COMP201 – Software Engineering I
Lecture 25

Lecturer: Dr T Carroll

Email: Thomas.Carroll2@Liverpool.ac.uk

Office: G.14

See Vital for all notes

Recap

Lecture 24 Recap

- **Verification and validation** are not the same thing.
  - Verification shows conformance with specification;
  - Validation shows that the program meets the customer's needs
- **Test plans** should be drawn up to guide the testing process.
- **Program inspections** are very effective in discovering errors
- Different types of systems and software development processes require different levels of verification and validation

Today - Testing

Defect Testing

- Defect testing involves testing programs to establish the **presence of system defects**

Objectives

- To understand testing techniques that are geared to discover program faults
- To introduce guidelines for interface testing
- To understand specific approaches to object-oriented testing
- To understand the principles of CASE tool support for testing

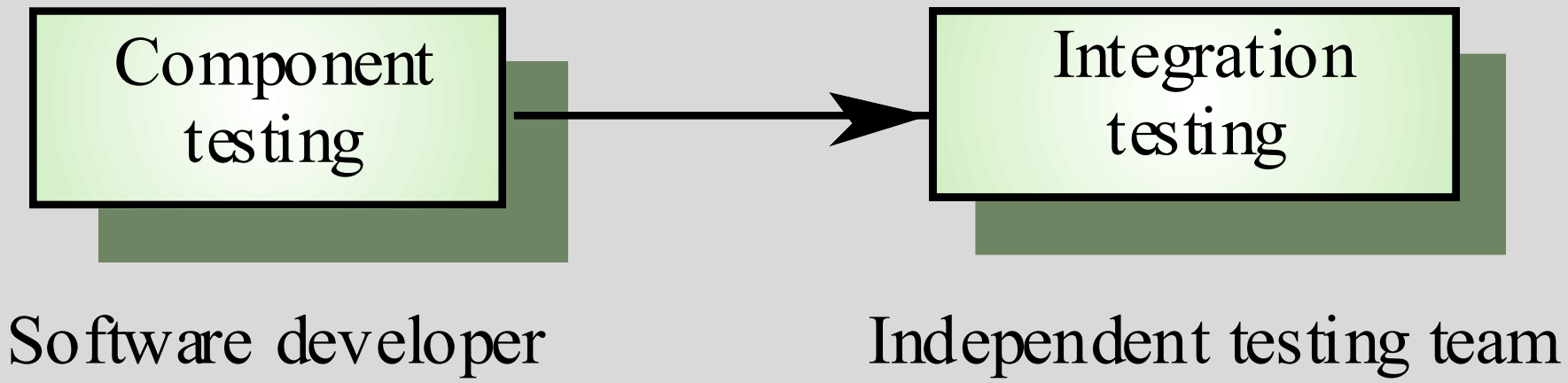
Topics Covered

- Defect testing
- Integration testing
- Object-oriented testing
- Testing workbenches

