			
Critical	1	0	0	0	0	1			
High	12	4	12	5	0	33			
Medium	6	0	6	2	0	14			
Low	16	0	4	2	0	22			
Total	35	4	22	9	0	70			

	Results table C2%							
	Phase=V							
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase= I/	0%	100%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	0%	0%	0%	0%	0%
	High	0%	0%	50%	0%	0%	50%	1%
	Medium	0%	0%	0%	0%	0%	0%	0%
	Low	0%	0%	50%	0%	0%	50%	1%
	Total	0%	0%	100%	0%	0%	1	3%
	Phase=P/R		57%	43%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	1%	0%	0%	0%	0%	1%	1%
	High	18%	6%	16%	7%	0%	47%	46%
	Medium	9%	0%	9%	3%	0%	21%	20%
	Low	24%	0%	4%	3%	0%	31%	30%
	Total	51%	6%	29%	13%	0%	1	97%
	Phase=M			0%	0%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase=F				0%	0%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical						0%	
	High						0%	
	Medium						0%	
	Low						0%	
	Total						0	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	0%	0%	100%	100%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	1%	0%	0%	0%	0%	1%	
	High	17%	6%	17%	7%	0%	47%	
	Medium	9%	0%	9%	3%	0%	20%	
	Low	23%	0%	6%	3%	0%	31%	
	Total	50%	6%	31%	13%	0%	1	

Results table C345									
Phase=V									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase= I/T									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High	5	4	1			10			
Medium	2	2	1			5			
Low						0			
Total	7	6	2	0	0	15	0	0	0
Phase=P/R									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	1	3	2	3		9			
High	1	1	10	1		13			
Medium	6		7			13			
Low	2	1	1			4			
Total	10	5	20	4	0	39	0	0	0
Phase=M									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	1					1			
High	2		2	1		5			
Medium	2		1	1		4			
Low						0			
Total	5	0	3	2	0	10	0	0	0
Phase=F									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Efficiency									
EV CORE	V	I/T	P/R	M					
EV	0	6	22	6					
post EV	22	5	3	0					
total	22	11	25	6					
Totals									
EV CORE	V	I/T	P/R	M	F	total			
Critical	2	3	2	3	0	10			
High	8	5	13	2	0	28			
Medium	10	2	9	1	0	22			
Low	2	1	1	0	0	4			
Total	22	11	25	6	0	64			

	Results table C345%							
	Phase=V							
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase= I/	47%	53%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	0%	0%	0%	0%	0%
	High	33%	27%	7%	0%	0%	67%	16%
	Medium	13%	13%	7%	0%	0%	33%	8%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	47%	40%	13%	0%	0%	1	23%
	Phase=P/R		38%	62%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	3%	8%	5%	8%	0%	23%	14%
	High	3%	3%	26%	3%	0%	33%	20%
	Medium	15%	0%	18%	0%	0%	33%	20%
	Low	5%	3%	3%	0%	0%	10%	6%
	Total	26%	13%	51%	10%	0%	1	61%
	Phase=M			80%	20%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	10%	0%	0%	0%	0%	10%	2%
	High	20%	0%	20%	10%	0%	50%	8%
	Medium	20%	0%	10%	10%	0%	40%	6%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	50%	0%	30%	20%	0%	1	16%
	Phase=F				0%	0%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical						0%	
	High						0%	
	Medium						0%	
	Low						0%	
	Total						0	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	0%	55%	88%	100%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	3%	5%	3%	5%	0%	16%	
	High	13%	8%	20%	3%	0%	44%	
	Medium	16%	3%	14%	2%	0%	34%	
	Low	3%	2%	2%	0%	0%	6%	
	Total	34%	17%	39%	9%	0%	1	

