Integration Testing

Chapter 13

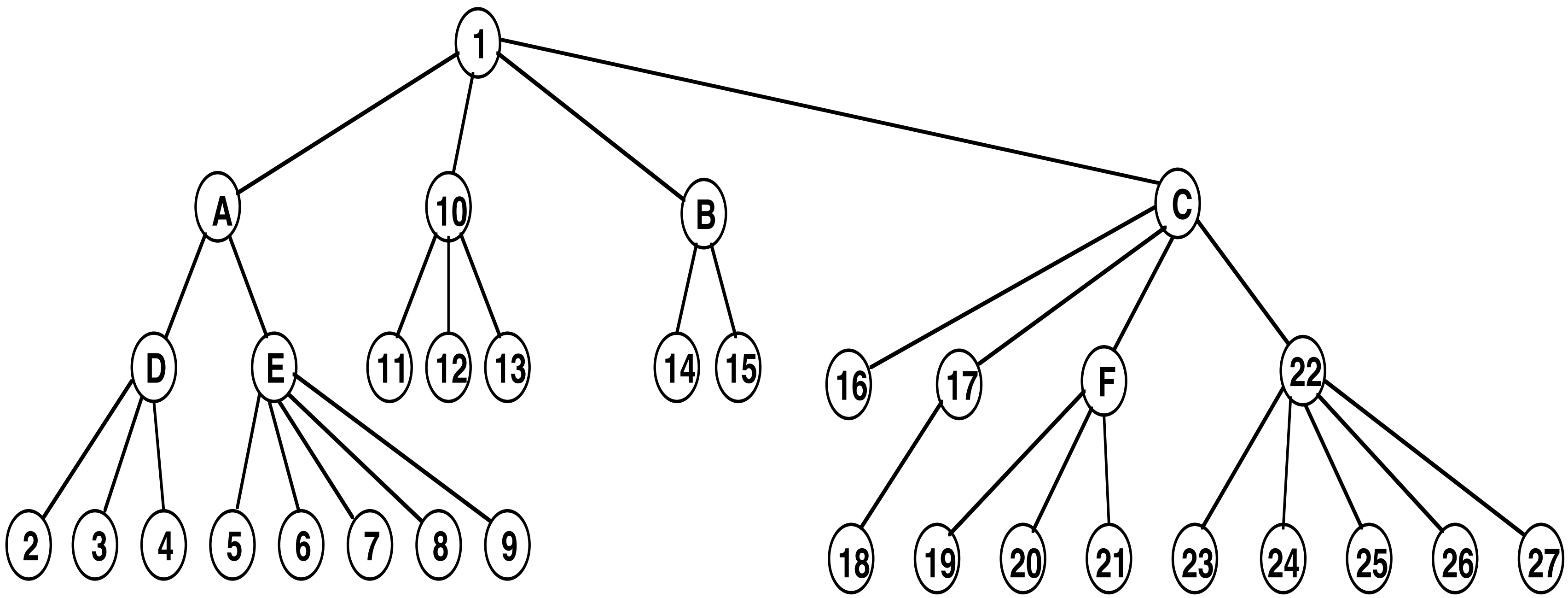
Integration Testing

- Test the interfaces and interactions among separately tested units
- Three different approaches
  - **Based on functional decomposition**
  - **Based on call graphs**
  - **Based on paths**

Functional Decomposition

- Functional Decomposition
  - **Create a functional hierarchy for the software**
  - **Problem is broken up into independent task units, or functions**
  - **Units can be run either**
    - **Sequentially and in a synchronous call-reply manner**
    - **Or simultaneously on different processors**
- Used during planning, analysis and design

Example functional decomposition



Decomposition-based integration

- Four strategies
  - **Top-down**
  - **Bottom-up**
  - **Sandwich**
  - **Big bang**

Top-Down Integration

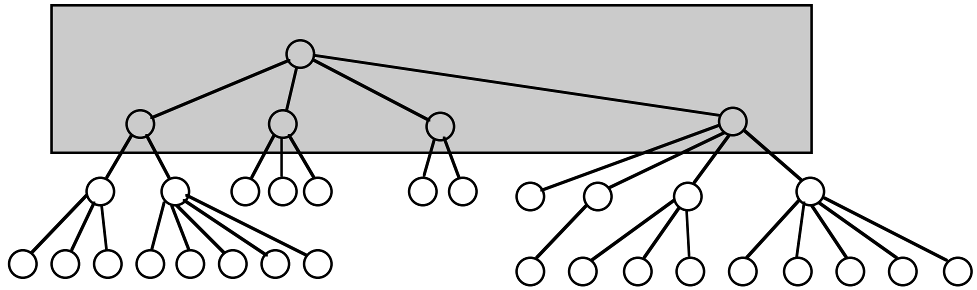
- Top-down integration strategy
  - **Focuses on testing the top layer or the controlling subsystem first (i.e. the main, or the root of the call tree)**
- The general process in top-down integration strategy is
  - **To gradually add more subsystems that are referenced/required by the already tested subsystems when testing the application**
  - **Do this until all subsystems are incorporated into the test**

