Chapter 13

Integration Testing

The Mars Climate Orbiter Mission

- mission failed in September 1999
- completed successful flight: 416,000,000 miles (665.600.600 km)
- 41 weeks flight duration
- lost at beginning of Mars orbit
- An integration fault: Lockheed Martin used English units for acceleration calculations (pounds), and Jet Propulsion Laboratory used metric units (newtons).
- NASA announced a US$ 50,000 project to discover how this happened.
- (We know!)

Goals/Purpose of Integration Testing

- Presumes previously tested units
- Not system testing
- Tests functionality "between" unit and system levels
- Basis for test case identification?
- Emphasis shifts from “how to test” to “what to test” (Model-Based Testing)

Testing Level Assumptions and Objectives

- Unit assumptions
  - All other units are correct
  - Compiles correctly
- Integration assumptions
  - Unit testing complete
- System assumptions
  - Integration testing complete
  - Tests occur at port boundary

- Unit goals
  - Correct unit function
  - Coverage metrics satisfied
- Integration goals
  - Interfaces correct
  - Correct function across units
  - Fault isolation support
- System goals
  - Correct system functions
  - Non-functional requirements tested
  - Customer satisfaction.

Approaches to Integration Testing
("source" of test cases)

- Functional Decomposition (most commonly described in the literature)
  - Top-down
  - Bottom-up
  - Sandwich
  - "Big bang"
- Call graph
  - Pairwise integration
  - Neighborhood integration
- Paths
  - MM-Paths
  - Atomic System Functions

