ED 192 757 IR 008 828

AUTHOR	Branstad, Martha A.; And Others
TITLE	Computer Science and Technology: Validation, Verification, and Testing for the Individual Programmer.
INSTITUTION	National Bureau of Standards (DOC), Washington, D.C. Inst. for Computer Sciences and Technology.
PUB DATE	Feb 80
NOTE	26p.
AVAILABLE FROM	Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402 (Stock No. 003-003-02159-8, $1.75).
EDRS PRICE	MF01/PC02 Plus Postage.
DESCRIPTORS	*Computer Programs; Design; *Evaluation Methods; *Evaluation Needs; *Formative Evaluation; Guidelines; *Program Development; Program Evaluation; Programing; Program Validation; Testing

ABSTRACT

Guidelines are given for program testing and verification to ensure quality software for the programmer working alone in a computing environment with limited resources. The emphasis is on verification as an integral part of the software development. Guidance includes developing and planning testing as well as the application of other verification techniques at each lifecycle stage. Relying upon neither automated tools nor formal quality assurance support, the guidelines should be appropriate for applications programmers doing small development projects. A brief bibliography is provided as an entry into the field of verification, validation, and testing. (Author/BK)

COMPUTER SCIENCE & TECHNOLOGY:

VALIDATION, VERIFICATION, AND TESTING FOR THE INDIVIDUAL PROGRAMMER

Martha A. Branstad
John C. Cherniavsky
W. Richards Adrion

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

U.S. DEPARTMENT OF COMMERCE, Philip M. Klutznick, Secretary
Luther H. Hodges, Jr., Deputy Secretary
Jordan J. Baruch, Assistant Secretary for Science and Technology
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, Director

Issued February 1980

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publicaton.

National Bureau of Standards Special Publication 500-56
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-56, 26 pages (Feb. 1980)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 80-600005

U. S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1980

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Stock No. 003-003-02159-8 Price $1.75
(Add 25 percent for other than U.S. mailing).

TABLE OF CONTENTS

Page

1. Introduction 1

2. Verification Through the Life Cycle 3

3. Testing 6

4. Requirements 9

5. Design 11

6. Construction 13

7. Operation 18

8. Summary and General Rules 18

References 21

Bibliography 22

VALIDATION, VERIFICATION, AND TESTING FOR THE INDIVIDUAL PROGRAMMER

Martha A. Branstad
John C. Cherniavsky
W. Richards Adrion

Guidelines are given for program testing and verification to insure quality software for the programmer working alone in a computing environment with limited resources. The emphasis is on verification as an integral part of the software development. Guidance includes developing and planning testing as well as the application of other verification techniques at each lifecycle stage. Relying upon neither automated tools nor formal quality assurance support, the guidelines should be appropriate for applications programmers doing small development projects.

Key words: Testing; Program Verification; Software Development.

1. Introduction

Testing, validation and verification of software is a difficult and arduous task even for a manager with automated tools and sufficient people to devote to the task. This report is intended as a guideline to those who are developing programs and software with limited resources. It is aimed at the production of quality software through the use of sound verification methods, techniques, and planning. The most common resource limited environment is that of the single programmer working alone on a development project.

The domain of the solitary programmer is unique and warrants a specialized guide to verification and testing. Neither the problems nor the solutions that exist for the development of medium or large computer systems apply. Problems of coordinating several programmers do not exist in this domain nor do massive integration efforts. All management is done by the programmer. No independent internal or external quality assurance groups exist. In addition, the problem is usually of a size which is intellectually manageable.

The verification and validation approaches used for medium and large systems are not always applicable to the environment of the single programmer. For instance, tool development is stressed for large and medium systems. For very small program development, time and cost restrictions do not allow for specialized tool development. Although this report assumes limited resources, tools which exist locally should not be ignored. A skilled craftsman uses the best tools available.

