UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

Software Testing

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/testing.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1. Software Testing Techniques

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test01.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.1 Testing Fundamentals

1.1.1 Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A good test is one that has a high probability of finding an as yet undiscovered error.
- A successful test is one that uncovers an as yet undiscovered error.

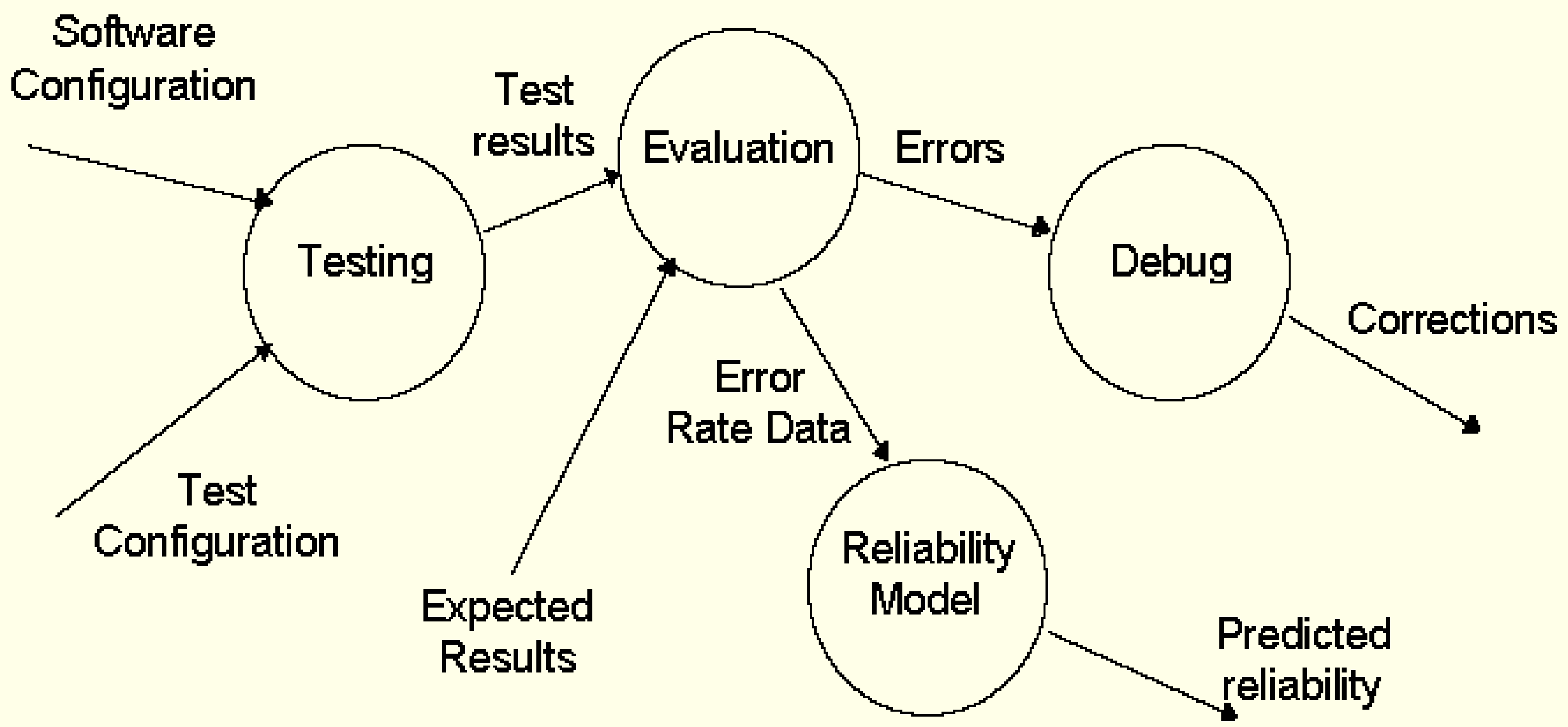
The objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.

Secondary benefits include

- Demonstrate that software functions appear to be working according to specification.
- That performance requirements appear to have been met.
- Data collected during testing provides a good indication of software reliability and some indication of software quality.

Testing cannot show the absence of defects, it can only show that software defects are present.

1.1.2 Test Information Flow



Notes:

- Software Configuration includes a Software Requirements Specification, a Design Specification, and source code.
- A test configuration includes a Test Plan and Procedures, test cases, and testing tools.
- It is difficult to predict the time to debug the code, hence it is difficult to schedule.

1.1.3 Test Case Design

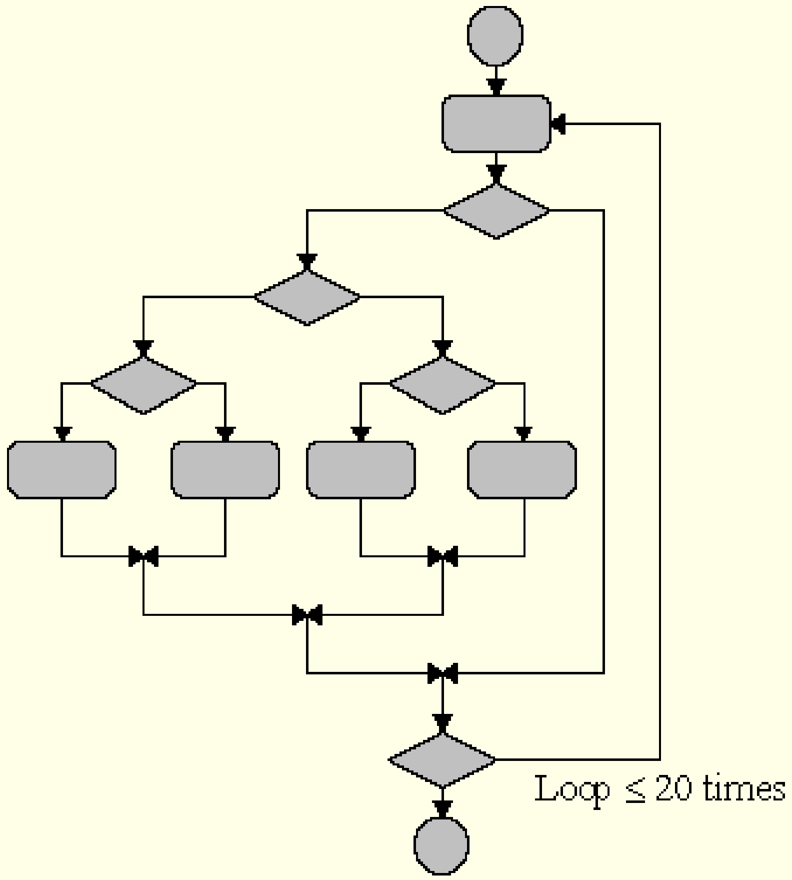
Can be as difficult as the initial design.

