Verification and Validation

- Assuring that a software system meets a user’s needs

Verification vs Validation

- **Verification**: “Are we building the product right”
  - The software should conform to its specification.
- **Validation**: “Are we building the right product”
  - The software should do what the user really requires.

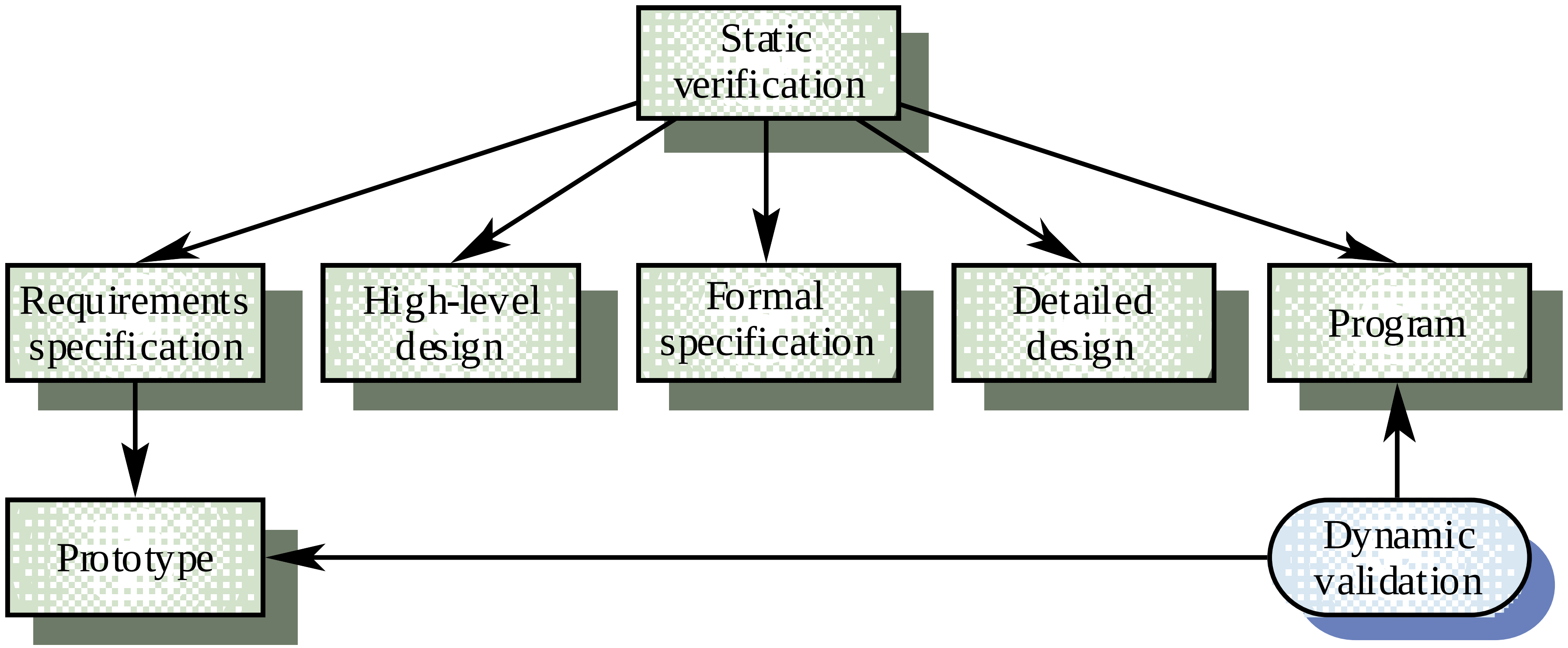
The V & V Process

- 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.

Dynamic and Static Verification

- *Dynamic V & V* Concerned with exercising and observing product behavior (testing).
- *Static verification* Concerned with analysis of the static system representation to discover problems.

Static and Dynamic V&V



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 Success

- Many different defects may be discovered in a single inspection. In testing, one defect, may mask another so several executions are required.
- They 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.

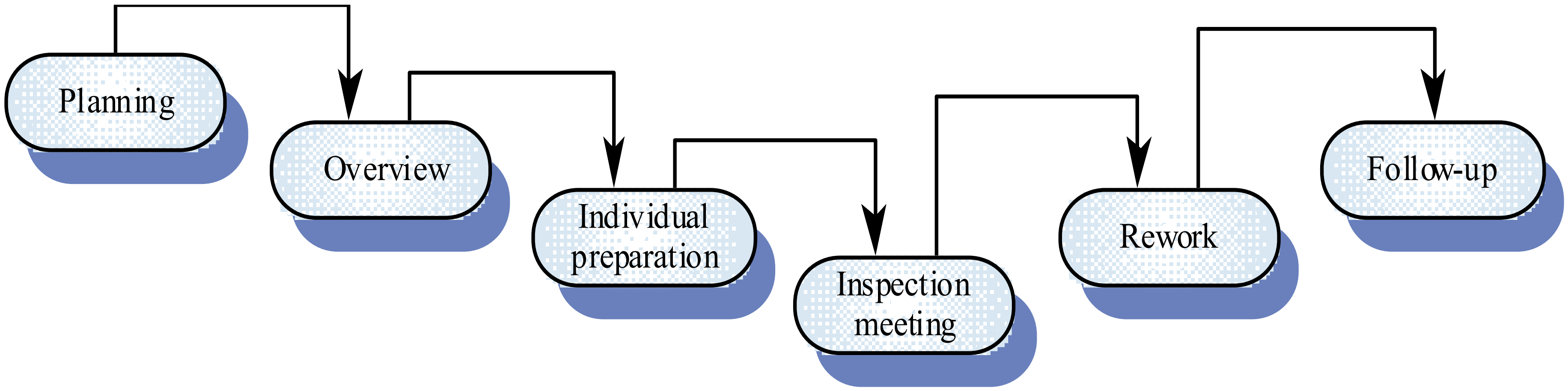
Program Inspections

- Formalised 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 uninitialised 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.

The inspection process



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: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection checks

Fault class	**Inspection check**
Data faults	Are all program variables initialized before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or size – 1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/Output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all functions and procedure calls have the correct number of parameters? Do formal and actual parameters types match? Are the parameters in the correct order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account.

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.

Program Testing

- Can reveal the presence of errors NOT their absence.
- A successful test is a test which discovers one or more errors.
- Only validation technique for non-functional requirements.
- Should be used in conjunction with static verification.

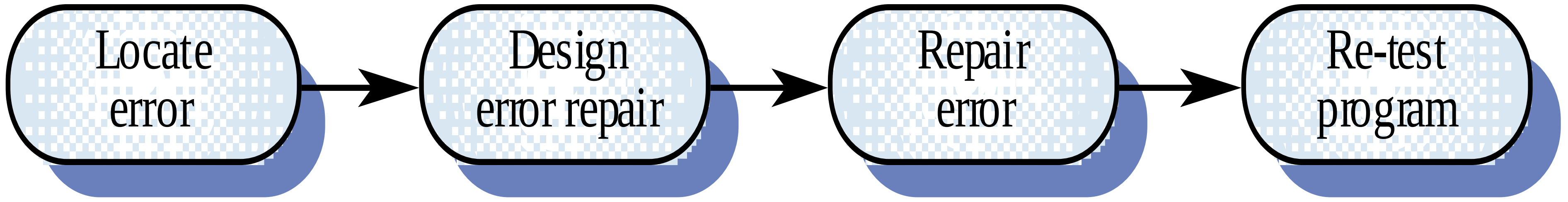
Types of Testing

- **Statistical testing**:
  - Tests designed to reflect the frequency of user inputs. Used for reliability estimation.
  - Covered in Chapter 17.1 - Software reliability.
- **Defect testing**:
  - Tests designed to discover system defects.
  - A successful defect test is one which reveals the presence of defects in a system.
  - Covered in Chapter 20.1 - Defect testing.

Testing and Debugging

- Defect testing and debugging are distinct processes.
- Defect testing is concerned with confirming the presence of errors.
- Debugging is concerned with locating and repairing these errors.
- Debugging involves formulating a hypothesis about program behavior then testing these hypotheses to find the system error.