Results table C6									
Phase=V									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase= I/T									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical		1				1			
High	1	2				3			
Medium	1					1			
Low						0			
Total	2	3	0	0	0	5	0	0	0
Phase=P/R									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	2	1	4	1		8			
High	10	4	14	4		32			
Medium	3		7	6		16			
Low	1		2	1		4			
Total	16	5	27	12	0	60	0	0	0
Phase=M									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High			1	5		6			
Medium			1			1			
Low						0			
Total	0	0	2	5	0	7	0	0	0
Phase=F									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Efficiency									
EV CORE	V	I/T	P/R	M					
EV	0	3	27	17					
post EV	18	5	2	0					
total	18	8	29	17					
Totals									
EV CORE	V	I/T	P/R	M	F	total			
Critical	2	2	4	1	0	9			
High	11	6	15	9	0	41			
Medium	4	0	8	6	0	18			
Low	1	0	2	1	0	4			
Total	18	8	29	17	0	72			

	Results table C6%							
	Phase=V							
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase= I/	40%	60%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	20%	0%	0%	0%	20%	1%
	High	20%	40%	0%	0%	0%	60%	4%
	Medium	20%	0%	0%	0%	0%	20%	1%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	40%	60%	0%	0%	0%	1	7%
	Phase=P/R		35%	65%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	3%	2%	7%	2%	0%	13%	11%
	High	17%	7%	23%	7%	0%	53%	44%
	Medium	5%	0%	12%	10%	0%	27%	22%
	Low	2%	0%	3%	2%	0%	7%	6%
	Total	27%	8%	45%	20%	0%	1	83%
	Phase=M			29%	71%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	0%	0%	0%	0%	0%
	High	0%	0%	14%	71%	0%	86%	8%
	Medium	0%	0%	14%	0%	0%	14%	1%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	0%	0%	29%	71%	0%	1	10%
	Phase=F				0%	0%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical						0%	
	High						0%	
	Medium						0%	
	Low						0%	
	Total						0	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	0%	38%	93%	100%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	3%	3%	6%	1%	0%	13%	
	High	15%	8%	21%	13%	0%	57%	
	Medium	6%	0%	11%	8%	0%	25%	
	Low	1%	0%	3%	1%	0%	6%	
	Total	25%	11%	40%	24%	0%	1	

Results table allC									
Phase=V									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	2	1	0	0	0	3			
High	5	0	0	0	0	5			
Medium	0	0	0	0	0	0			
Low	0	0	0	0	0	0			
Total	7	1	0	0	0	8	0	0	0
Phase= I/T									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	0	2	0	0	0	2			
High	6	7	2	0	0	15			
Medium	4	6	3	0	0	13	1		
Low	0	2	2	0	0	4	2		
Total	10	17	7	0	0	34	3	0	0
Phase=P/R									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	8	7	20	7	0	42	50		7
High	32	14	70	14	0	130	80		66
Medium	23	0	52	12	1	88	30		47
Low	19	1	11	4	0	35	12		19
Total	82	22	153	37	1	295	172	0	139
Phase=M									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	1	0	11	5	1	18	1		
High	2	0	26	9	3	40	6		
Medium	2	0	14	4	5	25	3		
Low	0	0	0	0	3	3			
Total	5	0	51	18	12	86	10	0	0
Phase=F									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	2	4	8	4	7	25	9		1
High	5	2	21	7	19	54	15		1
Medium	1	1	11	5	27	45	15		2
Low	2	3	4	0	12	21	1		1
Total	10	10	44	16	65	145	40	0	5
Efficiency									
EV CORE	V	I/T	P/R	M					
EV	7	18	160	55					
post EV	107	32	95	16					
total	114	50	255	71					
Totals									
EV CORE	V	I/T	P/R	M	F	total			
Critical	11	10	31	12	1	65			
High	45	21	98	23	3	190			
Medium	29	6	69	16	6	126			
Low	19	3	13	4	3	42			
Total	104	40	211	55	13	423			

