

Defect testing

- ◆ Establishing the presence of system defects

Defect testing

- ◆ The objective of defect testing is to discover defects in programs
- ◆ A successful defect test is a test which causes a program to behave in an anomalous way
- ◆ Tests show the presence not the absence of defects

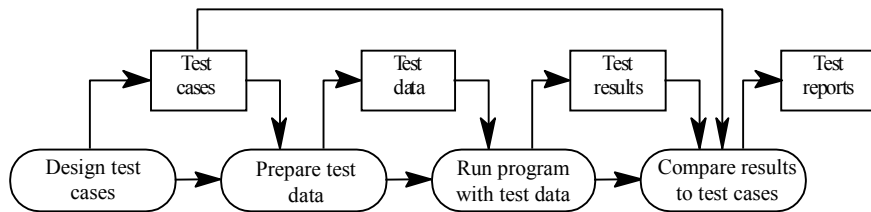
Testing priorities

- ◆ 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

The defect testing process

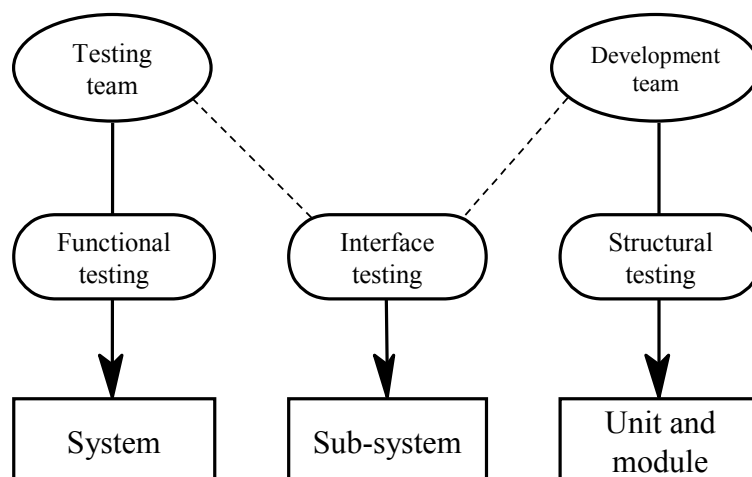


©Ian Sommerville 1995

Software Engineering, 5th edition. Chapter 23

Slide 7

Defect testing approaches



©Ian Sommerville 1995

Software Engineering, 5th edition. Chapter 23

Slide 8

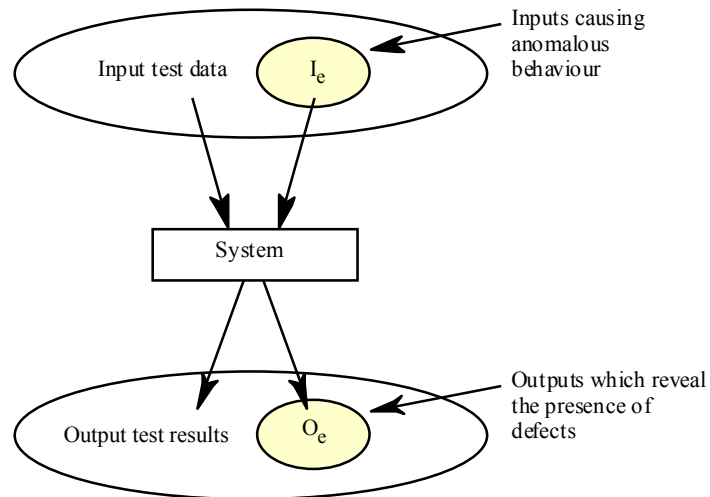
Testing effectiveness

- ◆ In an experiment, black-box testing was found to be more effective than structural testing in discovering defects
- ◆ Static code reviewing was less expensive and more effective in discovering program faults

