COMP201 – Software Engineering I
Lecture 26 – Software Testing

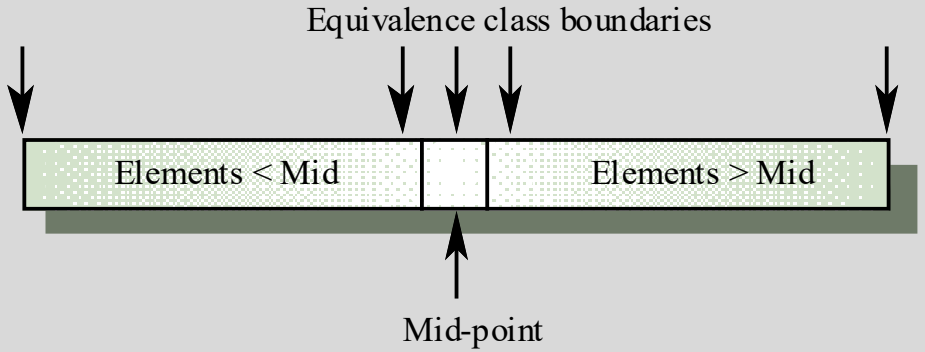
Lecturer: Dr. T. Carroll

Email: Thomas.Carroll2@Liverpool.ac.uk
Office: G.14
See Vital for All Notes

Recap

Lecture 25 Recap

- 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
- White box testing (glass-box testing) is based on the system implementation/structure

Today

Coming Up…

- Path testing
- Cyclomatic Complexity
- Integration Testing
- Interface Testing
- OO Testing

Path Testing

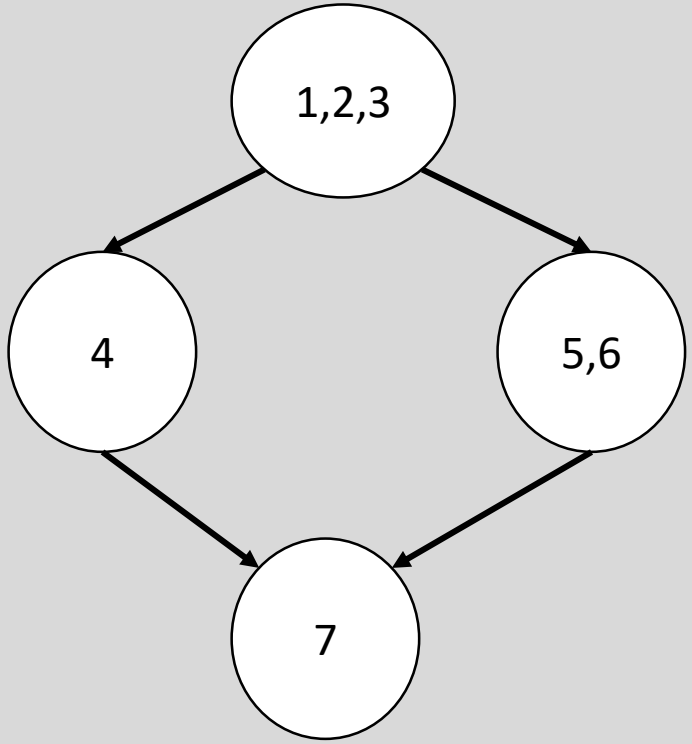
Path Testing… *Do all roads really leady to Rome?*

- The objective of path testing is to ensure each path through the program is executed at least once
- We must ensure that the set of test cases will test each path at least once.
- 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
- Loops are shown by arrows looping back to the loop condition node

```
1 read x;
2 read y;
3 if(x > y)
4       print “X is greater!”
5 else
6        print “Y is greater!”
7 print “We are done, for now…”
```