	Results table allC%						
	Phase=V						
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	25%	13%	0%	0%	0%	38%	1%
	High	63%	0%	0%	0%	0%	63%	1%
	Medium	0%	0%	0%	0%	0%	0%	0%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	88%	13%	0%	0%	0%	1	2%
	Phase= I/	29%	71%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	6%	0%	0%	0%	6%	0%
	High	18%	21%	6%	0%	0%	44%	4%
	Medium	12%	18%	9%	0%	0%	38%	3%
	Low	0%	6%	6%	0%	0%	12%	1%
	Total	29%	50%	21%	0%	0%	1	8%
	Phase=P/R		35%	65%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	3%	2%	7%	2%	0%	14%	10%
	High	11%	5%	24%	5%	0%	44%	31%
	Medium	8%	0%	18%	4%	0%	30%	21%
	Low	6%	0%	4%	1%	0%	12%	8%
	Total	28%	7%	52%	13%	0%	1	70%
	Phase=M			65%	35%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	1%	0%	13%	6%	1%	21%	4%
	High	2%	0%	30%	10%	3%	47%	9%
	Medium	2%	0%	16%	5%	6%	29%	6%
	Low	0%	0%	0%	0%	3%	3%	1%
	Total	6%	0%	59%	21%	14%	1	20%
	Phase=F				55%	45%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical	1%	3%	6%	3%	5%	17%	
	High	3%	1%	14%	5%	13%	37%	
	Medium	1%	1%	8%	3%	19%	31%	
	Low	1%	2%	3%	0%	8%	14%	
	Total	7%	7%	30%	11%	45%	1	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	6%	36%	63%	77%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	3%	2%	7%	3%	0%	15%	
	High	11%	5%	23%	5%	1%	45%	
	Medium	7%	1%	16%	4%	1%	30%	
	Low	4%	1%	3%	1%	1%	10%	
	Total	25%	9%	50%	13%	3%	1	

Results table D									
Phase=V									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase= I/T									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High	2		2			4			
Medium	2		1			3			
Low	1	2	1	1		5			
Total	5	2	4	1	0	12	0	0	0
Phase=P/R									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical	1		3			4			
High			4	1		5			
Medium	1	1	4			6			
Low			1			1			
Total	2	1	12	1	0	16	0	0	0
Phase=M									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Phase=F									
EV CORE	V	I/T	P/R	M	F	total	system	install	data
Critical						0			
High						0			
Medium						0			
Low						0			
Total	0	0	0	0	0	0	0	0	0
Efficiency									
EV CORE	V	I/T	P/R	M					
EV	0	2	16	1					
post EV	7	1	0	0					
total	7	3	16	1					
Totals									
EV CORE	V	I/T	P/R	M	F	total			
Critical	1	0	3	0	0	4			
High	2	0	6	1	0	9			
Medium	3	1	5	0	0	9			
Low	1	2	2	1	0	6			
Total	7	3	16	2	0	28			

Results table D%							
Phase=V							
EV CORE	V	I/T	P/R	M	F	Phase	Total
Critical						0%	0%
High						0%	0%
Medium						0%	0%
Low						0%	0%
Total						0	0%
Phase= I/	42%	58%					
EV CORE	V	I/T	P/R	M	F	Phase	Total
Critical	0%	0%	0%	0%	0%	0%	0%
High	17%	0%	17%	0%	0%	33%	14%
Medium	17%	0%	8%	0%	0%	25%	11%
Low	8%	17%	8%	8%	0%	42%	18%
Total	42%	17%	33%	8%	0%	1	43%
Phase=P/R		19%	81%				
EV CORE	V	I/T	P/R	M	F	Phase	Total
Critical	6%	0%	19%	0%	0%	25%	14%
High	0%	0%	25%	6%	0%	31%	18%
Medium	6%	6%	25%	0%	0%	38%	21%
Low	0%	0%	6%	0%	0%	6%	4%
Total	13%	6%	75%	6%	0%	1	57%
Phase=M			0%	0%			
EV CORE	V	I/T	P/R	M	F	Phase	Total
Critical						0%	0%
High						0%	0%
Medium						0%	0%
Low						0%	0%
Total						0	0%
Phase=F				0%	0%		
EV CORE	V	I/T	P/R	M	F	Phase	
Critical						0%	
High						0%	
Medium						0%	
Low						0%	
Total						0	
Totals							
EV CORE	V	I/T	P/R	M			
efficiency	0%	67%	100%				
Totals							
EV CORE	V	I/T	P/R	M	F	Total	
Critical	4%	0%	11%	0%	0%	14%	
High	7%	0%	21%	4%	0%	32%	
Medium	11%	4%	18%	0%	0%	32%	
Low	4%	7%	7%	4%	0%	21%	
Total	25%	11%	57%	7%	0%	1	

