

# Software Testing

# Defect testing

---

- Testing programs to establish the presence of system defects

# Test

---

- “Testing is the process of executing a program with the intention of finding errors.” – Myers
- “Testing can show the presence of bugs but never their absence.” - Dijkstra

# 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

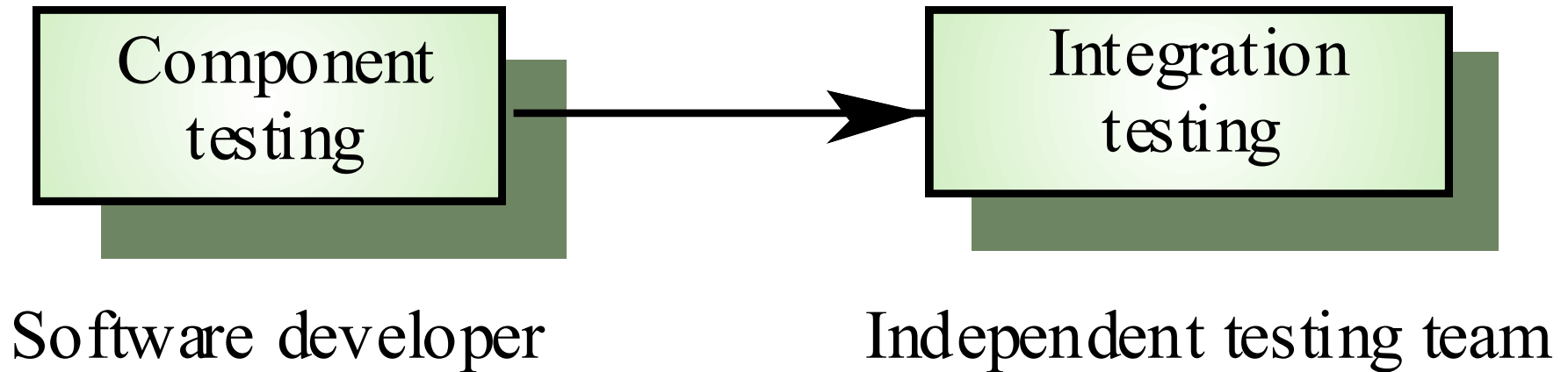
# The testing process

---

- **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

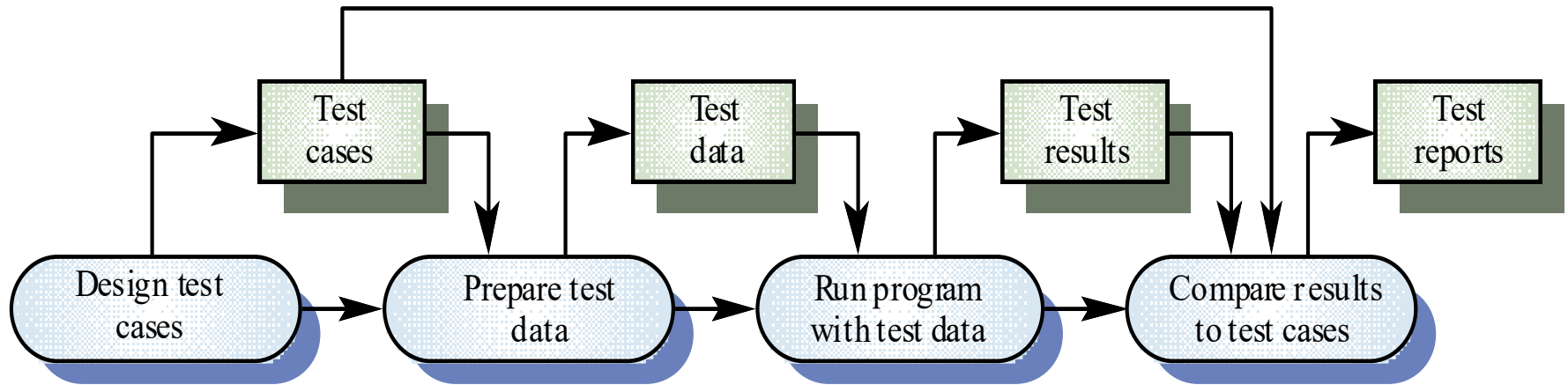
# Test data and test cases

---

- *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

# The defect testing process

---



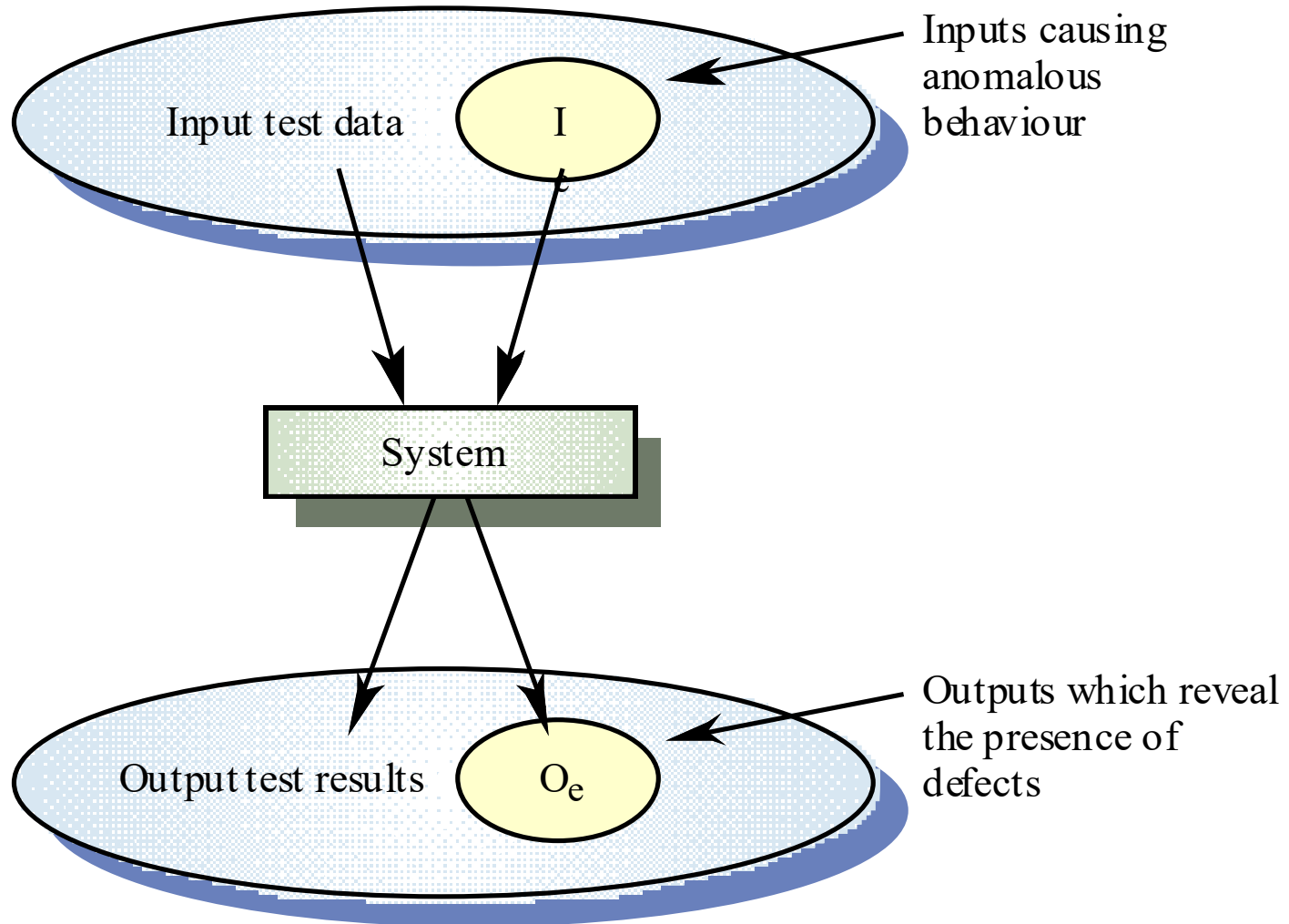
# Black-box testing

---

- 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

---