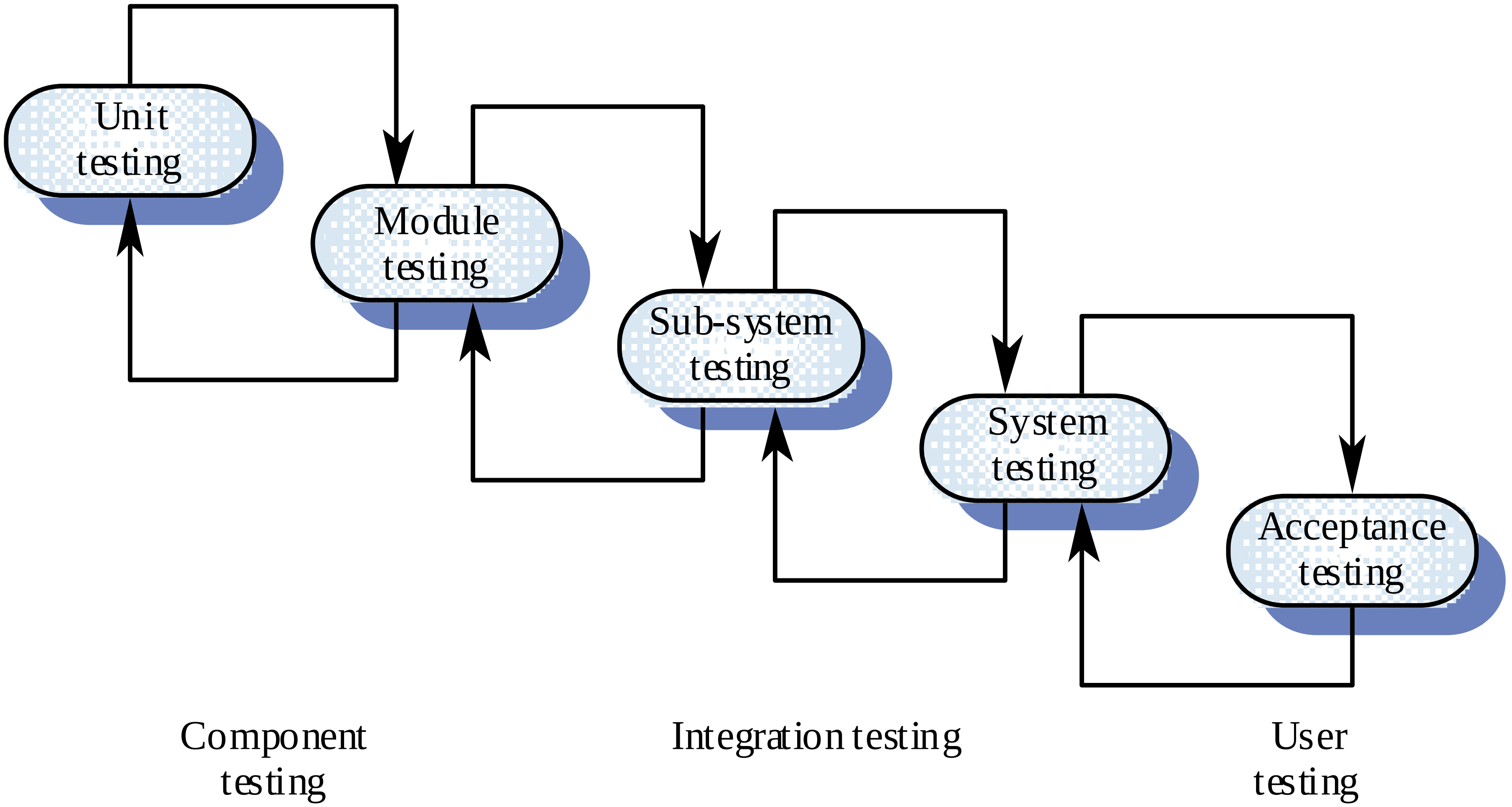
The Debugging Process



Testing Stages

- Unit testing
  - testing of individual components
- Module testing
  - testing of collections of dependent components
- Sub-system testing
  - testing collections of modules integrated into sub-systems
- System testing
  - testing the complete system prior to delivery
- Acceptance testing
  - testing by users to check that the system satisfies requirements. Sometimes called alpha testing

The Testing Process



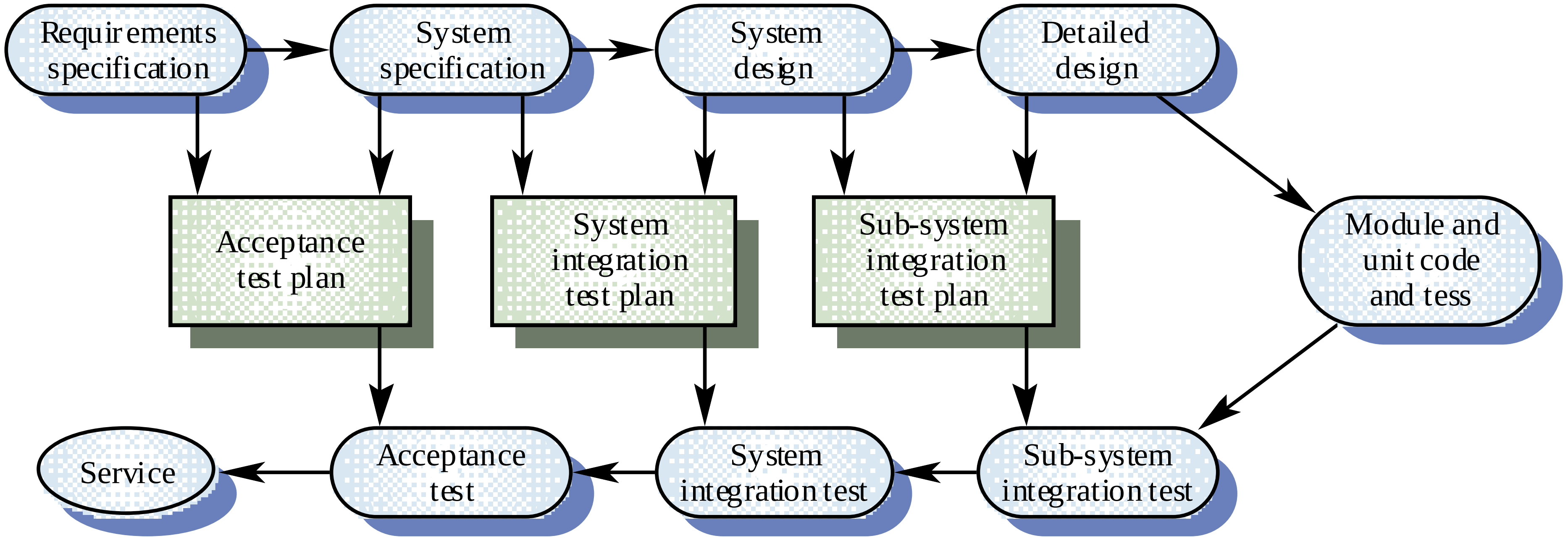
Object-oriented System Testing

- Less closely coupled systems. Objects are not necessarily integrated into sub-systems.
- Cluster testing. Test a group of cooperating objects.
- Thread testing. Test a processing thread as it weaves from object to object. Discussed later in real-time system testing.

Test Planning and Scheduling

- Describe major phases of the testing process.
- Describe traceability of tests to requirements.
- Estimate overall schedule and resource allocation.
- Describe relationship with other project plans.
- Describe recording method for test results.