			Results table 6							
	Phase=V									
	EV CORE	V	I/T	P/R	M	F	total	system	install	data
	Critical						0			
	High						0			
	Medium						0			
	Low						0			
	Total	0	0	0	0	0	0	0	0	0
	Phase= I/T									
	EV CORE	V	I/T	P/R	M	F	total	system	install	data
	Critical			1			1			
	High		3	1			4			
	Medium	2	1	4			7	1		
	Low						0			
	Total	2	4	6	0	0	12	1	0	0
	Phase=P/R									
	EV CORE	V	I/T	P/R	M	F	total	system	install	data
	Critical			1			1		2	
	High		1	7			8	1	2	
	Medium						0			
	Low						0			
	Total	0	1	8	0	0	9	1	4	0
	Phase=M									
	EV CORE	V	I/T	P/R	M	F	total	system	install	data
	Critical						0			
	High				3		3			
	Medium						0	2		
	Low						0			
	Total	0	0	0	3	0	3	2	0	0
	Phase=F									
	EV CORE	V	I/T	P/R	M	F	total	system	install	data
	Critical	1		6		7	14	1	2	
	High			1		2	3	1		
	Medium	1		2		2	5	2	3	
	Low						0			
	Total	2	0	9	0	11	22	4	5	0
	Efficiency									
	EV CORE	V	I/T	P/R	M					
	EV	0	4	14	3					
	post EV	4	1	9	0					
	total	4	5	23	3					
	Totals									
	EV CORE	V	I/T	P/R	M	F	total			
	Critical	0	0	2	0	0	2			
	High	0	4	8	3	0	15			
	Medium	2	1	4	0	0	7			
	Low	0	0	0	0	0	0			
	Total	2	5	14	3	0	24			

	Results table 6%							
	Phase=V							
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical						0%	0%
	High						0%	0%
	Medium						0%	0%
	Low						0%	0%
	Total						0	0%
	Phase= I/	17%	83%					
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	8%	0%	0%	8%	4%
	High	0%	25%	8%	0%	0%	33%	17%
	Medium	17%	8%	33%	0%	0%	58%	29%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	17%	33%	50%	0%	0%	1	50%
	Phase=P/R		11%	89%				
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	11%	0%	0%	11%	4%
	High	0%	11%	78%	0%	0%	89%	33%
	Medium	0%	0%	0%	0%	0%	0%	0%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	0%	11%	89%	0%	0%	1	38%
	Phase=M			0%	100%			
	EV CORE	V	I/T	P/R	M	F	Phase	Total
	Critical	0%	0%	0%	0%	0%	0%	0%
	High	0%	0%	0%	100%	0%	100%	13%
	Medium	0%	0%	0%	0%	0%	0%	0%
	Low	0%	0%	0%	0%	0%	0%	0%
	Total	0%	0%	0%	100%	0%	1	13%
	Phase=F				50%	50%		
	EV CORE	V	I/T	P/R	M	F	Phase	
	Critical	5%	0%	27%	0%	32%	64%	
	High	0%	0%	5%	0%	9%	14%	
	Medium	5%	0%	9%	0%	9%	23%	
	Low	0%	0%	0%	0%	0%	0%	
	Total	9%	0%	41%	0%	50%	1	
	Totals							
	EV CORE	V	I/T	P/R	M			
	efficiency	0%	80%	61%	100%			
	Totals							
	EV CORE	V	I/T	P/R	M	F	Total	
	Critical	0%	0%	8%	0%	0%	8%	
	High	0%	17%	33%	13%	0%	63%	
	Medium	8%	4%	17%	0%	0%	29%	
	Low	0%	0%	0%	0%	0%	0%	
	Total	8%	21%	58%	13%	0%	1	