This report concentrates on inexpensive verification and testing techniques to assure the quality of work. For a more comprehensive treatment of verification techniques and tools the reader is referred to a forthcoming NBS Special Publication [ADRI80].

Since we believe verification should be performed at every development stage, this report is organized around the life cycle stages given in Figure 1.

REQUIREMENTS	DESIGN	CONSTRUCTION	OPERATION & MAINTENANCE

Figure 1. Life Cycle Stages for Software Development

Section 2 introduces verification techniques and planning which should be employed throughout the life cycle. Section 3 discusses testing in general and why it is difficult. Section 4,5,6, and 7 discuss verification and testing during the requirements, design, construction, and operation stages, respectively. Section 8 summarizes and presents a collection of general rules.

The following definitions of common terms are used in this document. It should be noted that some of these terms may appear with slightly different meanings elsewhere in the literature.

1. VALIDATION: determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements. Validation is usually accomplished by verifying each stage of the software development lifecycle.

2. CERTIFICATION: acceptance of software by an authorized agent usually after the software has been validated by the agent, or after its validity has been demonstrated to the agent.

3. VERIFICATION: in general the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development lifecycle. Requirements are verified; successive iterations of the design specifications are verified internally, both against the requirements and the earlier iterations; modules are unit tested and verified against the design specifications; and the system undergoes intensive verification during integration at the construction stage. For small programs, verification is the process of determining the correspondence between a program and its specification. Static analysis, dynamic analysis, and program testing are used during verification.

4. TESTING: examination of the behavior of a program by executing the program on sample data sets.

5. PROOF OF CORRECTNESS: use of techniques of logic to infer that an assertion assumed true at program entry implies that an assertion holds at program exit.

6. PROGRAM DEBUGGING: the process of correcting syntactic and logical errors detected during coding. Debugging occurs before the program or module is fully executable and continues until the programmer feels the program or module is sound and executable. Testing is then used to exercise the code over a sufficient range of test data to verify its adherence to the design and requirements specifications.

2. Verification Through the Life Cycle

Problem solving is a creative process that follows a general paradigm. The first step is to carefully define the problem to be solved. Next a solution is hypothesized. The trial solution is then scrutinized and exercised to determine if it works. Based upon the results of the third step, modifications are made to the problem statement or to the trial solution. The revised solution is again exercised and scrutinized. Many iterations may occur before the problem solver is satisfied with the solution. When this state occurs, the solution is prepared in final form and final testing is performed.

If the original problem is so difficult that no trial solution comes immediately to mind, it should be transformed into a new problem as follows:

a. Identify the problem as a particular case of a problem whose solution is known.

b. Solve a specific instance of the problem in the hope that it will lead to the general solution.

Solution of this new problem proceeds as above.

Computer programming is a form of problem solving. Its solution paradigm is called the development life cycle and is organized into stages. There are many life cycle charts in the literature,[NBS76] [DOD77] but no one current chart is appropriate for all views of the development process. The one given in Figure 1 is representative. During the requirements stage, the problem to be solved is carefully defined. The design stage is when general solutions are hypothesized and data and process structures are organized. During construction the program modules are coded and debugged. The modules are integrated and the interfaces debugged. Testing begins as the program is exercised over the test data selected during this and earlier stages. The program is used and maintained during the final stages.

Some significant differences are apparent between the problem solution paradigm described in the first paragraphs and the development life cycle described above. The life cycle activity is expressed as a straight line solution whereas the previous paradigm emphasized iteration toward a solution. Anyone who has ever programmed knows that iteration is essential. Iteration is the result of the verification process and error assessment. Unless the correct problem is stated and the correct solution achieved at the first try, modification and iteration occur.

Verification should accompany each stage of the development life cycle. If the verification process is isolated in a single stage then problem statement errors or design errors discovered at that stage may exact an exorbitant price. Not only must the original error be corrected but the structure built upon it must be changed also.

