Software Testing

Software Testing

- *Error*: mistake made by the programmer/ developer
- *Fault*: a incorrect piece of code/document (i.e., bug)
- *Failure*: result of a fault

- Goal of software testing: Cause failures to uncover faults and errors
- Develop tests
- Execute tests

Quality & Testing

- Software Quality Assurance (SQA)
  - Evaluations to be performed
  - Audits and reviews
  - Standards
  - Procedures for error tacking/reporting
  - Documentation to be produced
  - Feedback
- Verification and Validation
  - Independent group (NASA IV&V)

Verification & Validation (V&V)

- **Verification**: The software should conform to its specification (Are we building the product right?)

- **Validation**: The software should do what the user really requires (Are we building the right product?)

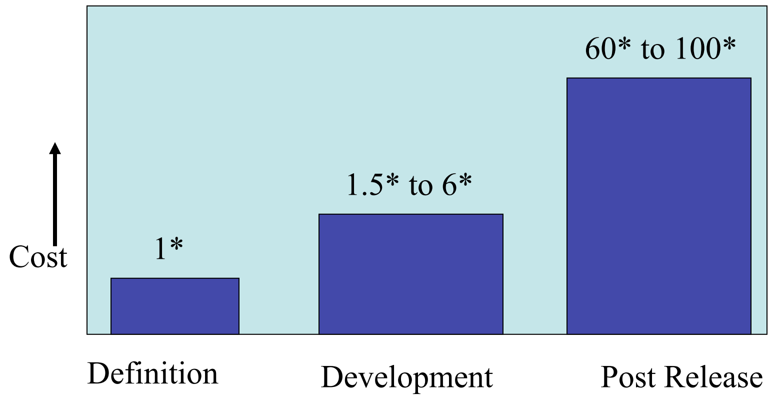
V & V Goals

- Verification and validation should establish confidence that the software is fit for its purpose
- This does NOT mean completely free of defects
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed

"Classical" lifecycle model

- Requirements Phase
- Specification Phase (Analysis)
- Planning Phase
- Design Phase
- Implementation Phase
- Integration and Testing
- Maintenance
- Retirement

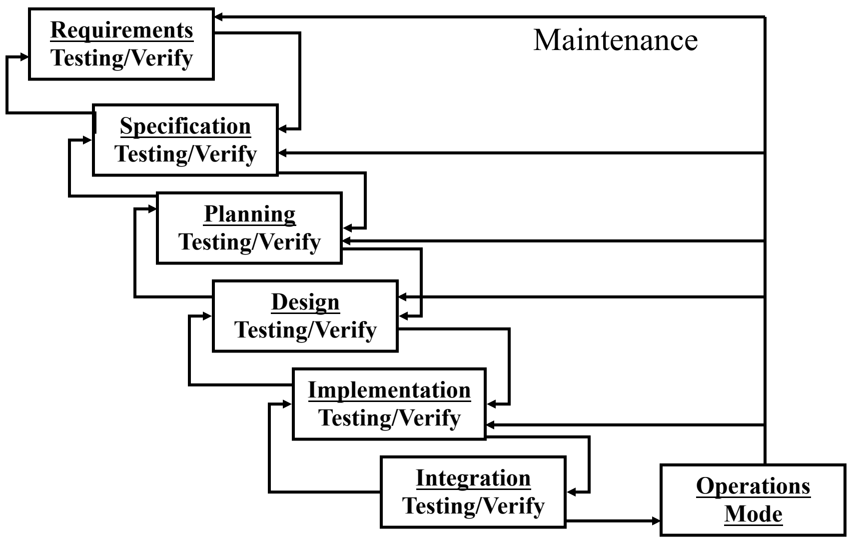
Cost to fix faults



The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.

