

308

/ A COMPREHENSIVE SOFTWARE TEST STRATEGY /

by

STEPHEN LOUIS KAHLE

B. S., University of Missouri at Columbia, 1972

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

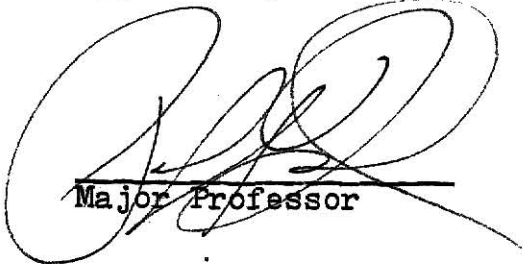
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

Approved by:



Major Professor

LD
2668
R4
1983
K33
C.2

A11202 244782

Acknowledgment

I would like to acknowledge the support of NCR Corporation, which subsidized my continued education and arranged this Masters Degree program with Kansas State University. I wish to also express my thanks to my major professor, David Gustafson, to whose sound advice I have done my best to adhere, and to all others at KSU who have been a part of the educational process.

A note of thanks goes to my wife, who has repetitively typed this document, to the rest of my family for their cooperation, and to my grandmother, whose interest in my progress, helped to keep me motivated.

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

Table of Contents

	<u>Page</u>
Chapter 1	
Introduction	1
1.1 Need for Software Reliability.	2
1.2 Factors Adversely Affecting Reliability.	5
1.3 Test Strategy Goals.	7
1.4 Test Characteristics	12
Chapter 2	
Current Testing Approaches	18
2.1 Test Methods	18
2.2 Test Coverage.	28
2.3 Type of Testing.	32
2.4 Test Ordering.	33
2.5 Summary.	35
Chapter 3	
Product Development Issues	37
3.1 Development Life Cycle	37
3.2 Software Structure	41
3.3 Testing Levels	42
3.4 Test Planning.	45
3.5 Responsibility	48
Chapter 4	
A Test Strategy.	52
4.1 Specification Phase Test.	59
4.2 Design Phase Test.	63
4.3 Coding/Debugging and Integration Phase Test.	69
4.4 Delivery Phase Test.	77
4.5 Maintenance Phase Test	78
4.6 Test Responsibility.	78
4.7 Summary.	80
Chapter 5	
Implementation of Testing.	81
5.1 Test Definition.	81
5.2 Test Environment Issues.	83
5.3 Conclusion	86
Chapter 6	
Concluding Remarks	87

Table of Contents (Cont.)

	<u>Page</u>
Appendices	
Appendix A	
Glossary	89
Appendix B	
Comprehensive Test Strategy.	94
B.1 Specification Phase.	95
B.2 Design Phase	96
B.3 Coding/Debugging and Integration Phase	99
B.4 Delivery Phase	102
B.5 Maintenance Phase.	103
Appendix C	
Implementation Documentation	106
C.1 Test Plan.	106
C.2 Test Specification	107
C.3 Test Log	107
C.4 Test Incident Report	108
C.5 Test Incident Resolution Report.	108
C.6 Test Summary Report.	109
Bibliography.	110

List of Illustrations

<u>Figure</u>	<u>Title</u>	<u>Page</u>
1-1	Test Strategy Goals	7
1-2	Testing Characteristics	13
1-3	Goals vs. Characteristics	16
1-4	Comprehensive Test Strategy Contents. . . .	17
2-1	Testing Methods	19
2-2	Testing Orders.	33
3-1	Software Development Stages	38
3-2	Testing Levels.	42
4-1	Example Software Product Structure.	56
4-2	Example Specification	57
4-3	Example Development Organization.	58
4-4	Software Component Specification Test . . .	65
5-1	Implementation Documentation.	82

Chapter 1

Introduction

This paper presents a comprehensive strategy for testing software throughout the development cycle which, it is hoped, will increase the reliability of software. This strategy, detailed in Appendix B, is a systematic plan for the application of testing methods and techniques to a software project. It is based on the increasing importance of reliability in software. Techniques which best satisfy reliability principles are selectively integrated into the strategy which is comprehensive in two ways. In the small sense, it encompasses all phases of the product development cycle. In a larger sense, it should be almost universally applicable to any development environment on any type of software product.

Software reliability encompasses two traits: fidelity and robustness. Fidelity is how well the product adheres to the behavior expected of it (NCR, 1982). This means that software must produce expected results for inputs and do so within specified performance constraints (for example, within system resource limits with a maximum response time). Robustness is complementary to fidelity and is the probability that the product will not misbehave or fail. This means the product must be capable of

detecting and handling invalid conditions. It may be required to execute under harsh environmental conditions detrimental to computing and, in the extreme case, be fault tolerant or able to detect its own mistakes and correct and adjust itself (Aviziensis, et al, 1971). Reliability is measured through testing which detects erroneous behavior but does not necessarily guarantee correct behavior. In order to yield the greatest reliability in the product, a well planned and integrated test activity is a necessity, so that the greatest number of errors will be detected.

1.1 Need for Software Reliability

While the reliability of software has always been a desirable goal in product development, it is now becoming absolutely essential. There are a number of reasons for this which arise from both the expansion of computer applications and the economics of product development.

As computers are used for more and more tasks, they are penetrating areas where public safety is affected. They are used in design computation for construction where a miscalculation could produce faulty buildings and endanger the public. Specialized applications are being used to control space flight where the lives of astronauts are dependent on them. Another critical area of application is

nuclear power control (Geiger, et al, 1979), where large numbers of people could be harmed in a failure situation.

Computers are also being used to handle large collections of sensitive data. In many cases, improper handling of this data could jeopardize a business or be very embarrassing to an individual. As an example, consider an incorrect business prediction algorithm which might generate a much larger number of units for production than required and would leave the manufacturer with an excessive inventory. As another example, the unauthorized disclosure of an individual's financial records could cause personal embarrassment. Lastly, if data transfers involving electronic funds transfer fail, at the very least a great deal of inconvenience may result. It can be seen from these examples that it is necessary from the public welfare point of view that software products be reliable.

Other factors which stimulate reliability concerns are the economics of software development. Of obvious concern is the fact that software failures result in user problems which can lead to law suits against the producers. These are costly regardless of their settlement. To protect himself from liability, the software producer must do everything within reason to insure reliability.

The actual value of the software to the user may depend on its quality or fitness for use (NCR, 1981). This means it must satisfy expectations as specified and comply with applicable standards. This is a variation of the reliability definition (adherence to expected behavior and robustness); hence product value and salability are enhanced through the demonstration of reliability. In addition, if the performance of the product can be shown to exceed that of the competition, a business advantage is gained.

The cost of software production is also a consideration which demands product reliability and emphasizes techniques which produce reliability in the development process. Errors (deviations from expected behavior) which are detected and corrected early in the development process are much less costly to fix than those allowed to propagate into later stages (NCR, 1980). In order to keep debugging costs to a minimum, each stage of development should include testing (Myers, 1979). This will help find errors and keep them from multiplying into extensive project rework later on. In addition, as a by-product of ongoing reliability efforts, software should be more understandable and better structured. This will not only help minimize

development costs but also help reduce maintenance and enhancement costs which can exceed development costs (Fairley, 1982). Thus by actively pursuing reliability the producer protects his users and the public, improves his product's image, and minimizes the resources he puts into development.

1.2 Factors Adversely Affecting Reliability

Many factors in software development may be detrimental to reliability and cause the product's behavior to deviate from that which is expected or desired. Among these are lack of standards, insufficient planning, and incomplete definition of requirements (Fairley, 1982). All of these factors contribute to misunderstanding and confusion in establishing an organized development which fulfills the desired objectives. Use of systematic processes which formalize development and reduce ambiguities tend to reduce errors resulting from this gap in understanding and thus may improve reliability.

Possibly the greatest enemies of reliability are, however, software scale and complexity (Fairley, 1982). Scale refers to the volume or size of the software product. Complexity is the intricacy of software, or the deviation from a simple flow and interface structure.

These factors not only make programming more difficult but hinder the entire development process with problems in understanding and remembering complex software. Complexity makes testing and debugging more difficult due to the subtlety and covertness of errors which may occur. Complexity also increases the potential for side effects, such as the introduction of new errors when corrections are made. Furthermore, simple probability predicts more opportunities for errors in larger volumes of code with more interfaces.

While scale and complexity can not be eliminated from software, their impact can be minimized. Systematic development techniques, such as structured programming, tend to keep flow paths as simple as possible (Zelkowitz, 1978). This also tends to keep the software product broken into component pieces of a more manageable size. Using a planned method of testing, which not only detects errors but also checks development techniques, is helpful to insure that scale and complexity are controlled. This not only allows error correction but also should help reduce error occurrence. In the end this contributes to reliability.

1.3 Test Strategy Goals

In order to establish a plan for testing which assures software reliability, some goals for this comprehensive strategy must be established. They are listed in figure 1-1.

TEST STRATEGY GOALS

1. Detect Errors Early
2. Detect All Errors
3. Reveal Complex Structures
4. Encourage Good Structure and Understandability
5. Provide Error Localization
6. Provide Maintenance Capability
7. Provide Positive Visibility to the Product (User Confidence (IEEE, 1981) and Performance)
8. Provide Data to Improve the Development Process
9. Be Compatible with Diverse Development Environments

Figure 1-1

The first goal is early error detection. The earlier in development that an error is detected, the less costly it is to correct, (NCR, 1980; Myers, 1979). This is because in early development stages changes may be made with relatively little back tracking to make the implementation complete. For example, if a system feature is found

missing at the time the product is specified, it need only be added to the specification; whereas, if it is not discovered until coding has begun, the system design may have to be altered affecting other parts of the system, in addition to changing the original specification. Early error detection not only exposes errors at a time when they are most easily fixed but also protects the developer from having errors propagate into later development stages which could cause an extensive rework effort.

The second goal of detecting all errors concerns the thoroughness of testing. To be thorough, a test must show up all possible errors (Goodenough, Gerhart, 1975). Thoroughness means that the test is reliable with respect to exposing errors; that is, if errors are present they are exposed. Without a systematic test method this would require an impractically large set of test cases, which include not only software inputs to the test environment, but also environmental conditions occurring in operation. However, almost reliable testing is a more practical approximation of thoroughness (Howden, 1976). Such testing can be performed using several of the test methods described later. Knowledge about the software's behavior is used to select test cases which cover all classes or groupings of

input types, thus testing all types of operation. The closer testing is to thorough, the less chance exists of errors going undetected.

A good test strategy should be a tool for detecting structure problems and encouraging good software structure and understandability, the third and fourth goals. It must first illustrate where these characteristics are lacking. There may be poorly structured areas which could be restructured when detected, or areas which are necessarily complex and need special attention in later testing. Understandability problems should be corrected as an aid to further development and maintenance. By revealing understandability and structure problems, attention is focused on these aspects of the product, and a feedback path to the developer is established. He is put under peer pressure to keep development as simple as possible. As these are subtle goals, they require a certain amount of subjective analysis and decision making but become tools for encouraging good programming structure and design.

It is insufficient for a test process to merely detect errors; hence the fifth goal of providing error localization (tool for locating and correcting errors) (Myers, 1979). The progression of the tests and methods

being used should help in locating errors. For instance, if a failure occurs immediately after a new portion of software is implemented, that portion may be suspect. Certain test methods also may point to failing software structures. In addition, the nature of the failure should provide some insight as to what the failure may be. As an example, if the software executes but produces an incorrect result, then a computation error may be suspect; whereas, if the program does not terminate, an error in flow may have occurred. At this point some special tools or procedures which will help pinpoint the error for correction may be applicable. The correction must then be tested to insure that it is indeed a correction and has no adverse side effects.

Once a system is completed and placed in service, it is probable that usage problems will arise or enhancements will be desired; hence the need for a maintenance capability in the strategy as a sixth goal. Software maintenance is the implementation and retesting (regression testing) of such changes. The test strategy should guide software maintenance testing and, if possible, supply a means of reusing previous tests so that the redundancy of regenerating test cases may be avoided.