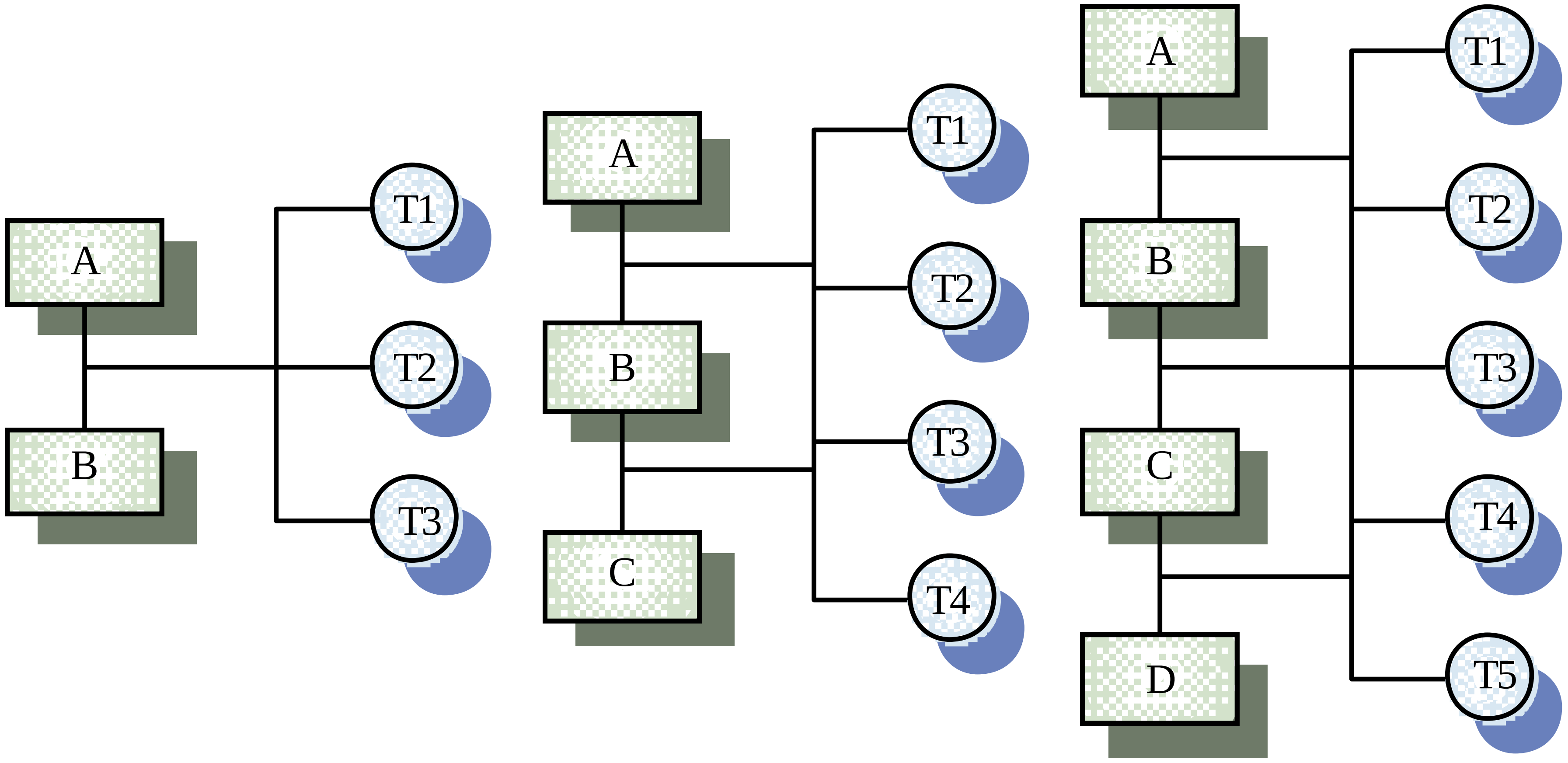
The V-model of development


Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and tess
Service
Acceptance test
System integration test
Sub-system integration test

Testing Strategies

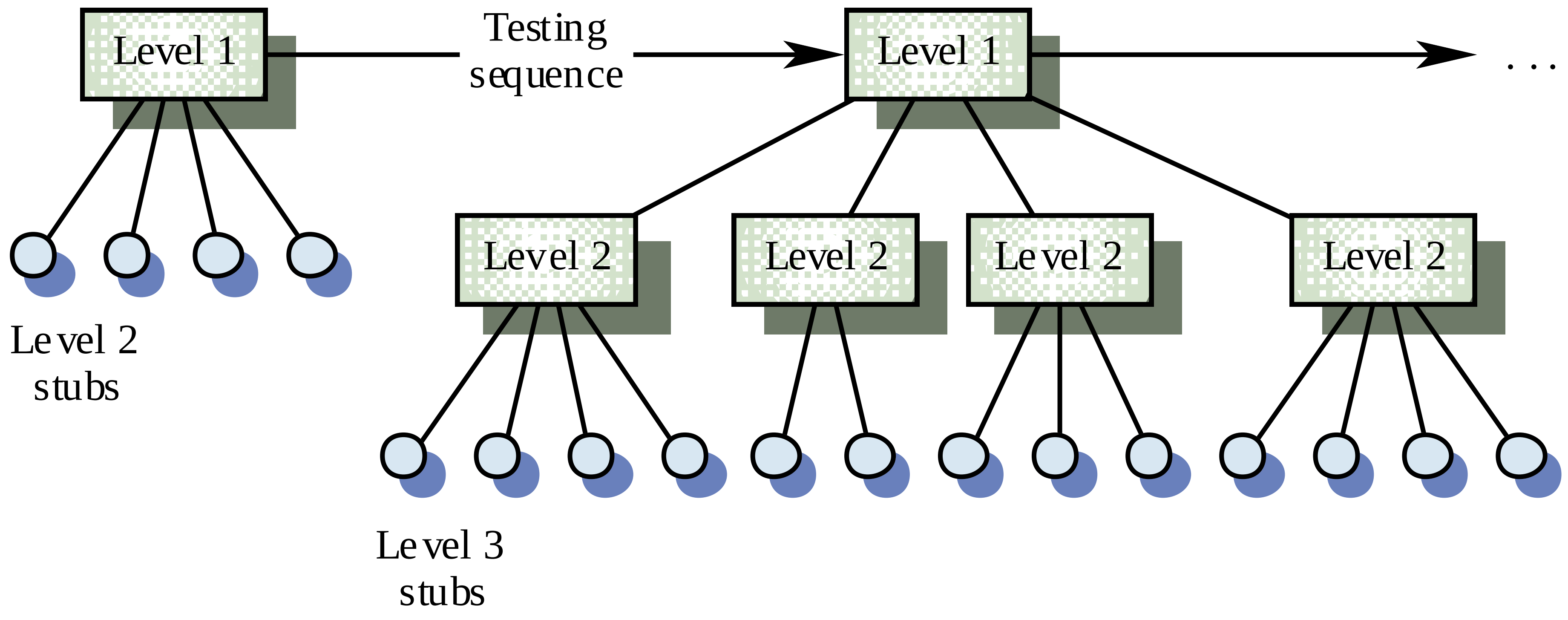
- Testing strategies are ways of approaching the testing process.
- Different strategies may be applied at different stages of the testing process.
- Strategies covered:
  - Top-down testing
  - Bottom-up testing
  - Stress testing
  - Back-to-back testing

Incremental Testing



Test sequence 1 Test sequence 2 Test sequence 3

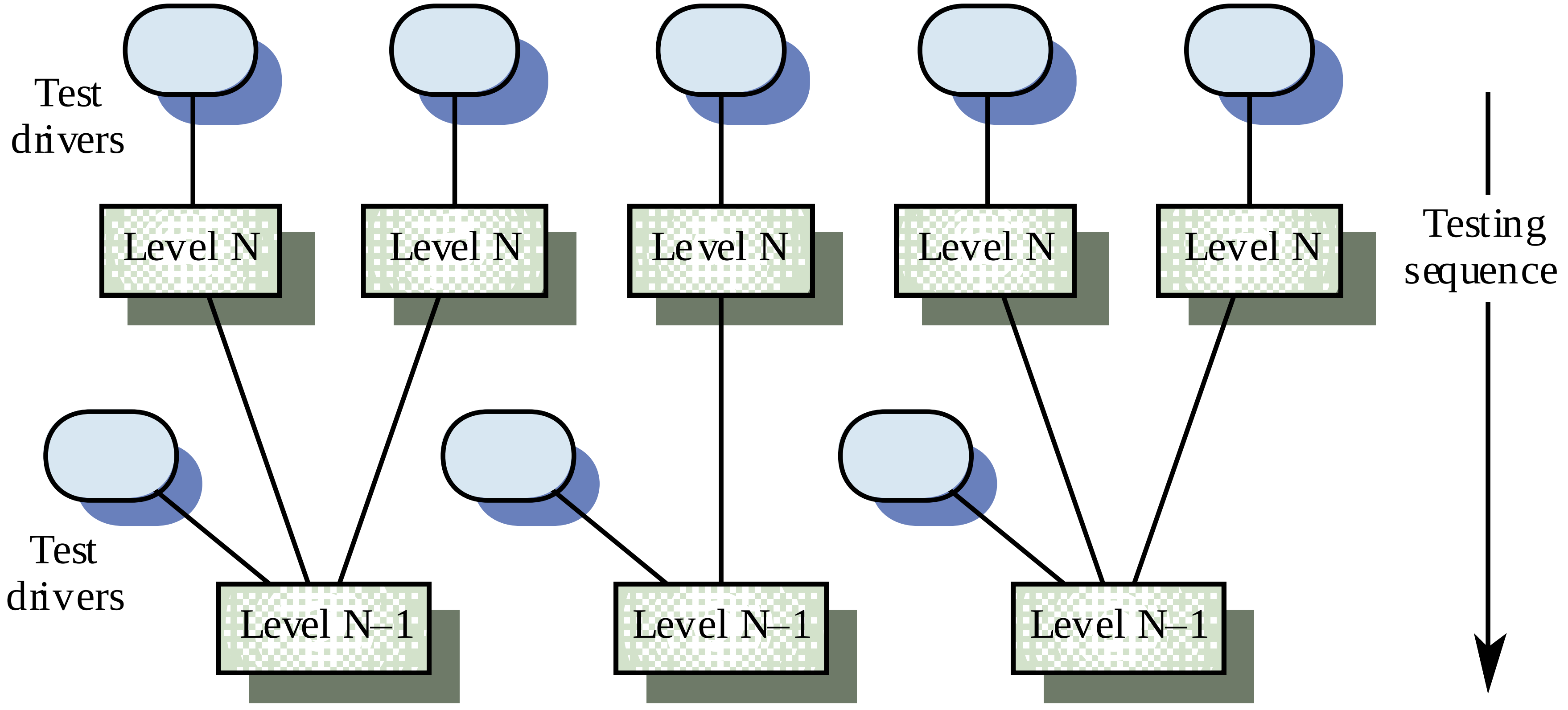
Top-down Testing



Top-down Testing

- Start with the high-levels of a system and work your way downwards.
- Testing strategy which is used in conjunction with top-down development.
- Finds architectural errors.
- May need system infrastructure before any testing is possible.
- May be difficult to develop program stubs.