Appendix E Bibliography

1 Norris M (1996) Understanding Network Technology - concepts, terms and trends. Artech House Inc.

2 Norris M, Rigby P. (1995) Software Engineering Explained. John Wiley & Sons.

3 Tice G D (1987) SAL Software testing. State of the art Ltd. Southampton Row, London.

4 Graham D R. (1991) Software testing tools: A new classification scheme. The Journal of Software Testing, Verification and Reliability. Vol 1,3. pg 17 - 34.

5 Leintz B P, Swanson E B. (1980) Software maintenance management. Addison-Wesley.

6 Ince D. (1988) Software Development. Fashioning the Baroque. Oxford Science Publications.

7 Pressman R. (1994) Software Engineering. A Practitiioner's Approach. McGraw-Hill Book Company Europe.

8 Sommerville I. (1982) Software Engineering. Addison-Wisley Publishing Company.

9 Jones C B. (1980) Software Development - A rigorous approach. Prentice Hall International.

10 Carre B. (1989) Reliable programming in standard languages. High-intergrity Software, edited by Sennett C. Pitman Publishing. pg 102 - 121.

11 Smith P, Bleech N. (1989) Practicle experience with a formal verification system. High-intergrity Software, edited by Sennett C. Pitman Publishing. pg 69 - 101.

12 Pryke W M, Whiting P J, Pang S, Kaur R, Wilmot C. (1997) Systems integration throughout the early life cycle. BT Technology Journal Vol.15 No.3 pg 9 - 18.

13 Osterweil L, Clarke L A (1989) Directions for U.S. research and development efforts on software testing and analysis. U.S. Office of Naval Reaserach and Office of Naval Technology.

14 Boehm B W. (1975) The high cost of software. Practical Strategies for developing large software systems. Ed Horowitz E. Addison-Wesley.

15 Howden W E (1987) Functional Program Testing & Analysis. McGraw-Hill International.

16 Boehm B W. (1981) Software engineering economics. Prentice-Hall.

17 Daily K. (1992) Quality Management for Software. NCC Blackwell.

18 Myers G. (1976) The art of software testing. John Wiley & Sons.

19 Ould M, Unwin C. (1986) Testing in software development. British Computer Society.

20 Abbott J. (1986) Software testing techniques. NCC publications.

21 Deutsch M. (1982) Software verification and validation. Prentice-Hall.

22 ANSI/IEEE 829std (1983) Software test documentation. John Wiley & sons.

23 DeMarco T. (1982) Controlling software projects. Yourdon Press.

24 Roper M. (1994) Software Testing. McGraw-Hill Book Company.

25 ANSI/IEEE 729std (1983) Glossary of software engineering terminology. IEEE.

26 Conte S, Dunsmore H, Shen V. (1986) Software Engineering Metrics and Models. Benjamin Cummings Publishing Co.

27 Gelperin D, Hetzel B (1988) The growth of software testing. Communications of the ACM Vol 31, 6. pg 687 - 695.

28 Duncan I (1993) Strong Mutation Testing Strategies. PhD thesis, University of Durham.

29 Jones R (1997) Model environments for integration and support. BT Tecnology Journal Vol 15 No.3 pg 100-115.

30 Wilmot C (1992) Analytical techniques for verification, validation and testing. BT Tecnology Journal Vol 10 No.2 pg 46-53.