Can test if a component conforms to specification - Black Box Testing.

Can test if a component conforms to design - White box testing.

Testing can not prove correctness as not all execution paths can be tested.

Example:



A program with a structure as illustrated above (with less than 100 lines of Pascal code) has about 100,000,000,000,000 possible paths. If attempted to test these at rate of 1000 tests per second, would take 3170 years to test all paths.

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test01_1.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.2 White Box Testing

Testing control structures of a procedural design.

Can derive test cases to ensure:

1. all independent paths are exercised at least once.
2. all logical decisions are exercised for both true and false paths.
3. all loops are executed at their boundaries and within operational bounds.
4. all internal data structures are exercised to ensure validity.

Why do white box testing when black box testing is used to test conformance to requirements?

- Logic errors and incorrect assumptions most likely to be made when coding for "special cases". Need to ensure these execution paths are tested.
- May find assumptions about execution paths incorrect, and so make design errors. White box testing can find these errors.
- Typographical errors are random. Just as likely to be on an obscure logical path as on a mainstream path.

"Bugs lurk in corners and congregate at boundaries"

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test01_2.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.3 Basis Path Testing

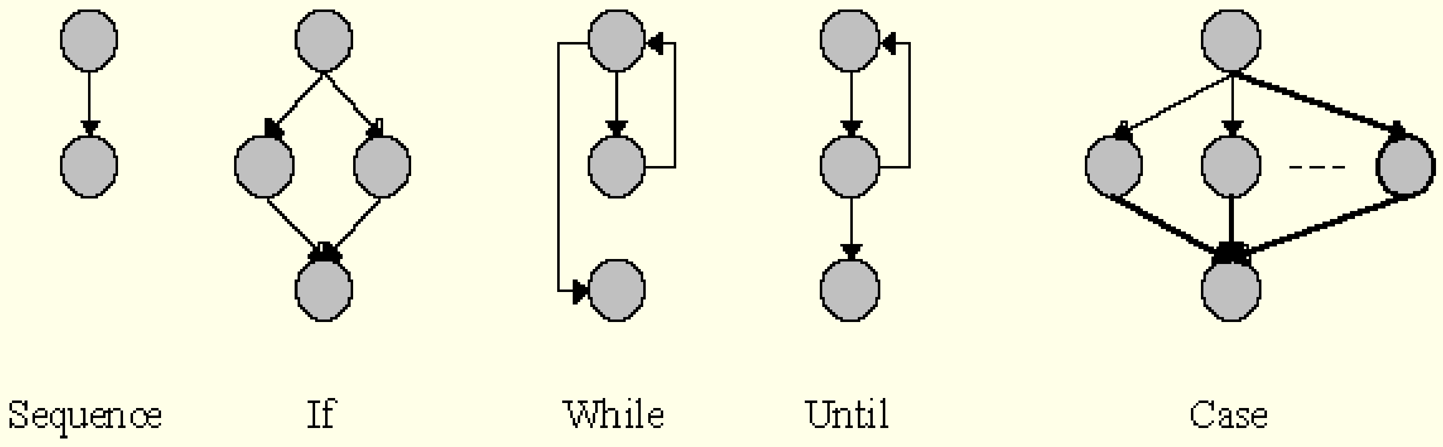
A testing mechanism proposed by McCabe.

Aim is to derive a logical complexity measure of a procedural design and use this as a guide for defining a basic set of execution paths.

Test cases which exercise basic set will execute every statement at least once.

1.3.1 Flow Graph Notation

Notation for representing control flow



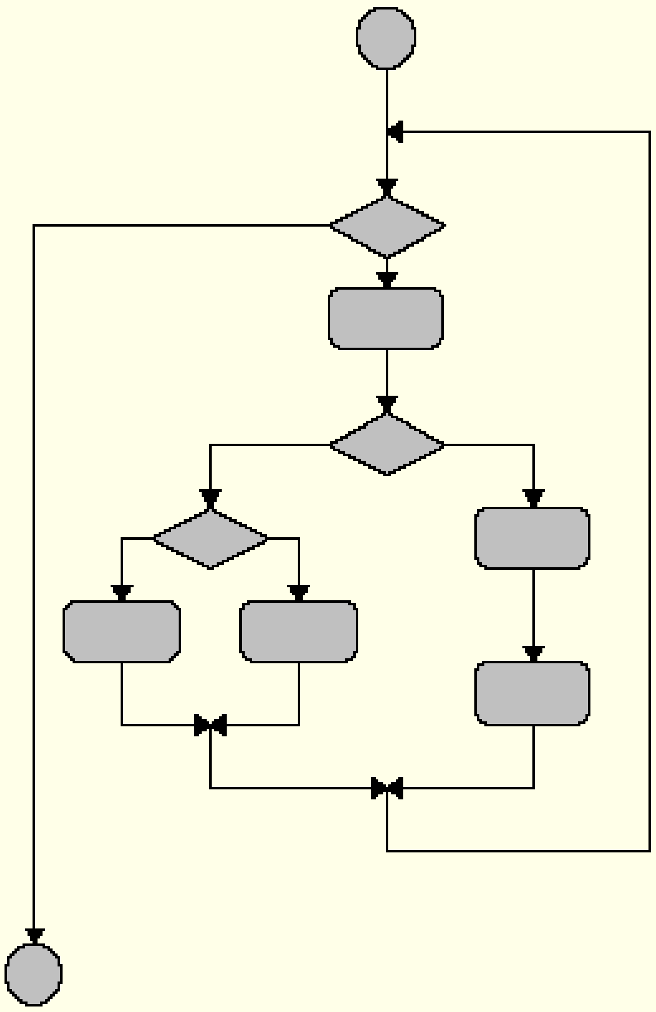
On a flow graph:

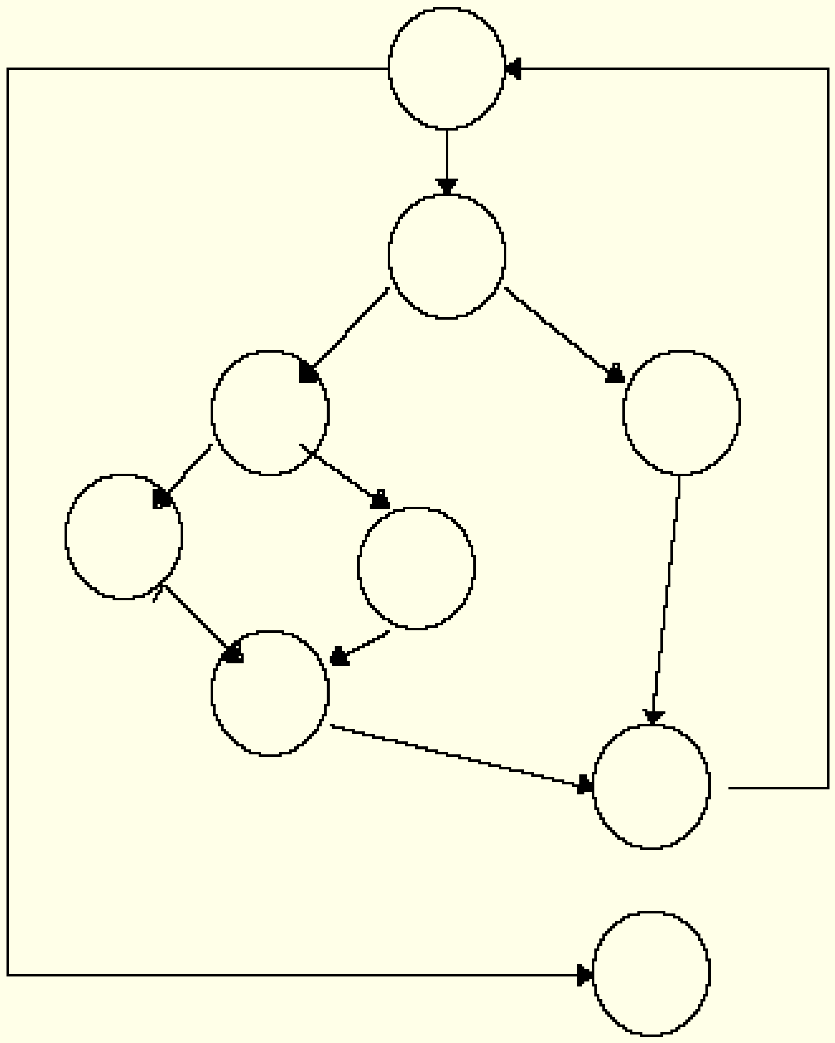
- Arrows called *edges* represent flow of control
- Circles called *nodes* represent one or more actions.
- Areas bounded by edges and nodes called *regions*.
- A *predicate node* is a node containing a condition

Any procedural design can be translated into a flow graph.

Note that compound boolean expressions at tests generate at least two predicate node and additional arcs.

Example:





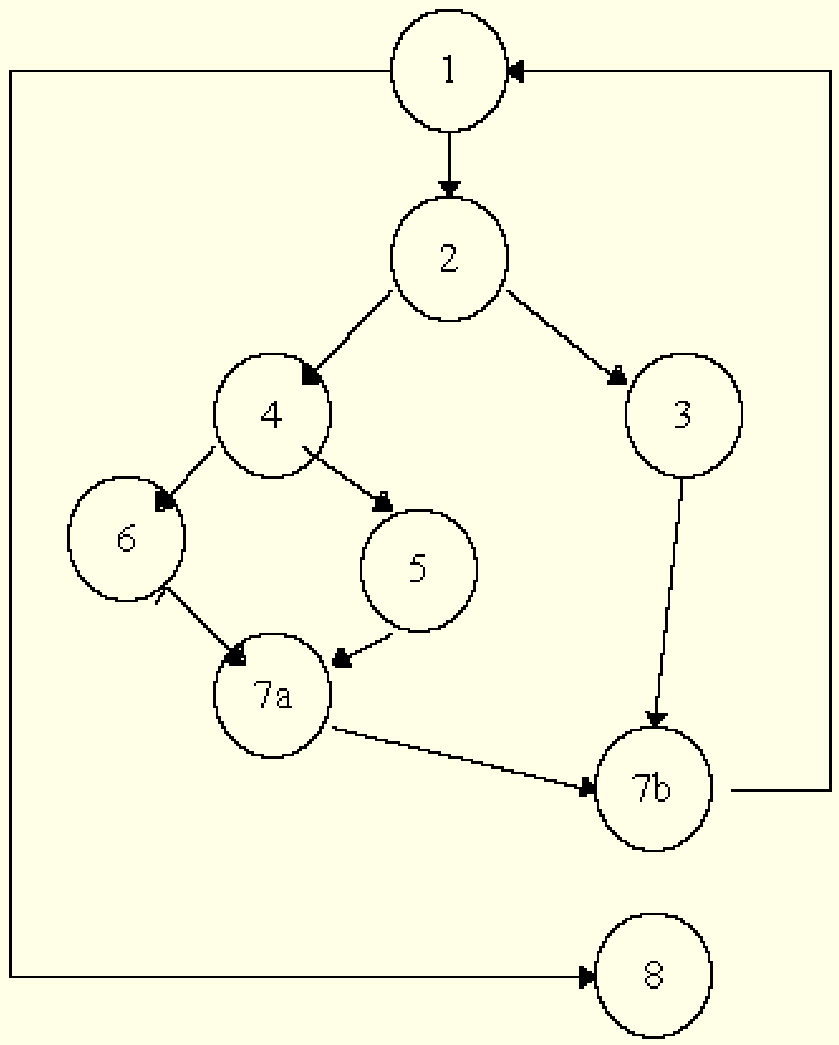
1.3.2 Cyclomatic Complexity

The cyclomatic complexity gives a quantitative measure of the logical complexity.