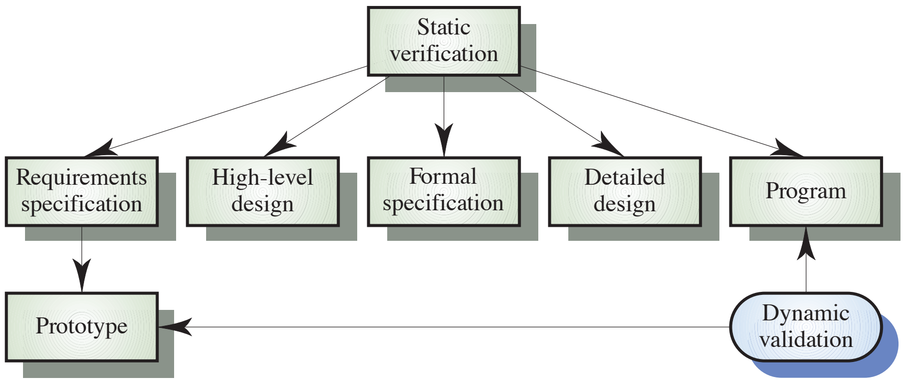
Sequential model



Static and dynamic verification

- *Software inspections and walkthroughs* - Concerned with analysis of the static system representation to discover problems (static verification)
- *Software testing* - Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

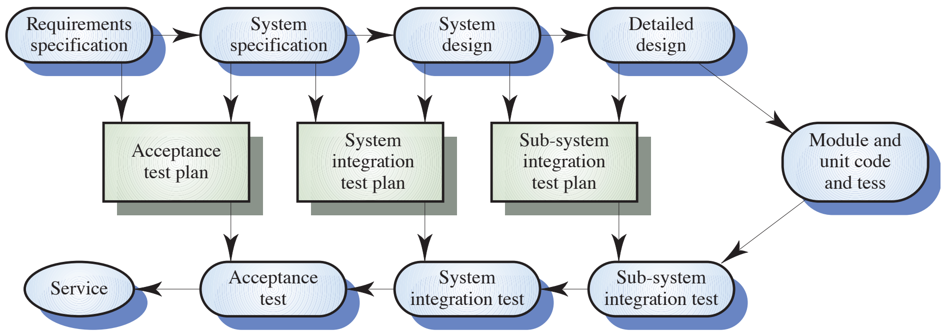
Static and Dynamic V&V



V & V planning

- Careful planning is required to get the most out of testing and inspection processes
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning is about defining standards for the testing process rather than describing product tests

The V-model of development



Software Test Plan

- The testing process
- Requirements traceability
- Tested items
- Testing schedule
- Test recording procedures
- Hardware and software requirements
- Constraints

Walkthroughs

- Informal examination of a product (document)
- Made up of:
  - developers
  - client
  - next phase developers
  - Software Quality Assurance group leader
- Produces:
  - list of items not understood
  - list of items thought to be incorrect

Software Inspections

- Involve people examining the source representation with the aim of discovering anomalies and defects
- Do not require execution of a system so may be used before implementation
- May be applied to any representation of the system (requirements, design, test data, etc.)
- Very effective technique for discovering errors

Inspection Process

- Overview - of the document is made
- Preparation - participants understand the product in detail
- Inspection - a complete walk through is made, covering every branch of the product. Fault finding is done
- Rework - faults are fixed
- Follow - up check fixed faults. If more than say 5% of product is reworked then a complete inspection is done again.

- Statistics are kept: *fault density*

Inspection Success

- Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise

Inspections and Testing

- Inspections and testing are complementary and not opposing verification techniques
- Both should be used during the V & V process
- Inspections can check conformance with a specification but not conformance with the customer's real requirements
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Program Inspections

- Formalized approach to document reviews
- Intended explicitly for defect DETECTION (not correction)
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an un-initialised variable) or non-compliance with standards

Inspection Pre-conditions

- A precise specification must be available
- Team members must be familiar with the organisation standards
- Syntactically correct code must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process
- Management must not use inspections for staff appraisal

Inspection Procedure

- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required

Inspection Teams

- Made up of at least 4 members
- Author of the code being inspected
- Inspector who finds errors, omissions and inconsistencies
- Reader who reads the code to the team
- Moderator who chairs the meeting and notes discovered errors
- Other roles are Scribe and Chief moderator

Inspection Checklists

- Checklist of common errors should be used to drive the inspection
- Error checklist is programming language dependent
- The 'weaker' the type checking, the larger the checklist
- Examples: Initialization, Constant naming, loop termination, array bounds, etc.