Cyclomatic Complexity

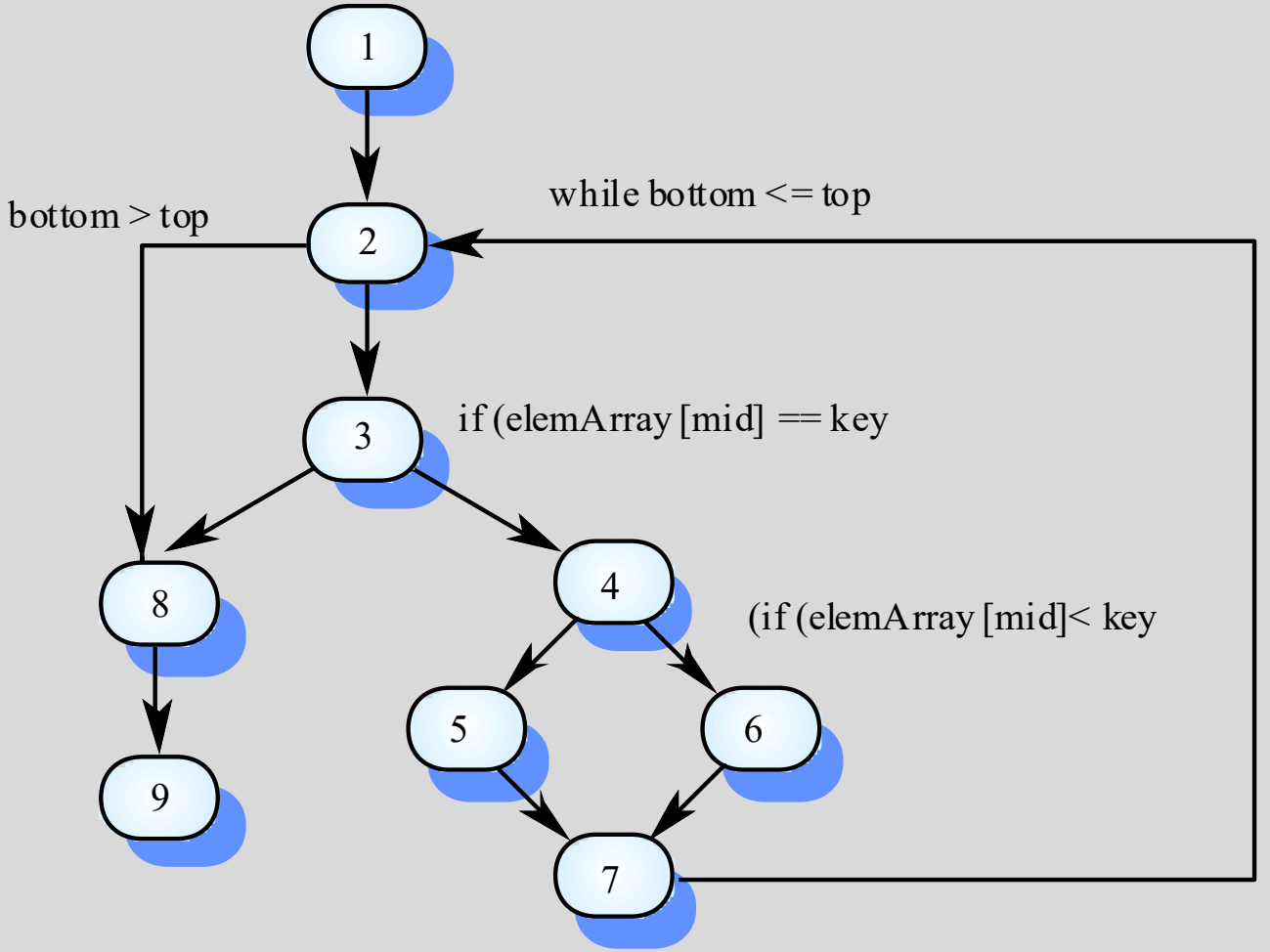
- The number of tests to test all control statements equals the **cyclomatic complexity**
- **Two ways to calculate:**

Cyclomatic complexity = |E| - |V| + 2	**Cyclomatic complexity** = #conditions + 1

- **Conditions** are any type of branching operation (if, for, while, etc…)
- Useful if used with care. Does not imply adequacy of testing.
- Executes all paths, but not all combinations of paths