Viewing each development stage as a sub-problem leads to a more productive paradigm for program development. The amended life cycle chart in Table 1 presents the verification activities that accompany each development stage.

Table 1. Life Cycle Verification Activities

Life Cycle Stage	Verification Activities
Requirements	Determine Correctness Generate Functional Test Data
Design	Determine Correctness and Consistency Generate Structural and Functional Test Data
Construction	Determine Correctness and Consistency Generate Structural and Functional Test Data Apply Test Data
Operation & Maintenance	Retest

At each stage the verification activities should include:

1. Determine the correctness and consistency of the structures produced at the stage, and

2. Generate test data based upon structures introduced at that stage.

For the design and construction stages it is also necessary to:

3. Determine that the structures are consistent with those at the previous stage, and

4. Refine and redefine test sets generated earlier.

Performing the above activities at each development stage should help to locate errors when they are introduced and will also partition the test set construction.

3. Testing

Great strides have been made toward the development of formal verification techniques. These techniques, based on ideas of formal semantics and proof techniques, while being promising research avenues are not easily applied without supporting tools (verifiers). Currently automated verifiers are expensive, not widely available, and limited in application. For the single programmer, testing is the most easily applied verification technique. Testing is, as we will discuss, limited in its ability to demonstrate correctness. Testing shows _the presence_ of errors, and generally (excluding exhaustive testing) cannot demonstrate _the absence_ of errors.

One view of a program is as a representation of a function taking elements from one set (called the domain) and transforming them into elements of another set (called the range). The testing process is then used to ensure that the implementation (the program) faithfully realizes the function. Since programs are frequently given inputs that are not in the domain, they should also act reasonably on such elements. Thus a program which realizes the function 1/x should not fail catastrophically when x=0, but instead should generate an error message. We call elements within the functional domain _valid_ inputs and those outside the domain _invalid_ inputs.

The goal of testing is to reveal errors not removed during debugging. The testing process consists of obtaining a valid value from within the functional domain or an invalid value from outside the functional domain, determining the expected (correct) value, running the program on the given value, observing the program's behavior, and finally comparing that behavior with the expected (correct) behavior. If the comparison is successful, the result of the testing process has revealed no errors. If the comparison is unsuccessful, then through the testing process the errors are revealed.

The key words in the paragraph above are _expected (correct) behavior_, _observation_, and _comparison_. An important phase of testing lies in planning how to apply test data, how to observe the results, and how to compare the results with desired behavior. Applying and observing tests are not always straightforward activities. Often extensive analysis is required to determine tests which adequately test design components, and often code must be instrumented to provide observation. Determining the desired (correct) behavior for comparison with observed results is very difficult. To test a program, we must have an "oracle" to provide the correct responses and to represent the desired behavior. This is a major role of a requirements specification. By providing a complete description of how the system is to respond to its environment, a good requirements specification may form a basis for constructing such an oracle. In the future, executable requirements languages may provide this capability directly, but currently they are still basic research topics, consequently we must be content with more ad hoc techniques. Some typical ones include:

1. Intuition.

2. Hand calculation.

3. Simulation, both manual and automated.

4. An alternate solution to the same problem.

Although we have been discussing testing through examples at the coding level, our intent is to be general enough so that the discussion of the validation methods applies to any stage in the program's life cycle. Testing in its narrowest definition is performed during the construction stage, and then later during operation and maintenance as revisions are made. Derivation of test data and test planning, however, are activities which should cover the entire life cycle.

If a broader meaning of testing is used, subsuming more of the verification process, then testing is an important activity in each life cycle stage. Simplified walkthroughs, code reading, and most forms of requirements and design analysis can be thought of as testing procedures. Each of these will be discussed in later sections.