31 West S, Norris M (1997) Media Engineering - A guide to developing Information Products. John Wiley & Sons.

32 Royer T C (1993) Software testing management. Life on the critical path. Prentice Hall.

33 DeMillo R A, McCracken W M, Martin R J, Passafiume J H (1987) Software testing and Evaluation. Benjamin/Cummings Publishing Company.

34 Hetzel W (1988) The Complete Guide to Software Testing - second edition. QED Information Sciences.

35 Hennell M A, Riddell I J (1989) Program analysis and systematic testing. High-intergrity Software, edited by Sennett C. Pitman Publishing. pg 159 - 175

36 Petschenik N H (1985) Practical priorities in system testing. IEEE Software. Vol 2/5 pg 18 - 24.

37 McDermid J (1989) Assurance in high-itegrity software. High-intergrity Software, edited by Sennett C. Pitman Publishing. pg 226 - 273.

38 Musa J D (1980) A theory of software reliability and its application. Tutorial on models and metrics for software management and engineering. IEEE computer society press. pg 157-180.

39 Atkins J, Norris M (1995) Total Area Networking. John Wiley & Sons.

40 Dobson J (1989) Modelling real-world issues for dependable software. High-intergrity Software, edited by Sennett C. Pitman Publishing. pg 274 - 316.

41 IEEE (1991) IEEE Computer Society's Task Force on Professional Computing Tools. A standard reference model for computing system tool interconnections. Draft. P1175/D11. IEEE Computer Society.

42 Roper M. (1990) The automatic generation of test cases. Proceedings of the seminar series on new directions in software development. Testing large systems. March 1990. Wolverhampton Polytechnic/BCS. pg 43 - 55.

43 Roper M, Bin Ab Rahim A. (1993) Software testing using analysis and design. The Journal of Software Testing, Verification and Reliability. Vol 3,4. pg 165 - 179.

44 Sennett C T (1989) Contractual specification of reliable software. High-intergrity Software, edited by Sennett C. Pitman Publishing. pg 317 - 354.

45 Buckle J. (1987) Managing software products. J & J Buckle computer Consultancy

46 Grady R.B, Caswell D.L. (1986) Software metrics: establishing a company-wide program. Prentice-Hall Inc.

47 Fenton N E, Pfleeger S L (1996) Software Metrics - A rigorous and practical approach. International Thompson Computer Press.

48 Fenton N E, Kitchenham B (1991) Validating Software Measures. The Journal of Software Testing, Verification and Reliability. Vol 1,2. pg 27 - 42.

49 Basili V.R. (1980) Resource Models. Tutorial on models and metrics for software management and engineering. IEEE computer society press. pg 4-9.

50 Ferdinand A E (1993) Systems, software, and quality engineering: applying defect behavior theory to programming. Van Nostrand Reeinhold.

51 British Standards (1979) BS5750 Quality Systems. Britisn Standards Institution. Milton Keynes.

52 ISO (1991) International standard ISO9000-3 Quality management and quality assurance standards part 3.

53 Rout T P. (1995) SPICE: A framework for software process assessment. Software Process Improvement and Practice. pg 57-66.

54 TickIT Guide. (1990) Guide to software quality management system construction and certification. TickIT Project Office. 68 Newman St London.

55 Gilb T. (1988) Principles of software engineering management. Addisn-Wesley publishing company.

56 Denhand K (1990) Software Metrics from a User's Perspective.The Journal of Systems and Sftware.Vol 13 No.2. Elsevier Science Publishing Co. pg 111 - 115.

57 Brooks P.B. (1982) The mythical man-month Essays on software engineering. Addison-Wesley publishing company.

58 Watts R.A. (1987) Measuring software quality. NCC publications.

59 Boehm B.W, Brown J.R, Lipow M. (1976) Quantitative evaluation of software quality. Proceedings IEEE Second International Conference on Software Engineering. pg 592 - 605.