The Testing Process

- 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 Phases



Defect Testing

- The goal 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

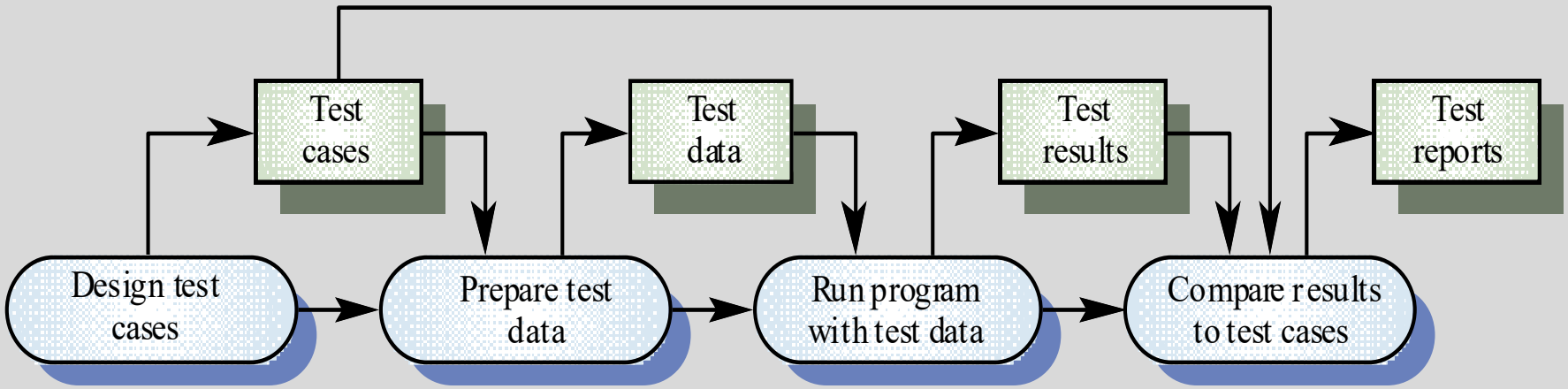
Test Data and Test Cases

- *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

Test plan template

Name of case	Description	Input data	Action	Expected output	Actual output	Success/ Fail
LoginOKPass	Tests login with a good username and password	Username=test 1 Password=pass 1	Click login	OK	Login OK	Success
LoginBadPass	Tests login with good username but wrong password	Username=test 1 Password=pass 2	Click login	Failed to Login	Failed to login	Success
LoginNoPass	Tests login with password field empty	Username=test 1 Password=	Click login	Failed to login	Login OK	Fail