Product testing visibility is the seventh goal of the strategy. Visibility of product reliability efforts is achieved through formal documentation of the test activity. This documentation illustrates measures taken to insure product reliability and may be used as a selling point. Documentation also provides evidence of performance claims about the product, an integral part of competitive business. The strategy should indicate what testing documentation is to be formalized and preserved.

The next goal is to improve the software development process through feedback of data from testing on previous products (Myers, 1979). Data such as the types and numbers of errors detected in various development stages, as well as factors which may have contributed to error occurrence, should be recorded. Information regarding error characteristics is not only helpful in avoiding errors but also in knowing how to test for them. It is also useful to know when the greatest percentage of errors are likely to be introduced, so that the intensiveness of testing may be increased at that point, maximizing early error detection. Furthermore, error quantities or occurrence rates are sometimes used in determining when to cease testing, making information on error quantities essential for such a completion criteria.

Finally, the test strategy must provide for compatibility with varying projects. It should be usable under a variety of conditions, such as, with various levels of language, on different hardware, under varying application environments, and in the absence of automated tools. It, therefore, should not dictate tests but should point out appropriate test methods throughout the development cycle. It is possible that special tools, such as flow path analyzers or test case generators, may be available in some instances and should be used. Tools must not be critical to the success of the strategy, however, as this would limit its use to environments where they are available.

If the nine goals described above are fulfilled, the strategy should represent an approach to testing which is a major asset to the development project. This approach should be capable of increasing software reliability through thorough testing in a wide variety of applications. In addition, it should be an aid to product definition and design as well as programming and maintenance.

1.4 Test Characteristics

A number of test characteristics are required in order to fulfill testing goals. These are listed in figure 1-2.

Some of these characteristics reflect test traits, while others are characteristics of the test strategy.

TESTING CHARACTERISTICS

1. Thorough
2. Flexible
3. Specifiable
4. Understandable
5. Measurable
6. Retainable

Figure 1-2

The first characteristic is thoroughness. This is the measure of how well the test's expected behavior objective of exposing as many software errors as possible is met (in the ideal case, all errors) (Howden, 1975). These errors include both errors in executable and nonexecutable phases of product development. This characteristic is essential to the goals of detecting and localizing all errors early; therefore the test strategy should guide in creating thorough tests. As structure and understandability problems can be considered contributors to error, thoroughness also contributes to the structure related goals.

In order to be of practical help in software development, the tests, created through the strategy, must be flexible. They must be usable under varied circumstances. The strategy must outline what testing is to be done without dictating any specifics of the actual test implementation. This allows the strategy to be applicable throughout a variety of development environments.

The test strategy must be specifiable. The process outlined must be uniformly and clearly described with regard to what each step in testing encompasses and accomplishes. This description must be done in a manner which leaves implementation of the tests to the testing activity (so that the flexibility characteristic is not degraded). Test progression paths and concepts must be well described to insure proper usage under varying circumstances.

Understandability is also required, so that tests implemented through the strategy will indeed meet the strategy goals. The strategy must be logically structured, so that its features and requirements are understood from its description.

Test visibility is provided through documentation which requires that testing be measurable. That is, there must be some manner of quantifying test results and coverage, indicating the correctness of results and the effectiveness

of testing. It is important that the test strategy provide a consistent means of test measurement and documentation at each stage of development so that recorded results will provide evidence of test effectiveness. Evidence of software improvements, gained through testing, should be available through systematic error tracking which records problems encountered and their solutions. Another effect may be the detection of holes in the strategy which must be corrected to improve the overall testing process.

Finally, the tests must be retainable. This means they must be usable with future developments of different projects or in the maintenance mode of programming. The strategy, through its flexibility, should be applicable to a wide range of projects and thus allow some standardization of the tests, the test process, and quantification of test results.

If the testing activity generated from the test strategy has these characteristics, the strategy is capable of fulfilling its goals. Figure 1-3 lists the test strategy goals and characteristics supporting them. The characteristics of "specificable" and "understandable" do not apply directly to the goals; rather they support the overall strategy by making its correct usage clear.

GOALS vs. CHARACTERISTICS

<u>Goals</u>	<u>Major Supporting Characteristics</u>
1. Detect Errors Early	Thorough, Flexible
2. Detect All Errors	Thorough
3. Reveal Complex Structures	Thorough
4. Encourage Good Structure and Understandability	Thorough
5. Provide Error Localization	Thorough
6. Provide Maintenance Capability	Retainable
7. Provide Positive Visibility to the Product	Measurable
8. Provide Data to Improve the Development Process	Measurable
9. Be Compatible with Diverse Development Environments	Flexible

Figure 1-3

Figure 1-4 presents an overview of the testing strategy. This illustration provides quick reference to the location of specific testing guides. Additional information describing these guides is located in chapter 4. This strategy is divided into sections covering each phase of the software development cycle. Within each phase, subsections describe what test methods are appropriate, at what level tests occur, the order of test progression, the type of testing, the documentation produced, where test responsibility lies, and what the overall accomplishment of testing in that phase is.

COMPREHENSIVE TEST STRATEGY CONTENTS

<u>Contents</u>	<u>Page</u>
Definition	94
B.1 Specification Phase	95
In the specification phase of product development, static analysis methods are used to determine that the product specification is complete and unambiguous and that it satisfies the customer's needs. This requires supplier/customer agreement.	
B.2 Design Phase	96
In testing the product design, static analysis methods of testing are used by the product developers to insure that the design provides the features necessary to satisfy the product specification and that the design is consistent.	
B.3 Coding/Debugging and Integration Phase	99
Product developers and quality assurance personnel work together in the phase to insure that the implemented product accurately matches the design and correctly performs the required functions. Static analysis and executional testing methods culminate in supplier certification of the product.	
B.4 Delivery Phase	102
The user, or his representatives, use functional tests to ascertain whether or not the product operates as required by the user's needs defined in the product specification.	
B.5 Maintenance Phase	103
Product changes are verified through static analysis and testing. Verification begins with analysis of the changes defined in the product specification and terminates with testing of the changes in the delivered product.	

Figure 1-4

Chapter 2

Current Testing Approaches

The comprehensive strategy for testing is to be built from current testing approaches. These include generalized test methods, as well as test generation techniques and test progression schemes. An understanding of those testing approaches as described in testing literature is necessary to select and use them appropriately. This allows strategic placement of tests within the development process in a manner which provides the most effective testing.

2.1 Test Methods

Two general methods of testing are currently in use: static analysis and dynamic analysis. Static analysis is testing done to determine product properties without execution, and dynamic analysis is testing done through execution. As listed in figure 2-1, both are further separated into specific test methods. These methods exhibit features which may make them desirable for inclusion in the test strategy. Their application at various stages of development and at various levels of the products' structure contributes to the overall strategy characteristics and, hence, to fulfilling the strategy goals.

TESTING METHODS

1. Static Analysis
 - Code Analysis
 - Program Proving
2. Dynamic Analysis
 - Function Testing
 - Structural Testing

Figure 2-1

Through static analysis the testing of nonexecutable portions of the product may be accomplished. It requires reading and examining documents and source code to verify their correctness in terms of algorithms, structure, understandability, and fidelity to desired functionality. This provides an initial test method which can be used prior to the availability of executable code. The two most common static analysis methods are code analysis and program proving.

Code analysis is a software verification method by which the program code is read and statements are made about its operation. This may be an inspection of software structure and algorithmic flow by a single programmer, or a

group of programmers may be assembled for a presentation of the flow structure by its author (Myers, 1979). In such cases, through having to analyze and present the actions of the code, many errors become evident. Another code analysis method is to actually simulate the code's operation for real inputs, known as a walkthrough (Myers, 1979). Intermediate and terminal variable values are calculated by the analysis personnel and recorded. Gross errors in the software are displayed, but due to the time required in analyzing and computing results, only a few test cases are possible, making the results incomplete.

A very sophisticated analysis method is program proving (Myers, 1979). In proving code, the statements of the algorithm are examined, and assertions are made about their action. By inductively and deductively proving that assertions are correct from preceding assertions, the effect of the algorithm can be proven correct (Myers, 1979). The assertions regarding these algorithms then prove the function of software components, which, when combined, prove the operation of the entire product. This is, however, a very complex and time consuming process and is impractical for all but simple strings of code (unless some form of automated proving can be used). In addition, the proof

depends on a strict definition of what the expected behavior of the code will be and the use of accurate assertions. It does, however, illustrate a building block approach to verification through analysis.

To actually test the execution of program code, dynamic analysis must be used. Inputs are selected and the code executed; then the output values are checked for suitability. To be absolutely thorough, exhaustive testing, where all possible input combinations are utilized, is necessary (Myers, 1979). This not only includes data inputs but all environmental conditions of the system. Even for small systems and programs this is such a large number of inputs as to be impossible.

Functional testing (Howden, 1980) is a method by which a reasonably thorough test of program execution may be implemented. The criteria here is to execute the software so that all possible functions, as described in the product specification, are verified. Test cases are selected from functionality specifications without considering software structure. The test inputs must not only check to insure that expected outputs are correlated to inputs, but also that inappropriate inputs are handled properly. This is actually a black box approach (Myers, 1979), where the

operation of the software is defined, inputs are sent to it, and the outputs are verified.

Another use of functional testing is in measuring product performance. When software has been found to operate correctly, profile type tests (which characterize execution) may be executed to determine how efficiently the product performs various functions in terms of throughput and resource (e.g. memory and peripheral) utilization. Various performance related functions specified, such as maximum execution time for an application or number of user instructions executed per second, can then be demonstrated.

One technique of generating functional test cases (the sets of inputs which exercise software functions) is to simply use random data patterns. This technique is practically useless (Myers, 1979), as it does nothing to attempt to force errors, insure thoroughness, or stress the software. It is, however, fairly simple to use.

Another technique is to use a random sample of customer data as test cases. Here again there is no attempt to force errors or stress the system. This does, however, simulate the software's operating environment well, and if real time conditions are accurate, such a test might be useful in determining whether or not a user's needs are

satisfied and what level of performance is achieved. The use of a mixture of customer data which has been determined to accurately simulate a typical run of the application gives a clearer view of average performance characteristics of the product. An input mixture which simulates the worst case conditions which the customer would experience will indicate the product's worst case performance in that environment.

In order to demonstrate the functional integrity of the software product, several techniques may be used to generate a reliable set of test cases. Such a set of test cases implies the absence of errors when the entire set is executed successfully (Goodenough, Gerhart, 1975). Boundary value analysis, cause/effect graphing, and error guessing are techniques which lend themselves to generating test cases.

Boundary value analysis is a methodology which requires a partitioning of input conditions into equivalence classes (Myers, 1979). These are subdomains of the inputs for which any element of the class is representative of the entire class. By selecting elements of the sets which are within the class, within the class but near its limits, on its limits, and just beyond its limits, an effort is made

to force errors (it is theorized that these boundaries are prone to errors). To extend this method the result's ranges can also be partitioned and test cases selected which produce outputs around the edges of their ranges. Test cases are generated until the desired level of testing is realized. For functional tests the boundaries of the equivalence classes are based on functional software characteristics.

The major deficiency of boundary value analysis is that test cases "Do not explore combinations of input circumstances" (Myers, 1979, p. 56). No provision is made to select input values from different equivalence classes which may conflict or produce inconsistent results.

Cause/effect graphing provides a systematic means of selecting combinations of input conditions which produce high-yield test cases (Myers, 1979). In this type of test case generation, input conditions are mapped to output results via graphic form. Many of these causes and effects fall immediately out of the specifications, and this process can be useful in detecting failures of the specifications. The graph is then converted to a table of decisions by tracing the graph back from effect to causes. The decision table entries are then used to generate test

cases. Product specifications can be used to build functional test cases, and in this way specification deficiencies may be revealed. The deficiencies of this technique are that it does not address boundary conditions specifically, and it involves a rather cumbersome task. It is a great deal of work to build the graph and then convert it to test cases.