Bottom-up Testing



Bottom-up Testing

- Necessary for critical infrastructure components.
- Start with the lower levels of the system and work upward.
- Needs test drivers to be implemented.
- Does not find major design problems until late in the process.
- Appropriate for object-oriented systems.

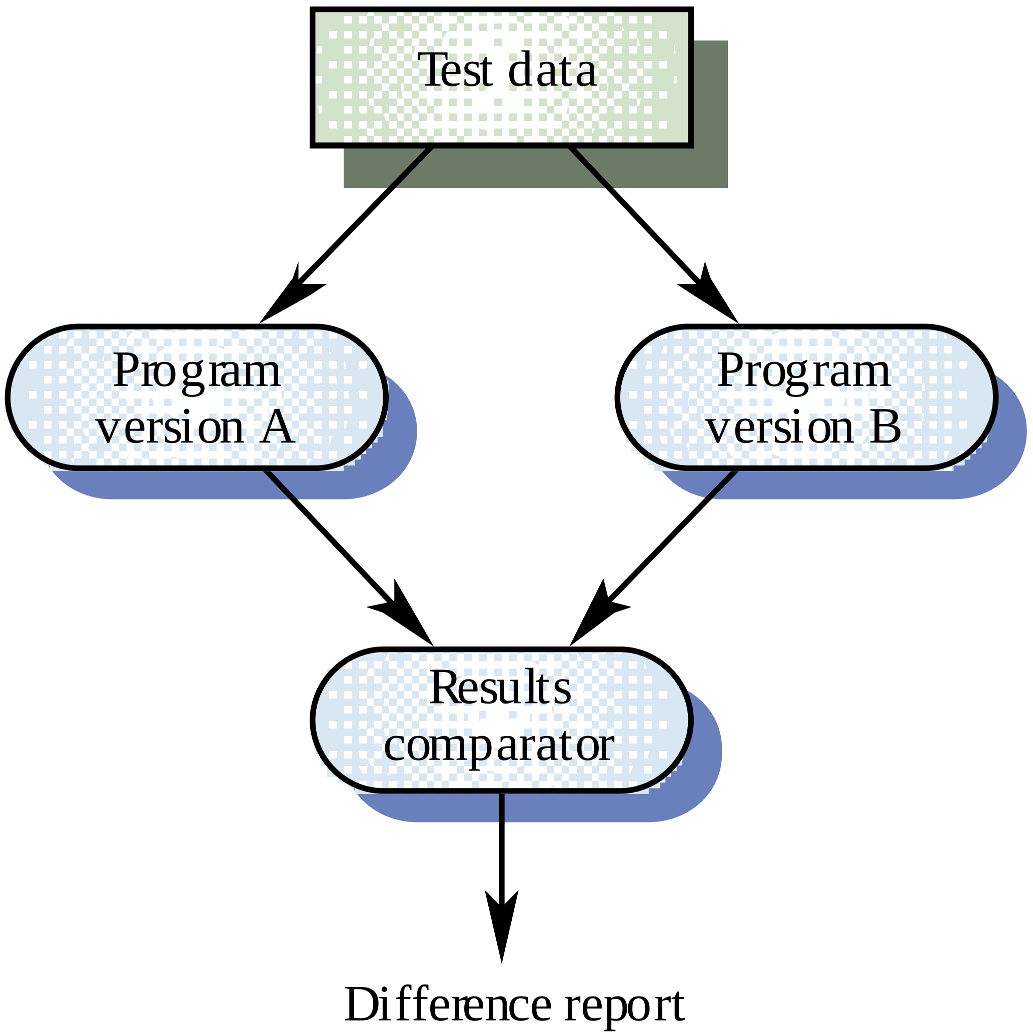
Stress Testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behavior. 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.

Back-to-back Testing

- Present the same tests to different versions of the system and compare outputs. Differing outputs imply potential problems.
- Reduces the costs of examining test results. Automatic comparison of outputs.
- Possible when a prototype is available or with regression testing of a new system version.

Back-to-back Testing



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.

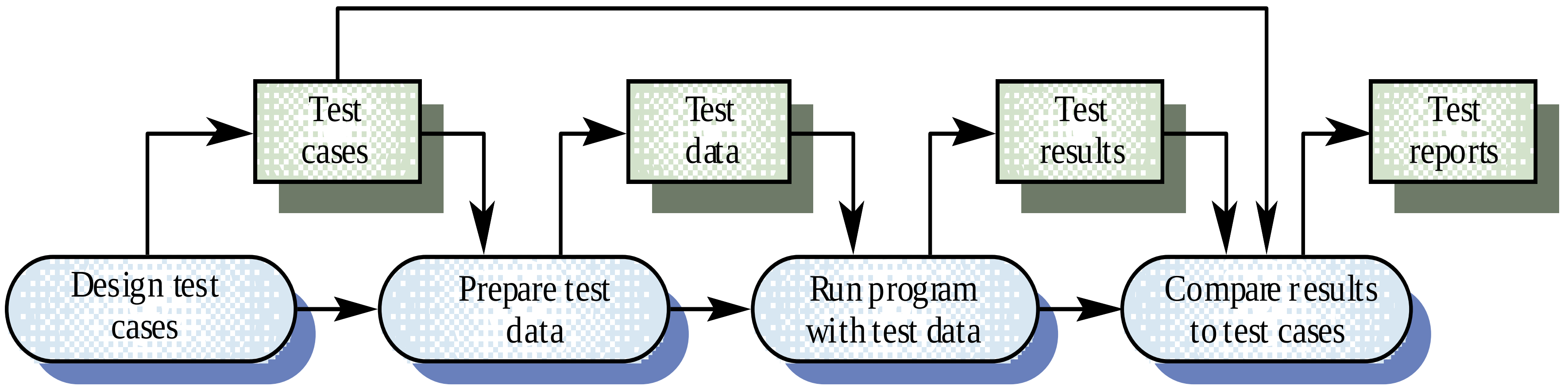
Testing Priorities

- 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.