Top-Down Integration

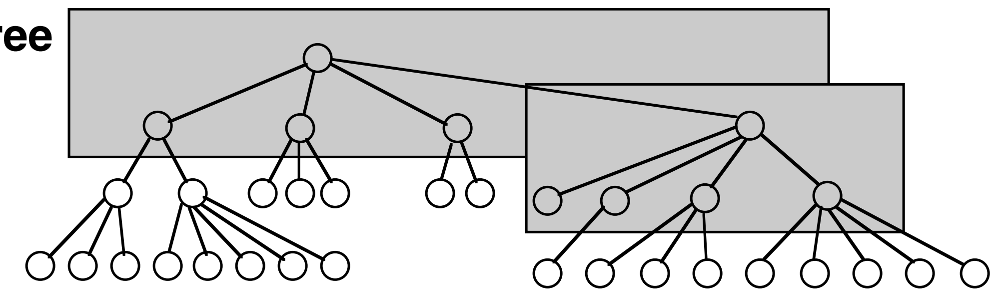
- Special code is needed to do the testing
- Test **stub**
  - **A program or a method that simulates the input-output functionality of a missing subsystem by answering to the decomposition sequence of the calling subsystem and returning back simulated data**

Top-Down integration example

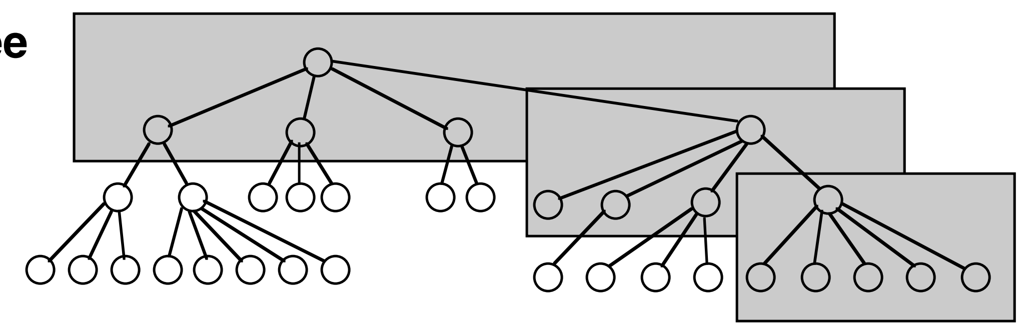
Top Subtree
(Sessions 1-4)



Second Level Subtree
(Sessions 12-15)



Botom Level Subtree
(Sessions 38-42)



Top-Down integration issues

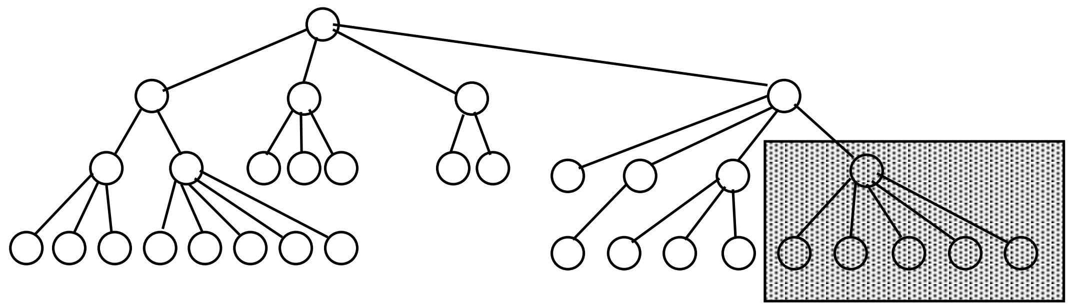
- Writing stubs can be difficult
  - **Especially when parameter passing is complex.**
  - **Stubs must allow all possible conditions to be tested**
- Possibly a very large number of stubs may be required
  - **Especially if the lowest level of the system contains many functional units**
- One solution to avoid too many stubs
  - **Modified top-down testing strategy**
  - **Test each layer of the system decomposition individually before merging the layers**
  - **Disadvantage of modified top-down testing**
    - **Both, stubs and drivers are needed**