A third test case generation technique is error guessing (Myers, 1979). Experience in locating errors and intuition are used to select test cases. Test values which have been troublesome in past projects may be especially helpful in testing, or special case values which have been identified in other areas of the development process may be useful. It would be difficult to generate a complete set of tests through this technique, but it provides a good supplement to tests generated by the other methods.

While these techniques primarily generate large sets of test cases for executional testing, they may also be helpful in static analysis. In performing walkthroughs and code inspections, some type of data processing must be simulated. Error guessing or boundary value analysis might be a satisfactory means of selecting limited tests for this purpose.

A method of generating test thoroughness through checking its effectiveness uses mutation testing (Demillo, Lipton, 1978). It is based on the assumption that any software product has a finite (although possibly very large) number of mutants or corrupted versions with a single error and that programs are normally close to being correct. Therefore the set of mutants represents an approximation of all erroneous program conditions. By executing the test cases on these mutants, only mutants which are equivalent forms of the correct program should produce correct results. This then indicates how thorough the test is. When a mutant does execute properly it must be resolved as to whether it is truly equivalent code (e.g. the condition $A \neq B$ can be the same as $B \neq A$), or is the result of a deficiency in the test data which does not detect the error. If it is an erroneous version, test coverage can be improved by writing a new test case; hence mutation testing is a test generation technique. Mutation testing involves a great deal of overhead, first, in producing a complete set of mutants and, then, in executing the set of test cases on all of them.

An alternative to functional testing is structural testing or path testing (Howden, 1976). This method

requires analysis of the product or component structure to distinguish flow paths (strings of statements and decision branches). By executing test inputs which selectively execute the flow paths, and by checking results, as well as intermediate variable values, and by tracing code flow, the operation of each path may be tested. When all paths in the structure have been tested, structural testing is completed. This type of testing is more cumbersome than functional testing, because it has the extra burden of analyzing flow-related errors; however, it may prove useful in debugging, as information on where the error originated is more visible from intermediate values.

In selecting test cases to execute structural tests, input values are selected based on the selection of flow paths to be traversed. Subdomains of input values or combinations of subdomains which are representative of each paths' traversal may be identified. Boundary values and error sensitive inputs may be especially helpful in ascertaining that the decisions which cause path selection are correct and that the paths execute correctly. If care is taken in selecting test cases, and functional results are observed, an overlap between functional test cases and structural test cases may occur.

The general consensus of most of the authors knowledgeable about testing is that no one test method is itself sufficient. Myers, Howden, Goodenough, and Gerhart all indicate that static analysis, functional testing, and structural testing are complementary and should be used together to improve testing.

2.2 Test Coverage

In order to discern whether or not a set of test cases is likely to be effective (thorough), test coverage must be quantified. Coverage is the extent to which the test cases exercise the software and is the basis for satisfying test completion criteria.

One coverage measure is function coverage. This is simply how much of the software's functional character is tested. It might be explained as how close the input cases come to producing all of the result effects possible, or how close the test cases are to 100% detection of all the mutants in mutation testing.

Structural tests are based on some form of path coverage. One form of path coverage is statement coverage which quantifies the total individual statements exercised. Statement coverage is made more stringent by adding branch coverage which quantifies the execution of decision outcomes.

By including not only branches but the conditions or combinations of conditions causing the branch decision, an even stronger coverage results. The most complete type of path coverage is one which, not only is based on statement coverage, branch coverage, and condition coverage, but also utilizes coverage of loops with varying iterations and external entry points (Myers, 1979).

In an effort to standardize structural test coverage measures, a set of levels has been defined for both modules and systems (Software Research Associates, 1981). Module test coverages start with C0 coverage which is simply execution of all statements within modules. With the addition of the condition that all segments are executed, path coverage designated C1, is attained. Levels of stringency continue to build to C_k coverage which requires execution of all module paths, including reiterative loops up to k times. C1 coverage is the coverage level most often advocated as minimum coverage level (Software Research Associates, 1981).

System coverage measures are similar to module levels. Coverage level S2 is similar to C1. It requires the invocation of each module, within a system, at least once for each possible value of logical expression parameters. This means that each path between modules must be exercised.

Various levels of stringency are used to define other system levels. The S2 level combined with C1 coverage for modules appears to provide a path coverage criteria throughout the system.

Besides having a criteria for test coverage, a means of measuring coverage is also needed. Any generation of a set of test cases should ideally yield 100% coverage for the criteria on which it is based. This is, however, unlikely. Special tools, such as program tracers and flow analyzers, can be used to check for structural types of coverage. It is also possible, although cumbersome, to insert flags and counters into flow paths which, when dumped on test completion, indicate what statements and branches were executed, but not under what conditions. Without special tools structural coverage appears difficult to quantify with certainty.

Test coverage can be used to determine satisfaction of test completion criteria. Completion criteria are standards which when met indicate that no more testing of a piece of software is required. These criteria have many forms. They may be based on achieving a given percentage of one of the structural test coverages, such as reaching 90% condition coverage or 100% statement coverage, or a

combination of the two. Completion criteria may also be based on functional test coverage (Howden, 1980).

Some slightly different completion criteria are based on error detection (Bowen, 1979). One example of this is a criteria which indicates that testing is finished when the rate at which errors are detected falls below a certain level. This level may be a ratio of errors to time or errors to test cases. Another criteria relies on errors which have been detected per lines of code, or simply the total number of errors found and corrected. A third criteria is not as concerned with finding errors but is based on reaching a period of execution time without encountering failures.

Criteria which are based on minimizing error counts may tend to dilute test stringency, while those which require errors to be found have the reverse effect. On the other hand, it is difficult to predict how many errors are a suitable threshold, and it may be argued that errors which seldom occur or are too difficult to produce are not valid errors. In general, there seems to be a great deal of doubt about how to apply error level related criteria, and a great deal of experience and historical data are needed.

2.3 Type of Testing

There are two general modes of testing: incremental testing which progressively builds program structure and tests simultaneously, and nonincremental testing which tests all individual components separately and then combines them for test all together (Myers, 1979).

Incremental testing through its building block nature provides increasing levels of confidence as software components are tested and added to the overall structure and retested. This not only repetitively tests the components through increasing usage, but also verifies interfaces between them. It also provides a great deal of insight as to the location of errors when they suddenly appear after adding a new component and is helpful in debugging.

Nonincremental or "Big Bang" testing does not provide the aid in debugging and confidence level of incremental testing. It does, however, allow more parallelism of testing and may use less machine time because of the eliminated test redundancy. This seems to be a poor excuse for sacrificing a "superior" (Myers, 1979, p. 92) type of testing which is a key to reliability.

2.4 Test Ordering

In addition to the selection of testing methods, a test activity must also have a logical testing order or hierarchy. Several common testing orders are listed in figure 2-2. One order may be used in one stage of development while another may be selected later.

TESTING ORDERS

1. Top Down
2. Bottom Up
3. Sequential
4. Parallel
5. Random

Figure 2-2

Top down testing order starts with the highest level component in the product and tests it first; then successively lower levels are tested by adding them to the higher components until the most basic block is reached (Myers, 1979). When a higher level block is under test, known data may be fed to it from below by stubbing the lower level blocks. This allows the block under test to be judged on its own merit and eliminates any possibility of an error from lower components propagating up at this time.

As a top down design flow is considered a good structured design method (Zelkowitz, 1978), close coordination between design and testing order is maintained. An early functional product skeleton is produced giving management a tangible accomplishment. However, "The production of stubs is not a trivial task" (Myers, 1979, p. 94) as they may entail logic to simulate the operation of the components they are replacing.

The reverse order is bottom up testing. Here, the basic functions are tested first, followed by higher level structures (Myers, 1979). Each time a higher level component is applied and tested it utilizes lower level components, providing additional testing of them. This may add to test thoroughness. If, however, a lower component does contain a latent error, isolating it may be more difficult, and its correction may involve more extensive retesting back up through the intervening blocks.

The sequential order of testing is very simple in that it merely tests components in the sequential order that the software is structured (Myers, 1979). This is a simple test order in that no parallelism is introduced in the process of integrating components. Testing may follow one leg

of the structure in a top down order or bottom up order to its termination. When one leg is completed the next leg is tested in a like manner. The difference between this and a strict bottom up or top down test is that only one chain is tested at a time.

Parallel testing also can follow a top down or bottom up structure. As opposed to sequential testing, it allows several chains of the program structure to be in test simultaneously (Myers, 1979). This, of course, requires a much larger test organization and more resources.

Random testing is a hit or miss selection of module testing order. Modules are selected at random for test, possibly in the order of code completion. The module itself is tested, but no real building of confidence in the program structure or of its interfaces is gained this way; consequently a great deal of additional testing is needed when integrating the modules. It may also be difficult to schedule testing.

2.5 Summary

From these testing methods and techniques, the elements of the test strategy must be selected. All stages of product development must employ the most effective test method, or combination of methods, and use the most cost

effective test generation techniques. Criteria used to select and evaluate tests must not only be significant in test satisfaction but must also be practically implementable.

Chapter 3

Product Development Issues

The general progression of software development must also be defined before the test approaches from chapter 2 may be placed within the comprehensive test strategy. Such factors as the level within the software structure undergoing test, the point in the development cycle where the test is occurring, and the purpose of the test are important in determining how to test. In addition, there are management issues such as test planning and test responsibility which must be addressed.

3.1 Development Life Cycle

The software development process can be divided into several phases, each of which encompasses related activities. One such set of phases is listed in figure 3-1 (Metzger, 1973; Zelkowitz, 1978; NCR, 1980; Fairley, 1982). At each stage certain sub strategies of testing may be required (Myers, 1979). Early stages will require analysis methods as there is no executable object to run. In later stages emphasis shifts to execution of the software as the executable object develops.

SOFTWARE DEVELOPMENT STAGES

1. Specification (Definition)
2. Design
3. Coding/Debugging (Programming)
4. Integration (System Test)
5. Delivery (Installation)
6. Maintenance

Figure 3-1

The specification stage is critical to the usability of the product (NCR, 1980). At this stage the user requirements, including objectives and constraints, are formally stated in a specification. The rest of product development will be based on this specification; therefore, if it is in error, the error will propagate through all of the remaining stages and may result in extensive rework or rejection of the product. It is, therefore, essential that the specification be tested with respect to its accuracy, comprehensiveness, understandability, and exactness. It may be that specification testing will be an ongoing process, with specification changes due to errors or redefinitions requiring additional specification tests in later development stages.

In the design stage the software structure is defined. The software components are established, the functions are designated and the interfaces conceived (Metzger, 1973). In effect, a number of lower level component specifications are created, as well as a mapping between them. Here again undetected errors will propagate into later stages, and every effort must be made to insure that the basic design is correct.

The coding/debugging stage of the development cycle produces the building blocks of the product. The components produced are debugged to at least an initially operational degree. Further debugging will occur when the components are integrated. Within this stage the individual programmers create software components which: one, implement the functions designed, within the specified constraints; two, match the required interfaces. They then test and reiterate their code.

Integration is the process of connecting software components together as designed, producing increasingly complex software structures, until the product is formed. Integration begins where coding/debugging terminates. At this point activity is no longer that of individual programmers but becomes a combined effort. Testing at this

stage most likely will reveal incompatibilities in interfaces and functions, and debugging will be required to revise the offending blocks of code. Integration is completed with final approval of the product.

Before becoming usable the product must be made available to the user and installed. This is the delivery stage. The user must test to determine that his needs are met and that the product is operable. Although it is still possible for specification errors to surface at this time, it is hoped that all such errors have been detected long previously. Consequently, all previous testing should be done with a view towards satisfying the delivery stage requirements and insuring that the desired objectives match those specified.

Once accepted and in operation any problems which develop, or enhancements requested, are accomplished through the maintenance stage of development. This stage continues throughout the useful life of the product. Any maintenance to be done begins with a specification stage and flows through delivery; consequently, the other five stages of the development cycle are included.

The nature of each stage in the development process produces specific testing needs. These are needs which may

be met through varying test methods and techniques. For this reason a progression of testing approaches must follow the software development cycle to make the comprehensive test strategy effective.