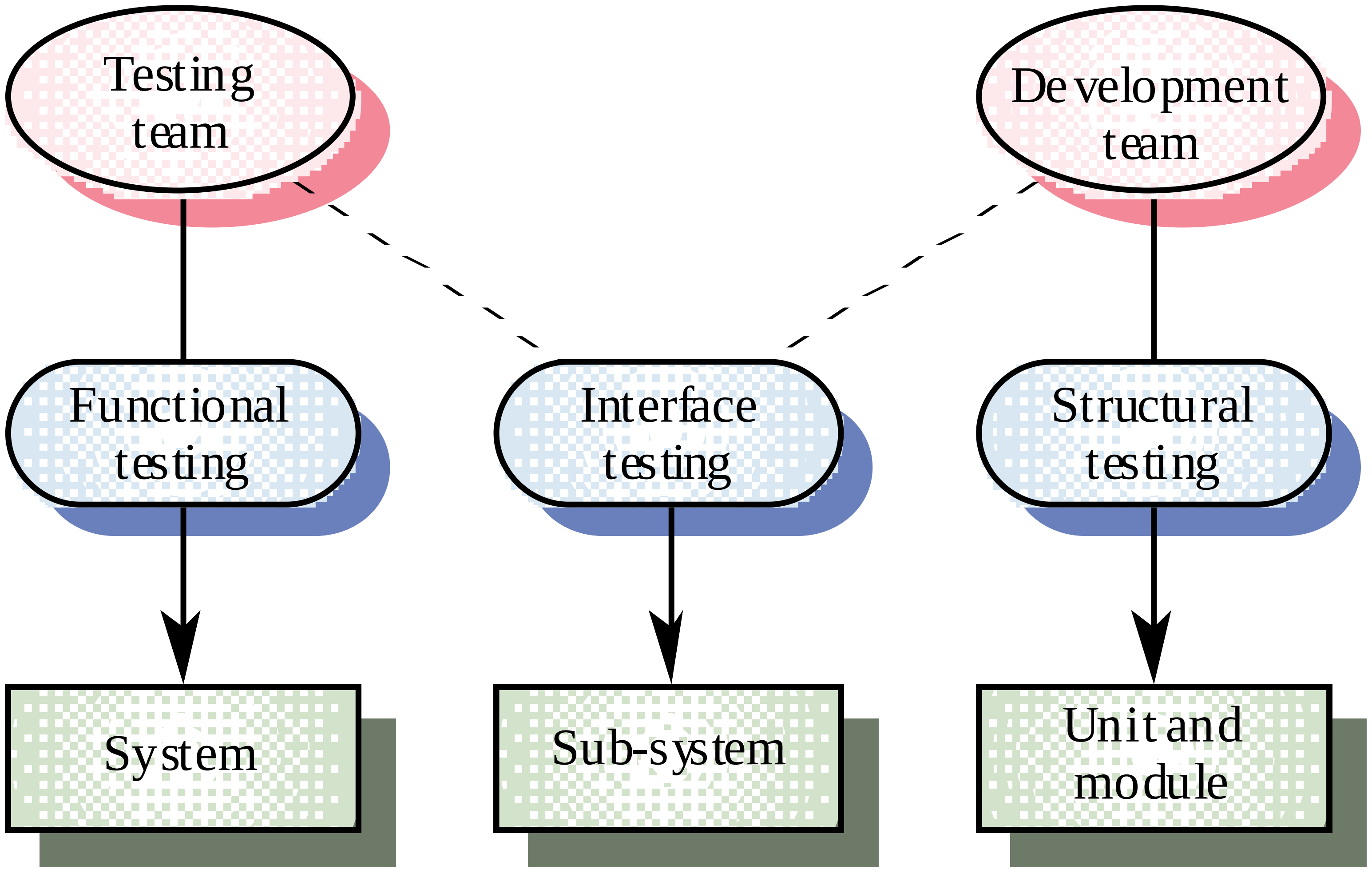
Test Data and Test Cases

- ***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



Defect Testing Approaches



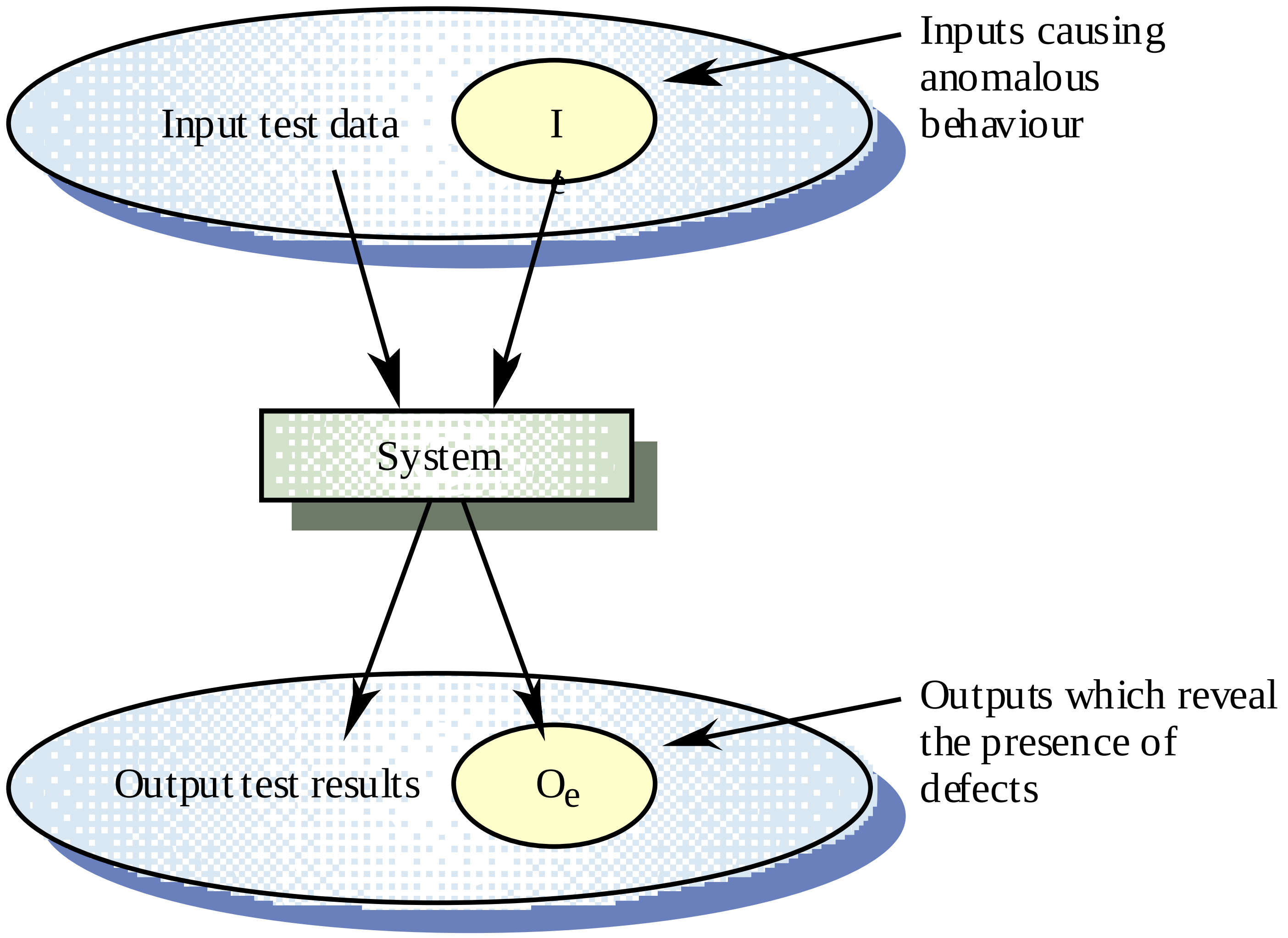
Testing Effectiveness

- 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.