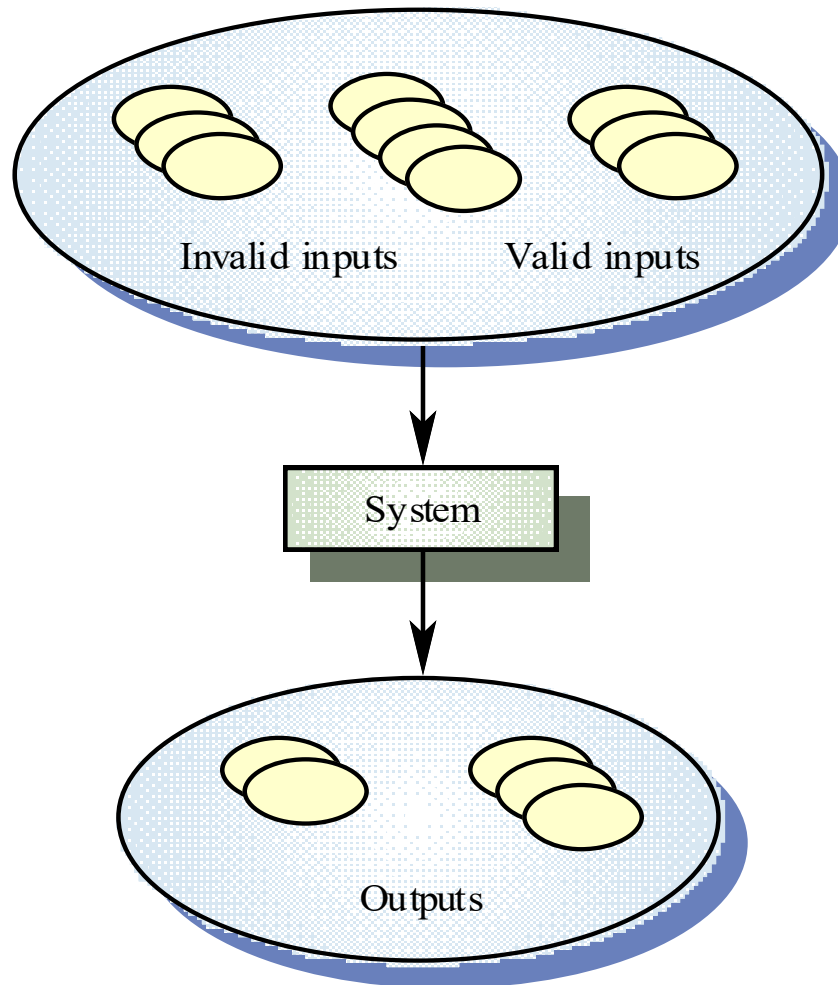
# Equivalence partitioning

---

- 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

---



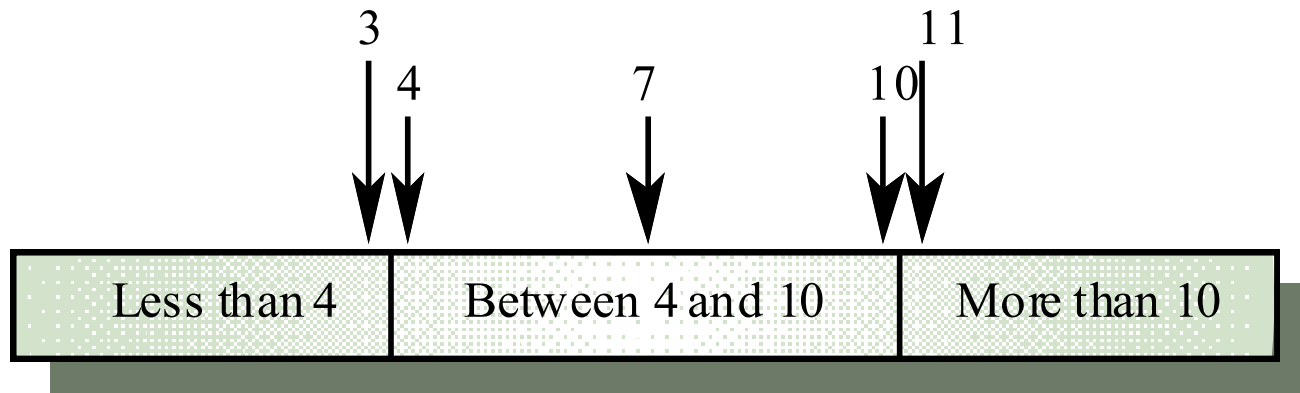
# Equivalence partitioning

---

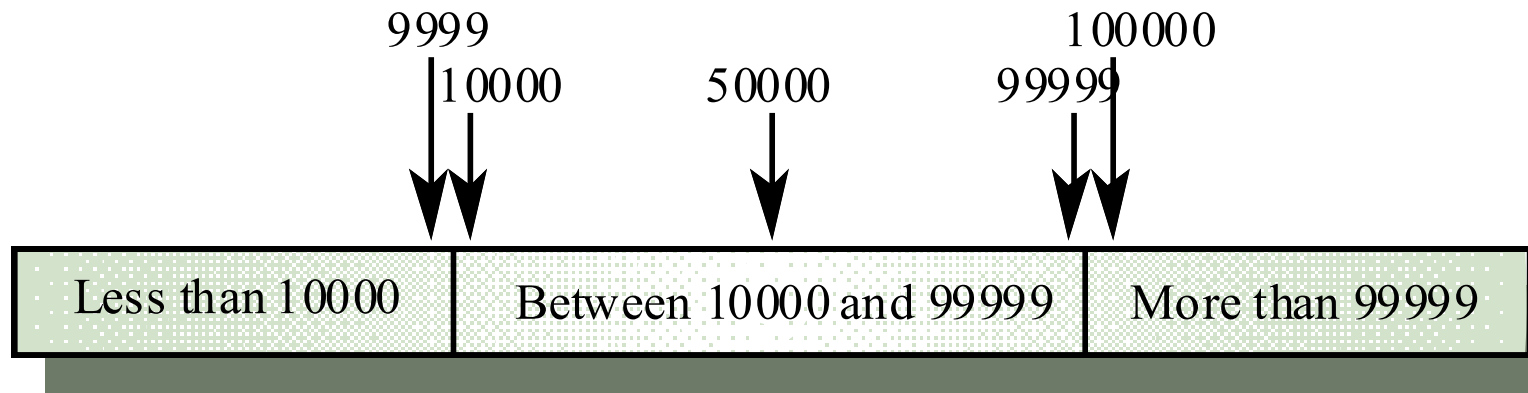
- Partition system inputs and outputs into 'equivalence sets'
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are  $<10,000$ ,  $10,000-99,999$  and  $>99,999$
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 10001



# Equivalence partitions



Number of input values



Input values

# 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

---

<b>Array</b>	<b>Element</b>
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

<b>Input sequence (T)</b>	<b>Key (Key)</b>	<b>Output (Found, L)</b>
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, ??