3.2 Software Structure

Software components are often building blocks comprising a software product. Considering components individually rather than looking at the entire product often makes development, including testing, much more manageable. Such components include modules and units within a system. The distinction between these structures is often not clear. A module normally consists of one or more functional operations which complete a specialized task in an independently compilable code segment. One or more modules are then combined into a unit of a software system. A unit performs related tasks which accomplish a product feature. The combination of units completes the software system. As may be seen the dividing line between modules and units may at times be ambiguous, and a software product may be a module, unit, or a system. What is important is that the total product may consist of a hierarchy of component blocks, each of which must be operational and reliable before the whole is reliable.

3.3 Testing Levels

There are a number of hierarchical testing levels which can be defined to fulfill specific purposes in testing. These levels are listed in figure 3-2.

TESTING LEVELS

<u>Level</u>	<u>Purpose</u>
Basic Component Levels Module Unit	Verify correctness of individual product building blocks.
Integration Levels Module Unit	Verify correctness of higher level system components through combination of components.
Certification Level	Verify overall product functionality and acceptability of testing.
Acceptance Level	Verify overall product functionality for contract and user satisfaction.
Regression Level	Reverification of product and components following the inclusion of modification.

Figure 3-2

The first test level, basic component level testing, verifies the functionality of individual product components; consequently, it has two sublevels: the module level and unit level. In nonincremental testing each module would be

tested separately for correctness. When the modules comprising a unit were all tested they would be combined into the unit, and the unit tested for correctness. All additional units would be tested in a like manner before combining the units and testing the product. In an incremental approach to testing however, after some initial modules are verified, the combination process would begin using the previously tested modules to test additional modules. At various points module combinations into units are completed allowing combinations in which units are integrated and tested.

Integration level testing is the testing of various stages in combining components and thus overlaps the module and unit basic component levels of incremental testing. The module basic component level of test is unique only as long as single components are involved. The module integration level is entered concurrently when the structure under test contains more than one module. Unit level integration testing involves the testing of unit combinations and is concurrent with basic component unit level testing after the initial units are completed. In the context of nonincremental testing, the module level of integration test assumes the connotation of testing modules

integrated into a unit, and unit integration testing verifies the completed integration into systems.

Once a product is complete and approaching delivery, a higher level of testing comes into play: testing for certification (Sorkowitz, 1979). Certification is stating that the software has been judged to conform to the specifications describing and constraining it and is therefore fit for use. This is like attaching the supplier's seal of approval. Certification may be the result of successful integration testing, or may be altogether another test stage which again verifies that the product does in fact meet its objectives properly. Since modules and units are sometimes products and are made available to other designers or systems, it is also possible that they may be individually certified. This certification should not occur too early. If, for example, in certification testing of a using unit, problems were detected in a certified component, difficulties could result if the component was already installed in other products. A good time for component certification might be at certification of the product for which the component was initially designed.

Acceptance testing is often done by or for the receiver of the software, after installation. Its concern

is with verifying the correct operation of the complete product in its actual environment, on the data situations to be processed. Testing should be based on the product specification and objectives and determines whether the product actually does what the receiver specified (Myers, 1979; NCR, 1980; Metzger, 1973). In many cases this testing is referred to as installation testing and is a user monitored extension of certification.

Once the software is accepted and operating, if changes are made to fix problems or provide enhancements, regression testing is required (Metzger, 1973). This tests the changes which are made to a working product. The concept implies repeat testing on previously tested software and it does in fact involve some or all of the other levels of testing. It may actually duplicate original tests or may use new tests at the same levels. Regression testing will be an ongoing process as a product is being maintained.

3.4 Test Planning

A plan of testing must be followed throughout testing. The purpose of the comprehensive test strategy is to provide a skeleton for implementing such a plan. A test plan implements the details of testing through test definition and ordering. It provides a master specification

of the entire testing activity and should identify the items to be tested, the goals in testing them, the tests to be executed, resource requirements, requirements to complete testing, testing organizational structure (personnel assignment), and procedures for managing data (IEEE, 1981).

Tests identified in the test plan are implemented through test specifications (IEEE, 1981). A good test specification should designate what component is to be tested and the testing to be done. To define the testing to be done, a test method should be selected. This could be one of the static analysis or execution techniques discussed previously. This, in part, will also define what technique should be utilized for generating test cases. In addition, some supplemental test case generation may also be indicated. The criteria for determining that the test is finished needs to be specified. Finally, the meaning of completion of the testing should be specified: for example, the module is complete, the program is certified, or the system is acceptable. The specifics of these items are all, of course, dependent on the character of the test, but their application is guided by the test strategy.

Subordinate to test specifications are test cases and procedures. Test cases provide the test data referenced

in the test specification, while procedures describe how to execute the test. It is helpful if each test case includes documentation on what it tests (e.g. what equivalence class of boundary conditions), and special conditions it may use or cause, as well as the input values and results expected. Test procedures should describe how to recover from errors and how to terminate.

Each test should have its own test specification. At each stage of development and level of testing one or more tests may be necessary to validate the component. This may cause several test specifications to be grouped together as a test. There should be a master test specification to associate such a group of tests.

Test specifications can take many forms. They may be written documents, automated computer files, or files that produce documents. They may provide descriptions of operation which include a general algorithm for determining whether errors are present. Equations may be used to specify expected results for inputs.

As mentioned, a computer file may be used to implement a specification, thus providing a media for utilizing automatic testers and generators and for the retention of data for regressive testing. These are very specialized tools

being developed for software engineering, which may be a part of a complex design automation system. Among automatic test tools being developed are program analyzers, test drivers for automated test beds, proving algorithms, flow and path traces, debuggers, and even automated data generators. Use of tools such as these, however, would tend to greatly restrict the application of the test strategy and are more pertinent to the implementation of a test system than to its overall strategy.

In addition to calling out the tests to be performed, the test plan must also describe or reference standards and procedures to manage the data generated by tests. Test activity must be monitored. One means of doing this is through test logs which keep a record of tests performed and interesting events. Test incident reports and error reports may be used to provide visibility to failures of the software or test mechanism and to the resolution or corrections of errors. Information about tests must be compiled into a summary report which not only evaluates the software tested but also the test itself.

3.5 Responsibility

Testing must be done with the right attitude. Basically, a program may be, and usually is, tested with the express

purpose of proving that it functions properly and does not contain errors. Any errors which are discovered and corrected are in effect only by-products of testing. An opposing attitude toward testing is to prove that a program does have errors and is therefore incorrect (Myers, 1979). In this case a valid software component is actually proven by failure of the test. Conversely, test cases are only successful if an error is uncovered. It is thought that the second test attitude probably produces more stringent testing, and it is under these circumstances that a possible test completion criteria might be the detection of a pre-determined number of errors.

A destructive attitude towards testing puts the tester in the position of doing everything he can to crash the software and consequently provides more potential for error detection than with the tester who merely wishes to see a correct output. This attitude is damaging to the ego of the developer who constructed the software and may be a difficult task for him. Furthermore, any errors resulting from his misunderstanding of the specifications or requirements may go undetected. This leads to the statement that programmers should not test their own programs (Myers, 1979).

On the other hand, especially at lower levels of testing, the programmer usually has insight into his design which allows him to select error prone test cases better than another, even though he may not be as capable of destroying his own creation through error proving. It appears that in testing the purpose should be a compromise, such as demonstration that the program works satisfactorily through examination of its operation for correctness, with a conscious effort to flush out errors. This effort to find errors will greatly influence the selection of the test sets used.

The question of who is to perform testing is one with differing opinions. As noted, it may be difficult for the creator of software to deliberately cause it to fail, while it may be difficult for a second party to become familiar enough with the software to see its subtleties. This situation may be improved if the code is well structured, understandable, and well documented. It appears, however, that the selection of the tester is at least in part dependent on the expected accomplishment of the test. A team approach seems to be a good compromise between the two approaches, and some variation of this may be appropriate. One such team would utilize the programmer to design test cases and a second party to execute them. Another structure

might be for a developer to do the testing while being audited by a quality representative. A third alternative is that all tests be well documented and turned over to a separate organization which will analyze the software, the test cases, and the results to determine whether or not testing has been sufficient (Sorkowitz, 1979). The team structure may be dynamic so that in early stages developers do the testing while in later stages, as certification time approaches, independent quality personnel become more active.

Chapter 4

A Test Strategy

A test strategy provides an overall tactical scheme for the testing activity. To be complete it should not be tied to only one set of testing principles; rather it should select as many principles as are useful and apply them in a manner which maximizes the benefit of testing. The comprehensive test strategy provides a skeletal guideline from which a test plan for a specific product development project can be developed. It provides guides for when in the development process testing is to be done, what should be accomplished, what methods are appropriate, what test case generation techniques are useful, how tests are to be measured, and where test responsibility lies. All of this information, as presented in chapters 2 and 3, is needed in the planning process. It is included in the test plan and in the descriptive test specifications referenced by the test plan.

In an attempt to be as universal as possible, this strategy is not concerned with details of implementation which might be influenced by the language, hardware, development organization, or tool availability. Rather,

it provides a skeleton from which a test activity may be built to suit a specific application. Appendix B outlines a comprehensive strategy. The following pages will present this test strategy.

Minimal Environment

Before describing the test strategy, some basic assumptions about the product development environment must be made. These assumptions provide a minimal capability of testing and therefore should not limit the use of the strategy.

First of all, it is assumed that some form of program tracing is available. This may be a sophisticated automated tool, or a crude method whereby breakpoints are manually inserted in the code. Any basic software debug tool should provide at least a minimal capability of this, and without it no real information about product performance can be obtained.

It is also assumed that expected outputs of the product can be predicted for given inputs (Howden, 1978), and compared with actual outputs. Here again, this may be, at the least, a manual means of calculating expected results and comparing them to observed values, or it can be an

automated system which models expected behavior and automatically compares results. It should be noted that the second case may actually involve a second version of the product which models expected behavior, and care should be taken in such a case not to replicate the product, which could cause replicated errors to occur in the tester.

Strategy Structure

The test strategy is composed of components which guide the sequence of testing. A component of the strategy describes testing to be done in each phase of the development process. The components for the phases include information regarding what types of testing should be done, to what level, and by whom, as well as how to document and report activity. Through this information the strategy is converted to a functioning test process described in the test documents of appendix C.

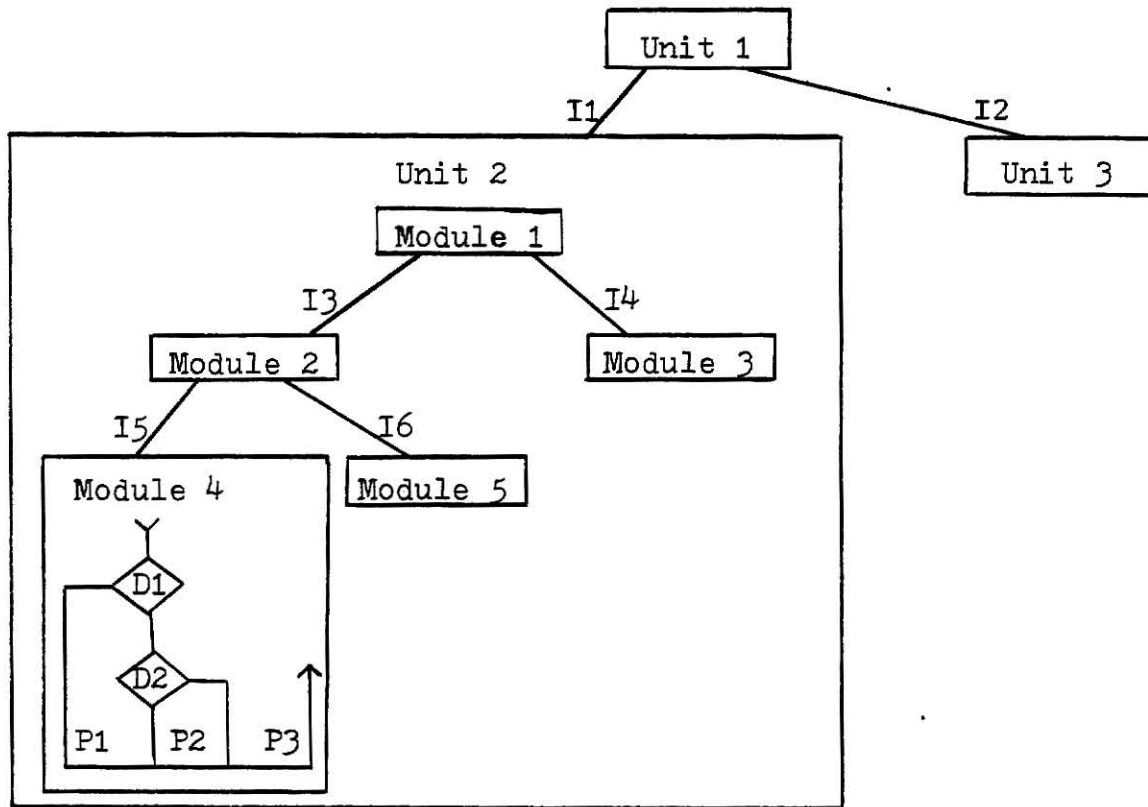
The test plan of appendix C is the overall view of the test activity and must call out all tests and their ordering. It must schedule when tests are to occur and what resources are needed. While this is basically a supervisory document, it is the thread which ties the individual tests together and minimizes roadblocks. Test specifications dictate what

test methods are used, how test cases are generated, and what goals or completion criteria apply to the individual tests. The test cases generated, and procedures for executing them, are also contained in the specifications. All of this information is guided by the test strategy, and actual form is dependent on the development organization. Chapter 5 covers implementation issues in greater detail.