Inspection Rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 500 lines costs about 40 man/hours effort (@ $50/hr = $2000!!!)

Program Testing

- Can reveal the presence of errors NOT their absence
- A successful test is a test which discovers one or more errors
- The only validation technique for non-functional requirements
- Should be used in conjunction with static verification to provide full V&V coverage

Execution Based Testing

"Program testing can be a very effective way to show the presents of bugs but is hopelessly inadequate for showing their absence"

[Dijkstra]

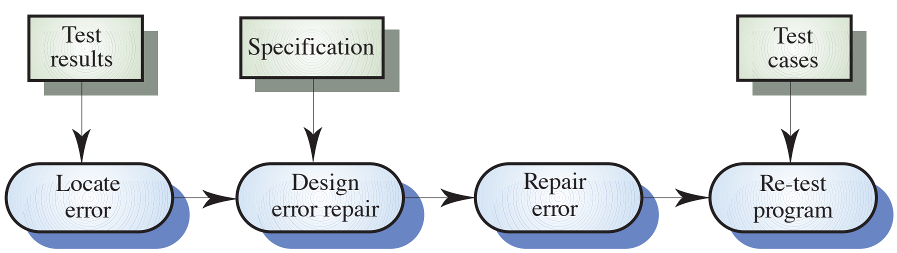
Behavioral Properties

- **Correctness** - does it satisfy its output specification?
- **Utility** - are the user's needs met
- **Reliability** - frequency of the product failure.
  - How long to repair it?
  - How lone to repair results of failure?
- **Robustness** - How crash proof in an alien environment?
  - Does it inform the user what is wrong?
- **Performance** - response time, memory usage, run time, etc.

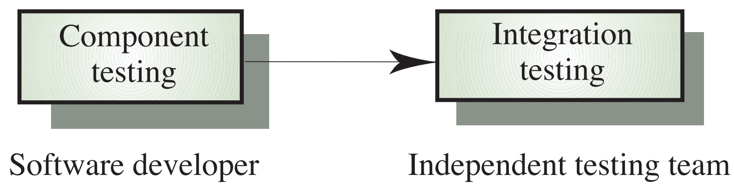
Testing and Debugging

- Defect testing and debugging are distinct processes
- Verification and validation is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

The Debugging Process



Testing Phases



Testing Phases

- Component testing
  - Testing of individual program components
  - Usually the responsibility of the component developer (except sometimes for critical systems)
  - Tests are derived from the developer's experience
- Integration testing
  - Testing of groups of components integrated to create a system or sub-system
  - The responsibility of an independent testing team
  - Tests are based on a system specification

Testing Priorities

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible
- Tests should exercise a system's capabilities rather than its components
- Testing old capabilities is more important than testing new capabilities
- Testing typical situations is more important than boundary value cases

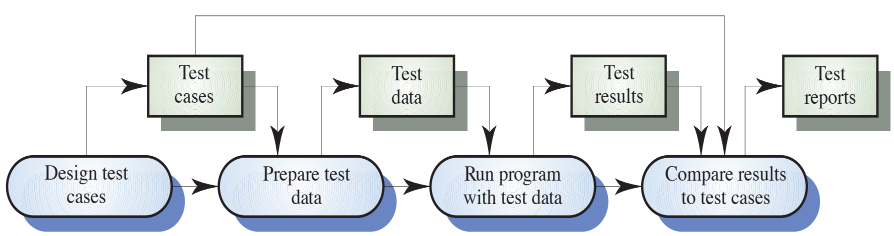
Test Data and Test Cases

- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

Development of test cases

- Test cases and test scenarios comprise much of a software systems *testware*.
- Black box test cases are developed by domain analysis and examination of the system requirements and specification.
- Glass box test cases are developed by examining the behavior of the source code.