Basis of Integration Testing Strategies

- Functional Decomposition

  applies best to procedural code
- Call Graph

  applies to both procedural and object-oriented code
- MM-Paths

  applies to both procedural and object-oriented code

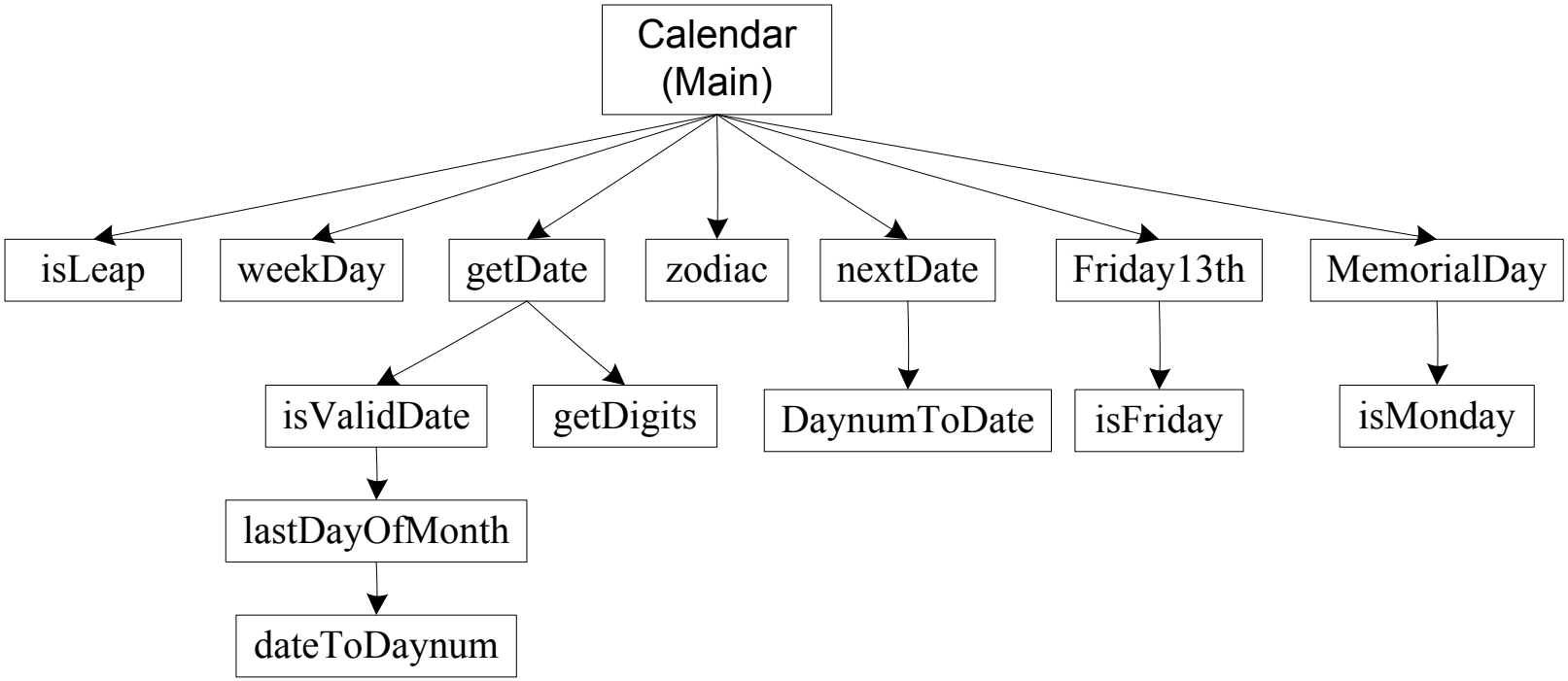
Continuing Example—Calendar Program

- (see text for pseudo-code version)
- Date in the form mm, dd, yyyy
- Calendar functions
  - the date of the next day (our old friend, NextDate)
  - the day of the week corresponding to the date
  - the zodiac sign of the date
  - the most recent year in which Memorial Day was celebrated on May 27
  - the most recent Friday the Thirteenth

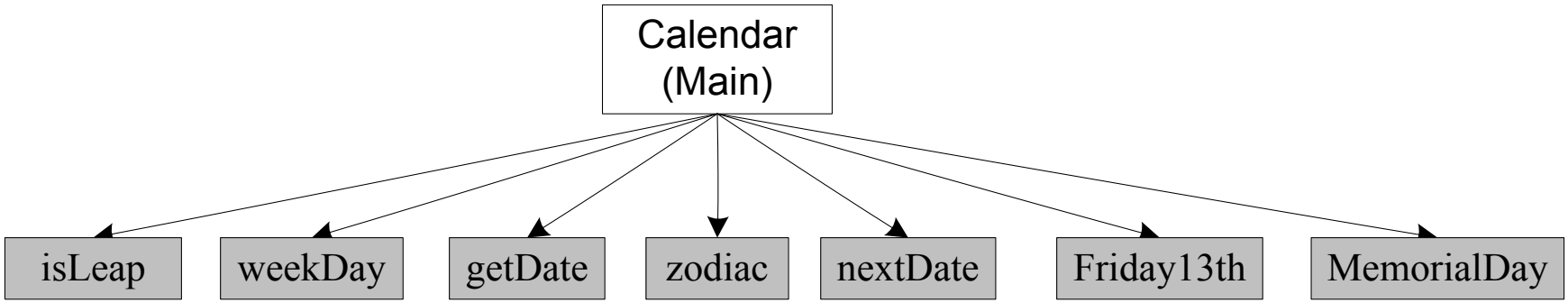
Calendar Program Units

Main Calendar
Function isLeap
Procedure weekDay
Procedure getDate
Function isValidDate
Function lastDayOfMonth
Procedure getDigits
Procedure memorialDay
Function isMonday
Procedure friday13th
Function isFriday
Procedure nextDate
Procedure dayNumToDate
Procedure zodiac

Functional Decomposition of Calendar



First Step in Top-Down Integration



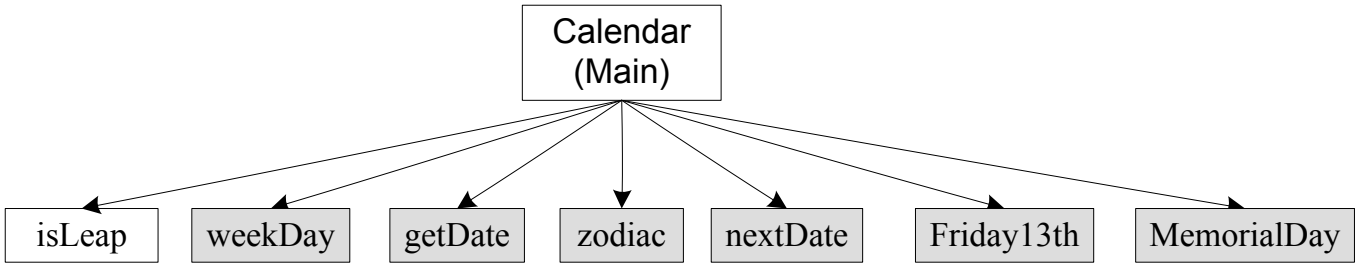
"Grey" units are stubs that return the correct values when referenced. This level checks the main program logic.