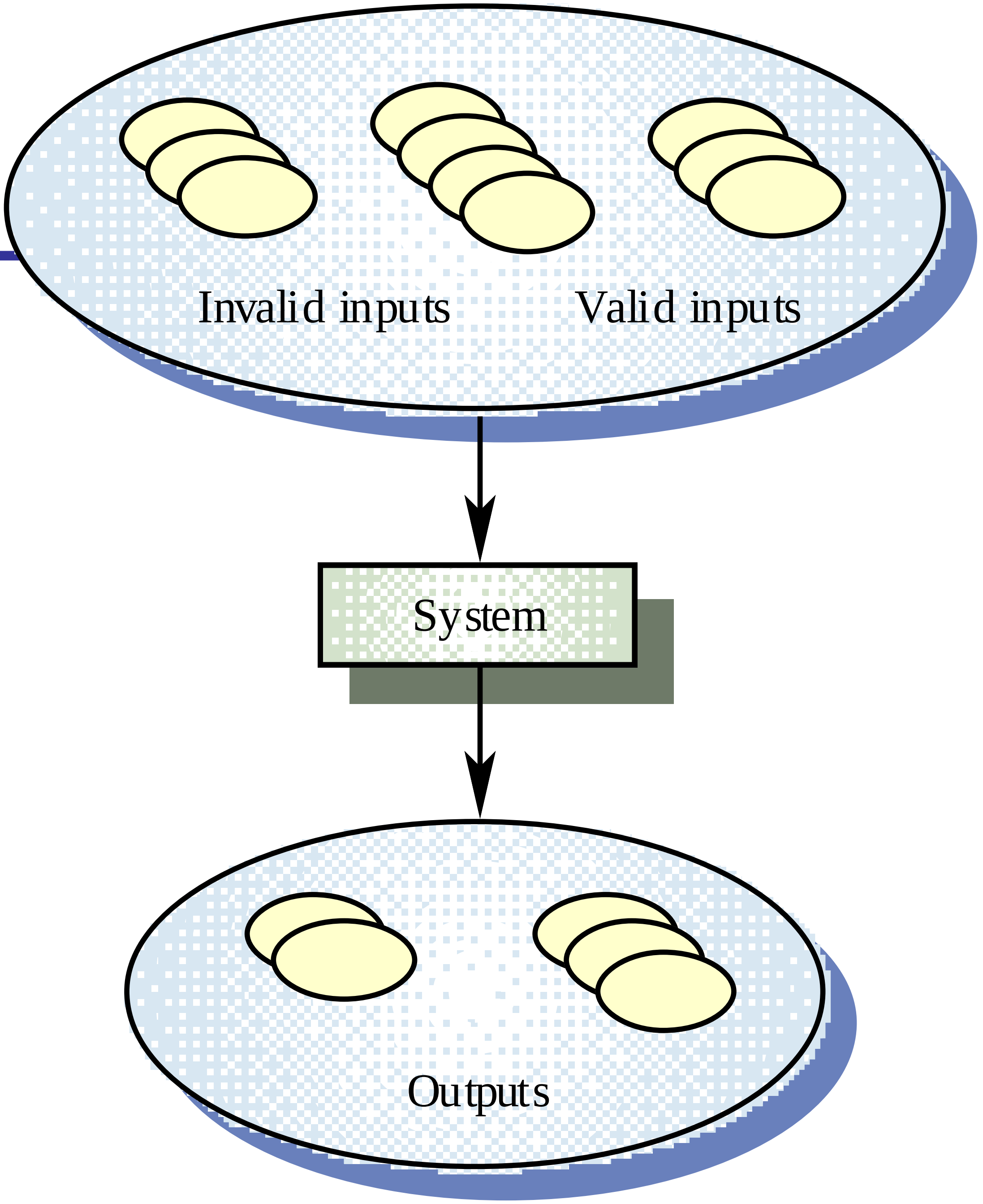
Black-box Testing

- 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


Input test data
I
e
Inputs causing anomalous behaviour
System
Output test results
O
e
Outputs which reveal the presence of defects