The main problem with testing is that it reveals only the presence of errors. A complete validation of a program can be obtained only by testing for every element of the domain. Since this process is exhaustive, finding the presence of no errors guarantees the validity of the program. This technique is the only dynamic analysis technique with this guarantee. Unfortunately, it is not practical. Frequently program domains are infinite or so large as to make the testing of each element of the domain infeasible. There is also the problem of deriving, for an exhaustive input set, the expected (correct) responses. Such a task is at least as difficult as (and possibly equivalent to) writing the program itself. The goal of a testing methodology is to reduce the potentially infinite exhaustive testing process to a finite testing process. This is done by choosing representative elements to exercise features of the problem under solution or of the program written to solve the problem.

A subset of the domain used in a testing process is called a test data set. Thus the crux of the testing problem is finding an adequate test data set, one that covers the domain and yet is small enough to use. This activity must be planned and carried out in each life cycle stage. Sample criteria for the selection of test data for test sets include:

1. The test data should reflect special properties of the domain such as extremal or ordering properties or singularities.

2. The test data should reflect special properties of the function that the program is supposed to implement such as domain values leading to extremal function values.

3. The test data should "exercise" the program in a specific manner, e.g., causing all branches to be executed or all statements to be executed.

The properties that the test data sets are to reflect are classified according to whether they depend upon the program's internal structure or the function the program is to perform. In the first two cases above, the test data reflect functional properties and in the latter case structural properties. Structural testing helps to compensate for the inability to do exhaustive functional testing.

While criteria for a test set to be adequate in a structural sense are often simple to state (such as branch coverage), the satisfaction of those criteria can usually only be determined by measurement. Due to the lack of analytical methods for deriving test data to satisfy structural criteria, most structural test sets are obtained using heuristics.

For functional analysis techniques, the major current difficulties are in the specification of such vague terms as extremal or exceptional value. Further, for some functional analysis techniques, it may not be possible to obtain a functional description from a requirement or specification statement. Thus once again a substantial amount of effort in test set construction for functional considerations is of a heuristic nature.

The general concepts of testing will be discussed below as they apply to each life cycle stage. Keep in mind throughout the discussions that planning is the key to successful testing.

4. Requirements

INVEST IN ANALYSIS AT THE BEGINNING OF THE PROJECT

Why bother with formal requirements for a small program written by a single programmer? The answer is that an investment in careful analysis at the very beginning of the project can reap benefits even for the programmer working alone. Having a clear, concise statement of the problem to be solved will facilitate construction, communication, error analysis, and test data generation.

What should be included in the requirements or problem statement? The following list suggests the information that should be recorded and the decisions that should be made at this stage in the development.

1. Functionally, what the program is to do.

2. What the input will be like such as the form, format, data types, and units for the input.

3. The form, format, data types, and units for the output.

4. How exceptions, errors, and deviations are to be handled.

5. For scientific computations, the numerical method or at least the required accuracy of the solution.

6. The computer environment required or assumed, e.g. the machine, operating system, and implementation language.

Making and recording decisions on the issues listed above are only part of what should be done during the requirements stage of development. In addition, the core of the test data set should be established and error analysis should be performed.

START DEVELOPING THE TEST SET AT THE REQUIREMENTS STAGE

The requirements state what the program is to do. Data should be generated which will determine if the requirements have been met. To do this the input domain should be partitioned into classes of values that the program will treat in a similar fashion. For each class a representative element should be included in the test data. Boundary values, elements at the edge of the input class, frequently require special treatment by the program and are thus a likely source of error. Therefore, boundary values for each class should be included in the test data set. In addition, any non-extremal input values that will require special handling by the program should be included in the test data set. Output classes should also be covered by input which causes output at each class boundary and within each class. Invalid input values require the same analysis provided for valid values.

For all elements in the test data set expected values should be determined. That is, how the program should treat each of the elements should be decided and recorded. The test data set and accompanying expected values form the core of the test set that will be refined to use with the implementation code. The test set generated during the requirements stage will address the functional aspects of the program. Data to test the structural aspects will be generated during the design and implementation stages.