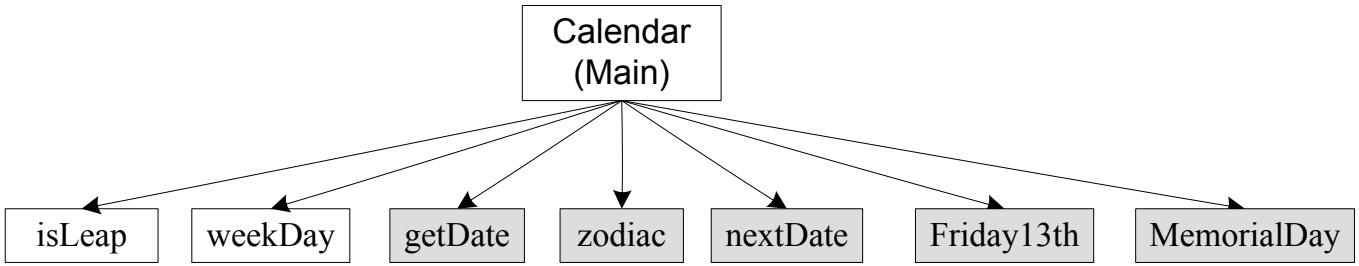
weekDayStub

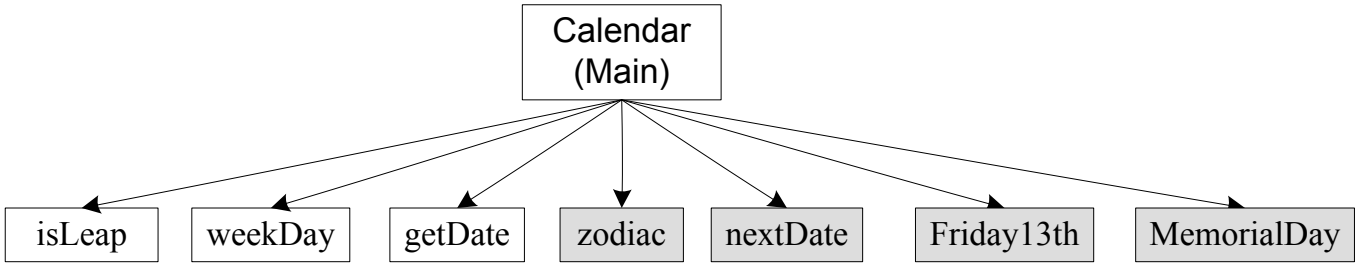
```
Procedure weekDayStub(mm, dd, yyyy, dayName)
If ((mm = 10) AND (dd = 28) AND (yyyy = 2013))
        Then dayName = “Monday”
EndIf
.
.
.
If ((mm = 10) AND (dd = 30) AND (yyyy = 2013))
        Then dayName = “Wednesday”
EndIf
```

Next Three Steps

(replace one stub at a time with the actual code.)







Top-Down Integration Mechanism

- Breadth-first traversal of the functional decomposition tree.
- First step: Check main program logic, with all called units replaced by stubs that always return correct values.
- Move down one level
  - replace one stub at a time with actual code.
  - any fault must be in the newly integrated unit