This value gives the number of independent paths in the basis set, and an upper bound for the number of tests to ensure that each statement is executed at least once.

An independent path is any path through a program that introduces at least one new set of processing statements or a new condition (i.e., a new edge)

```
      PDL for PROCEDURE SORT
1:        do while records remain
              read record;
2:            if record field 1 = 0
3:                then process record;
                      store in buffer;
                      increment counter;
4:            elsif record field 2 = 0
5:                then reset record;
6:                else process record;
                  store in file;
7a:           endif
          endif
7b:       enddo
8:    end
```



Example has:

- Cyclomatic Complexity of 4. Can be calculated as:
  1. Number of regions of flow graph.
  2. #Edges - #Nodes + 2
  3. #Predicate Nodes + 1
- Independent Paths:
  1. 1, 8
  2. 1, 2, 3, 7b, 1, 8
  3. 1, 2, 4, 5, 7a, 7b, 1, 8
  4. 1, 2, 4, 6, 7a, 7b, 1, 8

Cyclomatic complexity provides upper bound for number of tests required to guarantee coverage of all program statements.

1.3.3 Deriving Test Cases

1. Using the design or code, draw the corresponding flow graph.
2. Determine the cyclomatic complexity of the flow graph.
3. Determine a basis set of independent paths.

4. Prepare test cases that will force execution of each path in the basis set.

Note: some paths may only be able to be executed as part of another test.

1.3.4 Graph Matrices

Can automate derivation of flow graph and determination of a set of basis paths.

Software tools to do this can use a graph matrix.

Graph matrix:

- is square with #sides equal to #nodes
- Rows and columns correspond to the nodes
- Entries correspond to the edges.

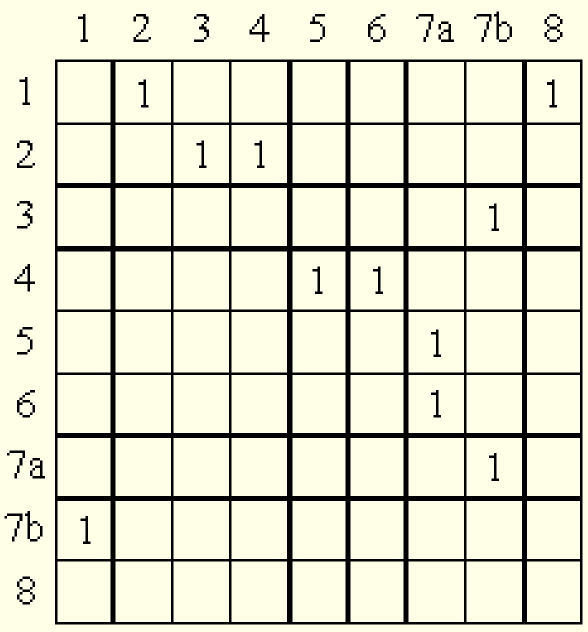
Can associate a number with each edge entry.

Use a value of 1 to calculate the cyclomatic complexity

- For each row, sum column values and subtract 1.
- Sum these totals and subtract 1.

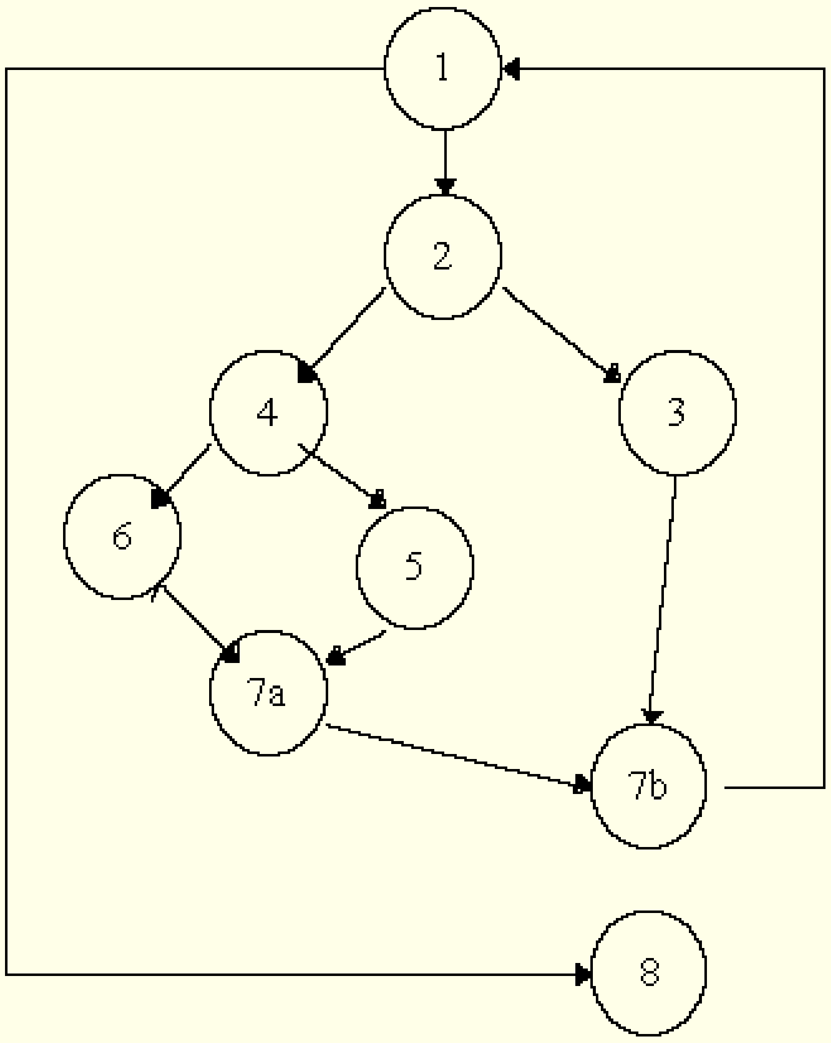
Some other interesting link weights:

- Probability that a link (edge) will be executed
- Processing time for traversal of a link
- Memory required during traversal of a link
- Resources required during traversal of a link



	1	2	3	4	5	6	7a	7b	8
1		1							1
2			1	1					
3								1	
4					1	1			
5							1		
6							1		
7a								1	
7b	1								
8									

Connection Matrix



This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test01_3.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.4 Control Structure testing.