Bottom-Up integration

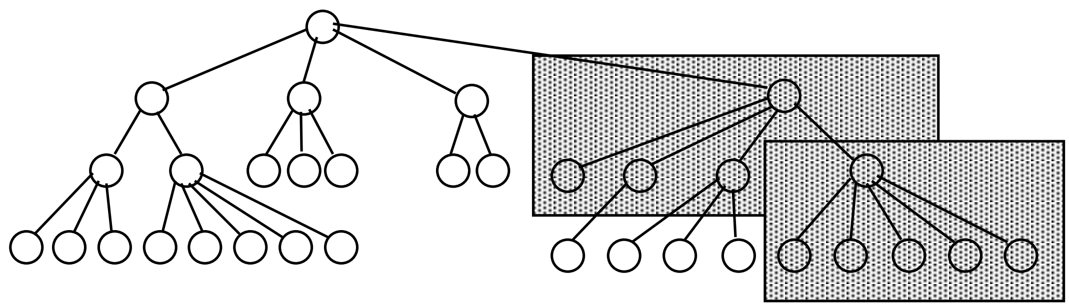
- Bottom-Up integration strategy
  - **Focuses on testing the units at the lowest levels first**
  - **Gradually includes the subsystems that reference/require the previously tested subsystems**
  - **Do until all subsystems are included in the testing**
- Special **driver** code is needed to do the testing
  - **The driver is a specialized routine that passes test cases to a subsystem**
    - **Subsystem is not everything below current root module, but a sub-tree down to the leaf level**

Bottom-up integration example

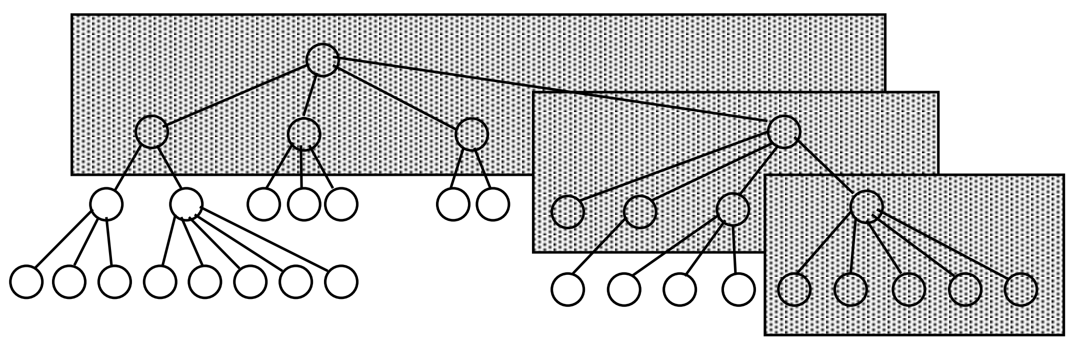
Bottom Level Subtree
(Sessions 13-17)



Second Level Subtree
(Sessions 25-28)



Top Subtree
(Sessions 29-32)