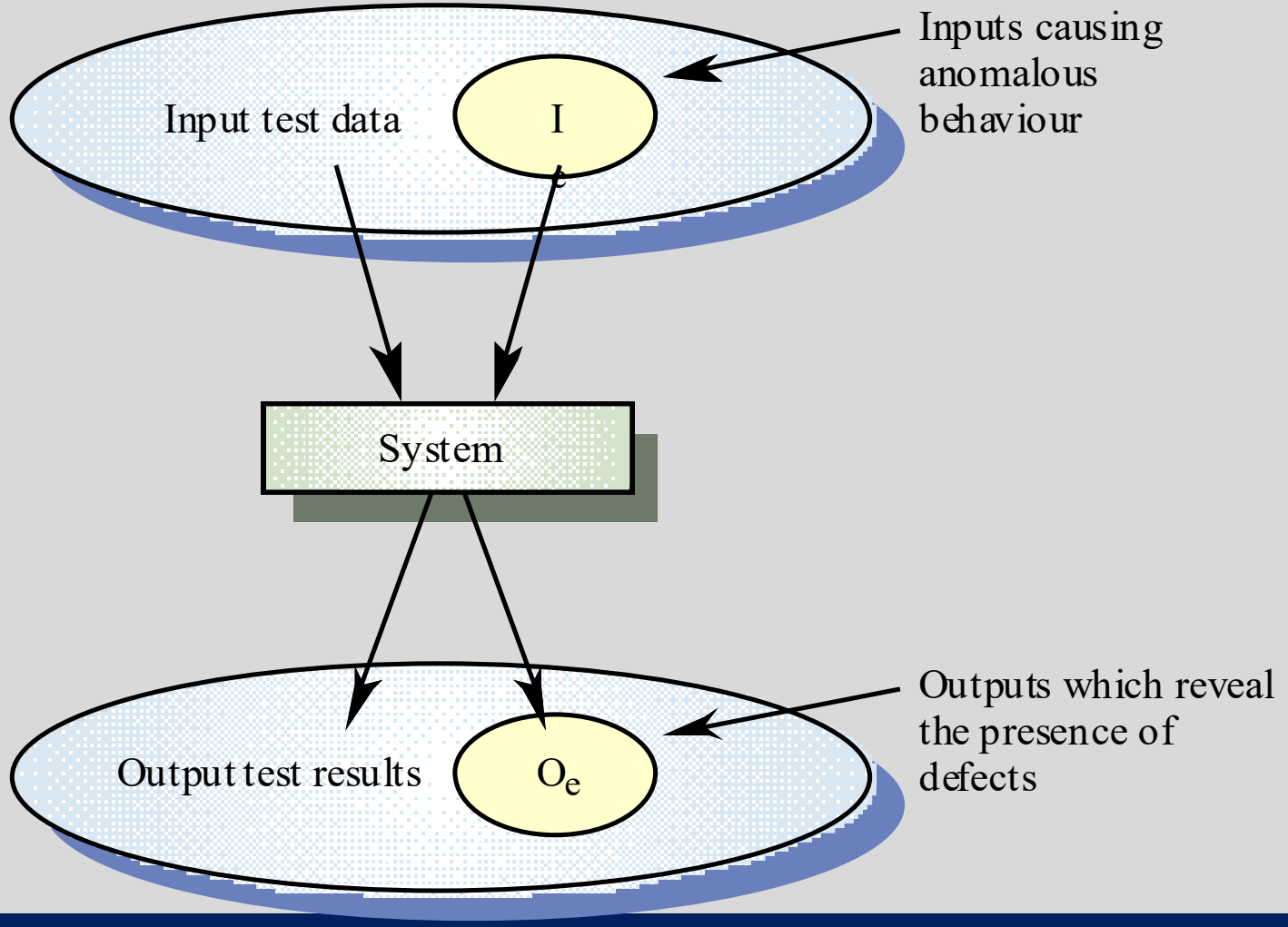
The Defect Testing Process



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

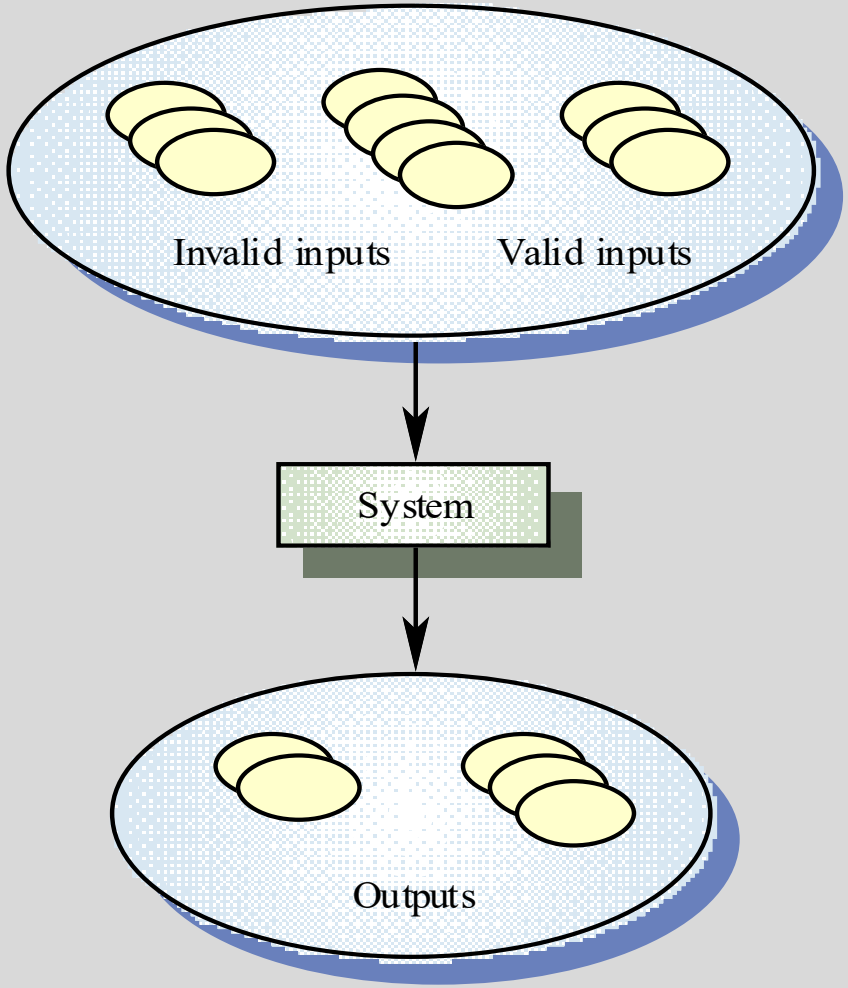
Black-box Testing



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

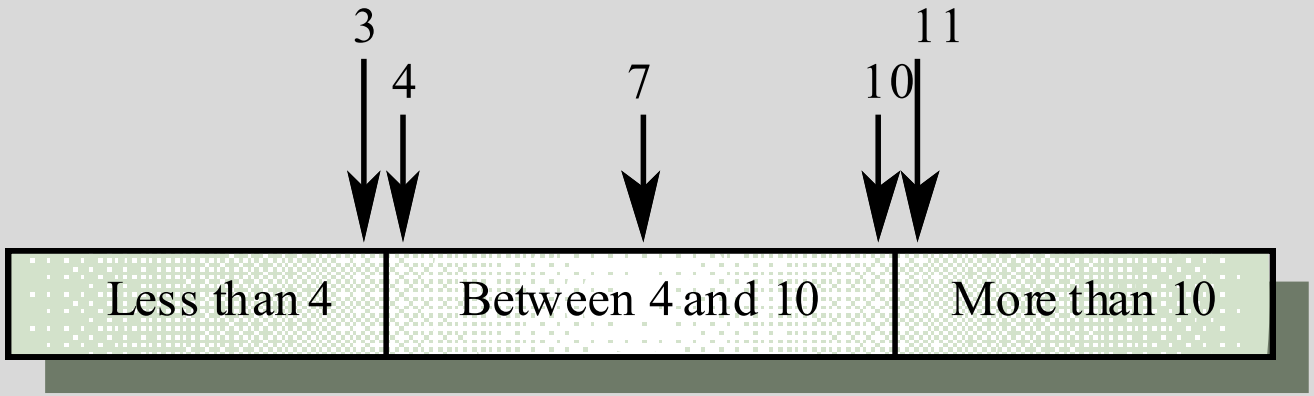