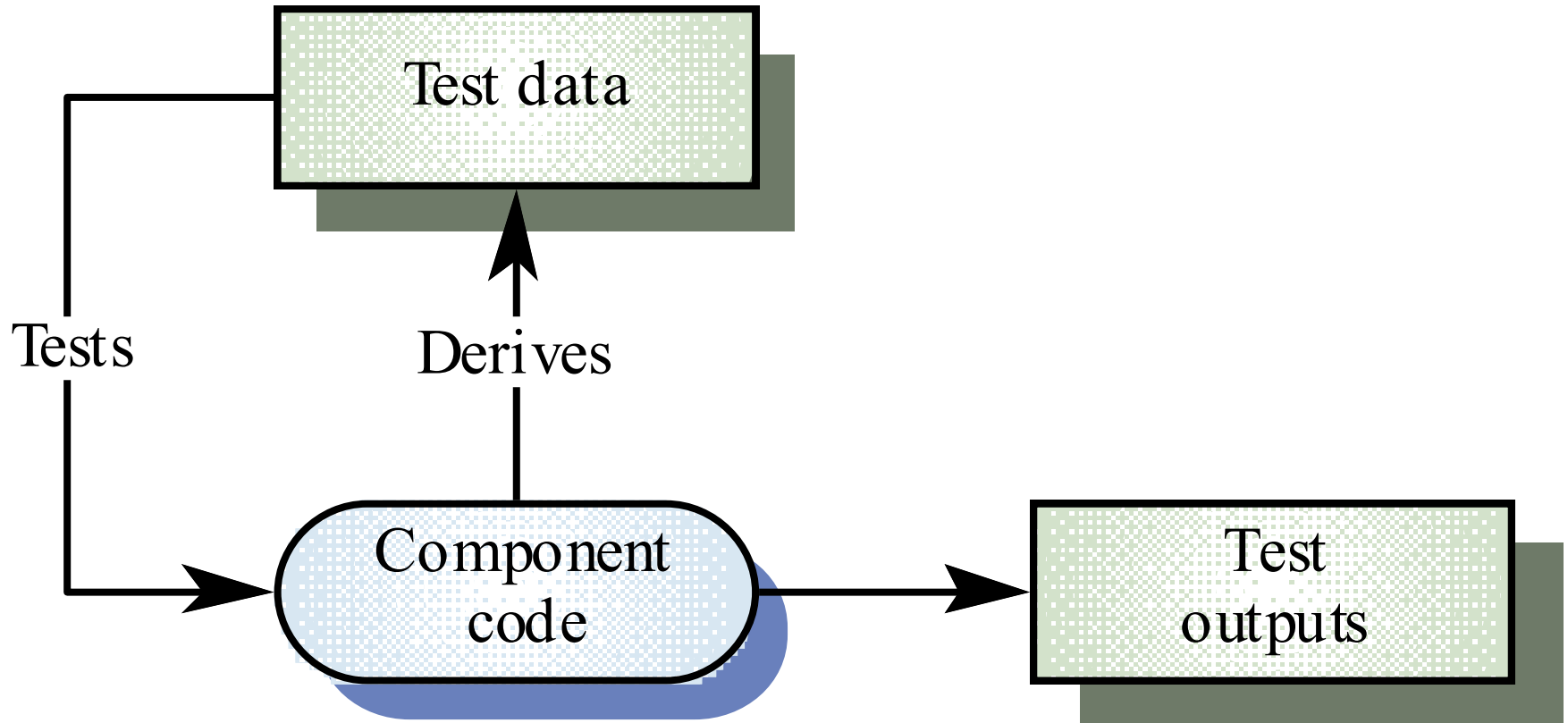
# Structural testing

---

- 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

---



```

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 (Java)



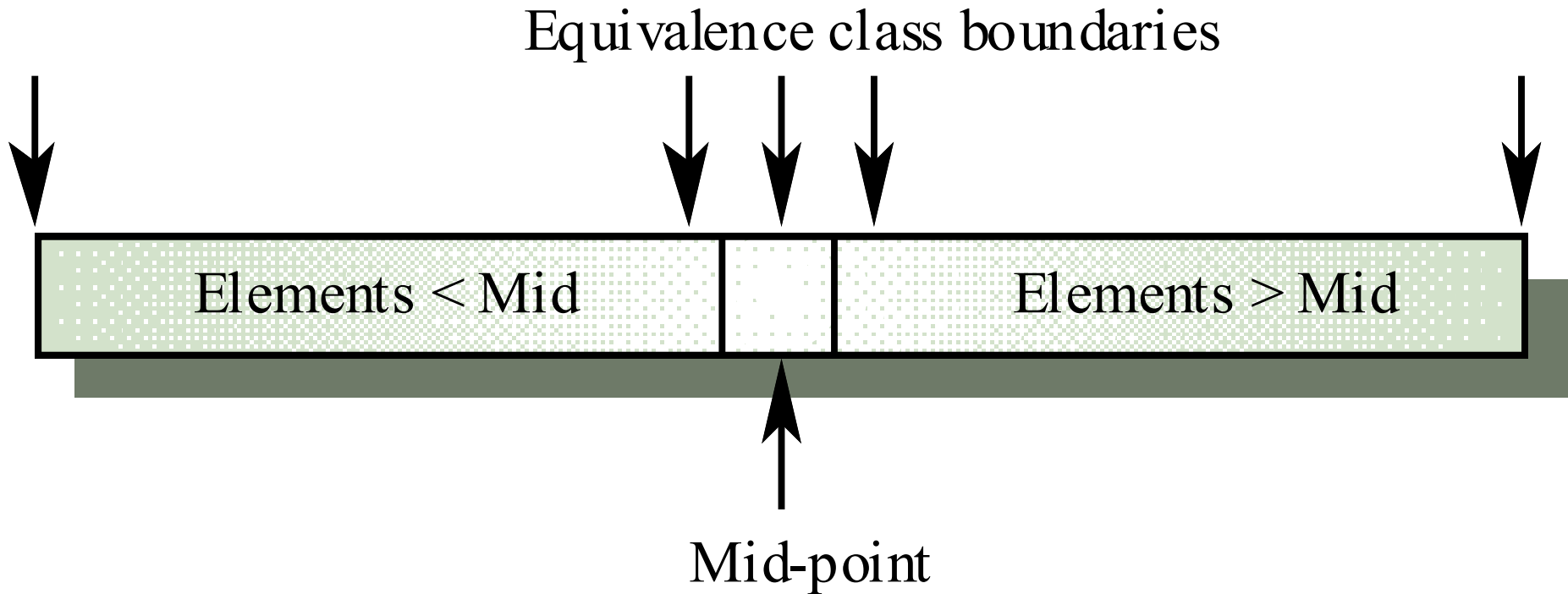
# Binary search - equiv. partitions

---

- 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

---

<b>Input array (T)</b>	<b>Key (Key)</b>	<b>Output (Found, L)</b>
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

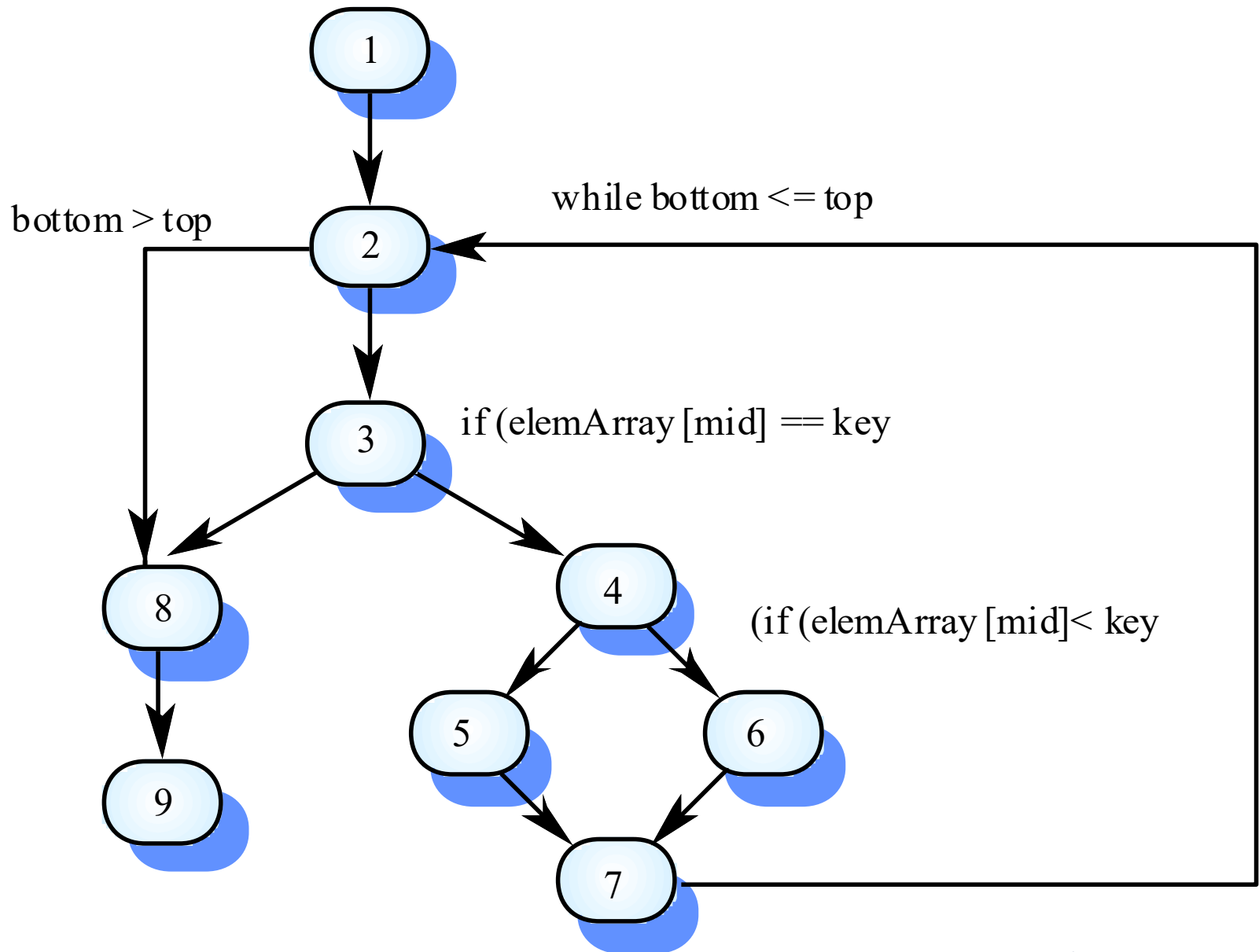
---

- 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

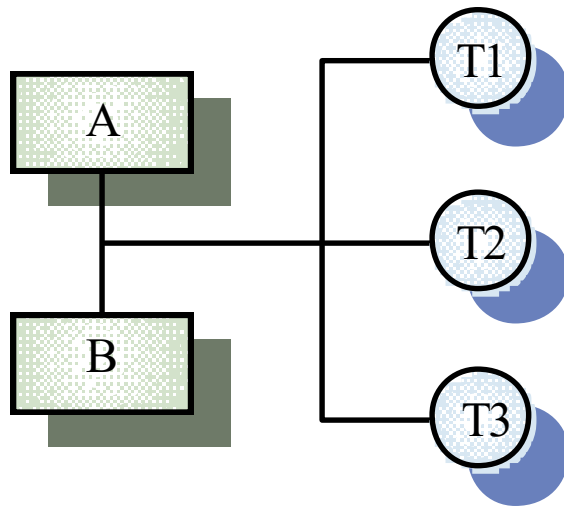


# Integration testing

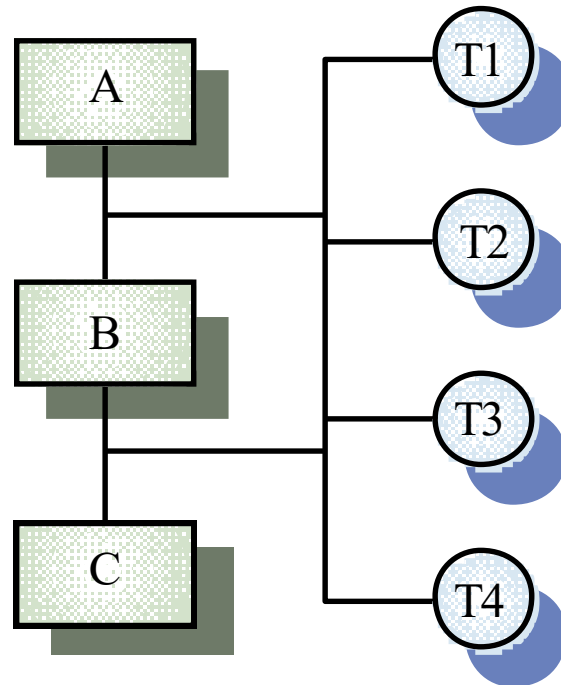
---

- 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

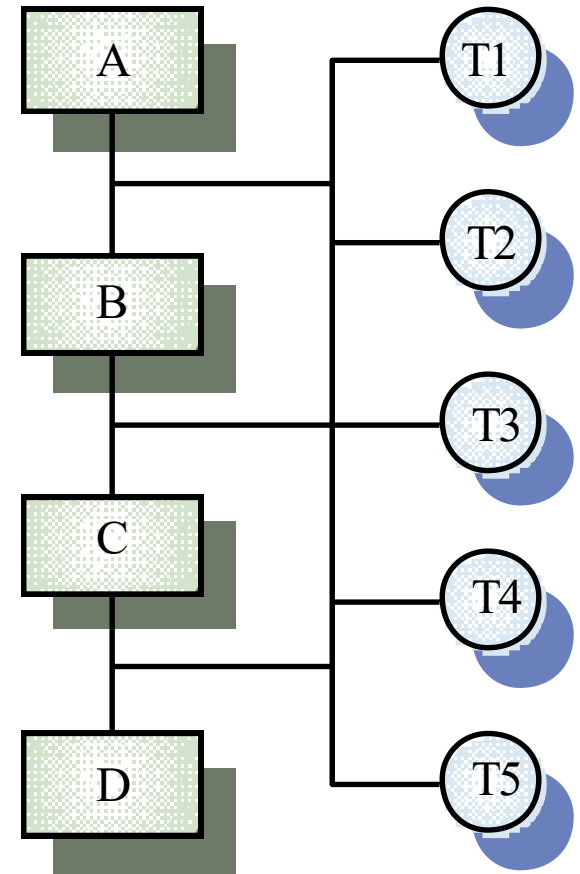
# Incremental integration testing



Test sequence  
1



Test sequence  
2



Test sequence  