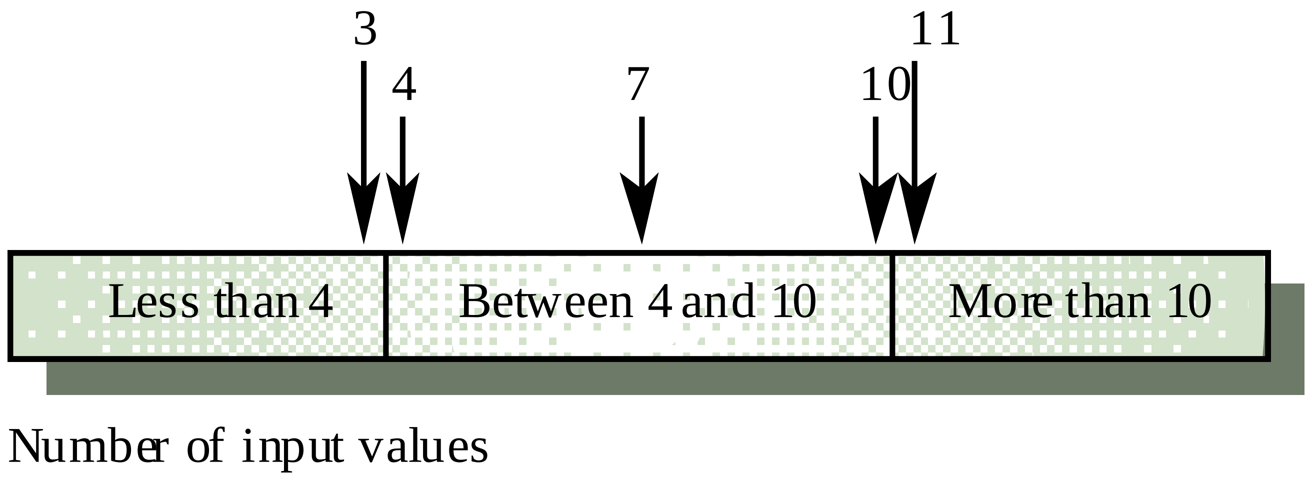
Equivalence Partitioning

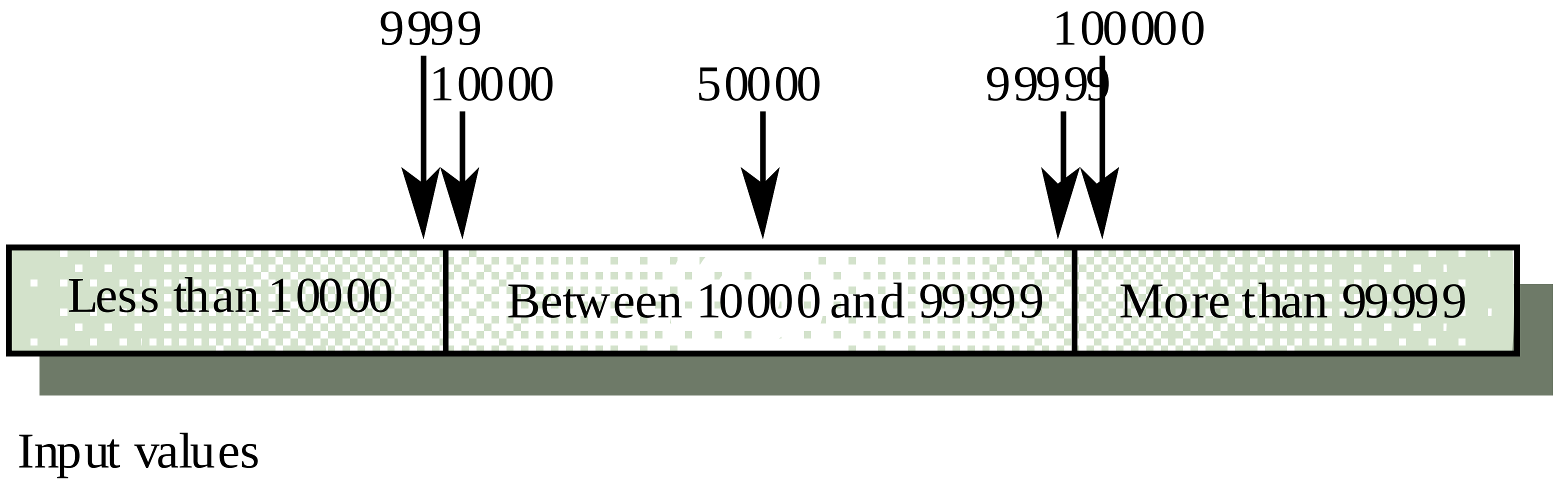


Equivalence Partitioning

- 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

Equivalence Partitions





Search Routine Specification

```
procedure Search (Key : INTEGER ; T: array 1..N of INTEGER;
    Found : BOOLEAN; L: 1..N) ;

Pre-condition
        -- the array has at least one element
        1 <= N
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, 1 >= i >= N, 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

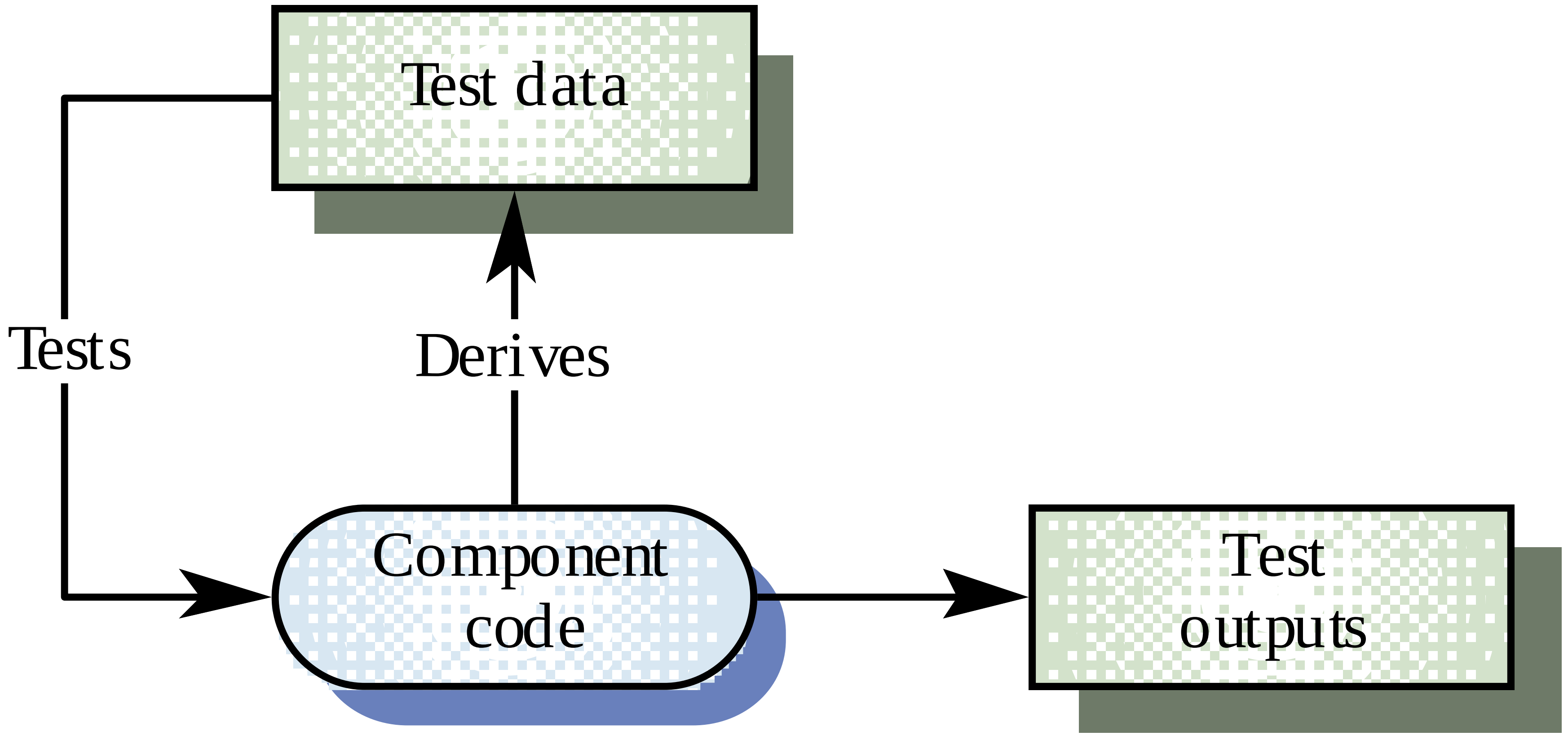
Search Routine - Test Cases

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

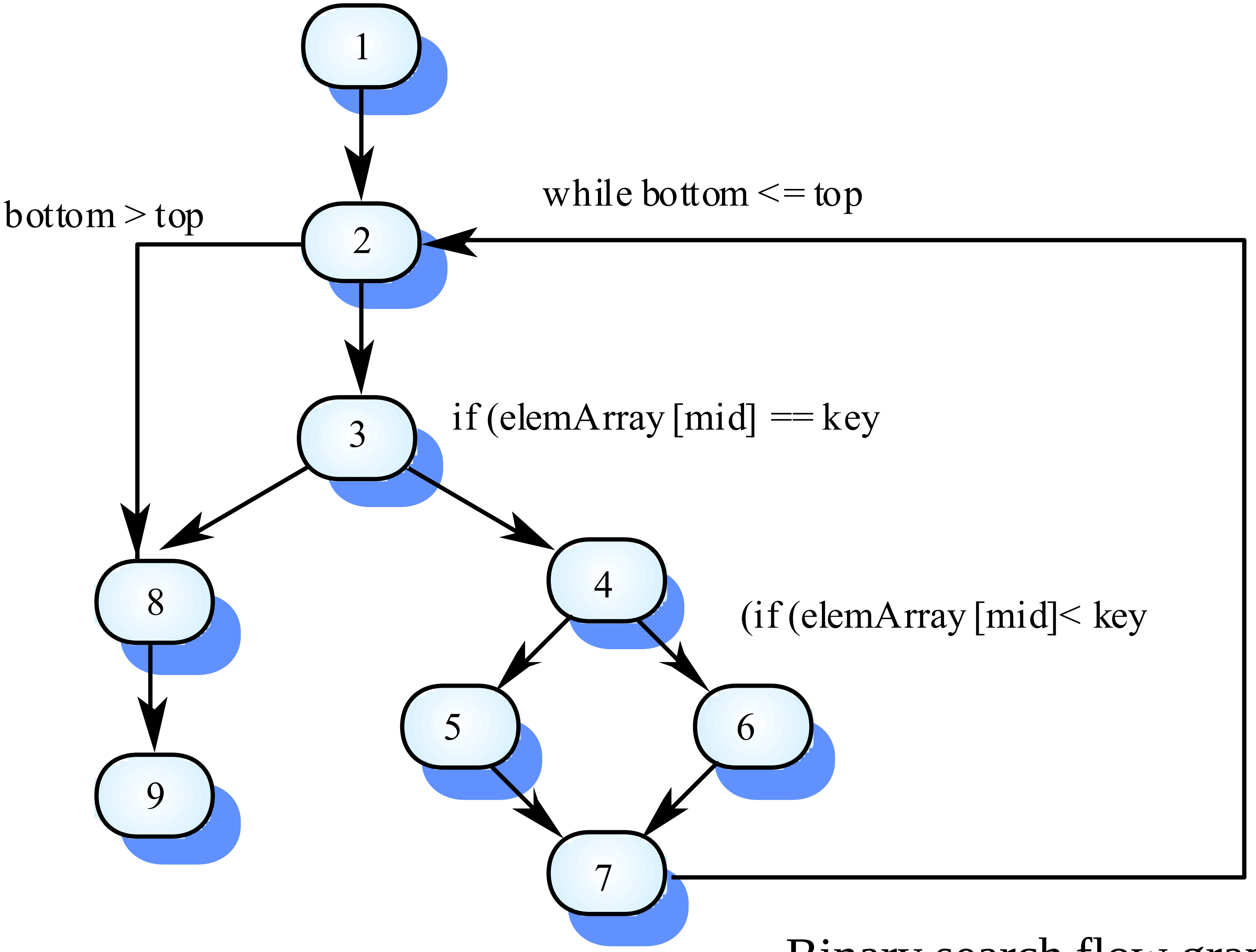


Binary search (Java)