Equivalence Partitioning

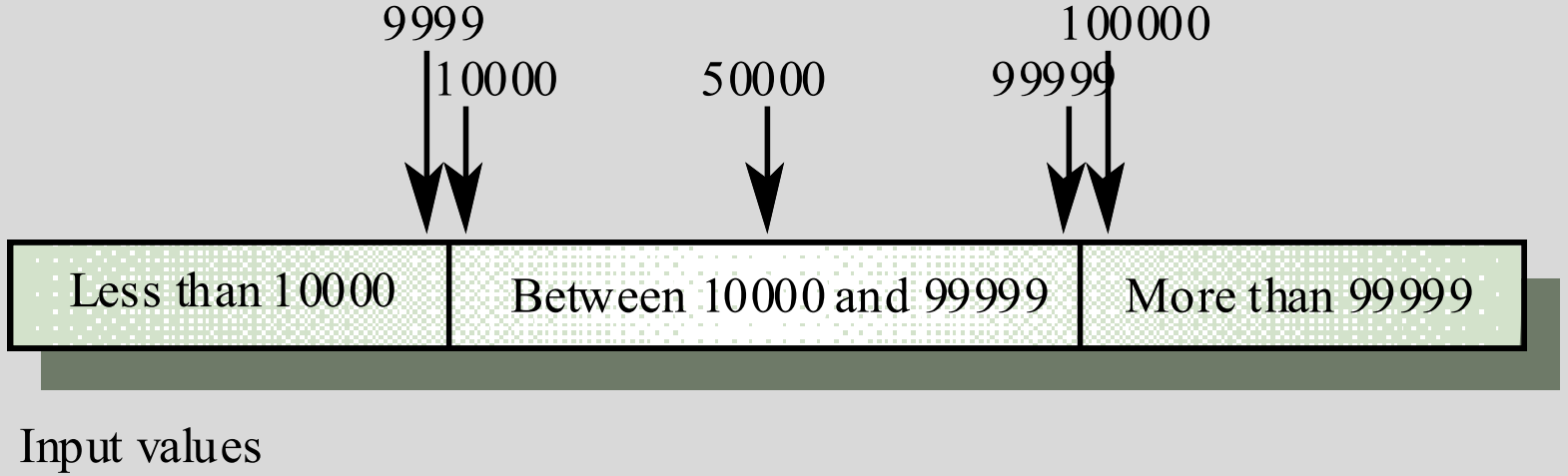


Equivalence Partitioning

- 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 > 10, 000
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 10001