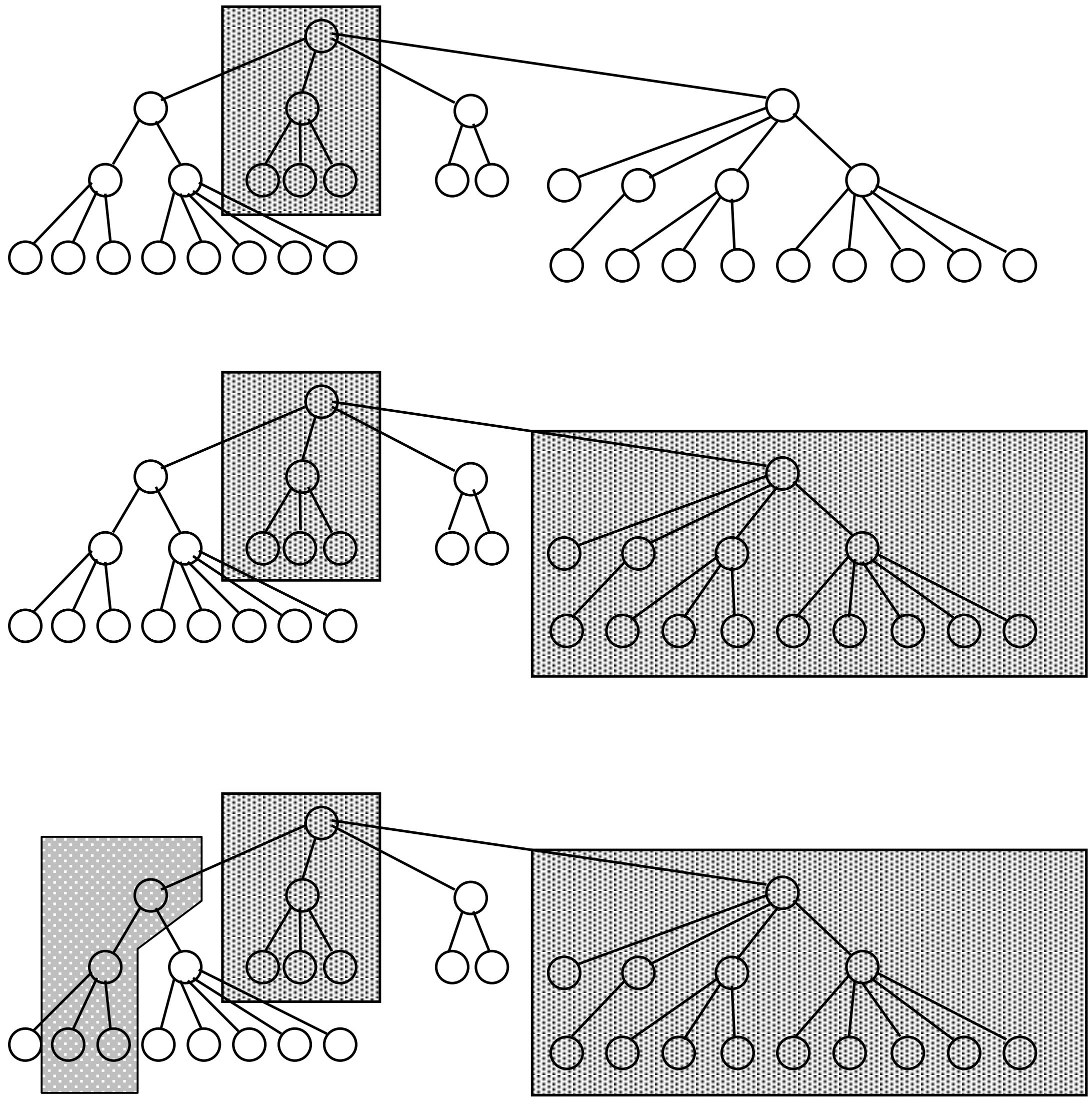
Bottom-Up Integration Issues

- Not an optimal strategy for functionally decomposed systems
  - **Tests the most important subsystem (user interface) last**
- More useful for integrating object-oriented systems
- Drivers may be more complicated than stubs
- Less drivers than stubs are typically required

Sandwich Integration

- Combines top-down strategy with bottom-up strategy
- Less stub and driver development effort
- Added difficulty in fault isolation
- Doing big-bang testing on sub-trees