Generating test data at this stage also serves another useful purpose, that of insuring that the requirements are testable. Requirements for which it is impossible to define test data or to determine the expected value for the test data are ineffective requirements and should be reformulated.

14

ARE YOU SOLVING THE CORRECT PROBLEM?

Generating the test set is only part of the verification activity that should accompany the requirements stage. Analysis for the correctness, consistency , and completeness of the requirements should also be done. Whether the correct problem is being solved should be considered. Perhaps a more general solution or a more specific solution would be more advantageous. The requirements should be carefully checked for conflicts and inconsistencies. The possibility of missing cases should be considered. Have functions been omitted that should have been included? Too frequently the results of such analysis aren't obtained until the program is executed with test data. At the implementation stage the cost of requirements modification is high and the rework is painful. Therefore, error analysis early in the development cycle is advocated.

5. Design

WRITE AN EXPLICIT DESIGN STATEMENT

Even for small projects, the programmer should spend the time and energy to produce an explicit statement of the design, for such a document will aid construction, communication, error analysis, and test data generation. The requirements and design documents together should describe the problem to be solved and organization of the solution. They should convey enough information so that the reader can determine what the program is to do and how it is to do it without having to resort to code reading.

Several different design techniques and methodologies have received wide coverage recently [EDP79]. Whether you use one of those or your own technique, the following information should be included in the design.

1. Principle data structures should be specified since the way key data are organized frequently shapes the structure of the entire program.

2. Functions, algorithms, heuristics, or special techniques used for processing should be recorded.

3. The basic program organization should be stated. How the program is to be sub-divided or modularized and the internal and external interfaces should be specified.

4. Additional information may be needed for particular projects.

Verification activity is very important in the design stage. If faults are not found until after the program is coded, fixes are often more costly and less satisfactory. Verification at the design stage follows the steps listed in section 2.

A. The design should be analyzed to determine if it is complete and consistent. The individual algorithms, heuristics, and special techniques should be checked to see if they really work for the given problem and data. This could involve hand calculations for representative test cases. The total process should be analyzed to determine that no steps or special cases have been overlooked. The module interfaces should be examined to assure that calling routines provide the information and environment required by the called routines and in the form, format, and units assumed. The data structure should be examined for inconsistencies or awkwardness. Input/output handling should be carefully analyzed for it is a frequent source of error.

B. The design should be analyzed to determine if it satisfies the requirements. Determine if all constraints specified by the requirements have been met. Determine if the design assumes the same form, format and units for input and output that are stated in the requirements. Check that all functions listed in the requirements have been included in the design. Selected test data generated during the requirements should be hand simulated to determine if the design would produce the expected values.

C. Test data based upon the design should be generated. During the design stage data structures, functions, algorithms, heuristics, and the general program structure have all been established; data to test these structures and functions should also be generated. Standard, extremal, and special values for the data structures should be included in the test data set. Boundary value analysis should be applied to both the structure and the values of the data structures. For example, if array size can vary, then a single element array, a maximum size array, and special valued arrays should be candidates for the test data set. Test data for externally visible functions has been generated at the requirements stage. For the internal functions introduced during design, test data should also be generated. The input and output analysis suggested in Section 4 should be used to generate the test data for these functions. If not already accomplished by the existing test data, new tests should be generated to exercise the modular structure of the design. For all the data in the test data set, expected values should be calculated. The test data generated during the design stage should test both the structure and the internal functions of the design.

D. The test set generated during the requirements stage should be re-examined in view of the design. Since additional decisions have been made during design, it is possible that data and expected values generated during requirements can be refined and made more exact.

Many programmers and managers want to rush directly to the coding stage when they begin a new project. However, the time spent in careful analysis during the first two life cycle stages can increase the quality of the total project. Following the above four steps will greatly facilitate program verification. Steps one and two should also be carried out by a colleague; specific techniques will be discussed in Section 6. It is so difficult to find your own errors that an independent analysis is strongly advocated.