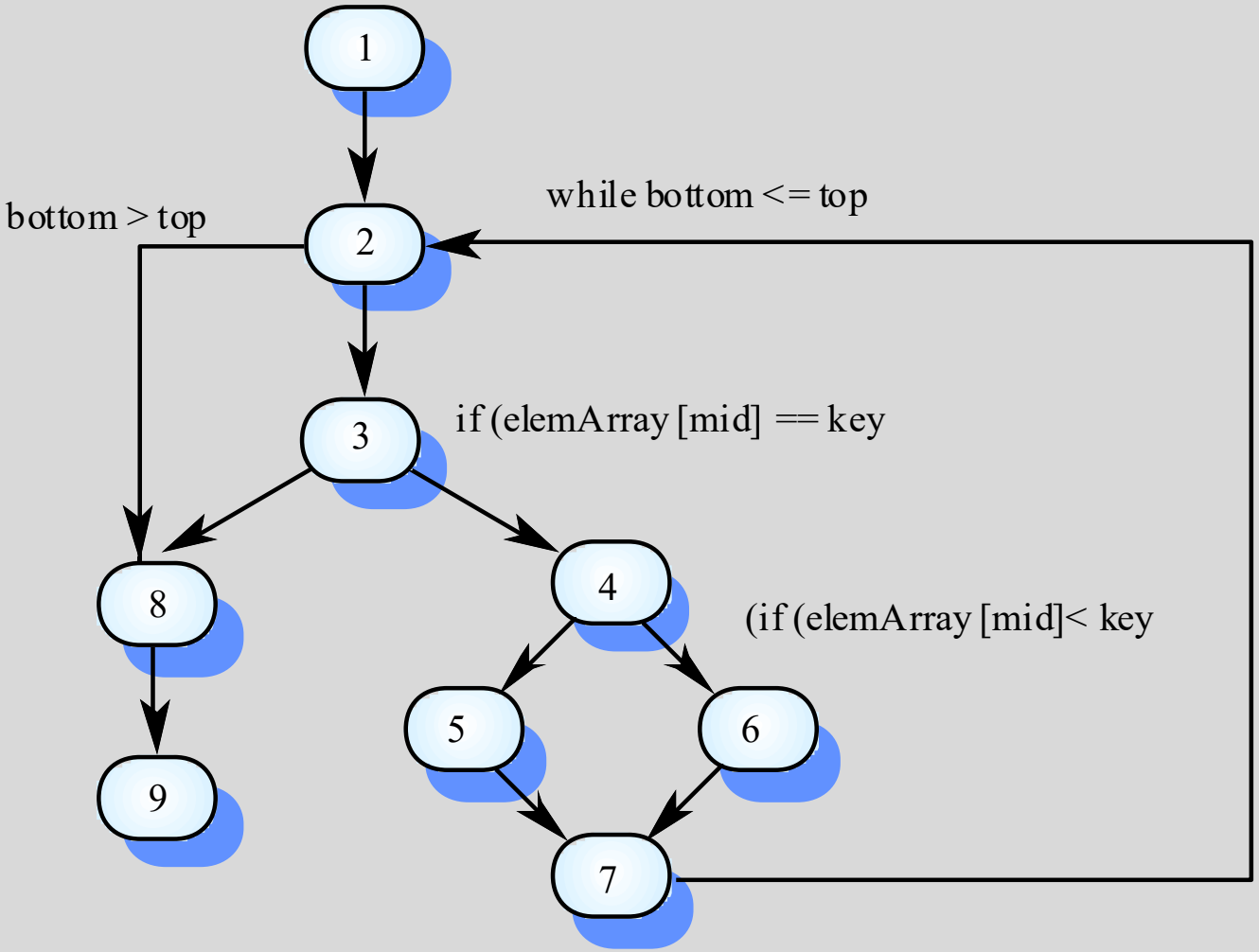
Binary search Example(Java)

```
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
```



Cyclomatic Complexity

Question: What is the Cyclomatic Complexity for this program?