Sandwich integration example



Integration test metrics

- The number of integration tests for a decomposition tree is the following

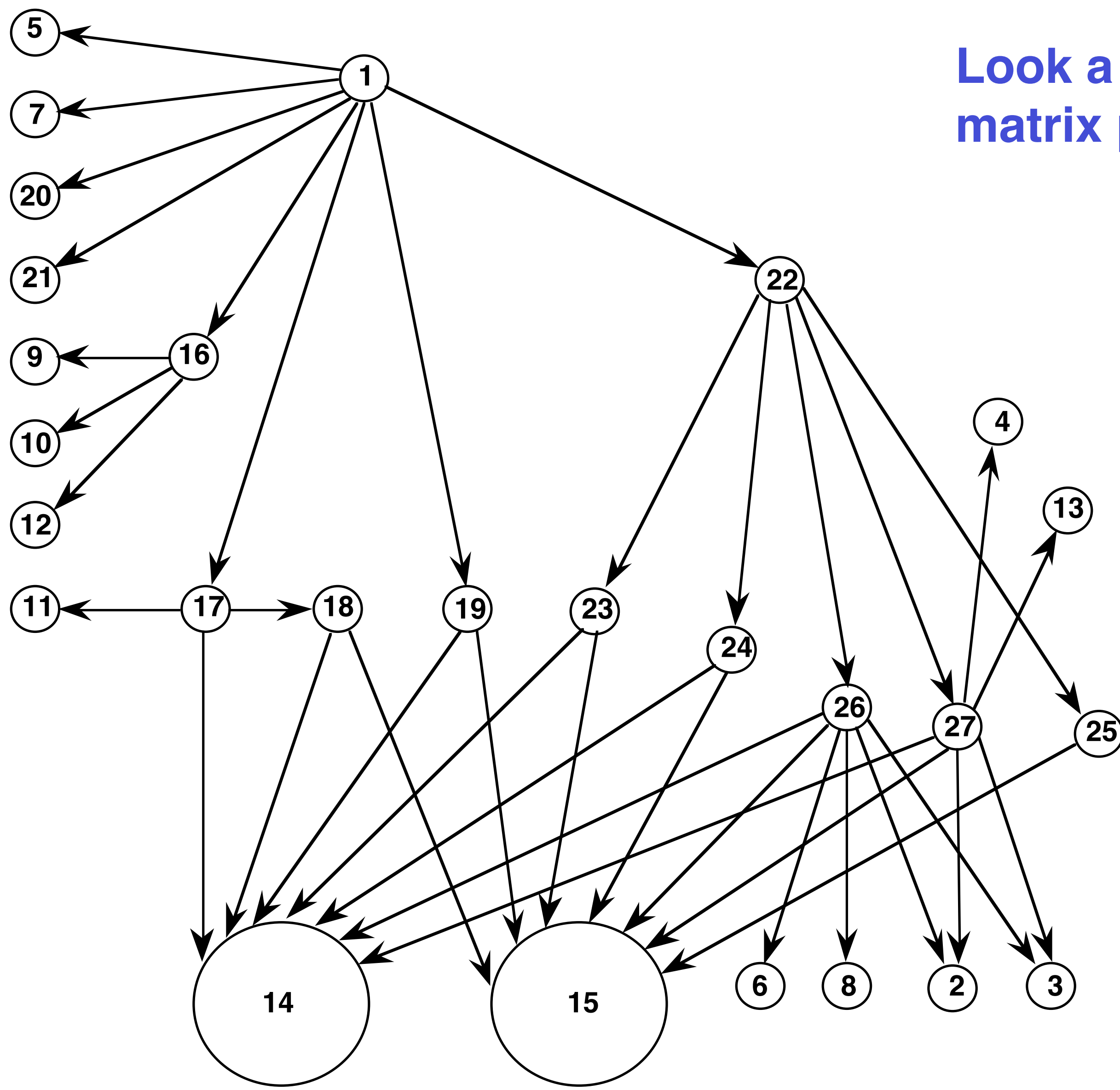
Sessions = nodes – leaves + edges

  - **For SATM have 42 integration test sessions, which correspond to 42 separate sets of test cases**
  - **For top-down integration nodes – 1 stubs are needed**
  - **For bottom-up integration nodes – leaves drivers are needed**
  - **For SATM need 32 stubs and 10 drivers**

Call Graph-Based Integration

- The basic idea is to use the call graph instead of the decomposition tree
- The call graph is a directed, labeled graph
  - **Vertices are program units; e.g. methods**
  - **A directed edge joins calling vertex to the called vertex**
  - **Adjacency matrix is also used**
  - **Do not scale well, although some insights are useful**
    - **Nodes of high degree are critical**

SATM call graph example



Look a adjacency matrix p204

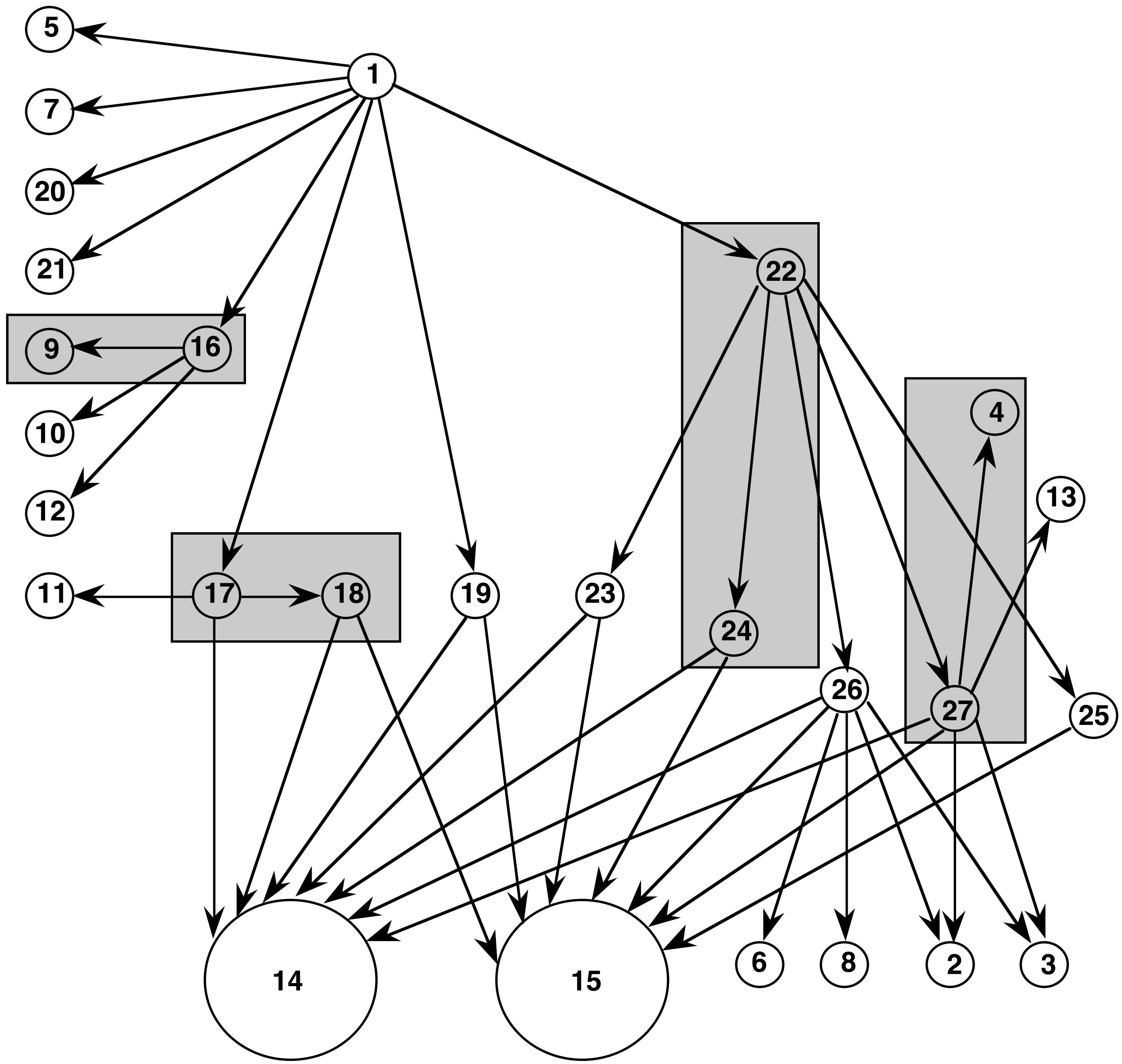
Call graph integration strategies

- Two types of call graph based integration testing
  - **Pair-wise Integration Testing**
  - **Neighborhood Integration Testing**