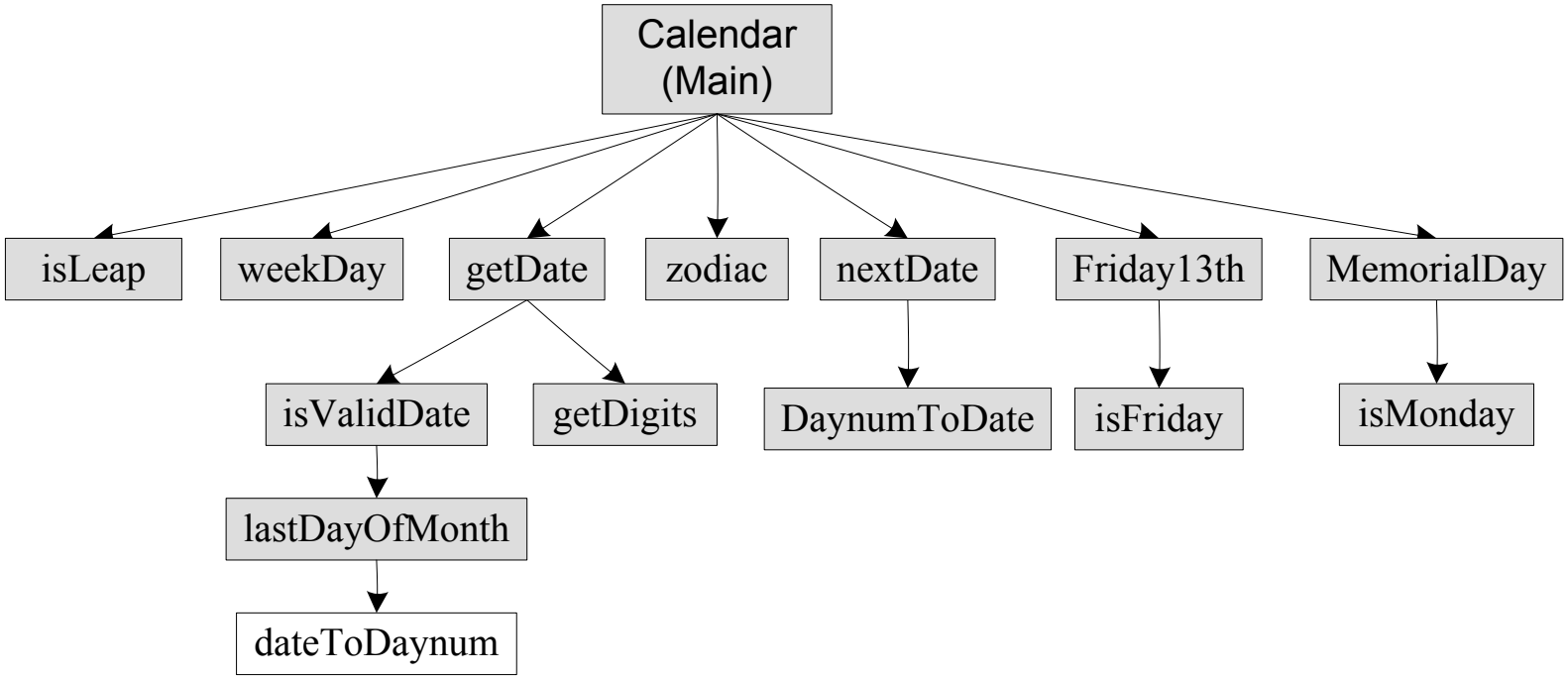
Bottom-Up Integration Mechanism

- Reverse of top-down integration
- Start at leaves of the functional decomposition tree.
- Driver units...
  - call next level unit
  - serve as a small test bed
  - “drive” the unit with inputs
  - drivers know expected outputs
- As with top-down integration, one driver unit at a time is replaced with actual code.
- Any fault is (most likely) in the newly integrated code.