3

# 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

**Q: For which types of system is bottom-up testing appropriate, and why?**

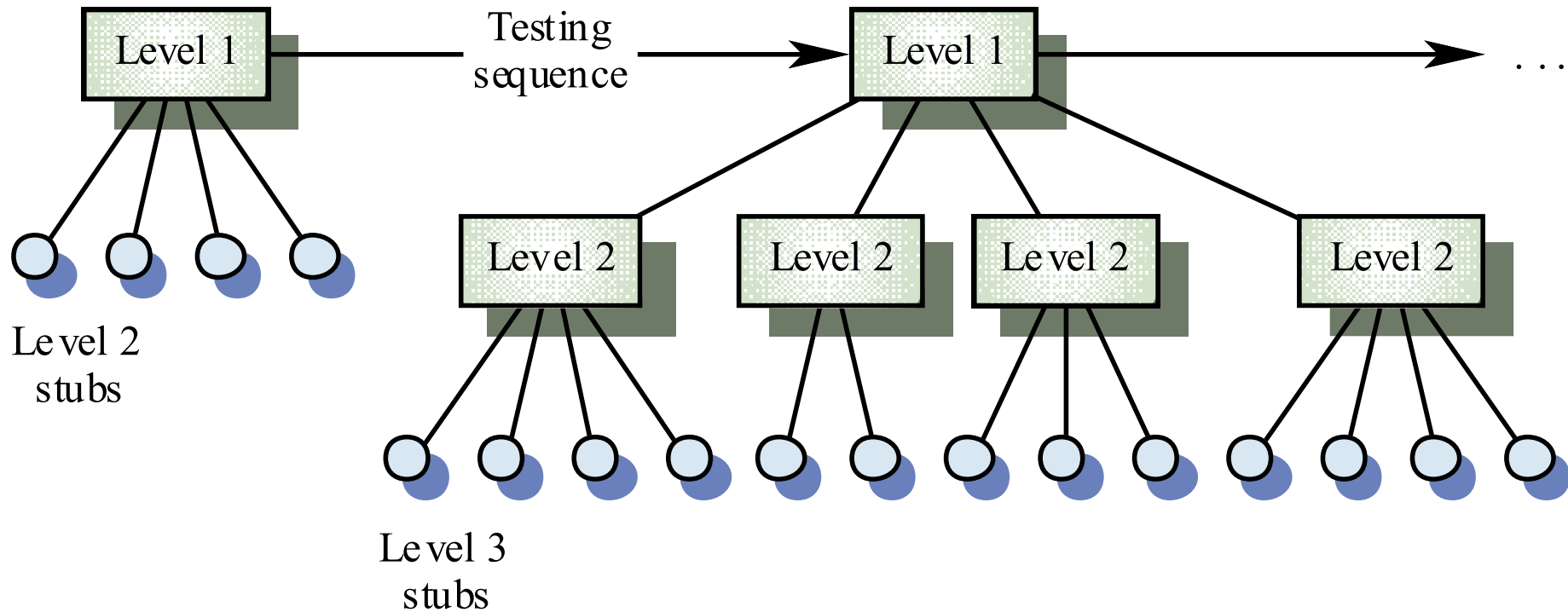
---

**Answer:**

1. **Object-Oriented Systems** – because these have a neat decomposition into classes and methods –makes testing easy
2. **Real-Time Systems** – because we can identify slow bits of code more quickly
3. **systems with strict performance requirements** – because we can measure the performance of individual methods early in the testing process

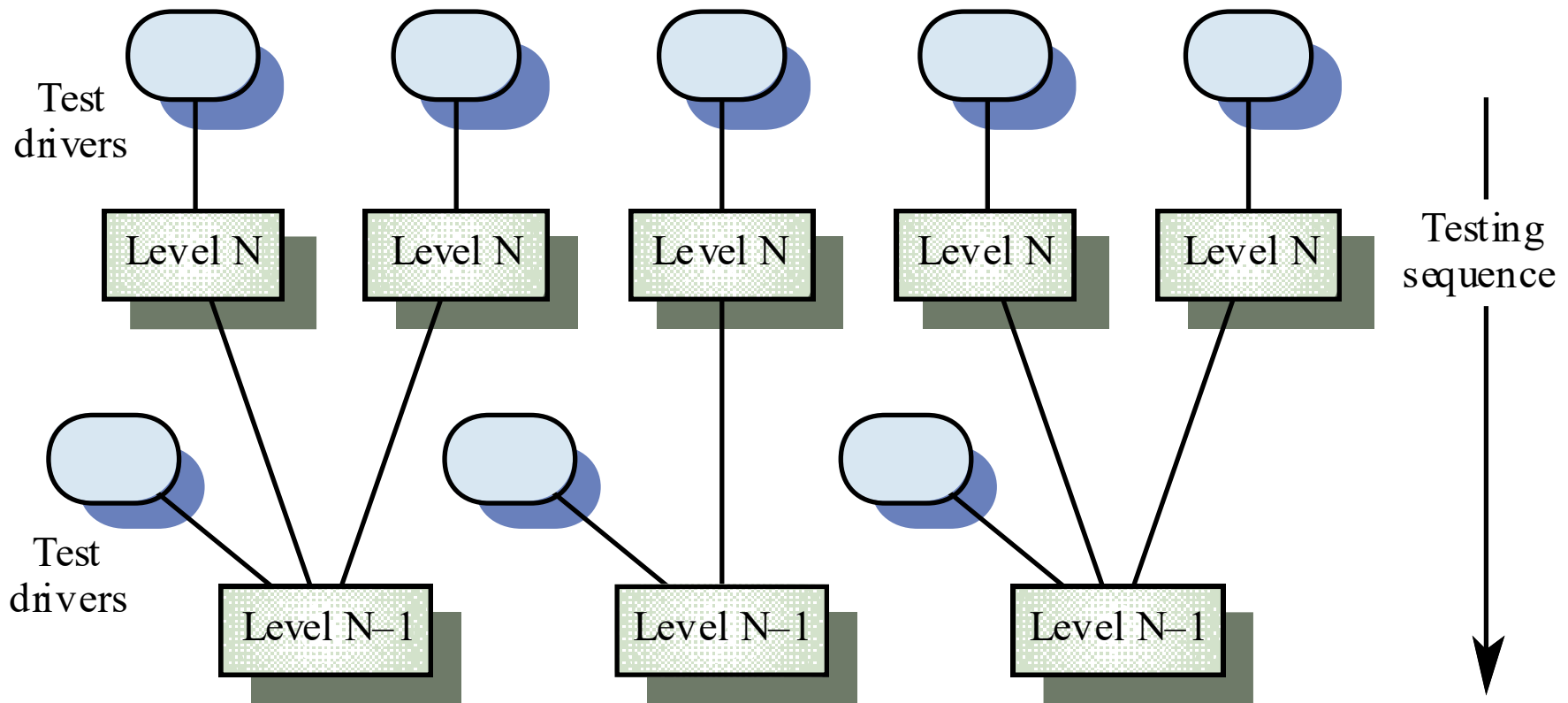
# Top-down testing

---



# Bottom-up testing

---



# Testing approaches

---

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration testing
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

# Interface testing

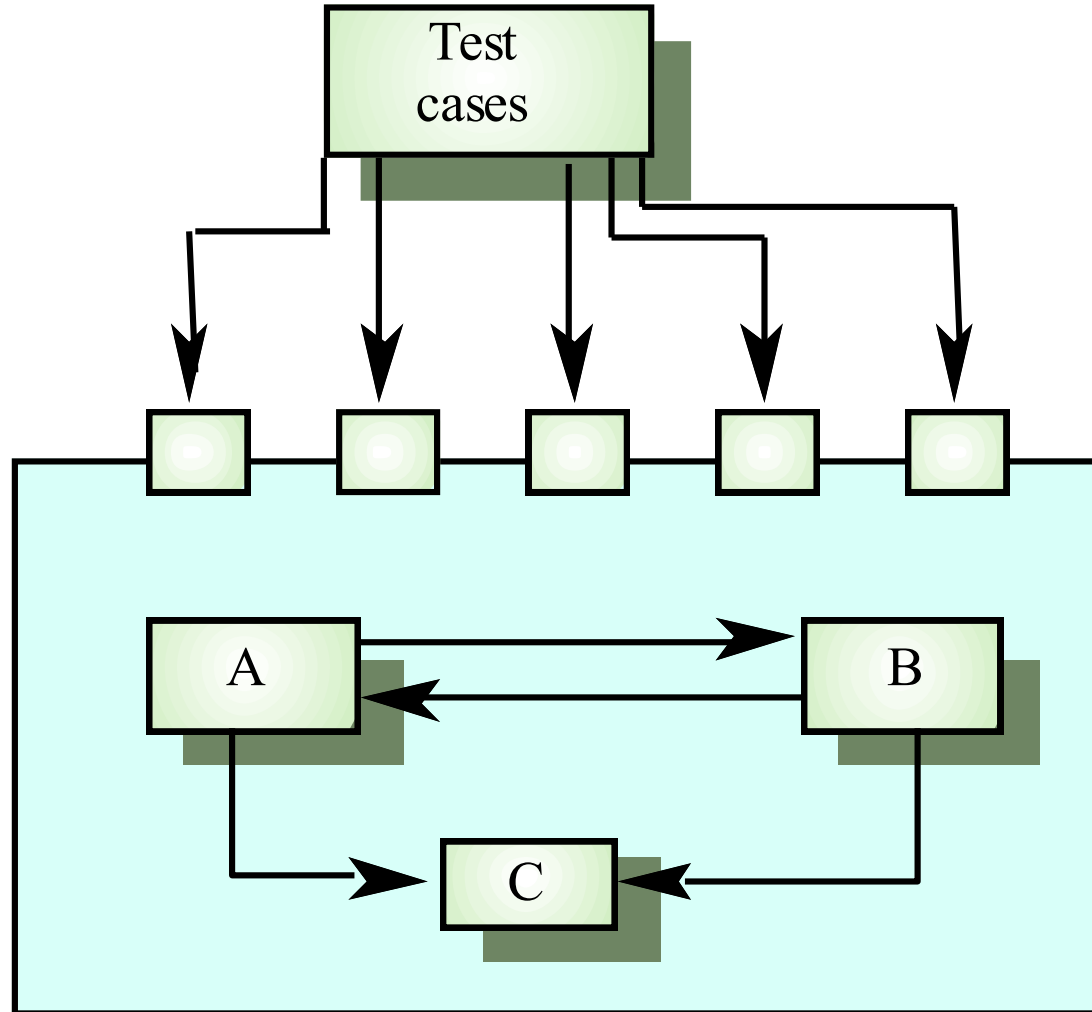
---