60 Siefert D.M. (1989) Implementing software reliability measures. NCR journal Vol3 No.1. pg 24 - 34.

61 ANSI/IEEE 982.1std (1988) Standard dictionary of measures to produce reliable software. IEEE.

62 ANSI/IEEE 982.2std (1988) Guide for the use of IEEE standard dictionary of measures to produce reliable software. IEEE.

63 Schneidewind N, Hoffmann H.(1979) An experiment in software error data collection and analysis. IEEE transactions on software engineering. SE5,3. pg 276 - 286.

64 Littlewood B. (1989) Predicting and measuring software reliability. City University (seminar 16/12/89, BT RT31).

65 USAF Air force cost centre. (1987) A descriptive evaluation of software sizing models. Data & analysis centre for software. IIT Research Institute.

66 Helmer O (1966) Social Technology. Basic Books. New York.

67 Albrecht A, Gaffney J. (1983) Software function, source lines of code, and development effort prediction: a software science validation. IEEE Transactions on software engineering SE9,6. pg 639 - 647.

68 Halstead M. (1977) Elements of software science. Elsevier North-Holland Incorporate.

69 Jones T C. (1978) Measuring programming quality and productivity. IBM System Journal 17,1. pg 39 - 63.

70 Thebaut S M. (1983) The saturation effect in large scale software development: its impact and control Ph.D Thesis. Department of Computer Science, Purdue University.

71 Norden P. (1970) Using tools for project management. Management of production. Penguin.

72 Putnam L H. (1978) A general empirical solution to the macro software sizing and estimating problem IEEE Transactions on software engineering SE4,4. pg 345 - 361.

73 Putnam L H. (1980) SLIM system description Quantitative Software Management Inc. McLean VA.

74 Parr F N. (1980) An alternative to the Rayleigh curve model for software development effort. IEEE Transactions on software engineering SE6,3. pg 291 - 296.

75 Tausworthe R C. (1981) Deep space network software cost estimation model. Jet propulsion laboratory, publication 81-7. Pasadena CA.

76 Carriere W M, Thibodeau R. (1979) Development of a logistics software cost estimation technique for military sales. General Research Corporation.

77 Walston C E, Felix C P. (1977) A method of programming measurement and estimation IBM system journal. Vol16,1. pg 54 - 73.

78 McCabe T J. (1976) A complexity measure. IEEE Transactions on software engineering. SE 2,4. pg 308 - 320.

79 McCabe T J. (1982) Structured testing: A testing methodology using the McCabe complexity measure US Department of Commerce, National bureau of standards.

80 McCabe T J, Schulmeyer G G. (1982) System testing aided by structured analysis (a practicle experience). IEEE COMPSAC proceedings. pg 51 - 49.

81 Woodward M R, Hennell MA, Hedley D. (1979) A measure of control flow complexity in program text. IEEE Transactions on software engineering SE5,1. pg 45 - 50.

82 Henry S, Kafura D, Harris K. (1981) On the relationships among three software metrics. Performance evaluation review Vol 10,1. pg 125 - 132.

83 Sunohara T, Takano A, Uehara K, Ohkawa T. (1981) Program complexity measure for software development management. Proceedings of the fifth IEEE International Software Engineering Conference. pg 118 - 124.

84 Curtis B, Sheppard S, Milliman P, Borst M A, Love T. (1979) Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on software engineering SE5,2. pg 96 - 104.

85 Walsh T J. (1979) A software reliability study using a complexity measure. AFIPS conference proceedings Vol 48. National Computer Centre, New York. pg 761 - 768.

86 Ottenstein L M. (1979) Quantitative estimates of debugging requirements. IEEE Transactions on software engineering SE5,5. pg 504 - 514.

87 The STARTS guide 1987. Vol 1 DTI.

88 Harrison W. (1988) Using software metrics to allocate testing resources. Journal of Management Information Systems Vol 4,4. pg 93 - 105.