Black-box testing

- ◆ 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



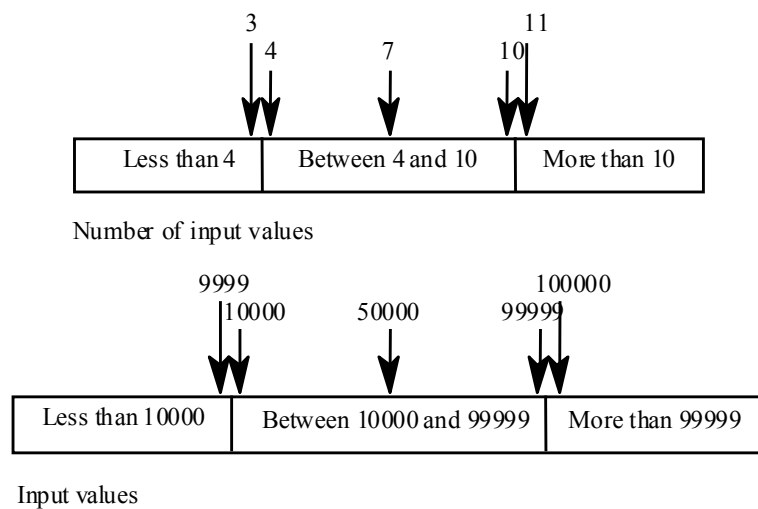
Equivalence partitioning

- ◆ Replace with portrait slide

Equivalence partitioning

- ◆ 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 $>99,999$
- ◆ Choose test cases at the boundary of these sets
 - 00000, 09999, 10000, 99999, 100000

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 (arrays)

- ◆ Test software with arrays which have only a single value
- ◆ Use arrays of different sizes in different tests
- ◆ Derive tests so that the first, middle and last elements of the array are accessed
- ◆ Test with arrays of zero length (if allowed by programming language)

Search routine - input partitions

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

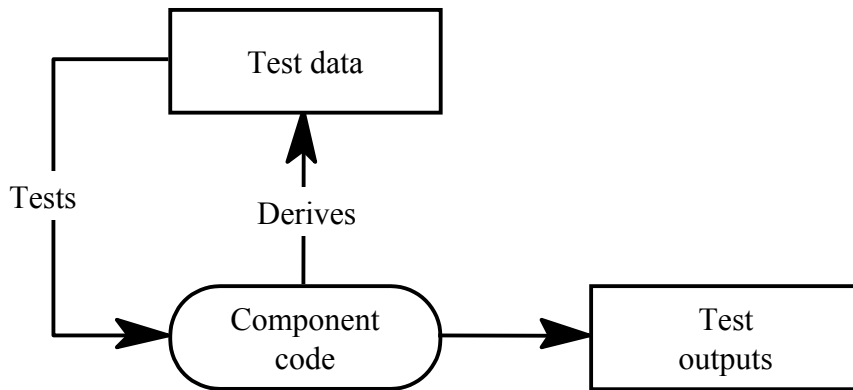
Search routine - test cases

Input array (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, 6
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Structural testing

- ◆ Sometime called white-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)

White-box testing



Binary search (Ada)

```
procedure Binary_search (Key: ELEM ; T: ELEM_ARRAY ;  
  Found in out BOOLEAN ; L: in out ELEM_INDEX ) is  
  - Preconditions  
  -- T'FIRST <= T'LAST and  
  -- forall i: T'FIRST..T'LAST-1, T (i) <= T(i+1)  
  Bott : ELEM_INDEX := T'FIRST ;  
  Top : ELEM_INDEX := T'LAST ;  
  Mid : ELEM_INDEX ;  
begin  
  L := (T'FIRST + T'LAST) / 2 ;  
  Found := T ( L ) = Key ;  
  while Bott <= Top and not Found loop  
    Mid := (Top + Bott) mod 2 ;  
    if T ( Mid ) = Key then  
      Found := true ;  
      L := Mid ;  
    elsif T ( Mid ) < Key then  
      Bott := Mid + 1 ;  
    else  
      Top := Mid - 1 ;  
    end if ;  
  end loop ;  
end Binary_search ;
```

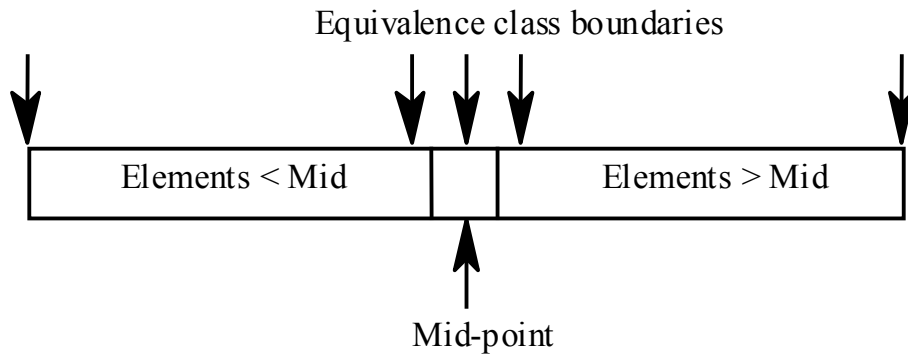
Binary search (C++)