Project Example

Figures 4-1, 4-2, and 4-3 define an example of a product development project which is used in this chapter. While it is a simplistic example it illustrates application of the strategy without implying limitations to its usage. Use of this example in describing the test strategy will clarify application of the strategy.

EXAMPLE SOFTWARE PRODUCT STRUCTURE



P1, P2, and P3 are logical flow paths in Module 4 selected by decisions D1 and D2. I1-I6 are interfaces between software components.

Figure 4-1

EXAMPLE SPECIFICATION

The requirements of the product are expressed in statements R1 and R2.

1. Product Specification: The product provides the functionality expressed as F1 and F2.
2. Unit 1 Specification: This unit provides the functionality expressed as F1 and F2. It places an external requirement of R3 on unit 2 through Interface I1^O, and a requirement of R4 on unit 3 through Interface I2^O.
3. Unit 2 Specification: This unit provides the functionality of F3 to unit 1 via interface I1^S. (The production of F3 is the subject of detailed design within Unit 2, items 5-9).
4. Unit 3 Specification: This unit provides the functionality of F4 to unit 1 via interface I2^S.
5. Module 1 Specification: This module produces functionality F3 for unit 2 through interface I1^S. It places an external requirement for R5 on module 2 through interface I3^O, and for R6 on module 3 via I4^O.
6. Module 2 Specification: This module produces F5 functionality through interface I3^S. It requires R7 via I5^O from module 4 and R8 via I6^O from module 5.
7. Module 3 Specification: This module produces F6 functionality through I4^S.
8. Module 4 Specification: This module provides F7 functionality via I5^S.
9. Module 5 Specification: This module produces functionality F8 through the I6^S interface.

Note: Interfaces IX^O and IX^S indicate the opposite sides (ordinate or calling and subordinate or returning) of interface IX.

Figure 4-2

EXAMPLE DEVELOPMENT ORGANIZATION

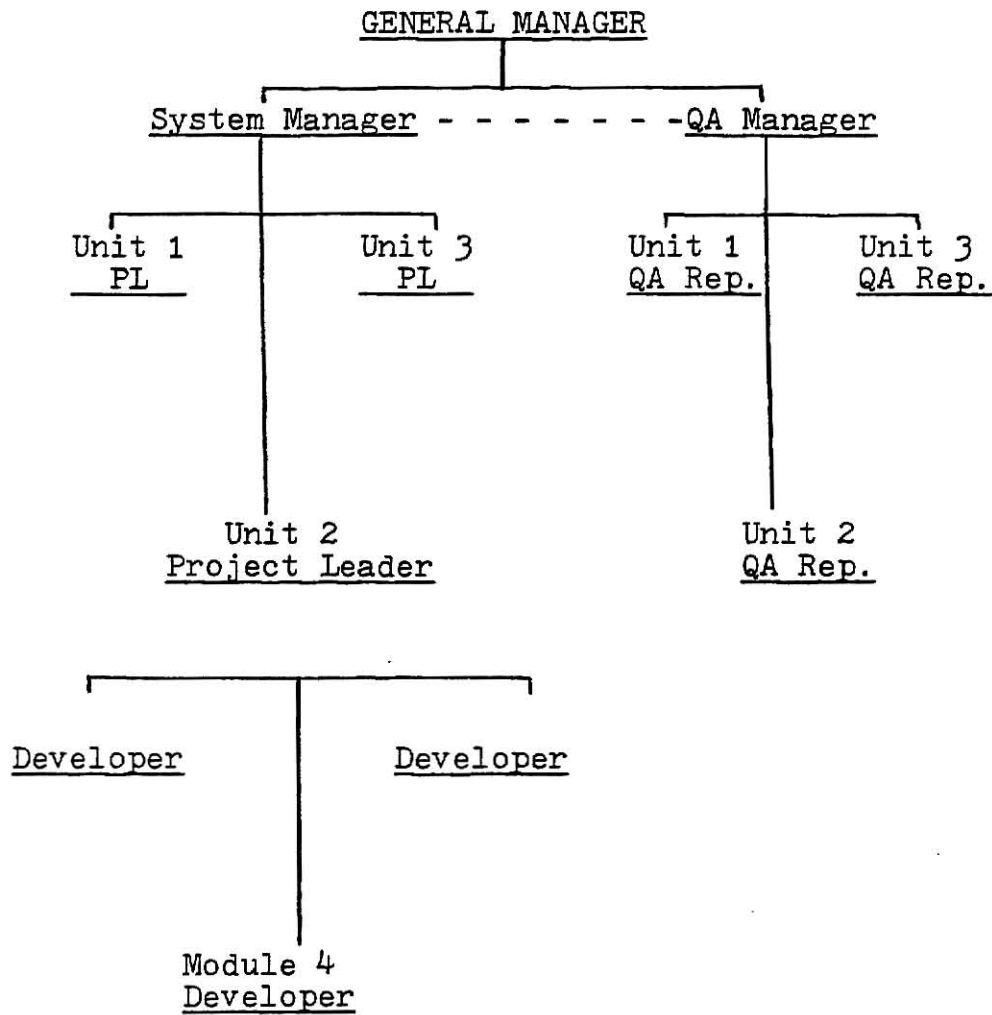


Figure 4-3

4.1 Specification Phase Test

Testing begins at the specification stage of software development. The specification is a concrete representation of what the software is to be and, therefore, must be correct before the product can be designed and built. The purpose of specification testing, therefore, is to insure agreement between the customer and developer regarding the product's nature. The proper application of an appropriate test method is necessary to accomplish this.

Test Method

Since the specification is not an executable type object, static analysis must be used to verify it through statements relating functionality produced to that required by the user. The difficulty of the analysis depends on the specification structure. If the specification is well structured and written in an unambiguous manner, the objectives it presents will fall out clearly. They should closely match the user's requirements, making the comparison almost trivial.

In the product specification of figure 4-2 there are two functions documented as the overall functional product definition. These functions might be performance levels provided, human interface features, or descriptions of

results provided by the product. They define how the product behaves and, as such, are basic statements about the product (F1 and F2). To show that the specification is complete the user's requirements should be stated in similar terms, (R1 and R2). There should then be a one to one correspondence between the functions and requirements (e.g. $F1=R1$, $F2=R2$). If the match is not perfect, for example, if there are requirements without supporting functions, or the equivalence of functions and requirements is in doubt, the specification may be incomplete or contain ambiguities. Both of these are product specification errors which this test stage should detect. The other possibility is that the requirements are incorrect, and this is also the time to resolve such misunderstandings.

Test Level

Verification of the specification is actually an acceptance level process. It involves verification that the customer or user concurs with the requirements and objectives before the development process continues. This is nonincremental in nature and does not involve any ordering of the process, other than that there may be some basic functions which are required to support the system concepts.

This test level is not finished until all discrepancies between the user requirements and the specification have been resolved.

Testers

The customer plays an important role in verifying the product specification. He must participate so that his approval can be given when verification is complete, signifying that the defined product is what he is prepared to accept. In addition, the customer personnel are more likely to see specification deficiencies or ambiguities, as they know their requirements, which are not necessarily what the developers perceive them to be, and they can extract what the specification states rather than what the author meant it to say. The developers' part in verification is to support analysis of the specification and insure that misunderstandings or discrepancies are corrected.

A suggested team approach includes the System Manager from Figure 4-3 and an experienced project leader along with their counterparts from the customer organization. The managers provide the contractual authority necessary for approval, while the technical level provides development and analysis. This keeps the team small enough to communicate and operate effectively.

Documentation

Formal documents produced should include lists of functions, inputs, outputs, performance levels, and constraints. All items should be specified in a manner which makes them quantifiable (IEEE, 1981). In addition, documentation of specification changes or areas of misunderstanding is helpful to provide background for any problems or controversies occurring at later stages. All of these documents must carry the end user's approval. If, in later stages, changes affecting the specification are desired, or problems with the specification are found, they should be documented and again submitted to analysis and acceptance with the user. Changes which are the result of specification error should not be combined with requirements changes, as this would not provide an accurate error history.

Accomplishment

The result of this type of analysis will be a definitive product specification from which the system can be designed. It encourages a well written specification with an accurate history of requirements and should insure that the user is aware of what he will receive. It also provides a basis for acceptance testing of the final product.

4.2 Design Phase Test

In the design stage of development the elements of the product specification are broken down and used to generate lower level specifications of the product's components. These lower level specifications must define the functions provided by each block and its interfaces and what performance limits apply to each block (e.g. maximum execution time allowed for a module). Before constructing the actual blocks, these specifications must also be verified in order to minimize false starts and code reiterations in the coding stage of development. Here again the correct combination of testing practices is necessary.

Test Method

Component specifications are still not executable structures and static analysis must be used. A combination of two techniques seems most appropriate. Functionality is proven for each block from its specification. In addition, the structure is walked through to verify interfaces and the functionality of the product.

In proving the design of the individual components, the elements of the product specification are compared to the functions defined by the component specifications. The product requirements have already been fulfilled in the

functional analysis of the product specification; therefore, a set of required functions is available. As additional functions are required during design, they are added to this list. As the design progresses, the component specifications provide fulfillment of these functions. This list should also contain information about what component (system level or another unit) requires the function and in which component it is provided. This produces a map of the designed functional pieces.

With respect to the example, a chart, such as figure 4-4, might be produced. The initial entries are the product requirements and the defined functions of unit 1 which satisfy them. Then the functions required by unit 1 from hierarchically subordinate components are entered (R3 and R4). F3 is supplied by unit 2 and F4 by unit 3 so they are added. This process continues down the hierarchy in a top down manner. As with product specification verification, there should be a correspondence between functional requirements and functions specified. If this match is not perfect, design errors have been detected and must be corrected.

<u>COMPONENT/REQUIREMENT</u>		<u>FUNCTION/COMPONENT</u>	
Product	R1, R2	F1, F2	Unit 1
Unit 1	R3	F3	Unit 2, Module 1
Unit 1	R4	F4	Unit 3
Module 1	R5	F5	Module 2
Module 1	R6	F6	Module 3
Module 2	R7	F7	Module 4
Module 2	R8	F8	Module 5

Software Component Specification Test

Figure 4-4

As the components are defined the consistency of their interfaces must also be verified. For example, the interface definition of $I1^O$ in the unit 1 specification should match the $I1^S$ definition in the unit 2 specification. If the match is not perfect design errors have been detected, and the interface must be resolved. It may be necessary to build tables defining the interfaces which include data items, their types, the direction they are being passed as well as flow information on how control passes to the other component. There would then be two matching definitions of $I1$, one derived from the unit 1 specification and the other from the unit 2 specification.