5. Construction

The construction or coding stage is what most people think of when they think of programming. However, if the requirements and design have been carefully done, the coding should be straightforward, almost mechanical. Since so much has been written about software engineering, structured programming, and various coding techniques, little will be said here except that we assume the programmer is using good programming techniques [ZELK79].

CHECK FOR CONSISTENCY WITH DESIGN

The first step in verification during the construction stage is to determine if the code is consistent with the design. Both code and design should exhibit the same modular structure and have the same module interfaces. Both should utilize the same data structures and implement the same functions using the same algorithms. Input/output handling in the code should be consistent with the decisions made during the design stage.

KEEP YOUR TESTS

Testing should be performed in an organized and systematic manner. Test runs should be dated, annotated, and saved. Not infrequently testing is slow and ineffective because it is performed in a random manner. The programmer manufactures test data on the spot and runs tests as the spirit moves him using his memory as the only record of what has been previously tested. Such random testing will produce random results. The systematic development of the test set through the requirements, design, and construction stages helps to produce an adequate test set; however, the actual execution of the program using the test data must also be performed in an orderly manner. A plan or schedule of which code pieces or modules are to be tested and in what order can be used as a check list to help the programmer organize his efforts. All test data and runs should be saved after being dated and suitably annotated. If errors are found and changes made to the program, retesting must be performed. Not only must the given test be rerun, but also any tests previously run and passed that involve the erroneous segment must be rerun since the program modifications may have invalidated the previous tests.

ASK A COLLEAGUE FOR ASSISTANCE

During the construction stage, code exists and testing can begin. However, additional manual analysis of the code should precede the first test runs. Desk checking, in which a programmer sits at a desk reading his code, looking for errors, is not a particularly effective verification technique. Usually programmers are poor at finding their own errors. Fundamental logic errors and missing cases elude the creator/programmer for if it seemed right when it was designed, it still seems right. As for syntactic errors, the programmer tends to see what was meant rather than what was coded.

Inspection, an exercise in disciplined error hunting, and walk-through,a form of manual simulation, are formalized manual techniques based upon desk checking. In both techniques a team of programmers examine the code (or design or requirements) and look for errors in a very organized manner.

For small programs, a technique partway between desk checking and inspections or walk-throughs seems appropriate. An independent party, usually a colleague, should be asked to analyze the development product at each stage, the requirements, the design, and the code. The programmer should explain the product to his colleague, while the colleague plays devil's advocate, questioning the logic and searching for errors. A check list of likely errors should be used to guide the search. The second party aspect of this technique is essential to its success for a new perspective is needed to locate the errors the programmer cannot perceive.

USE AVAILABLE TOOLS

At last the code is ready to be run. The programmer should be familiar with the various compilers and interpreters available on his system for the language he is using. If more than one exists, they undoubtedly differ in their error analysis and code generation capabilities. Some processors do syntax checking only with no code generation; others perform extensive error checking such as array bound checking and provide aids such as cross reference tables. If debugging compilers exist on your system, it is very helpful to use them.

APPLY STRESS TO THE PROGRAM

Testing should exercise and stress the program structure, the data structures, the internal functions, and the externally visible functions or functionality. Both valid and invalid data should be in the test set. The test set, which consists of test data and expected values generated during the requirements and design stages, forms the core of the test set to be used during construction. Additional data may be required to test structural and functional details visible only in the code. The test set generated during the first two life cycle stages may require refinement or redefinition in order to be applied to the code. The test data should be organized to conform with the general test organization to be used during construction. Intermediate values may be required in order to test incomplete pieces of code or individual modules. Testing requires as much creativity as does design.

TEST ONE AT A TIME

As with many tasks, divide and conquer is the rule for testing. Pieces of code, individual modules, and small collections of modules are exercised separately before they are integrated into the total program. Only after the pieces have been found error-free are they joined together, one at a time, and tested as a unit. It is easier to isolate errors when the number of potential interactions is kept small.