- These are more likely to display erroneous behaviour than choosing random values

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

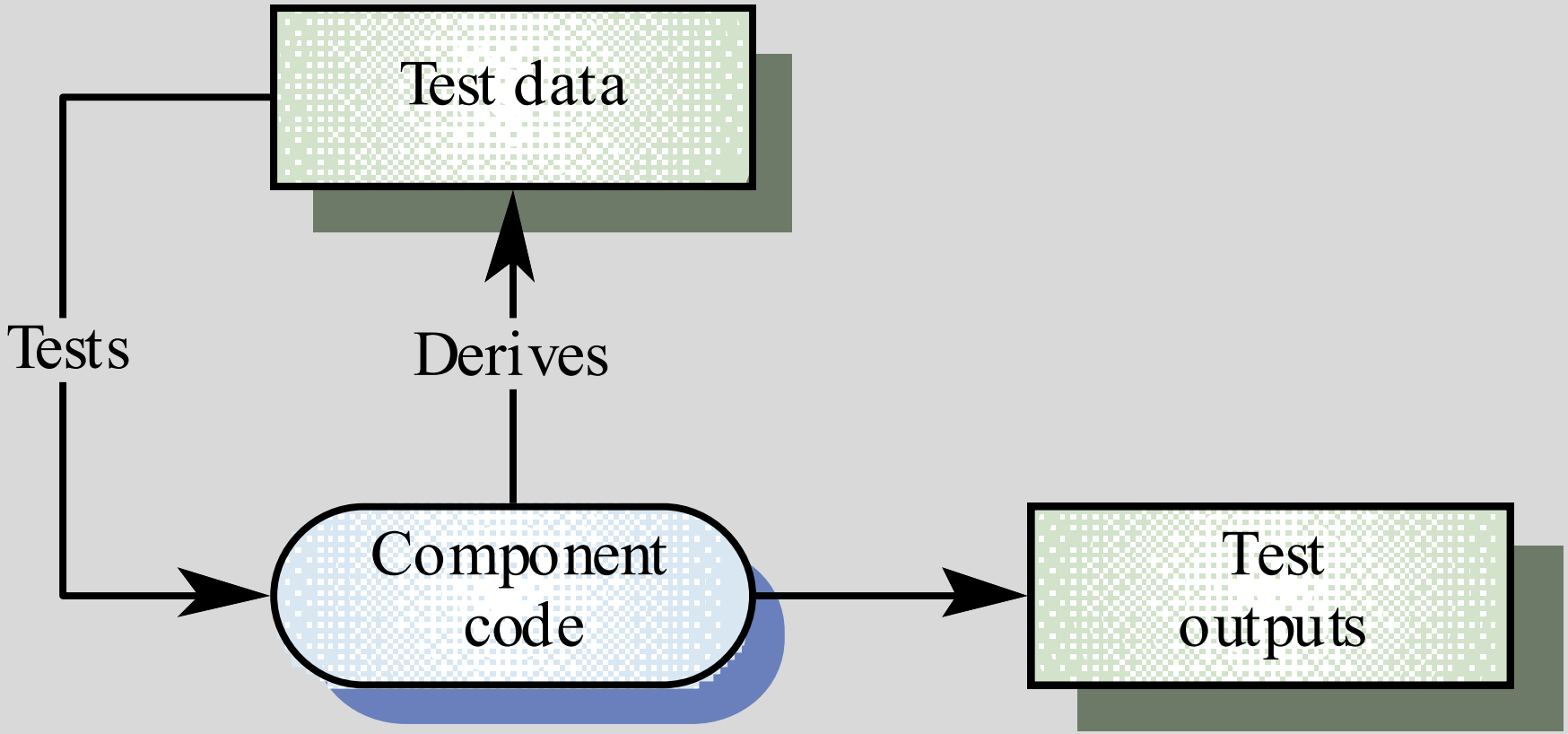
Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

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, ??