Cyclomatic complexity =
|E| - |V| + 2

Cyclomatic complexity =
#conditions + 1

Independent Paths

Independent paths introduce new nodes into test…

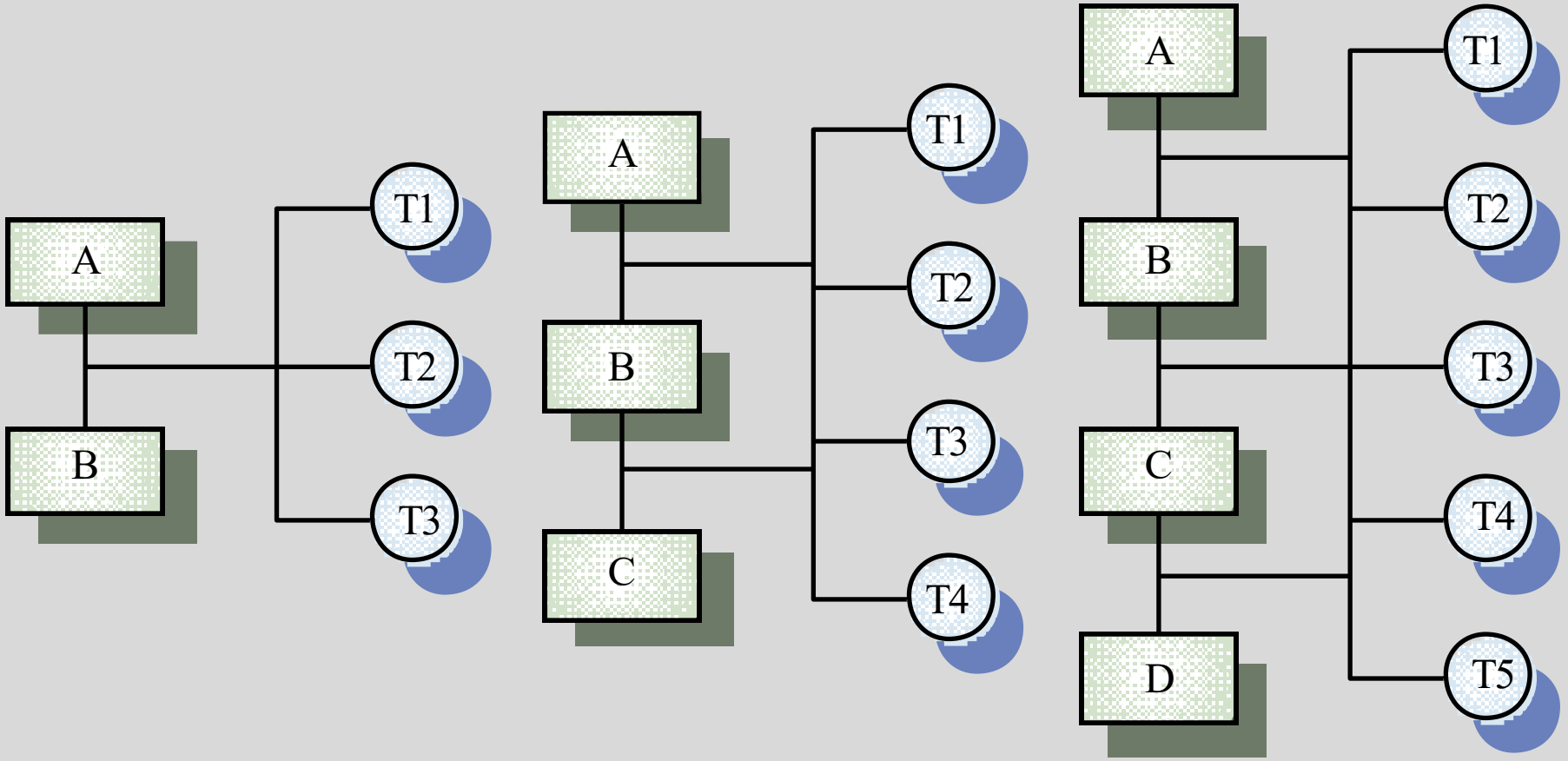
- 1,2,8,9 // new nodes 1,2,8,9
- 1, 2, 3, 8, 9 // new nodes 3
- 1, 2, 3, 4, 5, 7, 2,8,9 // new nodes 4,5,7
- 1, 2, 3, 4, 6, 7, 2,8,9 // new nodes 6
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Integration Testing