In order to test the smaller units of code, some additional code may be required. If the testing is done bottom-up, then drivers will need to be written. For small programs this may only entail producing code to call a procedure so the interface and control transfer can be tested. If construction is proceeding top-down, then stubs will be needed. In this case incomplete or dummy called routines must be constructed so the flow of control and argument passage can be tested before the entire program has been constructed. Often there is a reluctance with small programs to want to generate any "unnecessary" code solely for testing. However, it is wise to organize the testing so that small pieces of code are exercised and deemed error-free before aggregating them into ever larger units.

Instrumentation, the insertion of code into the program solely to measure various program characteristics, can be useful for program verification. For medium and large scale projects tools are often acquired or developed which will do automatic instrumentation. For small projects the programmer can do his own instrumentation. Array bound checks, checking of loop control variables, determining if key data values are within permissible ranges, tracing the execution, and counting the number of times a group of statements is executed are examples of the types of analysis that can be performed using instrumentation.

ARE YOU EVER THROUGH?

How do you know when you have tested enough? That's a fundamental question that unfortunately has no clear cut answer. If you are still finding errors everytime you execute your program, then testing should continue. As a matter of fact, errors tend to cluster, so those modules that appear particularly error prone should receive special scrutiny. If you have run your complete test set against the program and have found no more errors, it does not mean that your program is error free. Perhaps your test set is incomplete. The metrics that tend to be used to measure testing thoroughness include: statement testing, branch testing, and path testing. Statement testing determines if each statement in the program has been executed at least once. Branch testing determines if each exit from each branch in the program has been executed at least once. Path testing determines if all logical paths through the program, which may involve repeated execution of various segments, have been executed at least once. Each succeeding metric subsumes the others. Since the number of paths grows exponentially with the number of decision points, path testing tends to be too costly in time and money to utilize and often impossible to do. Statement testing , although known to be theoretically insufficient, is the coverage metric most frequently used because it is relatively simple to implement.

If no dynamic analysis tool that measures test coverage is available on your system, it is a straight forward exercise to instrument your own program to calculate statement coverage. Identify each program point into which control can be transferred and establish an array with as many slots as there are transfer points. At each transfer point insert a statement which will increment the appropriate slot in the array. The final values in the array can be printed at program exit. This technique will not only provide an indication of test coverage but will also show the portions of your program with the heaviest usage. Statement testing will not guarantee your program is correct; however, it is a testing minimum that is frequently used for if code has never been exercised, it is difficult to know if it is error free.

It should be emphasized that the amount of testing will depend upon the cost of an error. Critical programs or code segments will require more thorough testing than more insignificant functions do.

7. Operation

Why talk about testing during operation? For many production programs 50 to 85% of the total life cycle costs are accrued during the maintenance stage that is coincident with operation. These costs do not imply error studded programs that take forever to fix but evolving programs that are constantly undergoing modification. Even for small programs corrections, modifications, and extensions are bound to occur. Any time there is modification or change, testing is required. Testing during maintenance is termed regression testing. The systematic approach recommended in the rest of the report facilitates regression testing since much of the work should already have been done. The test set, test plan, and test results for the original program should exist. Modification to accommodate the program changes must be done and then all portions of the program affected by the modifications must be retested. After regression testing is complete, the program and test documentation must be updated to reflect the changes. To ignore the need to maintain consistent and current documentation on both programs and completed tests is to insure increasing instability of the programs at each successive modification.

8. Summary and General Rules

We have proposed a general approach to developing software in an environment with limited resources. This approach stresses that verification must take place throughout the development process. Even for relatively small, noncritical projects lifecycle planning for software quality is very important. A programmer's task of demonstrating that he has produced reliable and consistent code can be much simplified by beginning early to develop and plan verification activities.