- 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



# Interface testing

---



# 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 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

# Stress testing

---

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light
- Stressing the system test failure behaviour.. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data
- Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded

# Object-oriented testing

---

- The components to be tested are object classes that are instantiated as objects
- Larger grain than individual functions so approaches to white-box testing have to be extended
- No obvious 'top' to the system for top-down integration and testing

# Testing levels

---

- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

# Object class testing

---

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised



# Weather station object interface

---

WeatherStation
identifier
reportWeather () calibrate (instruments) test () startup (instruments) shutdown (instruments)

- Test cases are needed for all operations
- Use a state model to identify state transitions for testing
- Examples of testing sequences
  - Shutdown → Waiting → Shutdown
  - Waiting → Calibrating → Testing → Transmitting → Waiting
  - Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

# Object integration

---

- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

# Approaches to cluster testing

---

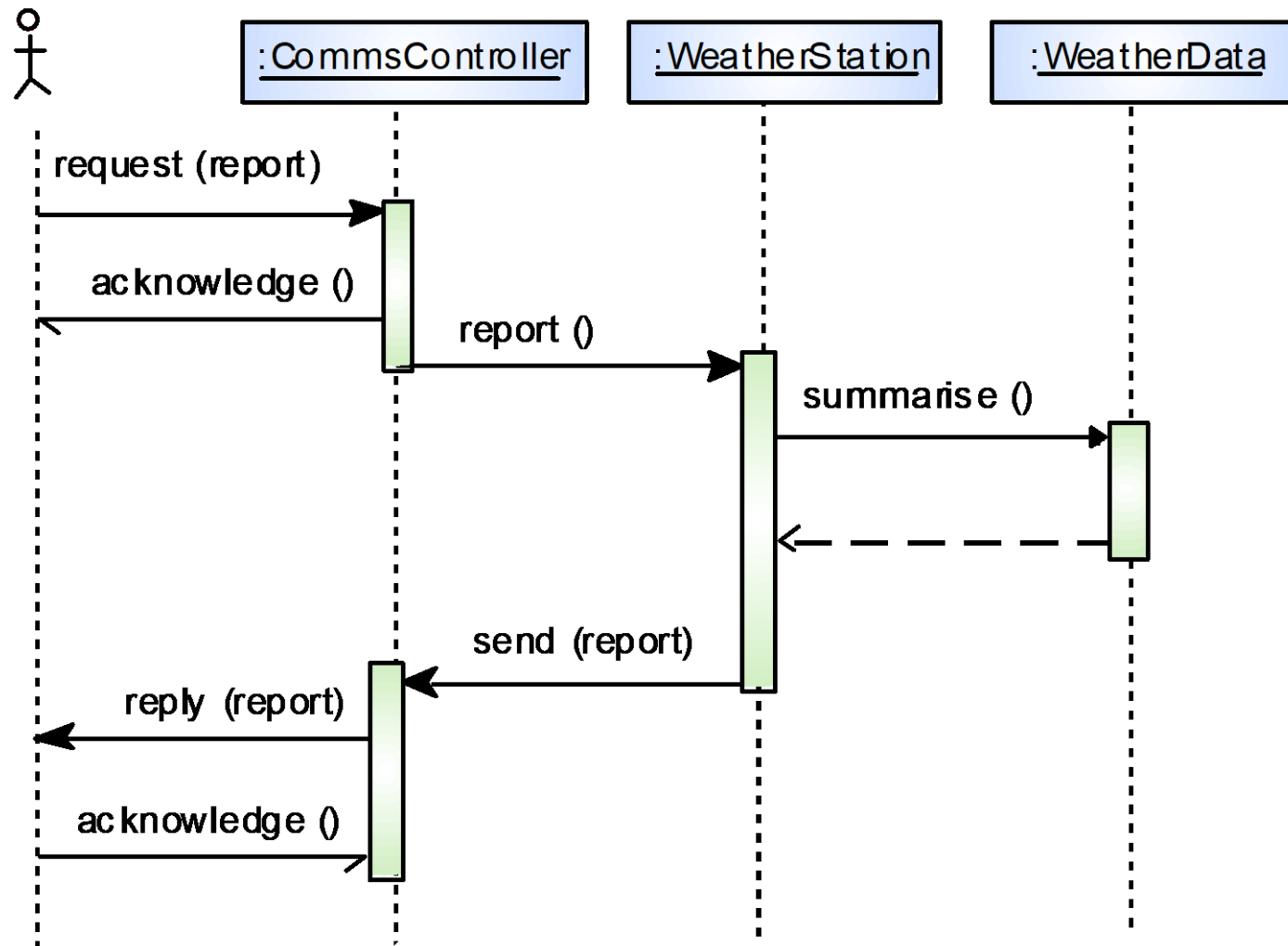
- Use-case or scenario testing
  - Testing is based on a user interactions with the system
  - Has the advantage that it tests system features as experienced by users
- Thread testing
  - Tests the systems response to events as processing threads through the system