Pair-Wise Integration

- The idea behind Pair-Wise integration testing
  - **Eliminate need for developing stubs / drivers**
  - **Use actual code instead of stubs/drivers**
- In order not to deteriorate the process to a big-bang strategy
  - **Restrict a testing session to just a pair of units in the call graph**
  - **Results in one integration test session for each edge in the call graph**

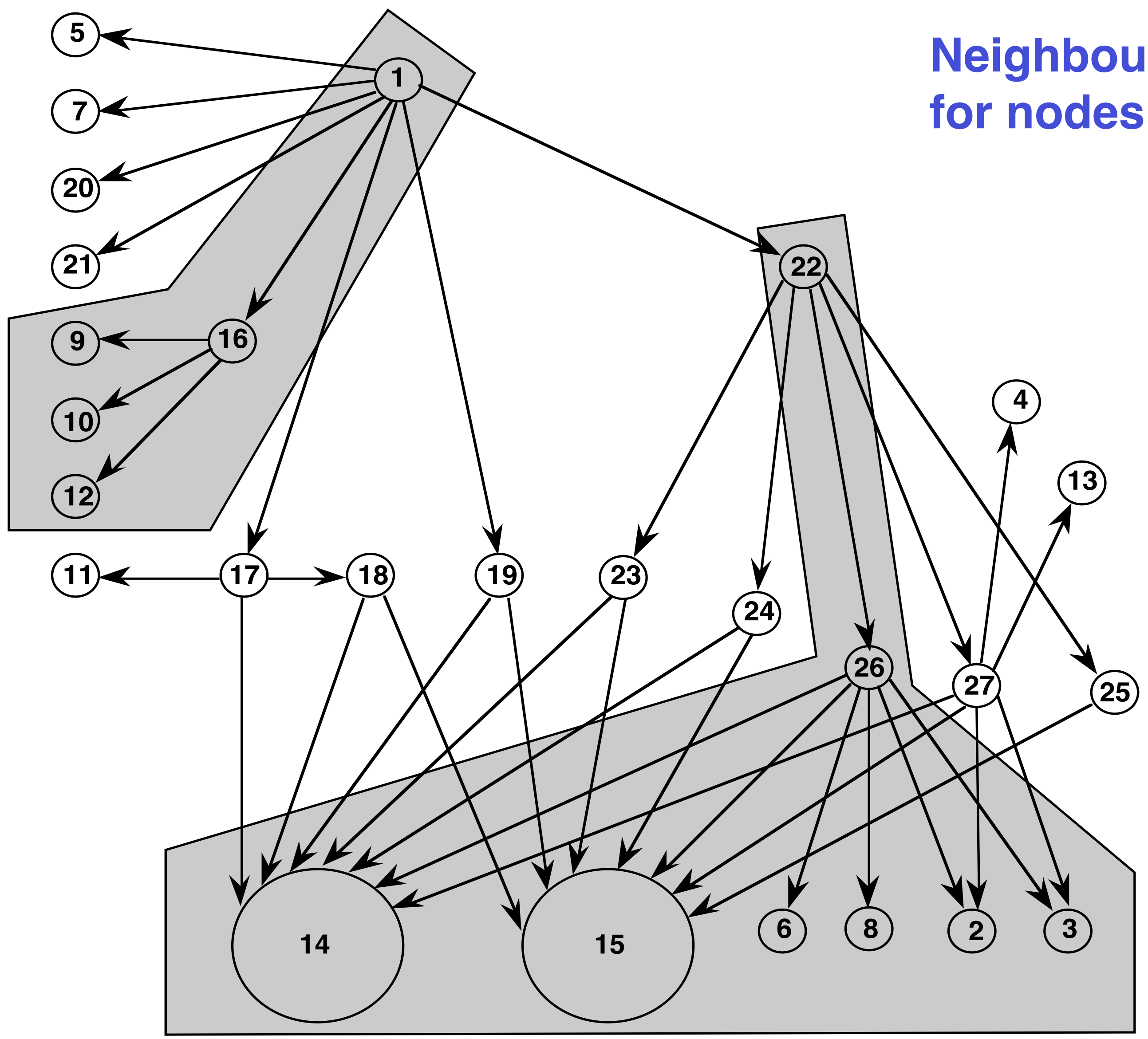
Pair-wise integration session example



Neighbourhood integration

- The neighbourhood of a node in a graph
  - **The set of nodes that are one edge away from the given node**
- In a directed graph
  - **All the immediate predecessor nodes and all the immediate successor nodes of a given node**
- Neighborhood Integration Testing
  - **Reduces the number of test sessions**
  - **Fault isolation is more difficult**