The Defect Testing Process



Methods of Testing

- Test to specification:
  - Black box,
  - Data driven
  - Functional testing
  - Code is ignored: only use specification document to develop test cases
- Test to code:
  - Glass box/White box
  - Logic driven testing
  - Ignore specification and only examine the code.

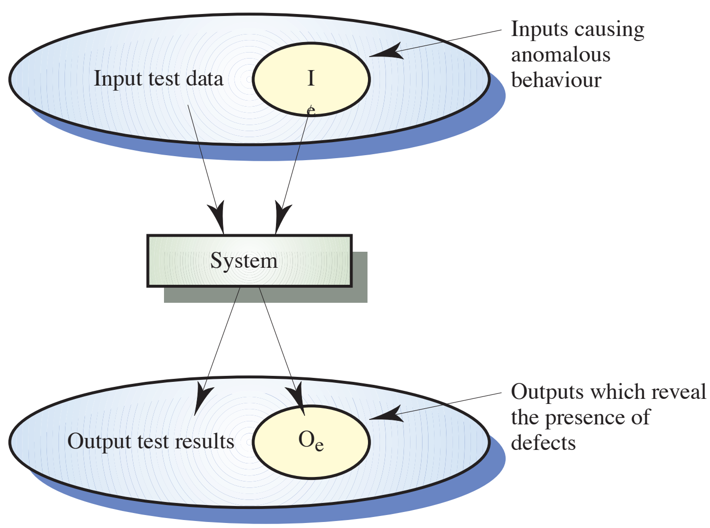
Guaranteeing a Program Correct?

- This is called the Halting Problem (in general)
- Write a program to test if any given program is correct. The output is correct or incorrect.
- Test this program on itself.
- If output is incorrect, then how do you know the output is correct?
- Conundrum, Dilemma, or Contradiction?

Black-box Testing

- An approach to testing where the program is considered as a 'black-box'
- The program test cases are based on the system specification
- Test planning can begin early in the software process

Black-box testing



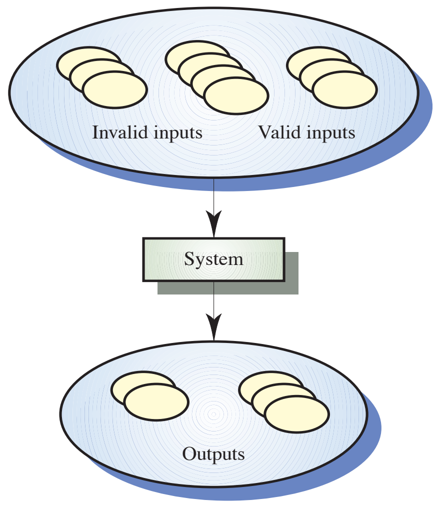
Pairing Down Test Cases

- Use methods that take advantage of symmetries, data equivalencies, and independencies to reduce the number of necessary test cases.
  - Equivalence Testing
  - Boundary Value Analysis
- Determine the ranges of working system
- Develop equivalence classes of test cases
- Examine the boundaries of these classes carefully

Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

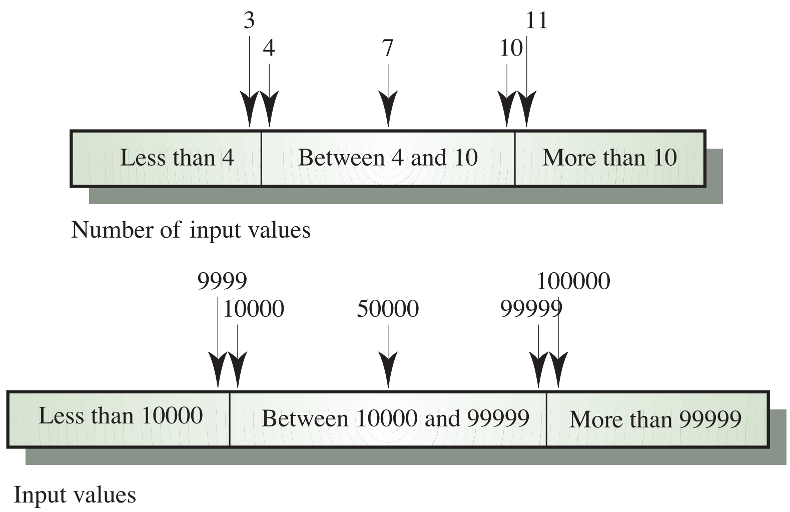
Equivalence Partitioning



Boundary Value Testing

- Partition system inputs and outputs into "equivalence sets"
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are < 10,000, 10,000 - 99, 999 and > 10, 000
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 10001