Test Flow

Design testing follows a top down flow. This closely follows a top down structured design flow and allows testing to overlap development. In fact, as functional requirements are extracted they may be used in developing specifications at the next lower level. When a component has been specified, verification of the next level above can be completed. This provides early visibility to the product structure so that project planning can be completed. If a design progression other than top down were used, top down verification would still be applicable, but the benefits of closely following design would not apply.

In following the structural hierarchy for testing, three levels of testing occur. An incremental integration test of the mating of unit components is accomplished as well as a test of the unit and module designs.

After the entire product is designed, a final integration test of the design is in order. This is done by walking through the component specifications. For example, a set of product flow scenarios would be selected from the specifications. One scenario would be traversal of interfaces I1, I3, and I5 of figure 4-1. In this traversal functional characteristics of unit 1, unit 2, module 1,

module 2 and module 4 are reverified, as well as the consistency of the interfaces. As this analysis is time consuming the number of scenarios possible is limited; however, all interfaces should be used at least once implying an S1 coverage of all module access paths. Design testing is completed when all functional requirements and interface matches are correct, and S1 walkthrough coverage is satisfied.

Testers

A small team of technical personnel within the development organization, not including the actual designers, should be used for design testing. These testers should have the expertise to judge the merits of the product design, while not being burdened with the details of implementation. Use of the technical individuals from the specification test team provides a knowledge base more closely coupled to the overall product requirements. Additional team members from the quality assurance organization may be used to help weed out ambiguities and to begin bringing the quality group up to speed on the product structure. The actual designers should be available to the team for explanation and resolution of design problems.

Documentation and Accomplishment

When the design is complete, a set of tested functional specifications describing the system design are complete. These probably include flow diagrams, function charts, and interface definitions, as well as timing and resource allocation tables. In addition, there should be documentation similar to that in the specifications phase showing functionality of the specifications. There should also be records of the walkthroughs. Any structure or understandability problems, as well as consistency gaps, should have been detected in this process, especially if individuals not intimate with the design were used to audit the walkthroughs. The use of quality assurance personnel provides this as well as allowing QA to start getting acquainted with the software they must later certify.

As the initial part of the design process is exploratory, changes made in progress need not be tracked. However, beginning with the walkthrough all inconsistencies detected should be treated as errors and so recorded. Differentiation should be made between design errors and latent specification errors detected. Changes made to correct deficiencies must be documented and made available to everyone concerned.

In addition to the testing of the design, completion of this stage provides a product structure from which the test plan and, at least, the beginning of the test specifications can be generated. Project planning is dependent on this design structure.

4.3 Coding/Debugging and Integration Phase Test

Testing in the coding/debugging stage of development and in the integration stage are very similar. In fact, testing of coded software components often requires some degree of integration, and errors are often found and corrected in the integration process. In actuality, this appears to be one test process which extends through both development stages and which produces correct functional software.

Test Methods

In the coding/debugging stage of development, verification begins with static analysis. During the initial coding, walkthroughs should again be used to verify algorithms and their interfaces. Code inspections should also be used to insure that good program structure is maintained. In addition, if the code inspections and walkthroughs are participated in by those to be involved in testing, a basis

for the selection of test cases will be established through this analysis. For example, the module 4 developer of figure 4-3 should continually inspect his code structure to maintain its understandability and simplicity where possible. At periodic points the unit 2 project leader should inspect module 4 code for the same things. As significant functional parts of module 4 are developed a structured walkthrough should be held. A small team consisting of the module 4 developer, the unit 2 project leader, and the QA representative verify the functionality of the algorithms by simulating their execution on sample data inputs.

As the coding of modules nears completion, executable objects become available, and the developer begins testing through execution. The functionality of the module must be verified; therefore, test cases must be executed which perform all the functions defined for the module. Also the flow behavior must be tested to verify that it is as expected. For this reason, both functional and structural testing is done throughout this test stage.

Test cases are selected through boundary value analysis and error guessing techniques. Functional test cases are selected to demonstrate the correct performance of all module functions with input values adjacent to and on the boundaries of their domains. Cases are also selected to

demonstrate the production of results adjacent to and on their range boundaries. Once the boundaries are established, inputs within their domains and results within ranges should also be tested. To supplement these test cases, additional cases should concern data which is especially critical or sensitive. Functional testing of module 4 in the example would involve verification of function F7 under all of these conditions.

Structural testing which may be combined with the functional test activity by intelligent test case selection indicates that the module structure is as expected. Here again, test values should be within acceptable ranges, approach limits from both sides, and lie outside the ranges. In addition, all combinations of input condition types (sets of conditions which cause a unique flow path to execute) should be tested. In determining that the paths of the module are indeed executed as expected, some form of program trace must be available to illustrate that correct path execution occurred. Path testing ideally is not complete until all acceptable module flow paths, including simple and reiterative loop cases, are executed, and unacceptable paths are shown to be unexecutable. C1 coverage which requires the execution of all code segments and decision

branches is more attainable. In the example, this would require the execution of paths P1, P2, and P3 through decision branches D1 and D2. In the event that path verification is not possible extra attention should be given to functional tests which would exercise the structure.

If certain flow paths can be identified as critical to performance, such as the lengthiest possible path, or one which utilizes a large number of resources, they should be executed with a view toward insuring that overall performance will be within limits. If performance goals can not be met, the modules can be optimized at this time to improve performance. In optimization, testing must be repeated, not only to test performance, but to insure that the flow paths and functionality are still valid.

Test Flow

Initially in coding/debugging, testing is at the module level. This begins as a parallel process with a number of modules under test simultaneously. Once the lower level modules of a unit hierarchy are complete they may be used to help test higher modules. At this point, testing becomes an incremental, bottom up process, and a phase of integration is entered, even though coding and debugging are not finished. In using lower level modules to test the higher

ones, the necessity for complex stubs to simulate subordinate modules is eliminated (Myers, 1979). The only external programming needed is a set of drivers to feed data to the module under test. In addition, the lower level modules are retested each time a higher one accesses them during test, and the module interfaces are tested during integration.

From the example, a possible test order would be individual testing of modules 3, 4, and 5 through test drivers which replace modules 1 and 2. The next step would be the integration of module 2 with modules 4 and 5. Testing would then be done on module 2 through a test driver replacing module 1. Integrating modules 1 and 3 and testing module 1 then completes module testing in unit 2. Concurrently module testing may be proceeding in the same manner for modules in unit 3. The end result of module testing then is units.

Throughout module testing both functional and structural tests are important, and boundary value analysis and error guessing are appropriate for test case selection. C1 test coverage is applied to internal module paths, but the external flow paths are also important after integration begins and S2 coverage may be applied. This requires the execution of all the paths between modules. In the example

this would be the following paths for unit 2: module 1, 3; module 1, 2, 4; and module 1, 2, 5.

As units are completed, integration continues combining units into a system. Testing proceeds as with module testing in a bottom up order with functional and structural tests. S2 coverage is applied to the inter unit access paths and the interfaces are tested. One problem occurs in higher level units like unit 1. This problem is the testing of modules within the unit, which require functions from outside the unit. One solution to this is to delay the module testing of this unit, or at least the modules affected, until the subordinate units are complete. The other solution is to go ahead and generate stubs to fill these gaps and permit unit development to proceed. Whatever the method used, the testing of coded components and the product is not finished until the desired coverage levels are satisfied, and integrated code is found to be functional.

Testers

The component developers perform module and unit testing themselves. For example, module 4 is tested by the module 4 developer. As integration occurs the developer whose module is entering the test receives responsibility

(e.g. the module 2 developer becomes the tester when module 2 is integrated with modules 3 and 4). In addition, a QA representative should work with the developers to promote rigor in test case selection and provide support with test tools. When unit testing begins it may be necessary to include the project leaders and additional developers on the team.

Certification completes module and unit testing and integration. This is approval of the product and its components by the QA organization. This is done either on the basis of knowledge about the module and unit tests or another functional type test. A technique found quite satisfactory by HUD (Sorkowitz, 1979) is to provide QA with the test cases and results for audit. QA then determines whether or not test criteria are met and if the testing was satisfactory, and then certifies or rejects the component with noted deficiencies. This saves the redundancy of retesting. One modification to this scheme might be to delay certification of product components until the product is certified, and to add a functional test of the product for certification purpose. When certifying the product or its components, quality assurance becomes the governing organization. Any discrepancies not agreed upon between the developers and the quality group must be

resolved through upper level management before development proceeds.

Documentation

When the integration development stage is complete, the product is complete and ready for delivery. A number of documents are now available. The test specifications containing test cases and procedures are now complete. There are additional documents of early code inspections and walkthroughs. There should also be a number of test summary reports which report the incidents and findings for each module and unit test as well as certification. All coding errors detected after a module enters module test should be noted in the error records, as well as any design or specification errors.

Accomplishment

In reaching product certification it may be necessary at several points to go back several stages to make corrections. When this happens all intervening tests must be repeated with supplemental tests to verify the correction. By the time the product is certified, while it still can not be guaranteed correct, it has been tested and examined

so intensively that it should by now be quite reliable. It also should perform as well or better than specified.

4.4 Delivery Phase Test

Once the user has the product available, his acceptance team will functionally test it. Their test specification will plan and outline procedures, test cases, and acceptance criteria for designating the system acceptable. Their tests should be concentrated on executing their requirements on live data. However, instead of using results in business, they are checked for fidelity to expected execution. In the final stages of acceptance testing, the system should actually be put in use for a period of time. A careful analysis of what the system accomplishes, in what time frame, and at what resource utilization level, should be made and verified against the requirements desired, because, once the product is accepted, the customer is committed to it. Any problems which occur after delivery must be proven to be development deficiencies or else changes must be negotiated as enhancements. Any errors detected at delivery should be given special attention to determine why they were not detected earlier. This is not to try and fix blame, but is an attempt to improve the test plan or the strategy as required.

4.5 Maintenance Phase Test

Software maintenance testing is actually a repeat of the process originally used in testing the product. It starts with analysis of changes to the design specification and culminates in the acceptance of the changes in the system by the customer. While only the tests applicable to areas of change need be repeated, even a small change may affect an entire system, and its overall functionality must be re-verified. This makes testing for program maintenance almost as large a task as the original development testing. For this reason, the test specifications and test case lists used originally are a big help in regression testing. Of even greater help would be a testing environment which uses computer data files and automated drivers to execute test cases.

4.6 Test Responsibility

This strategy utilizes a team approach to testing. In the specification phase of development, responsibility for verification rests with the customer and supplier, with input on feasibility and special considerations from their technical personnel. Later development stages are more developer intensive. In the design phase, the supplier, developers and the specification test team verify component design. This permits fast recovery as inconsistencies are

detected. When verifying the design of the overall unit, the inclusion of quality assurance in the team adds rigidity to conformance with the specification. Coding/debugging and integration testing is also a highly developer oriented activity allowing fast turn around of errors. In this phase, however, quality assurance becomes more actively involved and is, therefore, required on the team to insure thoroughness in testing and to give approval for certification. Finally, in product delivery the test responsibility shifts back to the customer whose technical personnel must evaluate the product.

The use of teams provides a means of using several types of experience to implement tests. The developers are there to provide basic design details and, at early stages, test their development step by step. Quality assurance provides auditing and expertise in the logistics of testing. Later, quality assurance provides authority and rigidity in testing, while developers give functional back up and analysis support.

In order for this team approach to be most effective, quality assurance personnel must be trained in testing. They must have a thorough background in the various test methods and tools. They must also be able to communicate

with developers and take any pressure applied to them when they become unpopular for finding faults. The developers, on the other hand, should at least be aware of the techniques of good testing and its purpose. They must be able to analyze their work objectively and communicate their knowledge to others. Finally, they must be willing to cooperate in a process which is designed to find hidden faults in their work and be able to react as necessary.