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



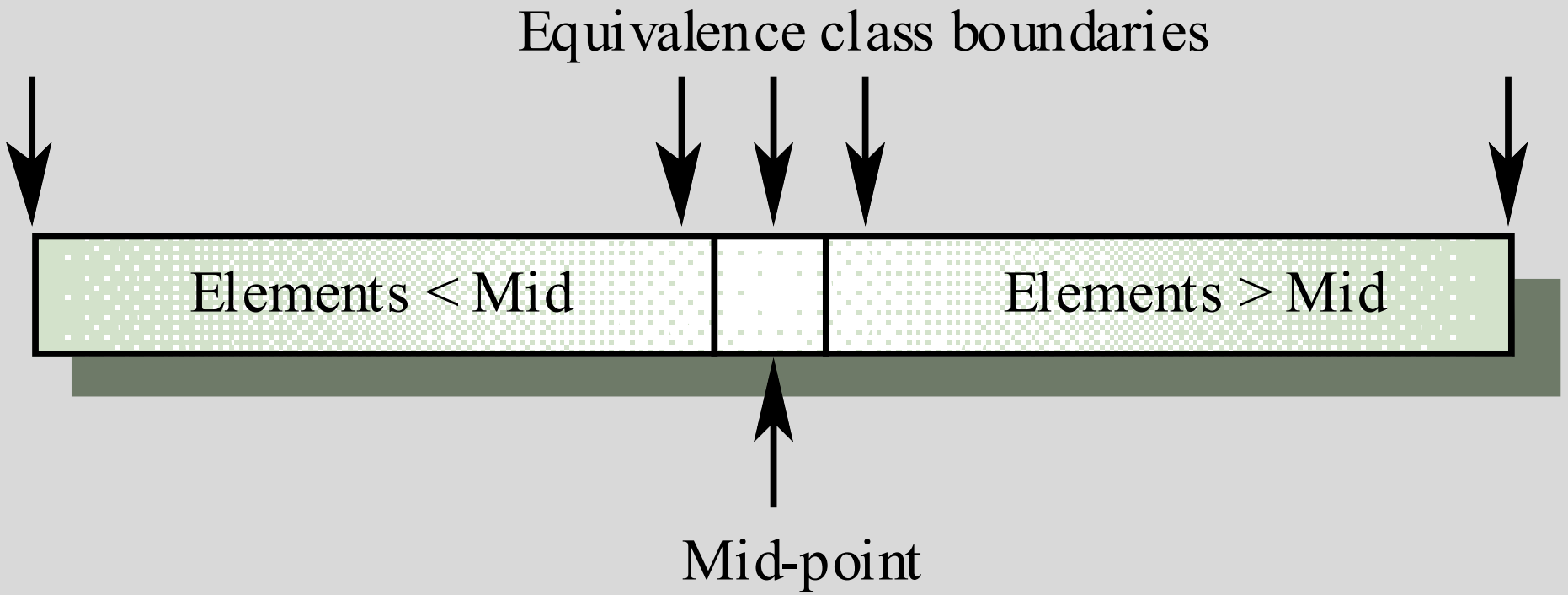
Binary search (Java)

```
class BinSearch {

// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types by
// reference to a function and so return two values
// the key is -1 if the element is not found

    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} //BinSearch
```

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 (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, ??

Lecture Key Points

- Test parts of a system which are commonly used rather than those which are rarely executed
- Equivalence partitions are sets of test cases where the program should behave in an equivalent way
- Black-box testing is based on the system specification
- Structural testing identifies test cases which cause all paths through the program to be executed