Integration Testing

- 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



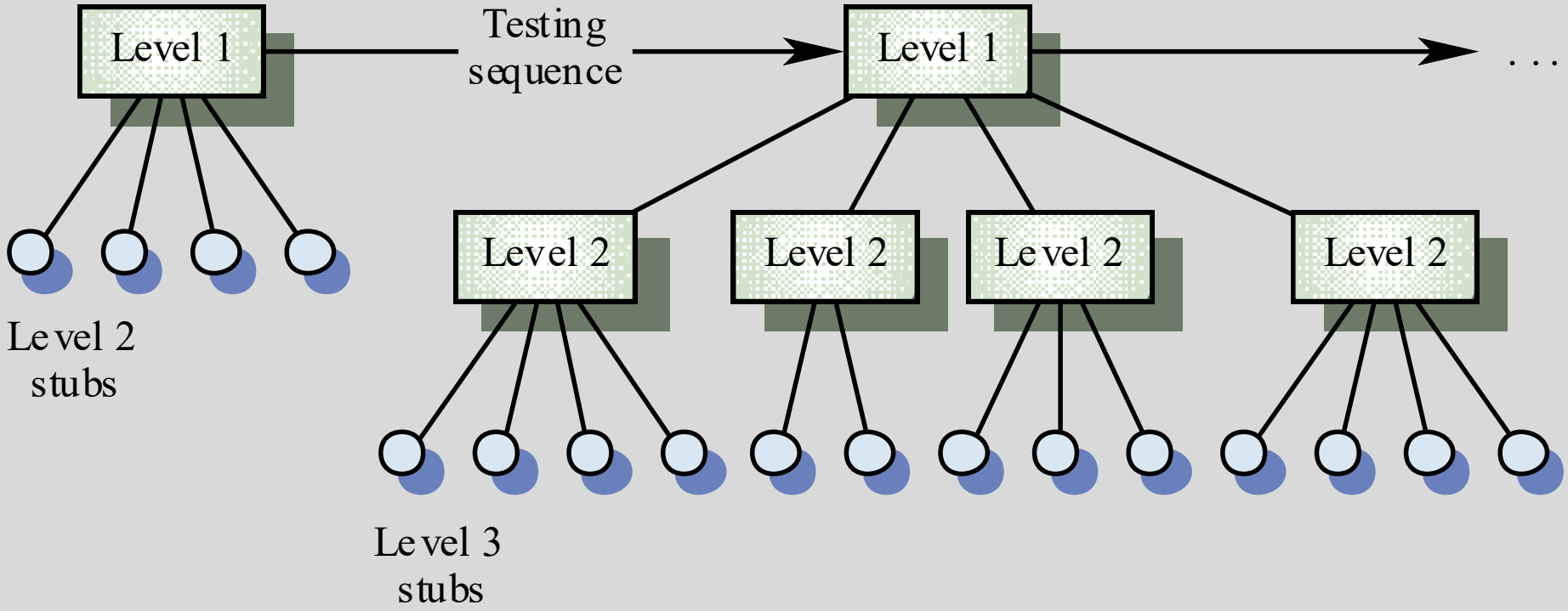
Incremental Integration Testing

- Note that incremental integration as on the previous slide uses the idea of regression testing
  - i.e. future tests also test previous test cases again.
- As a new module is added, we not only run a new test, we also make sure the addition of the new module does not “break” the previous test cases.
- This can sometimes by done automatically by using a test harness
  - (a program written to automatically generate test data and record their results)