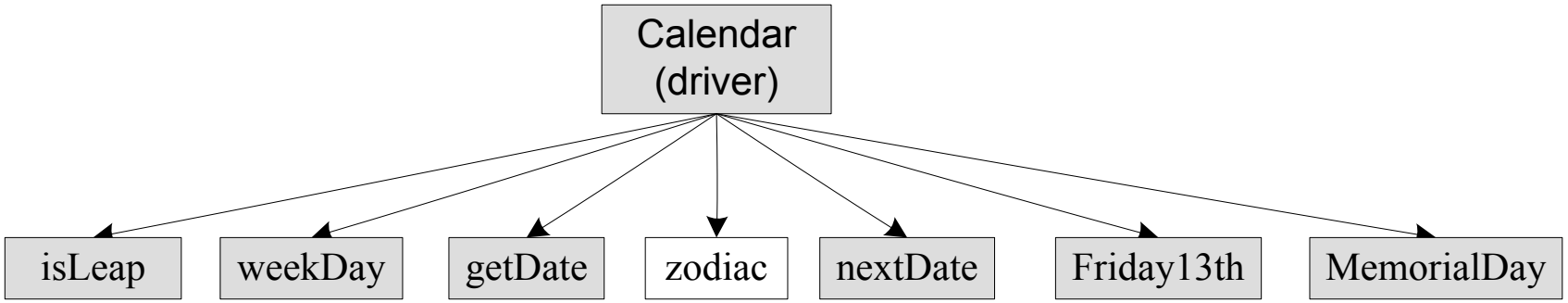
Top-Down and Bottom-Up Integration

- Both depend on throwaway code.
  - drivers are usually more complex than stubs
- Both test just the interface between two units at a time.
- In Bottom-Up integration, a driver might simply reuse unit level tests for the “lower” unit.
- Fan-in and fan-out in the decomposition tree results is some redundancy.