Basic path testing one example of *control structure testing*.

1.4.1Conditions Testing

Condition testing aims to exercise all logical conditions in a program module.

Can define:

- *Relational expression*: (E1 op E2), where E1 and E2 are arithmetic expressions.
- *Simple condition*: Boolean variable or relational expression, possibly preceded by a NOT operator.
- *Compound condition*: composed of two or more simple conditions, boolean operators and parentheses.
- *Boolean expression*: Condition without relational expressions.

Errors in expressions can be due to:

- Boolean operator error
- Boolean variable error
- Boolean parenthesis error
- Relational operator error
- Arithmetic expression error

Condition testing methods focus on testing each condition in the program.

Strategies proposed include:

Branch testing - execute every branch at least once.

Domain Testing - uses three or four tests for every relational operator.

Branch and relational operator testing - uses condition constraints

Example 1: C1 = B1 & B2

- where B1, B2 are boolean conditions..
- Condition constraint of form (D1,D2) where D1 and D2 can be true (t) or false(f).
- The branch and relational operator test requires the constraint set {(t,t),(f,t),(t,f)} to be covered by the execution of C1.

Coverage of the constraint set guarantees detection of relational operator errors.

1.4.2 Data Flow Testing

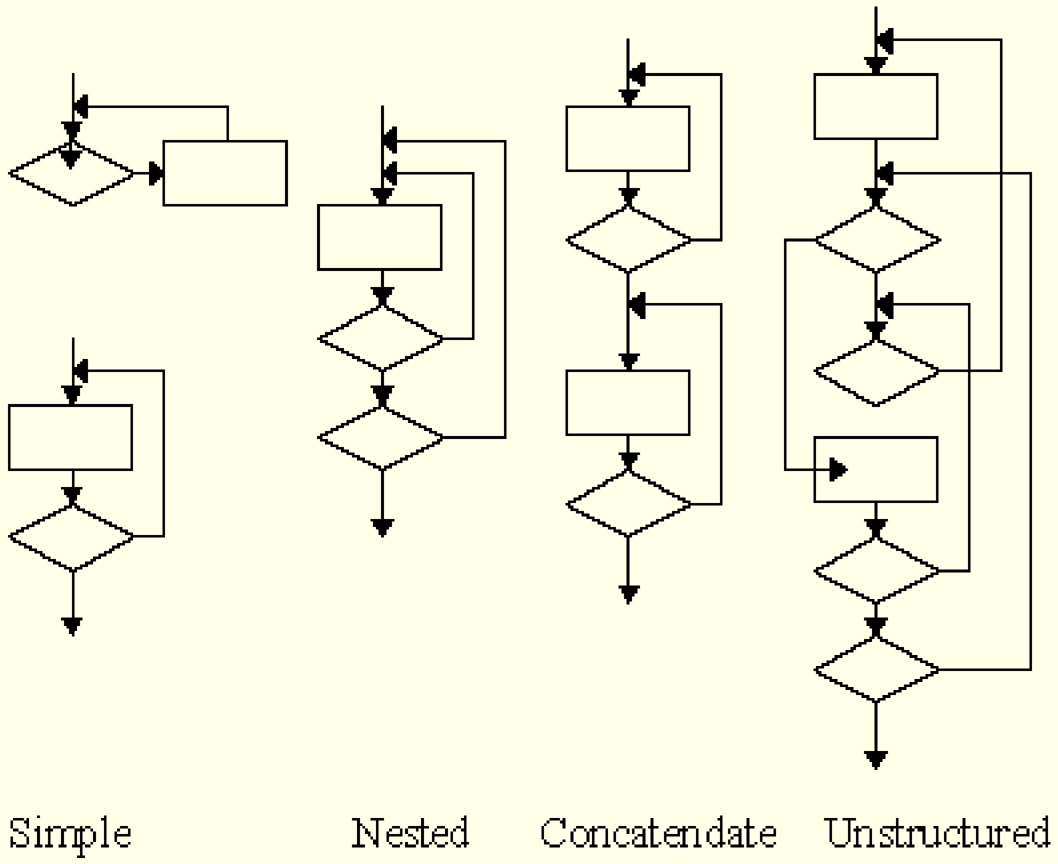
Selects test paths according to the location of definitions and use of variables.

1.4.3 Loop Testing

Loops fundamental to many algorithms.

Can define loops as simple, concatenated, nested, and unstructured.

Examples:



To test:

- **Simple Loops** of size n:
  - Skip loop entirely

- Only one pass through loop
- Two passes through loop
- m passes through loop where m<n.
- (n-1), n, and (n+1) passes through the loop.

- **Nested Loops**
    - Start with inner loop. Set all other loops to minimum values.
    - Conduct simple loop testing on inner loop.
    - Work outwards
    - Continue until all loops tested.
- **Concatenated Loops**
    - If independent loops, use simple loop testing.
    - If dependent, treat as nested loops.
- **Unstructured loops**
    - Don't test - redesign.

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.5 Black Box Testing

Focus on functional requirements.

Compliments white box testing.

Attempts to find:

1. incorrect or missing functions
2. interface errors
3. errors in data structures or external database access
4. performance errors
5. initialisation and termination errors.

1.5.1 Equivalence Partitioning

Divide the input domain into classes of data for which test cases can be generated.

Attempting to uncover classes of errors.

Based on equivalence classes for input conditions.

An equivalence class represents a set of valid or invalid states

An input condition is either a specific numeric value, range of values, a set of related values, or a boolean condition.

Equivalence classes can be defined by:

- If an input condition specifies a range or a specific value, one valid and two invalid equivalence classes defined.
- If an input condition specifies a boolean or a member of a set, one valid and one invalid

equivalence classes defined.

Test cases for each input domain data item developed and executed.

1.5.2 Boundary Value Analysis.

Large number of errors tend to occur at boundaries of the input domain.

BVA leads to selection of test cases that exercise boundary values.

BVA complements equivalence partitioning. Rather than select any element in an equivalence class, select those at the "edge' of the class.