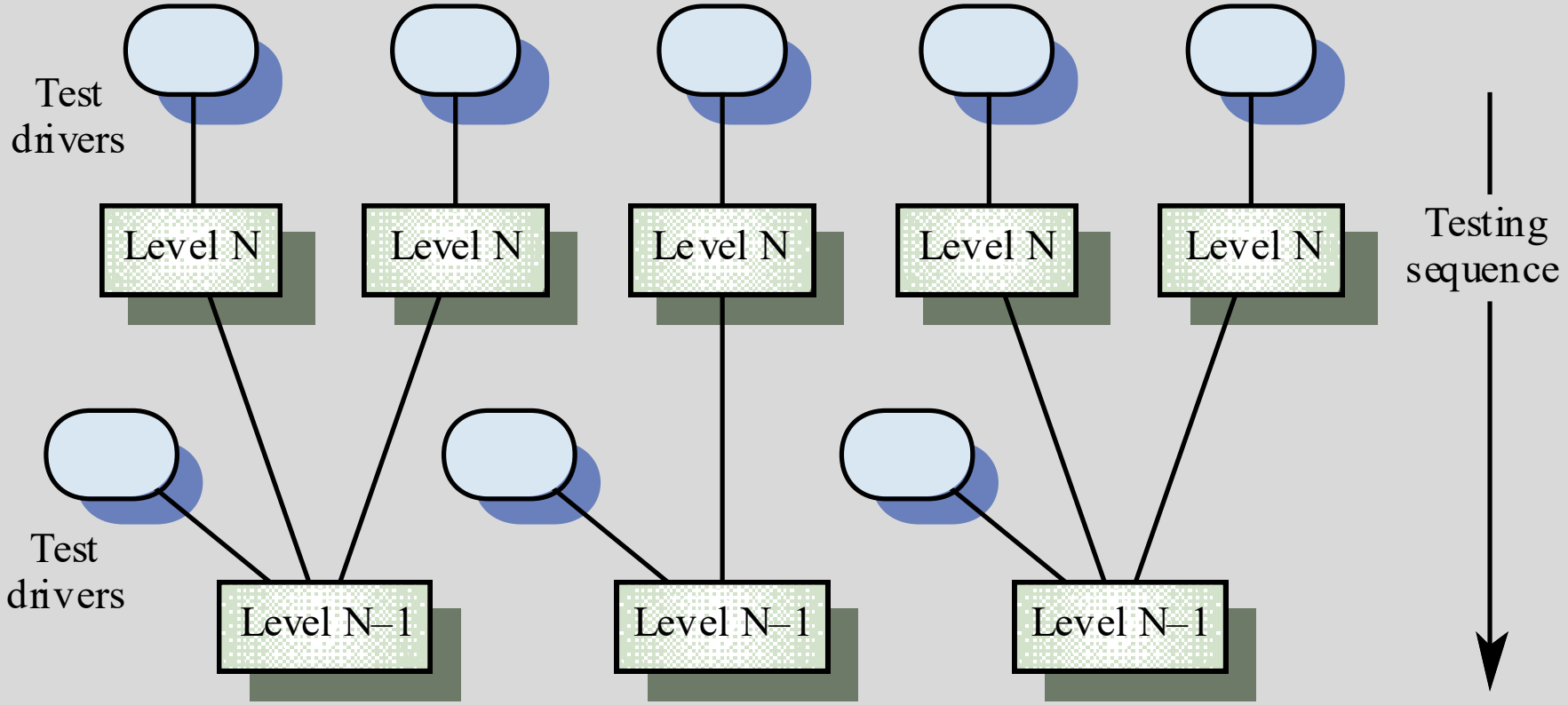
Approaches to Integration Testing

- 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



When to use Bottom-Up testing?