4.7 Summary

While software tested in this manner could still have flaws (absolute completeness in testing is not attainable), it is hoped that a high degree of reliability has been demonstrated and quality greatly improved. Testing has occurred throughout the product's evolution, so that fixes could be installed early, reducing the effort and risk involved in making changes. Much test redundancy has been provided, reducing the risks of erroneous testing and improving completeness (e.g. functional and structural testing at several levels). Much descriptive documentation has been produced to illustrate reliability measures and provide assistance with maintenance. Overall, an effective test activity should be the result of this strategy.

Chapter 5

Implementation of Testing

In order to implement a test activity from the comprehensive test strategy, a test plan must be generated. The test plan applies the strategy test principles to a real environment. The environment defines tool availability, resource allocation, development scheduling and personnel available, as well as product dependent characteristics, such as reliability requirements, and complexity. Figure 5-1 lists the documents outlined in Appendix C which implement a test plan. In general the test activity can not be fully defined until the product design stage is completed, as the test plan and test specifications will call out individual component tests. The test strategy is used to guide the selection of tests and the means of managing test data within the project.

5.1 Test Definition

An initial test plan should be generated at product inception. This plan can define the administrative aspects of testing and the product test goals. It also plans the specification testing phase. The plan should be open ended, so that as design proceeds the tests to be used for design

and code verification can be included. In addition to generating the tests mandated by the strategy, other tests might also be incorporated. As an example, mutation testing might be specified as a requirement for determining test coverage and generating additional test cases. Such additions could be helpful if a particularly critical product was being developed. This allows the application of the strategy to be dynamic and comprehensive.

IMPLEMENTATION DOCUMENTATION

Test Plan

Test Specification

Test Logs

Test Incident Reports (Error Reports)

Test Incident Resolution Reports

Test Summary Report

Figure 5-1

Application of the test strategy requires the definition of tests for each software structure. Each mode of testing on each structure must be described in a test specification. These documents are generated at the beginning of each components' development and later have appended to them test cases which designate the inputs and outputs

which should occur, as well as provide an execution flow. Test lists or scripts (NCR, 1980) are one acceptable means of specifying tests.

While executing tests, the events that occur must be recorded. A test log (IEEE, 1981) provides such a media. Events include satisfactory test case completion, problems or variances encountered in the test method, and software bugs detected. As problems are encountered they must be resolved, and any report of an incident, be it with the test method or the software under test, should have an associated resolution report. Upon completion, the execution documents provide a basis for a summary of the test which should report the findings of the test (the product is acceptable or not, and if not, its disposition), and analysis of the effectiveness of testing and recommendations regarding it.

5.2 Test Environment Issues

Rather than simply integrating a test activity based on the comprehensive strategy into an existing development process, some measures may be taken by management to optimize development. Several steps may be taken to provide a development environment which realizes the greatest benefits from the strategy.

One important step is the establishment of QA independence. While QA is still a part of product development, it is detrimental to product quality for the product developers to control QA. This is often the case when QA reports to the same manager as the developers. The tendency is often for the manager who is concerned with getting a product on the market to play down quality issues which delay his schedule. If QA is managed separately and at the same level as product development, quality or testing issues which can not be resolved must pass through a higher management level, permitting a more objective evaluation of quality concerns versus schedule concerns.

Another enhancement to development and testing is automated tools. An investment in effective tools can be very valuable in reducing the burden of testing, increasing test effectiveness, and maintaining tests. Automated test case generators and verifiers would greatly reduce the time required in these tasks and could be much more thorough than a human. Special cases could still be inserted manually as required. Automated test beds and data management systems could be very useful for executing the tests and reporting such things as test coverage, test results, and test criteria satisfaction. An almost essential tool for

structural test analysis is a flow tracer, which, coupled with a flow analyzer, could automate the verification of structural test results and test coverage. In addition, automated tools could generate reports on the tests. The problems with most tools at this time is that they are still at the edge of computer technology. In many cases the use of tools is still very theoretical, and if available at all, application is very limited. In the future, however, it is expected that some very powerful tools will be available.

Changes in the development process may also be desirable. In chapter 3, the design test was shown to follow a top down order, while integration tests are bottom up. If product design also follows a top down order, design verification can begin almost immediately and closely follows design. The verification process can even help supply information to lower levels of design. Use of the top down flow here produces an early overall design which is needed to complete project planning and minimize the lag time until the design is completely verified.

In integration, however, bottom up testing reduces the complexity of external code used only for testing and provides a building block approach to integration. To minimize the lag time of testing behind coding, a bottom up coding order should be established in the project plan.

5.3 Conclusion

By utilizing a strategy which provides systematic testing, the project development cycle and test activity can merge. This produces an improved development environment and order, not only beneficial to product design, but also to the production of high quality software.

In addition to test planning, associated software administrative plans which are beyond the scope of the test strategy, but relate to it, are also required. Administration includes such issues as project management (resource allocation), product configuration control, maintenance reporting, and release control (IEEE, 1980).

Chapter 6

Concluding Remarks

The characteristics of a test strategy required to fulfill the goals of testing as defined in chapter 1 are reliable, flexible, unambiguous, understandable, measurable, and retainable. This strategy provides for these characteristics and, thus, meets the goals of early error detection, complete error detection, complex structure revelation, encouragement of good structure and understandability, provision for debugging, maintenance capability, user confidence, improvement of testing, and compatibility with diverse development environments. Although the ultimate success of testing is dependent on test implementation, this strategy provides guidelines for all the elements essential to successful testing.

While many of the ideas presented in this test strategy have been theorized or practiced by experts on testing, the strategy as a whole is untried. A subsequent step should be its application to a live development project. In this way its applicability to generating a test activity may be tried, and the merit of the testing principles determined. Deficiencies in the strategy may be corrected, and it is possible that some specialization of the strategy

towards specific environments may occur. This would make the strategy less general in applicability but would add to its detail for a given project. It is even possible that a catalog of strategies, each geared toward a certain type of project, could be developed.

Appendix A

Glossary

In order to add consistency to this report, the usage of some standard terms is defined below. These are terms having a generalized usage which may vary slightly in literature. The usage in this report is a composite of the varied usages or is a referenced usage of the term.

Acceptance Testing - Testing to insure that the product compares favorably with the initial requirements and needs of the user (Myers, 1979).

Certification - Approval by a supplier representative that a product meets the supplier's quality standards and is available for delivery.

Code - A pattern, or group of patterns, of programming statements which are written to produce some behavior.

Code Segment - A sequential string of programming statements which are always executed together and contain no branches.

Complete - Contains all the possibilities of interest.

Complexity - The deviation from simplicity. This refers to understandability and size scale as well as the intricacy.

Domains - The sets of values acceptable as inputs of the product.

Driver - "A small module that must be coded to drive or transmit test cases through the module under test" (Myers, 1979, p. 89).

Error - A deviation from desired or correct behavior.

Fidelity - Faithfulness to expected software behavior.

Flow Decision - A branch, based on a conditional clause, which determines the flow path segments.

Flow Path - A chain of code segments which are sequentially executable.

Incremental Testing - An approach to testing in which components are combined in a step by step manner with previously tested components for testing purposes (Myers, 1979).

Interface - The mating portion of software components which is used to give or receive information and control.

Module - A separately compilable group of flow paths which produces a function or group of functions (NCR, 1978).

Module Testing - Testing of product modules to verify their correctness.

Mutation Testing - Effectiveness verification and generation of test cases through execution of test cases on software versions in which errors have been inserted.

Program - A group of one or more modules which performs a specific task or tasks.

Quality - The adherence of a product to its desired behavior and standards.

Ranges - The sets of values acceptable as outputs of the product.

Regression Testing - The retesting of portions of the product previously completed for the purpose of determining that changes have not caused unwanted side effects (Metzger, 1973).

Reliability - A measure of quality which determines how faithfully and consistently the desired behavior is produced.

Robustness - The ability of software to perform correctly under varied conditions.

Scale - The quantity or size of software objects. This includes such characteristics as number of lines of code, number of components, and amount of documentation required.

Software Component - A substructure of a software product which is uniquely identifiable, such as a module or unit.

Software Volume - The quantity of code in terms of statement lines, object instructions, and documentation.

Specification - A description of the details and of requirements of an object (NCR, 1978).
Specifications may apply to the following objects:

Requirements - Describes user needs and constraints.

Product - Describes characteristics of the product.

Components - Describes requirements and characteristics of product modules and units.

Stub - A component which simulates the functions of another component for the purpose of testing components externally utilizing those functions.

System - A group of software components (normally units) which cover the entire spectrum of an individual computer's applications.

Test - A set of data inputs, executed in a certain set of procedures, which determines correctness.

Test Activity - The overall process of testing which includes the testers, tests, and the object of the test.

Test Case - A set of input conditions which hopefully will expose an error when executed on by the product. Several techniques may be used to generate test cases such as follows:

Boundary Value Analysis - Selects test cases based on input and output proximity to domain and range limits.

Cause/Effect Graphing - Selects test cases based on a functional mapping of inputs to outputs.

Error Guessing - Selects test cases based on intuitive and historical error possibilities.

Test Coverage Measures - A criteria for determining the extent to which a set of test cases exercise or test the product. Some of the common measurement standards defined by Software Research Associates are defined below (Software Research Associates, 1981 pp. 3-5).

"C0	Execute all statements in a program."
"C1	Execute all segments in a program."
"C2	C1 and also one exterior and an upper and lower interior point."
"Cik	C1 plus one test for each iteration $i=1,2,\dots,K$ times."
"S0	Invoke all modules at least once."
"S1	All invocations to modules exercised at least once."
"S2	All invocations to modules for each possible value of logical expression (actual) parameters."

Test Level - A subset of a test activity which accomplishes one stage of software verification.

Test Script - A procedural description of how to execute a test. It contains information as to what actions must be taken, in what order and what the expectations are (Metzger, 1973).

Thorough - The degree to which all applicable cases are examined.

Unit - A grouping of software modules which perform a specific task or group of related tasks.

Unit Testing - Testing of product units to verify their correctness.

Appendix B

Comprehensive Test Strategy

Definition

This test strategy provides an integrated testing approach which encompasses the entire product development process. It outlines what testing is to be done, in what manner, what general test standards must be met, what records are required, where responsibility (the tester) lies, and what the purpose of each stage of testing is (IEEE, 1981).

In many stages of the plan, complimentary techniques of testing (Myers, 1979) are specified and various test levels are called for. Each level is to be carried out with each test technique. For example, in the coding/debugging phase, module, module integration, unit integration, and certification testing is to be carried out under each of the test techniques: analysis, path testing, and functional testing. This does not preclude combining tests where applicable as long as all tests are still provided. The levels build upward. After path testing at the module level, the unit level need not repeat the internal paths of the modules comprising it. The unit test must only

check the paths to and from the modules to insure their correctness. The same applies to the integration testing at a later time and to the other techniques.

All test records are to be kept in a project library section, so that test satisfaction records are available, as well as test cases which may be reused as necessary. Each test will have a detailed specification which implements the tests and specifies test cases and details of execution.

B.1 Specification Phase

A. Test Methods.

1. Static Analysis

- a. Proof that the product specification meets user requirements.

B. Test Levels.

1. Acceptance (Customer/Supplier agreement).

C. Test Order.

1. N/A.

D. Test Type.

1. Nonincremental.

E. Documentation.

1. Descriptive record showing that assertions about the specification fulfill product objectives.
2. Signed agreement on product definition between supplier and customer (users).
3. Record of all changes and corrections following signoff.

F. Responsibility.

1. Team of supplier and customer management and technical personnel.

G. Accomplishment.

1. Clearly defined contractual product specification.

B.2 Design Phase

A. Test Methods.

1. Static Analysis.
 - a. Verification that design specifications fulfill the product specification.
 - i. Test Cases - each requirement and constraint specified (including performance). S1 logic coverage of interfaces.

- ii. Completion Criteria - verification
that each interface is correct; all
functions provided within constraints.

B. Test Levels.

- 1. Module.
- 2. Unit.
- 3. Integration.