Examples:

1. For a range of values bounded by a and b, test (a-1), a, (a+1), (b-1), b, (b+1).
2. If input conditions specify a number of values n, test with (n-1), n and (n+1) input values.
3. Apply 1 and 2 to output conditions (e.g., generate table of minimum and maximum size).
4. If internal program data structures have boundaries (e.g., buffer size, table limits), use input data to exercise structures on boundaries.

1.5.3 Cause Effect Graphing Techniques.

Translation of natural language descriptions of procedures to software based algorithms is error prone.

Example: From US Army Corps of Engineers:

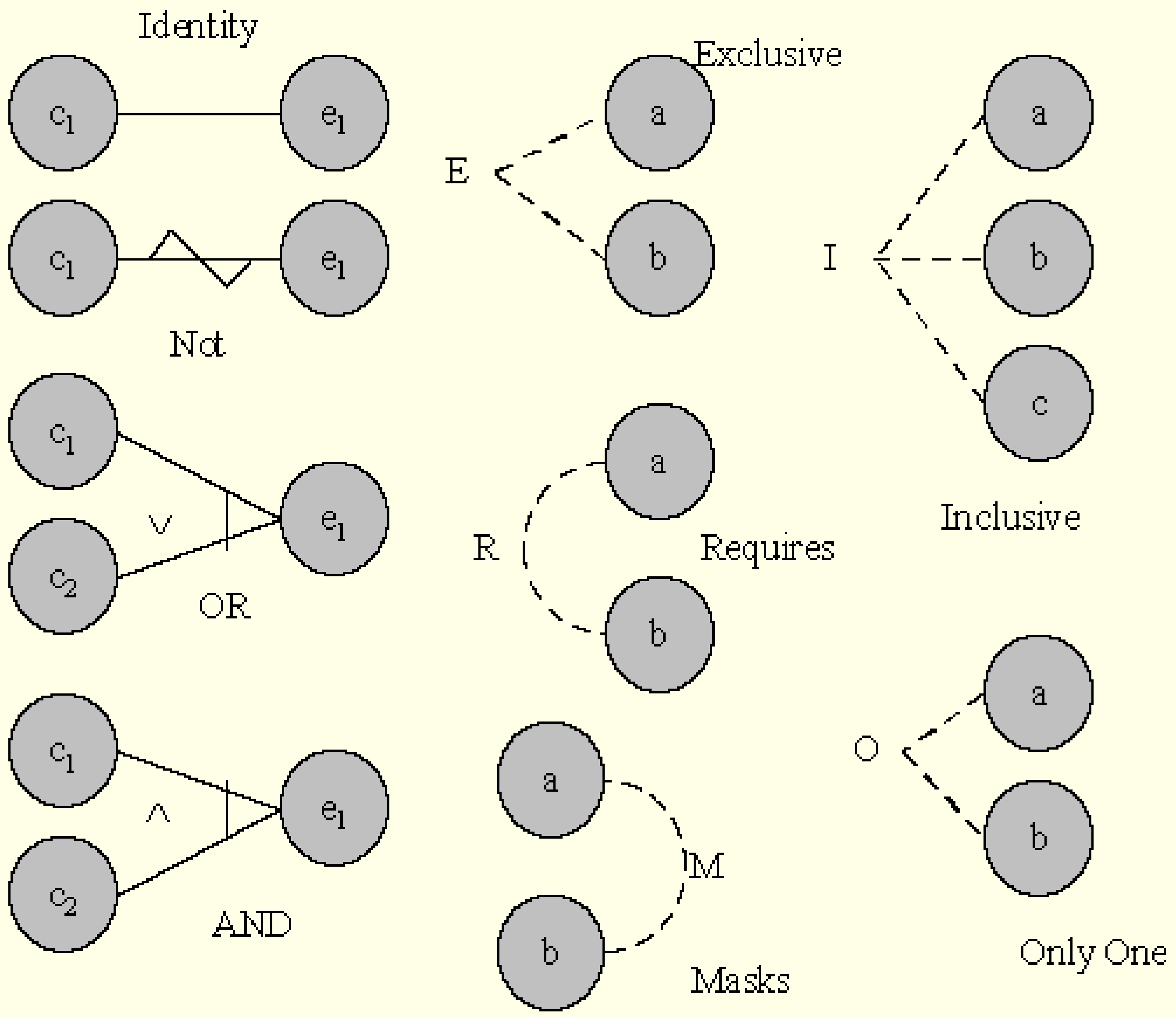
Executive Order 10358 provides in the case of an employee whose work week varies from the normal Monday through Friday work week, that Labor Day and Thanksgiving Day each were to be observed on the next succeeding workday when the holiday fell on a day outside the employee's regular basic work week. Now, when Labor Day, Thanksgiving Day or any of the new Monday holidays are outside an employee's basic workbook, the immediately preceding workday will be his holiday when the non-workday on which the holiday falls is the second non-workday or the non-workday designated as the employee's day off in lieu of Saturday. When the non-workday on which the holiday falls is the first non-workday or the non-workday designated as the employee's day off in lieu of Sunday, the holiday observance is moved to the next succeeding workday.

How do you test code which attempts to implement this?

Cause-effect graphing attempts to provide a concise representation of logical combinations and corresponding actions.

1. Causes (input conditions) and effects (actions) are listed for a module and an identifier is assigned to each.
2. A cause-effect graph developed.
3. Graph converted to a decision table.
4. Decision table rules are converted to test cases.

Simplified symbology:



1.5.4 Comparison Testing

In some applications the reliability is critical.

Redundant hardware and software may be used.

For redundant s/w, use separate teams to develop independent versions of the software.

Test each version with same test data to ensure all provide identical output.

Run all versions in parallel with a real-time comparison of results.

Even if will only run one version in final system, for some critical applications can develop independent versions and use *comparison testing* or *back-to-back testing*.

When outputs of versions differ, each is investigated to determine if there is a defect.

Method does not catch errors in the specification.

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).

Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test01_5.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.6 Static Program Analysis

1.6.1 Program Inspections

Have covered under SQA.

1.6.2 Mathematical Program Verification

If the programming language semantics are formally defined, can consider program to be a set of mathematical statements

Can attempt to develop a mathematical proof that the program is correct with respect to the specification.

If the proof can be established, the program is verified and testing to check verification is not required.

There are a number of approaches to proving program correctness. Will only consider axiomatic approach.