The approach we recommend relies heavily on testing as the main verification technique with code reading and inspection techniques employed as supporting activities. Testing cannot be performed as an after-the-fact activity. Test data sets are derived throughout the development process beginning at the requirements stage. It is at the first stage that test data can be derived from functional analysis of the system requirements. Inconsistency in the requirements is often detected during this analysis. During the design stage, test data which stress the program control and data structures are generated. When the final code generation and construction stage begins, the test data set should be nearly complete. The remaining test data should be chosen to exercise those "special functions" used by the programmer in the actual implementation. Care should be taken to insure that the data and control structures represented by the final programs are covered as extensively as possible by the test data.

The actual testing should proceed in an organized manner with small pieces being tested individually, then aggregated and retested. Test coverage metrics are very useful for determining a "confidence level" for the test data. Such metrics can be implemented by instrumenting your own code. Since testing is a process aimed at discovering errors, it must be noted that for each design or code modification resulting from such a discovery, retesting is necessary. The process becomes an iterative one towards a goal of achieving a high level of confidence in the finished product through intelligent testing.

Inspection, close reading and analysis of the code, and manual simulation are important verification techniques which should also be used throughout development. Thorough analysis and hand simulation are employed during both requirements and design in order to discover errors early in the development. Each stage must be analyzed for consistency with prior stages and for internal consistency. You should seek independent review of your work by having a colleague inspect your design and code for errors. We believe that quality can be improved through planning, a staged development, and the use of sound verification techniques at each stage.

The following list summarizes the approach presented in this report.

1. Generate test data at all stages and in particular the early stages.

2. Develop a means for calculating the expected values for test data to compare with test results.

3. Inspect requirements, design, and code for errors and for consistency.

4. Be systematic in your approach to testing.

5. Test pieces and then aggregate.

6. Save, organize, and annotate test runs.

7. Concentrate testing on modules that exhibit the most errors and on their interfaces.

8. Retest when modifications are made.

9. Discover and use available tools on your system.

References

[ADRI 80] ADRION, W. R.; BRANSTAD, M. A.; and CHERNIAVSKY, J. C., forthcoming National Bureau of Standards, Institute for Computer Science and Technology Special Publication, _Verification and Validation_, 1980.

[DOD77] Department of Defense, _Automated Data Systems Documentation Standards_, Standard 7935.1-S, September 1977.

[EDP79] "Program Design Techniques," EDP ANALYZER, Vol. 17 No. 3, Canning Publications, Inc., March 1979.

[NBS76] National Bureau of Standards, _Guidelines for Documentation of Computer Programs and Automated Data Systems_, Federal Information Processing Standards Publication 38, February 1976.

[ZELK79] ZELKOWITZ, M. V., "Perspectives on Software Engineering," _ACM Computing Surveys_, Vol. 10 No. 2 (June 1978), 197-216.

Bibliography

This collection is meant to provide the reader with an entry into the field of verification, validation, and testing. Each document lists further references and an annotated bibliograghy on validation appears in [YEH77].

[GOOD75] GOODENOUGH, J. B.; and GERHART, S., "Toward a Theory of Test Data Selection," _IEEE Transactions on Software Engineering_ 1,2 (June 1975), 156-173.

[HETZ73] HETZEL, W. C. (Ed.), _Program Test Methods_, Prentice-Hall, Engelwood Cliffs, N.J., 1973.

[KERN78] KERNIGHAN, B. W.; and PLAUGER, P. J., _The Elements of Programming Style_, McGraw-Hill, New York, 1978.

[MILL78] MILLER, E.; and HOWDEN, W. E., _Tutorial: Software Testing and Validation Techniques_, IEEE Computer Society, Long Beach, CA, 1978.

[MYER79] MYERS, G. J., _The Art of Software Testing_, John Wiley & Sons, New York, 1979.

[YEH77] YEH, R. T. (Ed.), _Current Trends in Programming Methodology_ Vol. 2, Prentice-Hall,Engelwood Cliffs, N.J., 1977.