- ◆ Replace with portrait slide

Binary search - equiv. 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 equiv. partitions



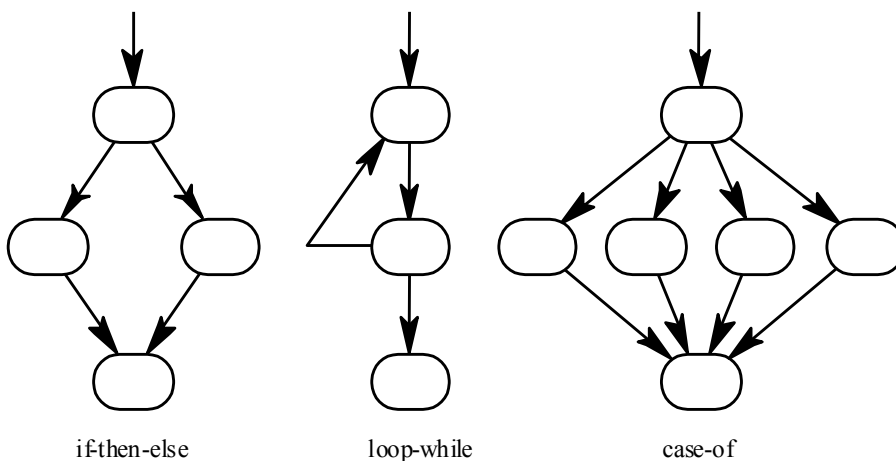
Binary search - test cases

Input array (I)	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, ??

Program flow graphs

- ◆ Describes the program control flow
- ◆ Used as a basis for computing the cyclomatic complexity
- ◆ Complexity =
Number of edges - Number of nodes + 1

Flow graph representations



Binary search flow graph

- ◆ Replace with portrait slide

Independent paths

- ◆ 1, 2, 3, 4, 12, 13
- ◆ 1, 2, 3, 5, 6, 11, 2, 12, 13
- ◆ 1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13
- ◆ 1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13
- ◆ 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

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
- ◆ Does not take into account data-driven programs

Control and data-driven programs

- ◆ Replace with portrait slide

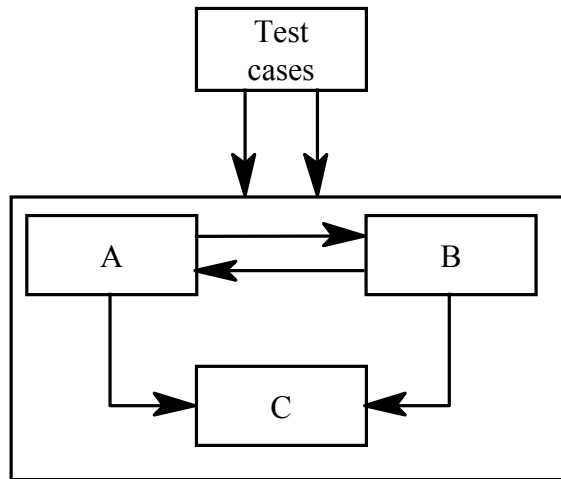
Interface testing

- ◆ Takes place when modules or sub-systems are integrated to create larger systems
- ◆ Objectives are to detect faults due to interface errors or invalid assumptions about interfaces
- ◆ Particularly important for object-oriented development as objects are defined by their interfaces

Interfaces types

- ◆ Parameter interfaces
 - Data passed from one procedure to another
- ◆ Shared memory interfaces
 - Block of memory is shared between procedures
- ◆ Procedural interfaces
 - Sub-system encapsulates a set of procedures to be called by other sub-systems
- ◆ Message passing interfaces
 - Sub-systems request services from other sub-systems

Interface testing



Interface errors

- ◆ **Interface misuse**
 - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- ◆ **Interface misunderstanding**
 - A calling component embeds assumptions about the behaviour of the called component which are incorrect
- ◆ **Timing errors**
 - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

- ◆ Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- ◆ Always test pointer parameters with null pointers
- ◆ Design tests which cause the component to fail
- ◆ Use stress testing in message passing systems
- ◆ In shared memory systems, vary the order in which components are activated