```
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
```

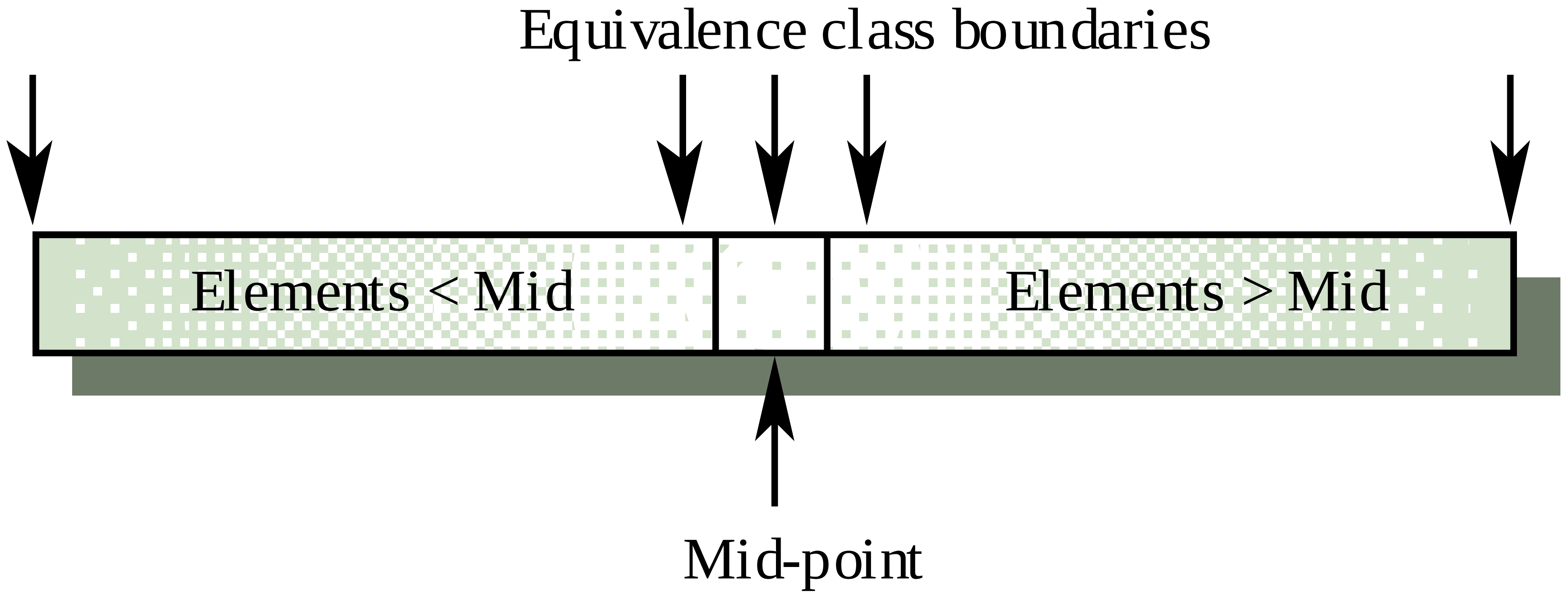


Binary search flow graph

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



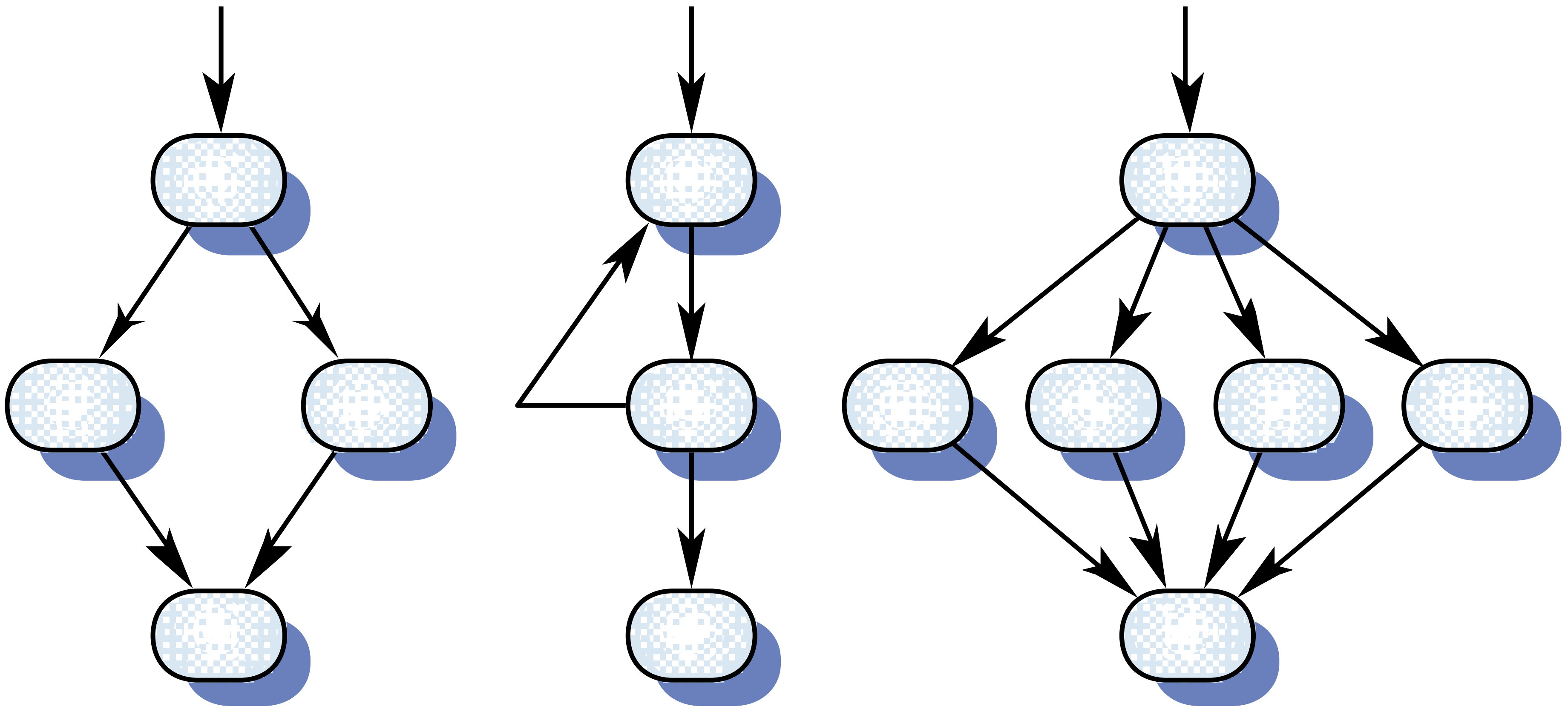
Binary Search - Test Cases

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, ??

Program Flow Graphs

- Describes the program control flow.
- Used as a basis for computing the cyclomatic complexity.
- Complexity = Number of edges - Number of nodes + 2N

Flow Graph Representations



if-then-else   loop-while   case-of

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.

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
- Subsystem 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.

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.

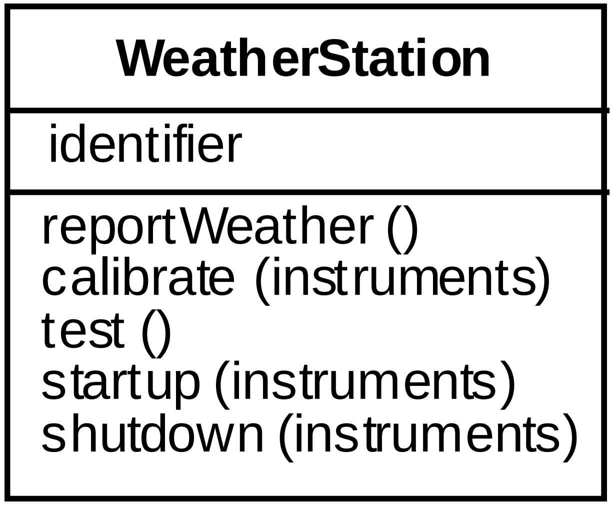
Testing Levels

- 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



- 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.

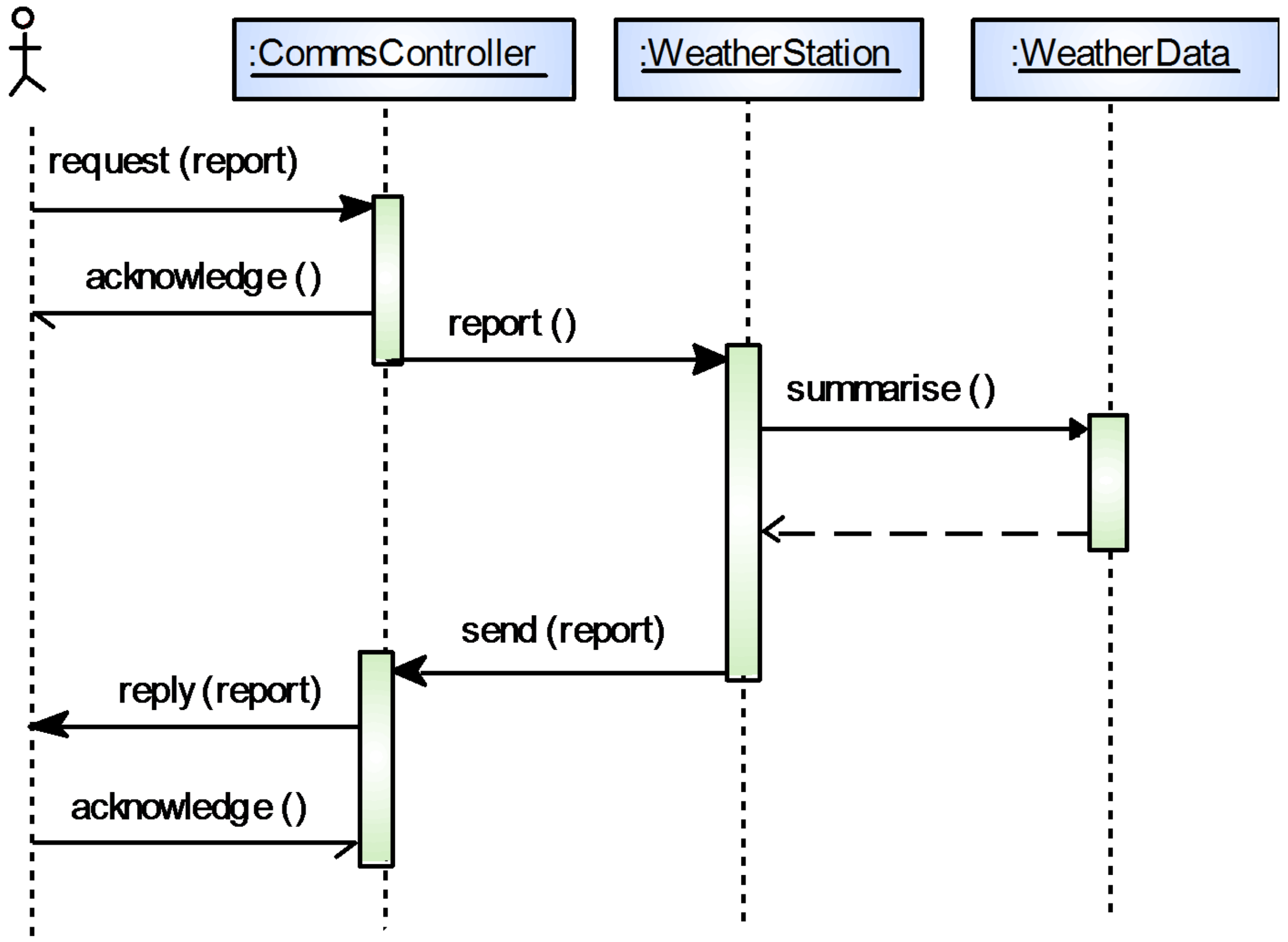
Approaches to Cluster Testing

- 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