C. Test Order.

- 1. Parallel at module level.
- 2. Top down at module integration level.
- 3. Top down at unit integration level.

D. Test Type.

- 1. Nonincremental at module level.
- 2. Incremental at module integration level.
- 3. Incremental at unit integration level.

E. Documentation.

- 1. Descriptive record showing that each requirement
and constraint is satisfied.
- 2. Descriptive record or chart (HIPO) showing the
meshing of interfaces and combination of lower
level functions to produce more complex functions
which satisfy specified requirements and
constraints.

3. Record of errors found in product specification and corrections made.
4. Record of errors and corrections in design following completion of component designs.
5. Test plan and test specifications (may not include test cases and processes).

F. Responsibility.

1. Project leaders and developers at module level.
2. Project leaders, developers, quality assurance, and management team at module and unit integration level.

G. Accomplishment.

1. Verification that design will satisfy specified objectives.
2. Verification that design will fit together.
3. Verification that design structure is good.
4. Set of software component specifications (design specifications).
5. Sub specification of each process and data item to be implemented to produce overall objectives of product.

B.3 Coding/Debugging and Integration Phase

A. Test Methods.

1. Static Analysis.

- a. Code Inspection - desk check of structure, understandability and functionality.
- b. Code Walkthrough - verify algorithms.
 - i. Test Cases - basic inputs which cause normal execution of algorithms; special inputs which algorithm treats as exceptions; special inputs under which execution is doubtful (error guessing).
 - ii. Completion Criteria - verification that algorithms are correct in normal and exception conditions and as many user conditions as practical.

2. Path Testing.

- a. Execution of every path through product and component structure, as determined from code structure.
 - i. Test Cases - inputs to execute each path (including repetitive loops). C1 coverage at module level, S2 coverage at module and unit integration level.

Inputs to execute modular paths through each unit. Must select cases which are at boundaries of input domains and cases where results may be expected to be more likely to fail. Also determine paths which limit performance.

- ii. Completion Criteria - correct selection and execution of every path. Execution of paths critical to performance must be within limits allowed by design.

3. Functional Testing.

- a. Execution of every function provided by software, as determined from design specifications.
 - i. Test Cases - inputs which cause all functions of each component to be exercised. Inputs which exercise complex functions of units. Must include inputs both within and outside of valid input domain limits from both directions (boundary value analysis). Also values which execute in a manner which may be particularly error prone (error guessing).

- ii. Completion Criteria - successful verification of correct results for each specified function.

B. Test Levels.

- 1. Module.
- 2. Module Integration.
- 3. Unit Integration.
- 4. Certification.
 - a. On completion, components and product are certifiable as functionally correct.

C. Test Order.

- 1. Parallel at module level.
- 2. Bottom up at module integration level.
- 3. Bottom up at unit integration level.
- 4. Parallel at certification level.

D. Test Type.

- 1. Nonincremental at module level.
- 2. Incremental at module integration level.
- 3. Incremental at unit integration level.
- 4. Incremental at certification level.

E. Documentation.

- 1. Descriptive records of walkthrough.

2. Tables of test cases versus observed and expected results for path and functional tests, and test procedures.
3. Record of all bugs found and corrected in debugging.
4. Record of errors and corrections to product specification.
5. Record of errors and corrections to design.
6. Record of errors and corrections to coded components following debug completion.

F. Responsibility.

1. Module developers and QA representative at module and module integration levels.
2. Module developers, project leaders, and QA representative at unit integration level.
3. QA at certification level.

G. Accomplishment.

1. Certified product and components. Verification of functional and operational correctness.

B.4 Delivery Phase

A. Test Methods.

1. Functional Testing.
 - a. Verification by user.

- i. Test Cases - boundary value analysis.
Error guessing, sample data runs.
- ii. Completion Criteria - satisfaction of
all objectives under constraints (including performance) specified in Phase I.
X hours of error free operation.

B. Test Levels.

- 1. Acceptance.

C. Test Order.

- 1. N/A.

D. Test Type.

- 1. Nonincremental.

E. Documentation.

- 1. Acceptance approval by customer.
- 2. Record of latent errors found.

F. Responsibility.

- 1. User technical personnel.

G. Accomplishment.

- 1. Product delivery.

B.5 Maintenance Phase

A. Test Methods.

- 1. Static Analysis.
 - a. Proof of design and specification changes.

- b. Interface and algorithm change walkthroughs.
 - c. Code inspection to generate new test cases.
- 2. Path Testing.
 - a. Of modules and units changed.
 - b. Of completed system modification.
 - c. Test Cases - previous test cases as required with appropriate changes to test alterations.
 - d. Completion Criteria - successful selection and traversal of new and altered paths as well as original paths.
- 3. Functional Testing.
 - a. Of module, unit, and system functionality.
 - b. Test Cases - previous test cases with changes as required.
 - c. Completion Criteria - successful execution of all functions affected.
- B. Test Levels (Regression Levels).
 - 1. Module.
 - 2. Module Integration.
 - 3. Unit Integration.
 - 4. Certification.
 - 5. Acceptance.

- C. Test Order.
 - 1. Same as other stages.
- D. Test Type.
 - 1. Same as other stages.
- E. Documentation.
 - 1. Same as other stages.
- F. Responsibility.
 - 1. Same as other stages.
- G. Accomplishment.
 - 1. Acceptance of change definitions, verification of design changes, certification of changes, and delivery of changes.

Appendix C
Implementation Documentation

- C.1 Test Plan (IEEE, 1981; Myers, 1979)
 - A. Identification of Software Product.
 - B. Identification of Goals of Test Activity.
 - C. Identification of Tests to be Performed.
 - 1. Components.
 - 2. Levels.
 - 3. Development stages where applicable.
 - D. Schedule of Tests.
 - 1. Test ordering based on strategy.
 - E. Organizational Structure.
 - 1. Controlling organization.
 - 2. Assign test responsibility.
 - F. Criteria for Judging Testing Complete and Product Acceptance.
 - G. Administrative Issues.
 - 1. Change control and error tracking.
 - 2. Resource requirements.
 - a. Equipment requirements.
 - b. Time requirements.
 - c. Tool requirements.

3. Risk analysis and contingency plans.
4. Project management issues.

C.2 Test Specification

- A. Identification of Test Goal.
- B. Identification of Test Method (s).
- C. Identification of Test Coverage (s).
- D. Identification of Test Completion Criteria.
- E. Listing of Test Cases.
 1. Inputs.
 2. Conditions.
 3. Results and activities.
 4. Identifications of features tested.
- F. Execution Process.
 1. Test set up.
 2. Test initiation.
 3. Test continuance.
 4. Error recovery.
 5. Test termination.

C.3 Test Log

- A. Identification of Test.
 1. Component.

- 2. Level.
- 3. Development stage.
- B. Data & Time Event Occurred.
- C. Event Description (Test Started, Error Logged, Power Failure, etc.).

C.4 Test Incident Report

- A. Identification of Test.
 - 1. Component.
 - 2. Level.
 - 3. Development stage.
- B. Data & Time of Incident.
- C. Identification of Incident.
 - 1. Incident nature.
 - 2. *Component effected.
 - 3. *Origin of incident - component and development stage.
 - 4. *Cause of incident.

*May not be available until incident resolved.

C.5 Test Incident Resolution Report

- A. Attached to Incident Report.
- B. Identification as to Software Error, Equipment Failure, Test Failure, or Operator Failure.

- C. Identification of Cause of Error and when it Occurred.
- D. Description of Error Correction or Resolution.

C.6 Test Summary Report

- A. Identification of Software Product.
- B. Summary of Results of Tests.
- C. Evaluation of Tests.
 - 1. Summary of errors detected (e.g. how many errors).
 - 2. Summary of completeness (e.g. all completion criteria satisfied or variances and justification).
- D. Evaluation of Product - Pass/Fail.
- E. If Failed, Definition of Additional Requirements or Final Disposition.

Bibliography

- Fairley, R. E. Colorado State University Software Engineering Videotape Course. Ft. Collins, Colorado: Colorado State University, 1982.
- Metzger, Phillip W. Managing a Programming Project. Englewood Cliffs, N.J.: Prentice-Hall, 1973.
- Myers, Glenford J. The Art of Software Testing. New York, N.Y.: Wiley and Sons, Inc., 1979.
- NCR Customer & Support Education Corporate Education. Programming Project Management Course. Dayton, Ohio: NCR Corporation, 1980.
- NCR Fundamental English Dictionary. Dayton, Ohio: NCR Corporation, 1978.
- Aviziensis, A.; Gilley, G. C.; Mathur, F. P.; Rennels, D. A.; Rohr, J. A.; and Rubin, D. K. "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design", IEEE Transaction on Computers, Vol. C-20, No. 11 (November, 1971), pp. 1312-1321.
- Bowen, John B. "A Survey of Standards and Proposed Metrics for Software Quality Testing", Computer, Vol. 12, No. 8 (August, 1979), pp. 37-42.
- Demillo, Richard A.; Lipton, Richard J.; and Sayward, Frederick G. "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, Vol. 11, No. 4 (April, 1978), pp. 34-41.
- Geiger, Werner; Gmeiner, Lothar; Trauboth, Heinz; and Voges, Udo. "Program Testing Techniques for Nuclear Reactor Protection Systems", Computer, Vol. 12, No. 8 (August, 1979), pp. 10-18.
- Goodenough, John B. and Gerhart, Susan L. "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2 (June, 1975), pp. 156-173.

Howden, William E. "Functional Program Testing", IEEE Transactions of Software Engineering, Vol. SE-6, No. 2 (March, 1980), pp. 162-169.

_____. "Introduction to Software Validation", Tutorial: Software Testing and Validation Techniques, (1978).

_____. "Introduction to the Theory of Testing", Tutorial: Software Testing and Validation Techniques, (1978).

_____. "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3 (September, 1976), pp. 38-45.

Huang, J. C. "An Approach To Program Testing", Computing Surveys, Vol. 7, No. 3 (September, 1975), pp. 113-128.

Miller, Edward. "Introduction to Software Testing Technology", Tutorial: Software Testing and Validation Techniques, (1978).

Sorkowitz, Alfred R. "Certification Testing: A Procedure to Improve the Quality of Software Testing", Computer, Vol. 12, No. 8 (August, 1979), pp. 20-24.

Zelkowitz, Marvin V. "Perspectives on Software Engineering", Computer Surveys, Vol. 10, No. 2 (June, 1978), pp. 197-216.

NCR World Magazine, Vol. 17, No. 3 (June/July, 1982), NCR Corporation.

Software Engineering Technical Committee of the IEEE Computer Society. Draft American National Standard For Software Quality Assurance Plans, ANSI/IEEE Std. 730 (January, 1980).

Draft Standard For Software Test Documentation, IEEE Inc., Revised October 15, 1981.

Technical Note Summary of Software Testing Measures, TN-843, Software Research Associates, May 1, 1981.

IEEE - Software Requirements Guideline, Rough Draft, IEEE Inc., July 17, 1981.

A COMPREHENSIVE SOFTWARE TEST STRATEGY

by

STEPHEN LOUIS KAHLE

B. S., University of Missouri at Columbia, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

Abstract

This report introduces a test strategy which is applicable to testing in the generalized software development process. It is comprehensive in that it is adaptable to any product development project, and it covers all phases of development. The strategy provides testing guidelines to provide a standardized, systematic product test activity, which should improve the quality of software produced.

Within the report are presented some of the basic foundational concepts from which the test strategy is conceived, as well as the strategy itself. The motivation behind high quality software is described in order to develop a list of desirable strategy traits. A survey of testing theories and methods is then presented to provide the practices useful within the strategy. This is followed by a discussion of pertinent aspects of product development. The methods are then applied to the development process in a manner which produces the desired strategy. In concluding the report, some comments on how the strategy may be used to implement testing are included.

The strategy, as presented, provides for testing over the entire product life. If properly implemented, in

accordance with the strategy, this testing should be a major contributor towards software reliability. This makes the strategy valuable both in the areas of public welfare and development economics, areas which are becoming increasingly critical to software production.