Equivalence Partitions



Search Routine Specification

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition
-- the array has at least one element
T'FIRST <= T'LAST
Post-condition
-- the element is found and is referenced by L
(Found and T (L) = Key)
or
-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Search Routine - Input Partitions

- Inputs which conform to the pre-conditions
- Inputs where a pre-condition does not hold
- Inputs where the key element is a member of the array
- Inputs where the key element is not a member of the array

Testing Guidelines - Sequences

- Test software with sequences which have only a single value
- Use sequences of different sizes in different tests
- Derive tests so that the first, middle and last elements of the sequence are accessed
- Test with sequences of zero length

Search Routine - Input Partitions

Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Sorting Example

- Example: sort (lst, n)
  - Sort a list of numbers
  - The list is between 2 and 1000 elements
- Domains:
  - The list has some item type (of little concern)
  - n is an integer value (sub-range)
- Equivalence classes;
  - n < 2
  - n > 1000
  - 2 <= n <= 1000

Sorting Example

- What do you test?
- Not all cases of integers
- Not all cases of positive integers
- Not all cases between 1 and 1001

- Highest payoff for detecting faults is to test around the boundaries of equivalence classes.

- Test n=1, n=2, n=1000, n=1001, and say n= 10
- Five tests versus 1000.

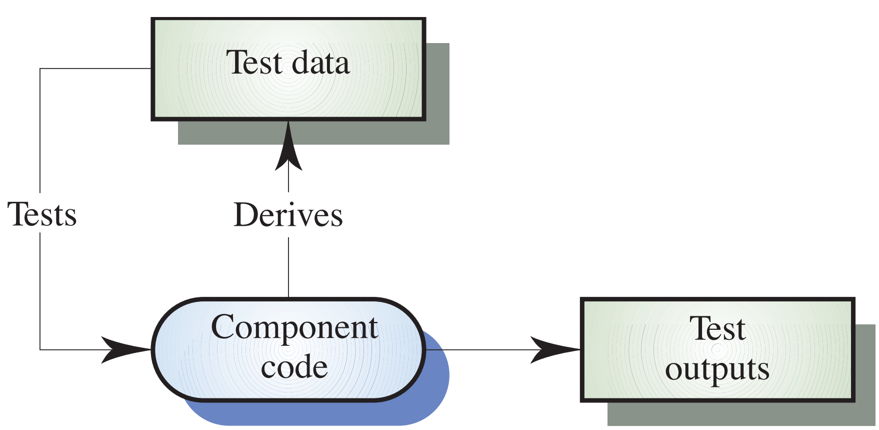
White-box Testing

- Sometime called structural testing or glass-box testing
- Derivation of test cases according to program structure
- Knowledge of the program is used to identify additional test cases
- Objective is to exercise all program statements (not all path combinations)

Types of Structural Testing

- Statement coverage -
  - Test cases which will execute every statement at least once.
  - Tools exist for help
  - No guarantee that all branches are properly tested. Loop exit?
- Branch coverage
  - All branches are tested once
- Path coverage - Restriction of type of paths:
  - Linear code sequences
  - Definition/Use checking (all definition/use paths)
  - Can locate dead code