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

Testing Approaches

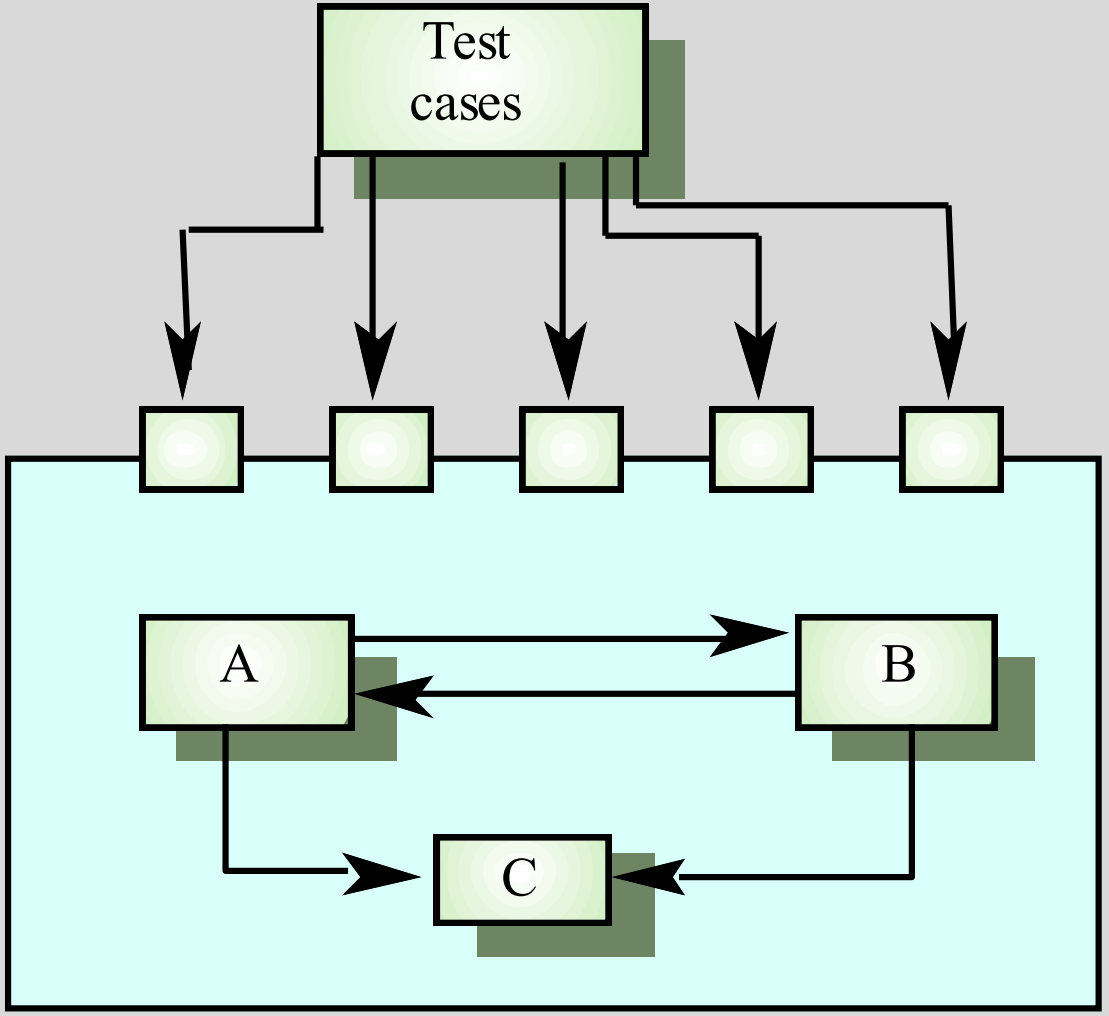
- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration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

Interface Testing

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 OO development: **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

OO Testing

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

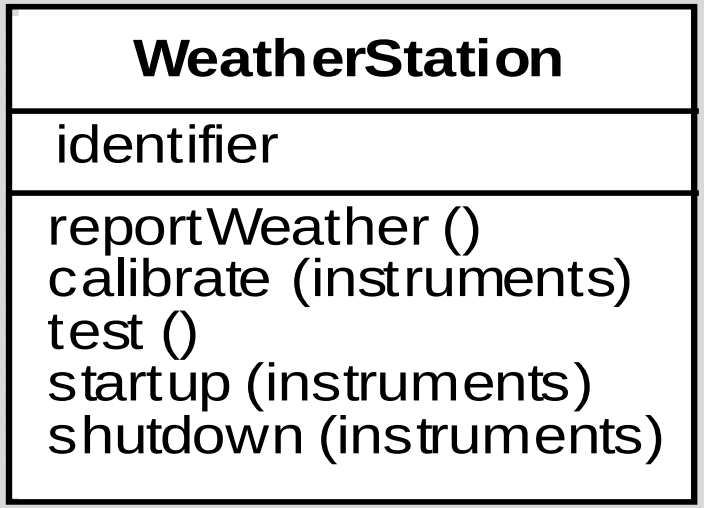
- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