Suppose that at points P(1), .. , P(n) assertions concerning the program variables and their relationships can be made.

The assertions are a(1), ..., a(n).

The assertion a(1) is about inputs to the program, and a(n) about outputs.

Can now attempt, for k between 1 and (n-1), to prove that the statements between

P(k) and P(k+1) transform the assertion a(k) to a(k+1).

Given that a(1) and a(n) are true, this sequence of proofs shows partial program correctness. If it can be shown that the program will terminate, the proof is complete.

Read through example in text book.

1.6.3 Static Program Analysers

Static analysis tools scan the source code to try to detect errors.

The code does not need to be executed.

Most useful for languages which do not have strong typing.

Can check:

1. Syntax.
2. Unreachable code
3. Unconditional branches into loops
4. Undeclared variables
5. Uninitialised variables.
6. Parameter type mismatches
7. Uncalled functions and procedures.
8. Variables used before initialisation.
9. Non usage of function results.
10. Possible array bound errors.
11. Misuse of pointers.

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

1.7 Automated Testing Tools.

Range of tools may be available for programmers:

1. Static analyser
2. Code Auditors
3. Assertion processors
4. Test file generators
5. Test Data Generators
6. Test Verifiers
7. Output comparators.

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

2. Software Testing Strategies.

So far have considered testing for specific components.

How are the component tests organised?

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).

Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test02.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

2.1 A Strategic Approach to Testing.

Testing should be planned and conducted systematically. What are the generic aspects of a test strategy?

- Testing begins at the module level and works 'outward'.
- Different testing techniques are used at different points in time.
- Testing conducted by developer and (for larger projects) by an independent test group.
- Testing and Debugging are two different activities, but debugging should be incorporated into any testing strategy.

2.1.1 Verification and Validation.

Testing is part of verification and validation.

- Verification: Are we building the product right?
- Validation Are we building the right product?

V&V activities include a wide range of the SQA activities.

2.1.2 Organising for Software Testing.

Can use an independent test group to do some of the testing.

Developer do unit testing & most likely the integration testing

Developer & Independent test group both contribute to validation testing and system testing.

ITG become involved at the specification stage. Contribute to planning and specifying test procedures. May report to the SQA group.

2.1.3 A Software Testing Strategy

System development proceeds with steps:

1. System engineering
2. Requirements
3. Design
4. Coding

The testing is usually in the reverse order:

1. Unit testing
    - Module level testing with heavy use of white box testing techniques.
    - Exercise specific paths in the modules control structures for complete coverage and maximum error detection.
2. Integration Testing
    - Dual problems of verification and program construction.
    - Heavy use of black box testing techniques.
    - Some use of white box testing techniques to ensure coverage of major control paths.
3. Validation Testing
    - Testing of validation criteria (established during requirements analysis).
    - Black box testing techniques used.
4. System Testing
    - Part of computer systems engineering.
    - Considering integration of software with other system components.

2.1.4 Criteria for Completion of Testing.

When do you stop testing?

Two responses could be:

- Never. The customer takes over after delivery.
- When you run out of time/money.

Can use statistical modeling and software reliability theory to model software failures (as a function of execution time) uncovered during testing

One model uses a logarithmic Poisson execution time of the form:

$$f(t) = \left(\frac{1}{p}\right) \ln(l_0 p t + 1)$$

where:

- *t* is the cumulative testing execution time
- *f(t)* is the number of failures expected to occurring after testing for execution time *t*
- *l0* is the initial software failure intensity (failures per unit time)
- *p* is the exponential factor for the rate of discovery of errors.

The derivative gives the instantaneous failure intensity *l(t):*

$$l(t) = \frac{l_0}{l_0 pt + 1}$$

Can plot the actual error intensity against the predicted curve. Can use this to estimate testing time required to achieve a specified failure intensity.

2.2 Unit Testing

2.2.1 Unit Test Considerations

Can test:

1. interface
2. local data structures
3. boundary conditions
4. independent paths
5. error handling paths

Some suggested check lists for these tests:

Interface

- Number of input parameters equal to number of arguments?
- parameter and argument attributes match?
- parameter and argument units system match?
- parameters passed in correct order?
- input only parameters changed?
- Global variable definitions consistent across modules
- If module does I/O:
    - File attributes correct?
    - Open / Close statements correct?
    - Format specifications match I/O statements?
    - Buffer size match record size?
    - Files opened before use?
    - End of file condition handled?
    - I/O errors handled?
    - Any textual errors in output information?

Local Data structures (common source of errors!):

- Improper or inconsistent typing
- Erroneous initialisation or default values
- Incorrect variable names
- Inconsistent data types
- Overflow, underflow, address exceptions.

Boundary conditions - see earlier

Independent paths - see earlier

Error Handling:

- Error description unintelligible
- Error noted does not correspond to error encountered
- Error condition handled by system run-time before error handler gets control.
- Exception condition processing incorrect
- Error description does not provide sufficient information to assist in determining error.

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).

Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test02_2.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

2.3 Integration Testing

Can attempt *non-incremental integration* - putting everything together at once and test as a whole.

Usually a disaster.

Incremental Testing - integrate and test in small doses.

2.3.1 Top Down Integration.

Modules integrated by moving down the program design hierarchy.

Can use *depth first* or *breadth first* top down integration.

Steps:

1. Main control module used as the test driver, with stubs for all subordinate modules.
2. Replace stubs either depth first or breadth first.
3. Replace stubs one at a time.
4. Test after each module integrated.
5. Use regression testing (conducting all or some of the previous tests) to ensure new errors are not introduced.

Verifies major control and decision points early in design process.

Top level structure tested the most.

Depth first implementation allows a complete function to be implemented, tested and demonstrated.

Can do depth first implementation of critical functions early.

Top down integration forced (to some extent) by some development tools in programs with graphical user interfaces.

2.3.2 Bottom Up Integration.

Begin construction and testing with atomic modules (lowest level modules).

Use driver program to test.

Steps:

1. Low level modules combined in clusters (builds) that perform specific software subfunctions.
2. Driver program developed to test.
3. Cluster is tested.
4. Driver programs removed and clusters combined, moving upwards in program structure.

2.3.3 Comments on Integration Testing

In general, tend to use combination of top down and bottom up testing.