White-box testing



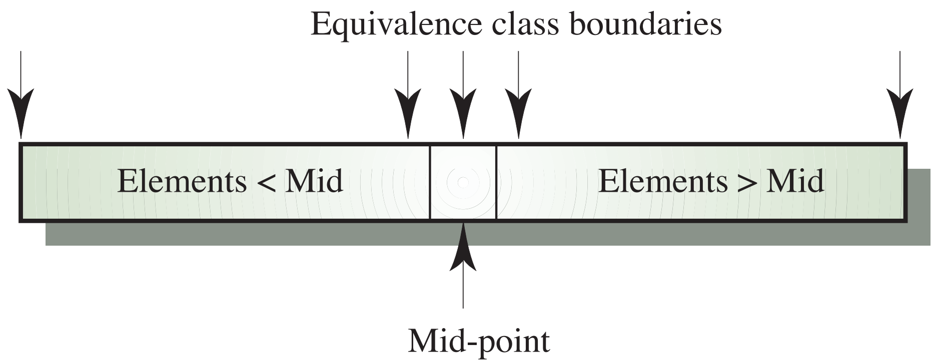
White Box Testing - Binary Search

```
int search ( int key, int [] elemArray)
{
   int bottom = 0;
   int top = elemArray.length - 1;
   int mid;
   int result = -1;
   while ( bottom <= top )
   {
        mid = (top + bottom) / 2;
        if (elemArray [mid] == key)
        {
          result = mid;
          return result;
        } // if part
        else
        {
          if (elemArray [mid] < key)
             bottom = mid + 1;
          else
             top = mid - 1;
        }
   } //while loop
   return result;
} // search
```

Binary Search Equivalence Partitions

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array
- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values
- Input array has an odd number of values

Binary Search Equivalence Partitions



Binary Search - Test Cases

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Path Testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control
- Statements with conditions are therefore nodes in the flow graph

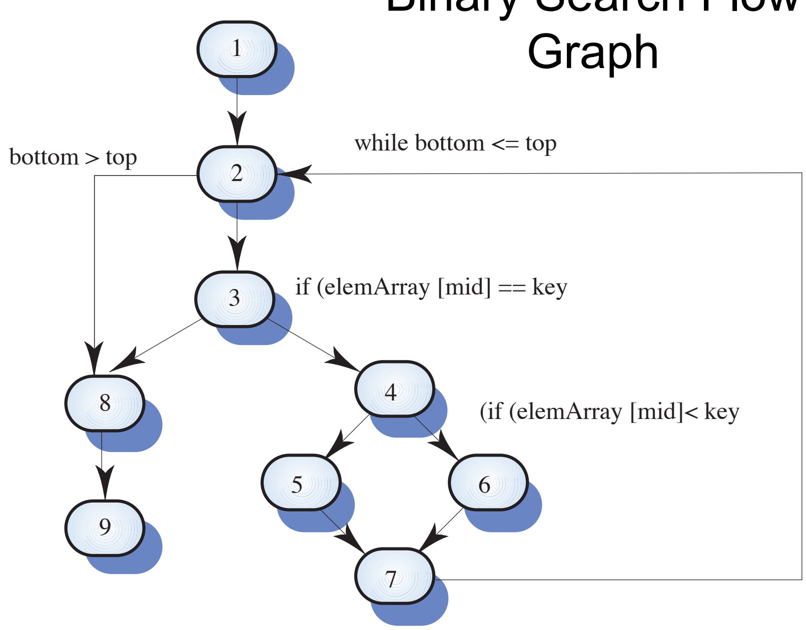
Program Flow Graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node
- Used as a basis for computing the cyclomatic complexity
- Cyclomatic complexity = Number of edges - Number of nodes +2

Cyclomatic Complexity

- The number of tests to test all control statements equals the cyclomatic complexity
- Cyclomatic complexity equals number of conditions in a program
- Useful if used with care. Does not imply adequacy of testing
- Although all paths are executed, all combinations of paths are not executed