89 Dunn R H (1984) Software defect removal. McGraw-Hill Inc.

90 Beizer B (1984) Software systems testing and quality assurance. Van Nostrand Reinhold Inc.

91 Xie M (1993) Software reliability models - a selected annotated bibliography. The Journal of Software Testing, Verification and Reliability. Vol 3,1. pg 3 - 28.

92 Dale C. (1987) Data requirements for software reliability prediction. Software reliability achievement and assessment. Blackwell scientific.

93 Akijama F. (1971) An example of software system debugging. Information processing 71. pg 353 - 379.

94 Motley R W, Brooks W D. (1977) Statistical prediction of programming errors. Rome air development centre.

95 Lipow M. (1982) Number of faults per line of code. IEEE Transactions on software engineering SE8,4. pg 437 - 439.

96 Gaffney J E. (1984) Estimating the number of faults in code. IEEE Transactions on software engineering SE10,4. pg 459 - 464.

97 Potier D, Albin J, Ferreol R, Bilodean A. (1982) Experiments with computer software complexity and reliability. Proceedings of the sixth international conference on software engineering. pg 94 - 103.

98 Shen V, Yu T, Thebaut S, Paulsen L. (1985) Identifying error prone software - an empirical study. IEEE Transactions on software engineering SE11,4. pg 317 - 324.

99 Basili V, Perricone B. (1984) Software errors and complexity: an empirical investigation. Communications of the ACM 27. pg 42 - 52.

100 Musa J. (1980) Software reliability measurement. The journal of systems and software. Vol13,2. pg 223 - 241.

101 Musa J D, Ackerman A F. (1989) Quantifying software validation: when to stop testing. IEEE software. pg 19 - 27.

102 Jelinski Z, Moranda P. (1972) Software reliability research. Statistical computer performance evaluation. edited Freiberger W. New York Academic press. pg 465 - 484.

103 Shooman M L. (1983) Software engineering. McGraw-Hill.

104 Schick G J, Wolverton R W. (1978) An analysis of competing software reliability models. IEEE Transactions on software engineering SE4,2. pg 104 - 120.

105 Ramamoorthy C V, Bastani F B. (1982) Software reliability - status and perspectives. IEEE Transactions on software engineering SE8,4. pg 354 - 371.

106 Littlewood B, Verrall J. (1973) A Bayesian reliability growth model for computer software. Applied statistics 22.

107 Remus H, Zilles S. (1979) Prediction and management of program quality. Proceedings of the fourth international conference on software engineering. pg 341 - 350.

108 Jones T C , Turk C W. (1976) Mathematical quality model of the programming process. IBM technical bulletin Vol 19,4.

109 Littlewood B. (1987) How good are software reliability predictions? Software reliability - achievement and assessment. Blackwell scientific.

110 Norris M, Rigby P, Payne M (1993) The healthy software project: a guide to successful development and management. John Wiley & Sons Ltd.

111 Sroka J V, Gosling C A (1987) Using quantitive activity models in project management: a case study. Measurement for software control and assurance, edited Kitchenham B A, Littlewood B. Elsevier Applied Science. pg 181 - 199.

112 Davey S, Huxford D, Liddiard J, Powley M, Smith A (1993) Metrics collection in code and unit test as part of continuous quality improvement. The Journal of Software Testing, Verification and Reliability. Vol 3,4. pg 125 - 148.

113 Hon III (1990) Assuring software quality through measurements: a buyer's perspective. The journal of systems and software. Vol13,2. pg 117 - 130.

114 Rombach D H. (1990) Quality improvements at NASA's software engineering LAB. Presentation to BTRL, Ipswich 11/5/90.

115 Moller K (1991) Increasing software quality by objectives and residual fault prognosis. Software Quality and Reliability - tools and methods, edited Ince D. UNICOM/Chapman & Hall. pg102 - 112