Neighbourhood integration example



Neighbourhoods for nodes 16 & 26

Pros and Cons of Call-Graph Integration

- Aim to eliminate / reduce the need for drivers / stubs
  - **Development effort is a drawback**
- Closer to a build sequence
- Neighborhoods can be combined to create “villages”
- Suffer from fault isolation problems
  - **Specially for large neighborhoods**

Pros and Cons of Call-Graph Integration – 2

- Redundancy
  - **Nodes can appear in several neighborhoods**
- Assumes that correct behaviour follows from correct units and correct interfaces
  - **Not always the case**
- Call-graph integration is well suited to devising a sequence of builds with which to implement a system

Path-Based Integration

- Motivation
  - **Combine structural and behavioral type of testing for integration testing as we did for unit testing**
- Basic idea
  - **Focus on interactions among system units**
  - **Rather than merely to test interfaces among separately developed and tested units**
- Interface-based testing is structural while interaction-based is behavioral

Extended Concepts – 1

- Source node
  - **A program statement fragment at which program execution begins or resumes.**
    - **For example the first "begin" statement in a program.**
    - **Also, immediately after nodes that transfer control to other units.**
- Sink node
  - **A statement fragment at which program execution terminates.**
    - **The final "end" in a program as well as statements that transfer control to other units.**

Extended Concepts – 2

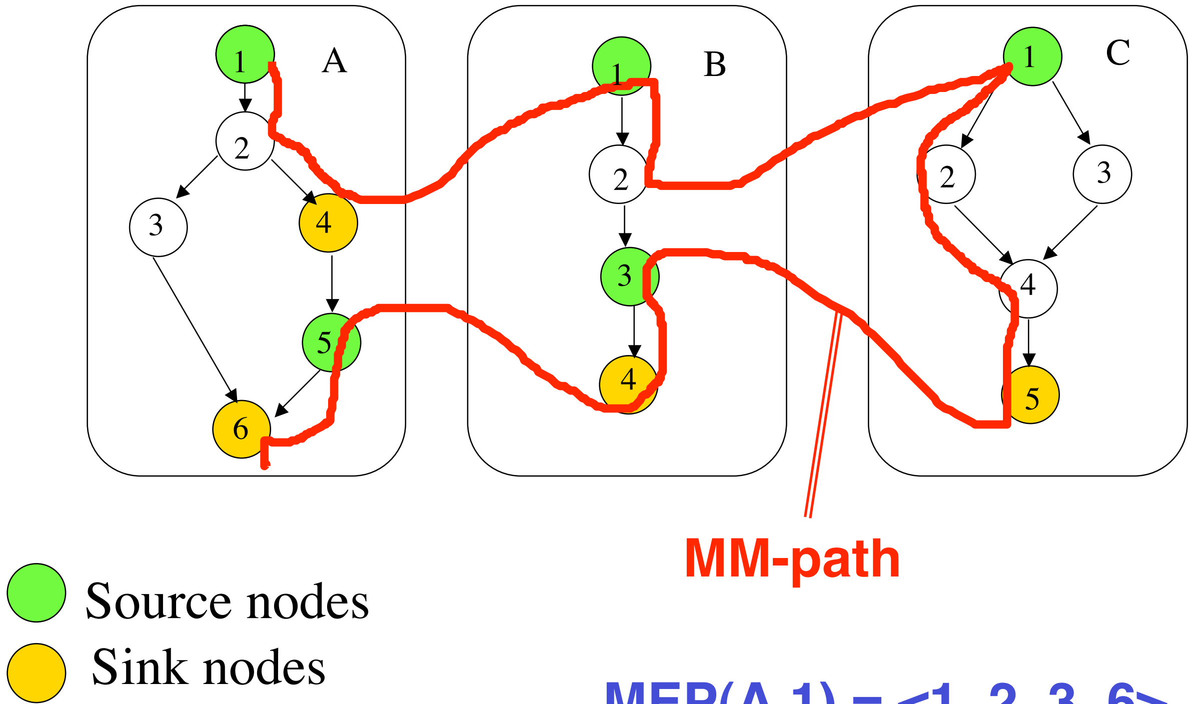
- Module execution path
  - **A sequence of statements that begins with a source node and ends with a sink node with no intervening sink nodes.**
- Message
  - **A programming language mechanism by which one unit transfers control to another unit.**
  - **Usually interpreted as subroutine invocations**
  - **The unit which receives the message always returns control to the message source.**

MM-Path

- An interleaved sequence of module execution paths and messages.
- Describes sequences of module execution paths that include transfers of control among separate units.
- MM-paths always represent feasible execution paths, and these paths cross unit boundaries.