Binary Search Flow Graph



Independent Paths

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

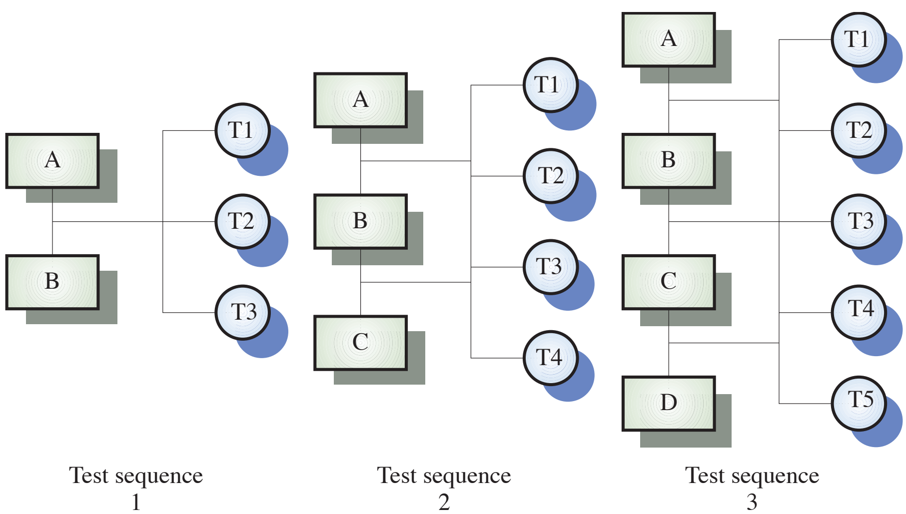
Feasibility

- Pure black box testing (specification) is realistically impossible because there are (in general) too many test cases to consider.
- Pure testing to code requires a test of every possible path in a flow chart. This is also (in general) infeasible. Also every path does not guarantee correctness.
- Normally, a combination of Black box and Glass box testing is done.

Integration Testing

- Tests complete systems or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- Incremental integration testing reduces this problem

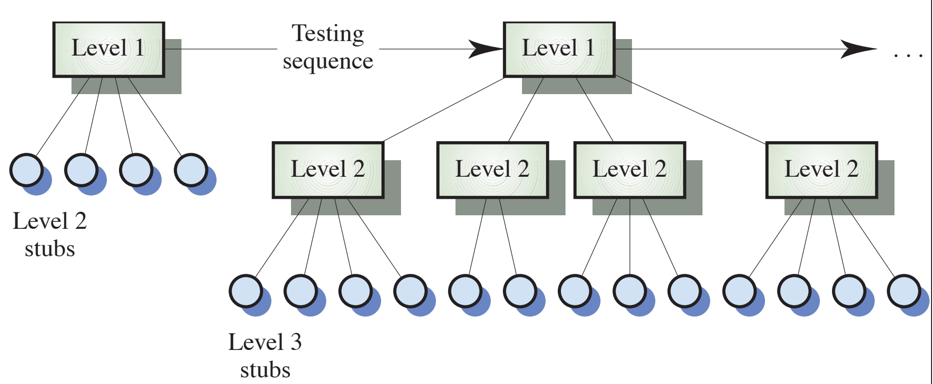
Incremental integration testing



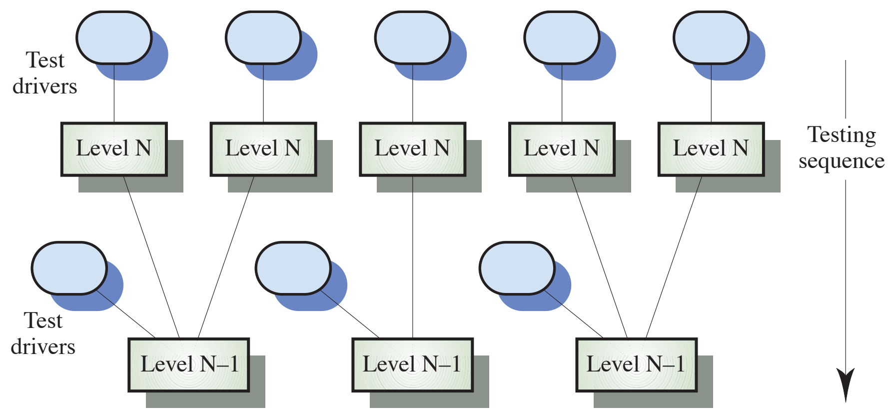
Approaches to Integration Testing

- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Top-down Testing



Bottom-up Testing



Software Testing Metrics

- Defects rates
- Errors rates
- Number of errors
- Number of errors found per person hours expended
- Measured by:
  - Individual, module, during development
- Errors should be categorized by origin, type, cost