Critical modules should be tested and integrated early.

2.3.4 Integration Test Documentation

Test specification describes overall plan for integration of the software and the testing.

Possible outline:

1. Scope of testing
2. Test Plan
    1. Test phases and builds
    2. Schedule
    3. Overhead software
    4. Environment and resources
3. Test procedure n (description of tests for build n)
    1. Order of integration
        1. Purpose
        2. Modules to be tested
    2. Unit tests for modules in build
    3. Description of test for module m
    4. Overhead software description
    5. Expected results
    6. Test Environment
    7. Special tools or techniques
    8. Overhead software description

9. Test case data
10. Expected results for build n

4. Actual test results
5. References
6. Appendices

Scope of Testing provides a summary of the specific functional, performance and internal design characteristics which will be tested. The testing effort is bounded, completion criteria for each test phase described, and schedule constraints documented.

The Test Plan describes the strategy for integration. Testing divided into phases and builds. Phases and builds address specific functional and behavioural characteristics of the software.

Example: CAD software might have phases:

User interaction - command selection, drawing creation, display representation, error processing.

Data manipulation and analysis - symbol creation, dimensioning, transformations, computation of physical properties.

Display processing and generation - 2D displays, 3D displays, graphs and charts.

Database management - access, update, integrity, performance.

Each phase and sub-phase specifies a broad functional category within the software which should be able to be related to specific parts of the software structure.

The following criteria and tests are applied for all test phases:

- *Interface integrity*: Internal and external interfaces tested as each module (or cluster) added to software structure.
- *Functional validity*: Tests for functional errors.
- *Information content*: Test local and global data structures
- *Performance*: Test performance and compare to bounds specified during design.

Test plan also includes

- a schedule for integration. Start and end dates given for each phase.
- a description of overhead software, concentrating on those that may require special effort.
- a description of the test environment.

Test plans should be tailored to local requirements, however should always contain an integration strategy (in the Test Plan), testing details (in Test Procedure) are essential and must appear.

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).

Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test02_3.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

2.4 Validation Testing

Validation testing is aims to demonstrate that the software functions in a manner that can be reasonably expected by the customer.

Tests conformance of the software to the *Software Requirements Specification*. This should contain a section "Validation Criteria" which is used to develop the validation tests.

2.4.1 Validation Test Criteria

A set of black box tests to demonstrate conformance with requirements.

To check that: all functional requirements satisfied, all performance requirements achieved, documentation is correct and 'human-engineered', and other requirements are met (e.g., compatibility, error recovery, maintainability).

When validation tests fail it may be too late to correct the error prior to scheduled delivery. Need to negotiate a method of resolving deficiencies with the customer.

2.4.2 Configuration Review

An audit to ensure that all elements of the software configuration are properly developed, catalogued, and have the necessary detail to support maintenance.

2.4.3 Alpha and Beta Testing

Difficult to anticipate how users will really use software.

If there is one customer, a series of *acceptance tests* are conducted (by the customer) to enable the customer to validate all requirements.

If software is being developed for use by many customers, can not use acceptance testing. An alternative is to use *alpha* and *beta testing* to uncover errors.

Alpha testing is conducted at the developer's site by a customer. The customer uses the software with the developer 'looking over the shoulder' and recording errors and usage problems. Alpha testing conducted in a controlled environment.

Beta testing is conducted at one or more customer sites by end users. It is 'live' testing in an environment not controlled by the developer. The customer records and reports difficulties and errors at regular intervals.

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

2.5 System Testing

Software only one component of a system.

Software will be incorporated with other system components and system integration and validation tests performed.

For software based systems can carry out recovery testing, security testing, stress testing and performance testing.

2.5.1 Recovery Testing

Many systems need to be fault tolerant - processing faults must not cause overall system failure.

Other systems require recovery after a failure within a specified time.

Recovery testing is the forced failure of the software in a variety of ways to verify that recovery is properly performed.

2.5.2 Security Testing

Systems with sensitive information or which have the potential to harm individuals can be a target for improper or illegal use. This can include:

- attempted penetration of the system by 'outside' individuals for fun or personal gain.
- disgruntled or dishonest employees

During security testing the tester plays the role of the individual trying to penetrate the system. Large range of methods:

- attempt to acquire passwords through external clerical means
- use custom software to attack the system

- overwhelm the system with requests
- cause system errors and attempt to penetrate the system during recovery
- browse through insecure data.

Given time and resources, the security of most (all?) systems can be breached.

2.5.3 Stress Testing

Stress testing is designed to test the software with abnormal situations. Stress testing attempts to find the limits at which the system will fail through abnormal quantity or frequency of inputs. For example:

- Higher rates of interrupts
- Data rates an order of magnitude above 'normal'
- Test cases that require maximum memory or other resources.
- Test cases that cause 'thrashing' in a virtual operating system.
- Test cases that cause excessive 'hunting' for data on disk systems.

Can also attempt sensitivity testing to determine if particular combinations of otherwise normal inputs can cause improper processing.

2.5.4 Performance Testing

For real-time and embedded systems, functional requirements may be satisfied but performance problems make the system unacceptable.

Performance testing checks the run-time performance in the context of the integrated system.

Can be coupled with stress testing.

May require special software instrumentation.

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test02_5.htm

UNIVERSITY OF SOUTH AUSTRALIA
School of Computer and Information Science
Software Engineering 1A/1B/1M

2.6 Debugging.

Debugging is not testing.

Debugging occurs because of successful testing.

Less well 'understood' than software development.

Difficulties include:

- Symptom and cause may be 'geographically' remote. Large problem in highly coupled software structures.
- Symptoms may disappear (temporarily) and another error is corrected.
- Symptom may not be caused by an error (but for example, a hardware limitation).
- Symptom may be due to human error.
- Symptom may be due to a timing problem rather than processing problem.
- May be hard to reproduce input conditions (especially in real-time systems)
- Symptom may be intermittent - especially in embedded systems.

Not everyone is good at debugging.

This page is maintained by Dr. A.J. Sobey (a.sobey@unisa.edu.au).
Access: Unrestricted.
Created: Semester 2, 1995
Updated: 8th June, 1997
URL: http://louisa.levels.unisa.edu.au/se/testing-notes/test02_6.htm