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

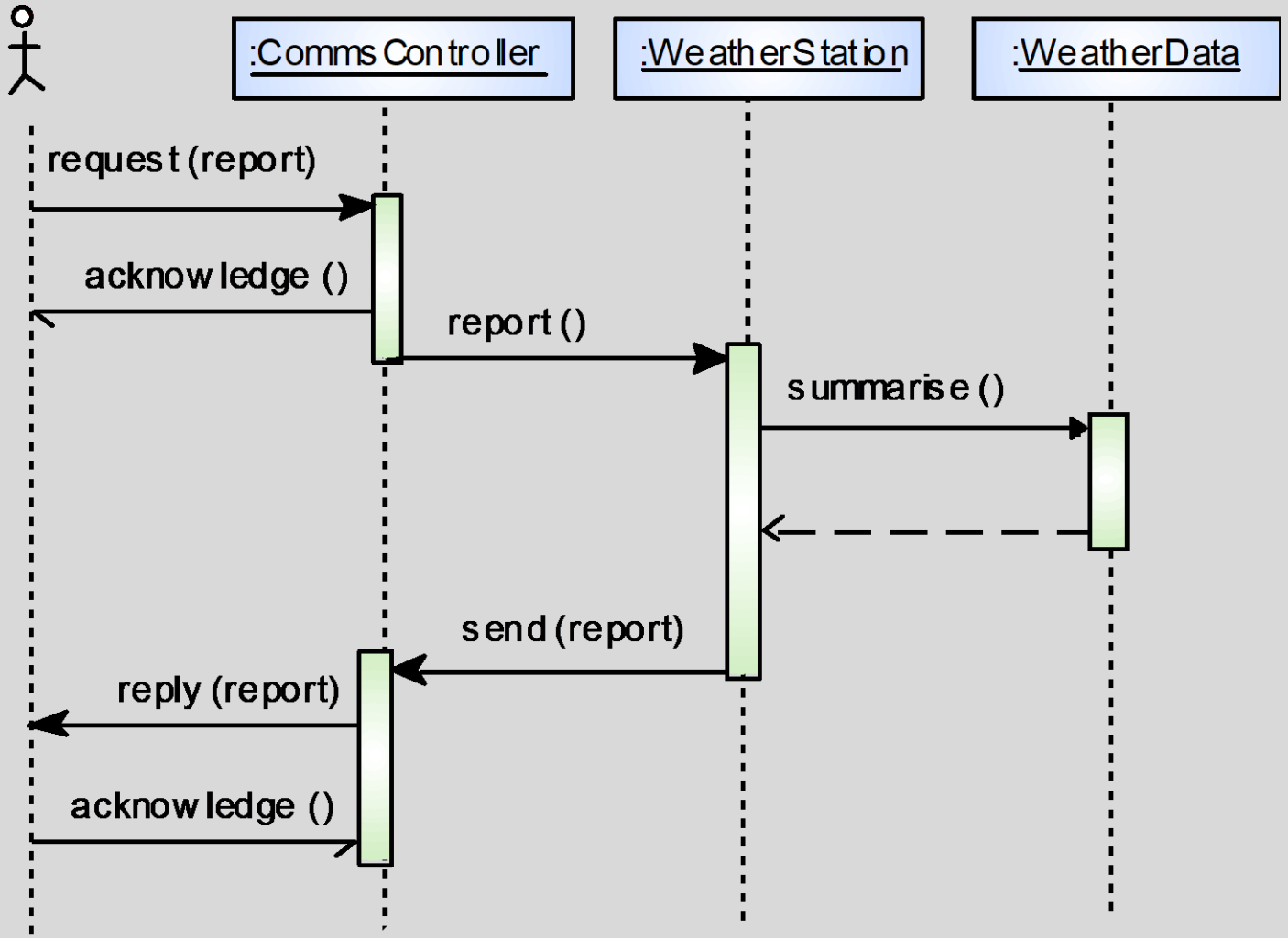
Approaches to Cluster Testing

- 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

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

Lecture Key Points

- Test coverage measures ensure that all statements have been executed at least once – we can use program flow graphs and cyclomatic complexity.
- 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