More Metrics

- Direct measures - cost, effort, LOC, etc.
- Indirect Measures - functionality, quality, complexity, reliability, maintainability

- Size Oriented:
  - Lines of code - LOC
  - Effort - person months
  - errors/KLOC
  - defects/KLOC
  - cost/KLOC

Proofs of Correctness

- Assertions, preconditions, post conditions, and invariants are used
- **Assertion** – something that is true at a particular point in the program
- **Pre conditions** must be true before something is executed
- **Post conditions** are true after something has executed
- **Invariants** are always true with a give scope (e.g., construct, loop, ADT)

Logical Properties

- Assertions describe the logical properties which hold at each statement in a program
- Assertions can be added to each line to describe the program
- Utilize a formal approach (e.g., first order predicate calculus, Z, etc.)

Example

```
//PRE: n in {1,2,3…}
int k, s;
int y[n];
k=0;
//ASSERT: k==0
s=0;
//ASSERT: s==0 && k==0
//LOOP INV: (k<=n) && (s==y[0]+y[1]+…+y[k-1])
While (k<n)
{
  //ASSERT: (k<n) && (s==y[0]+y[1]+…+y[k-1])
  s=s+y[k];
  //ASSERT: (k<n) && (s==y[0]+y[1]+…+y[k])
  k=k+1;
  //ASSERT: (k<=n) && (s==y[0]+y[1]+…+y[k-1])
}
//POST: (k==n) && (s==y[0]+y[1]+…+y[n-1])
```

Proving the Program

- Prove correct based on the loop invariant
- Use induction

- Basis:
  - Before loop is entered
  - k=0 and s=0 therefore
  - s=y[0-1]=y[-1]=0
  - Also k<=n since n in {1,2,3,…}

Using Induction

- Inductive Hypothesis
  - Assume for some k>=0,
  - s = y[0]+y[1]+…y[n-2]+y[n-1]
  - when ever n<=k

- Inductive step show s = y[0]+y[1]+…y[n-2]+y[n-1] is true for k+1
  - s = y[0]+y[1]+…+y[k+1-2]+y[k+1-1]
  - s = y[0]+y[1]+…+y[k-1]+y[k]
  - s = (y[0]+y[1]+…+y[k-1]) + y[k] Q.E.D

Proving can be Problematic

- Mathematical proofs (as complex and error prone as coding)
- Need tool support for theorem proving

- Leavenworth ‘70 did an informal proof of correctness of a simple text justification program. (Claims it’s correct!)
- London ‘71 found four faults, then did a formal proof. (Claims it’s now correct!)
- Goodenough and Gerhar ‘75 found three more faults.

- Testing would have found these errors without much difficulty

Automated Testing Tools

- Code analysis tools

- Static analysis
  - No execution
- Dynamic analysis
  - Execution based

Static Analysis

- Code analyzers: syntax, fault prone
- Structure checker
  - Generates structure graph from the components with logical flow checked for structural flaws (dead code)
- Data analyzer – data structure review. Conflicts in data definitions and usages
- Sequence checker – checks for proper sequences of events (open file before modify)

Dynamic Analysis

- Program monitors record snapshot of the state of the system and watch program behaviors
- List number of times a component is called (profiler)
- Path, statement, branch coverage
- Examine memory and variable information

Test Execution Tools

- Capture and replay
  - Tools capture keystrokes, input and responses while tests are run
  - Verify fault is fixed by running same test cases
- Subs and drivers
- Generate stubs and drivers for integration testing
  - Set appropriate state variables, simulate key board input, compare actual to expected
  - Track paths of execution, reset variables to prepare for next test, interact with other tools

Test Execution Tools

- Automated testing environments
- Test case generators
  - Structural test case generators based on source code – path or branch coverage
  - Data flow