- There is no correspondence between MM-paths and DD-paths
- The intersection of a module execution path with a unit is the analog of a slice with respect to the MM-path function

MM-Path Example



Module Execution Paths

MEP(A,1) = <1, 2, 3, 6>
MEP(A,2) = <1, 2, 4>
MEP(A,3) = <5, 6>

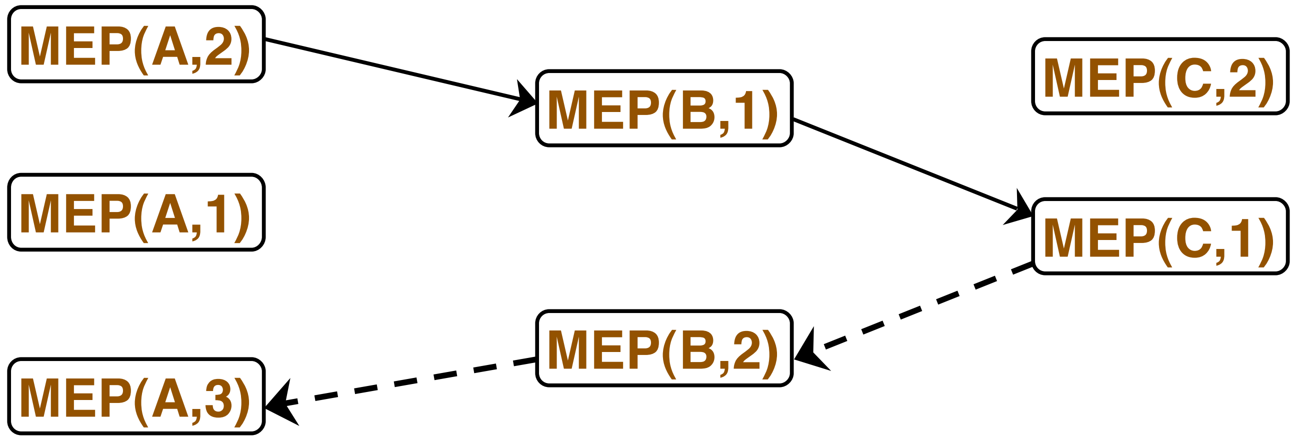
MEP(B,2) = <3, 4>
MEP(B,1) = <1, 2>

MEP(C,2) = <1, 3, 4, 5>
MEP(C,2) = <1, 3, 4, 5>

MM-path Graph

- Given a set of units their **MM-path graph** is the directed graph in which
  - **Nodes are module execution paths**
  - **Edges correspond to messages and returns from one unit to another**
- The definition is with respect to a set of units
  - **It directly supports composition of units and composition-based integration testing**

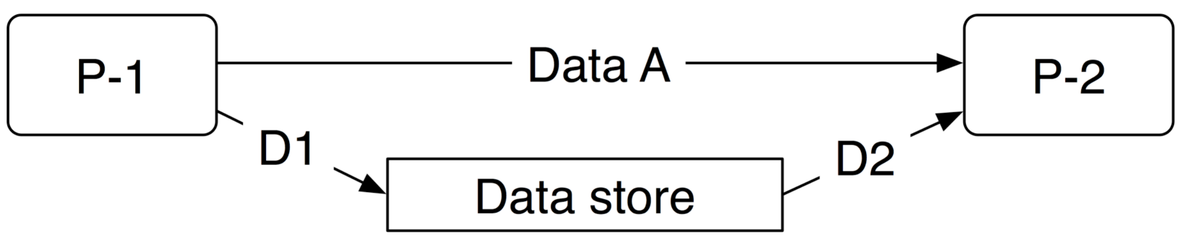
MM-path graph example



Solid lines indicate messages (calls)
Dashed lines indicate returns from calls

MM-path guidelines

- How long, or deep, is an MM-path? What determines the end points?
  - **Message quiescence**
    - **Occurs when a unit that sends no messages is reached**
      - Module C in the example
  - **Data quiescence**
    - **Occurs when a sequence of processing ends in the creation of stored data that is not immediately used (path D1 and D2)**



- Quiescence points are natural endpoints for MM-paths

MM-Path metric

- How many MM-paths are sufficient to test a system
  - **Should cover all source-to-sink paths in the set of units**
- What about loops?
  - **Use condensation graphs to get directed acyclic graphs**
    - **Avoids an excessive number of paths**

Pros and cons of path-based integration

- Hybrid of functional and structural testing
  - **Functional – represent actions with input and output**
  - **Structural – how they are identified**
- Avoids pitfall of structural testing (???)
- Fairly seamless union with system testing
- Path-based integration is closely coupled with actual system behaviour
  - **Works well with OO testing**
- No need for stub and driver development
- There is a significant effort involved in identifying MM-paths

MM-path compared to other methods

Strategy	Ability to test interfaces	Ability to test co-functionality	Fault isolation resolution
Functional decomposition	Acceptable, can be deceptive	Limited to pairs of units	Good to faulty unit
Call-graph	Acceptable	Limited to pairs of units	Good to faulty unit
MM-path	Excellent	Complete	Excellent to unit path level