- Object interaction testing
  - Tests sequences of object interactions that stop when an object operation does not call on services from another object

# Scenario-based testing

---

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Consider the scenario in the weather station system where a report is generated

# Collect weather data



# Weather station testing

---

- Thread of methods executed
  - CommsController:request → WeatherStation:report → WeatherData:summarise
- Inputs and outputs
  - Input of report request with associated acknowledge and a final output of a report
  - Can be tested by creating raw data and ensuring that it is summarised properly
  - Use the same raw data to test the WeatherData object

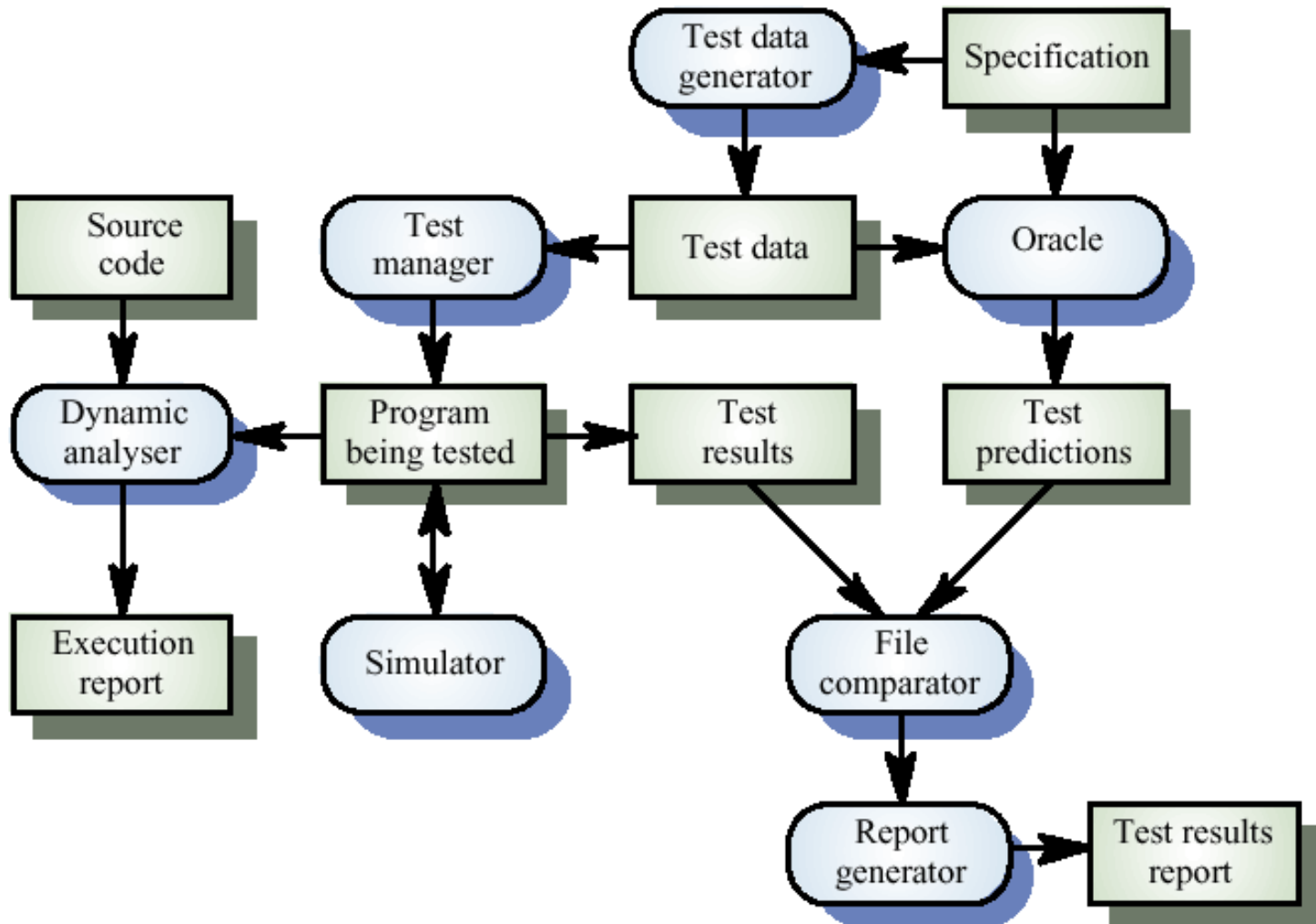
# Testing workbenches

---

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs
- Most testing workbenches are open systems because testing needs are organisation-specific
- Difficult to integrate with closed design and analysis workbenches

# A testing workbench

---





# Testing workbench adaptation

---

- Scripts may be developed for user interface simulators and patterns for test data generators
- Test outputs may have to be prepared manually for comparison
- Special-purpose file comparators may be developed

# 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

# Key points

---

- Test coverage measures ensure that all statements have been executed at least once.
- Interface defects arise because of specification misreading, misunderstanding, errors or invalid timing assumptions
- To test object classes, test all operations, attributes and states
- Integrate object-oriented systems around clusters of objects