Starting Point of Bottom-Up Integration



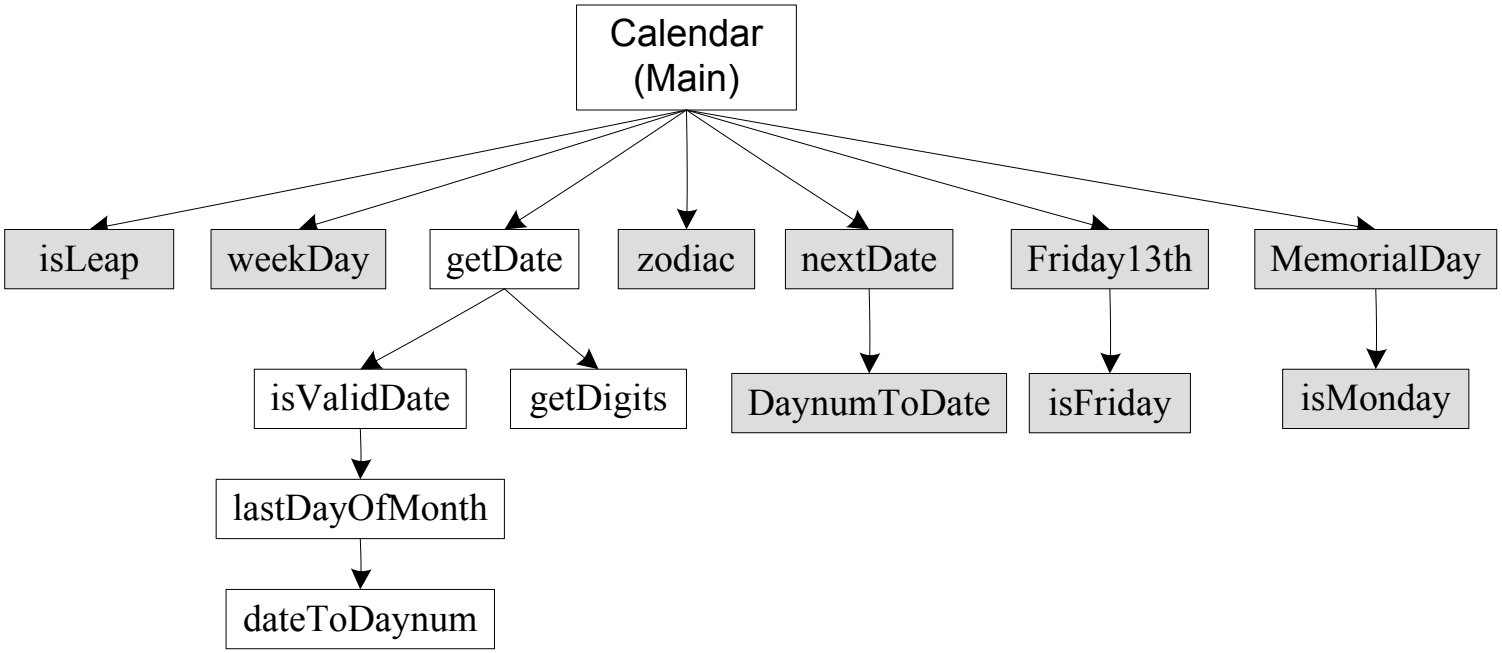
Bottom-Up Integration of Zodiac



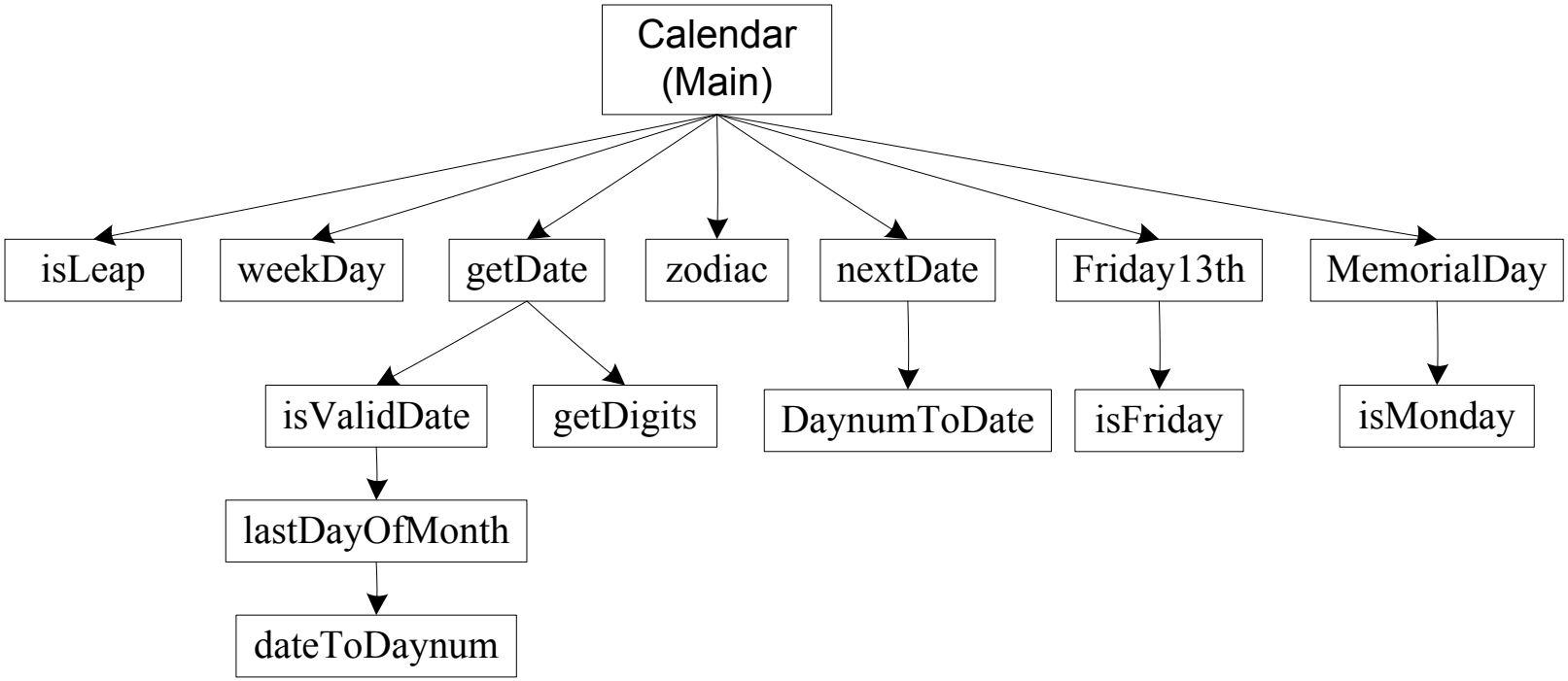
Sandwich Integration

- Avoids some of the repetition on both top-down and bottom-up integration.
- Nicely understood as a depth-first traversal of the functional decomposition tree.
- A “sandwich” is one path from the root to a leaf of the functional decomposition tree.
- Avoids stub and driver development.
- More complex fault isolation.

A Sample Sandwich



“Big Bang” Integration



“Big Bang” Integration

- No...
  - stubs
  - drivers
  - strategy
- And very difficult fault isolation
- (Named after one of the theories of the origin of the Universe)
- This is the practice in an agile environment with a daily run of the project to that point.

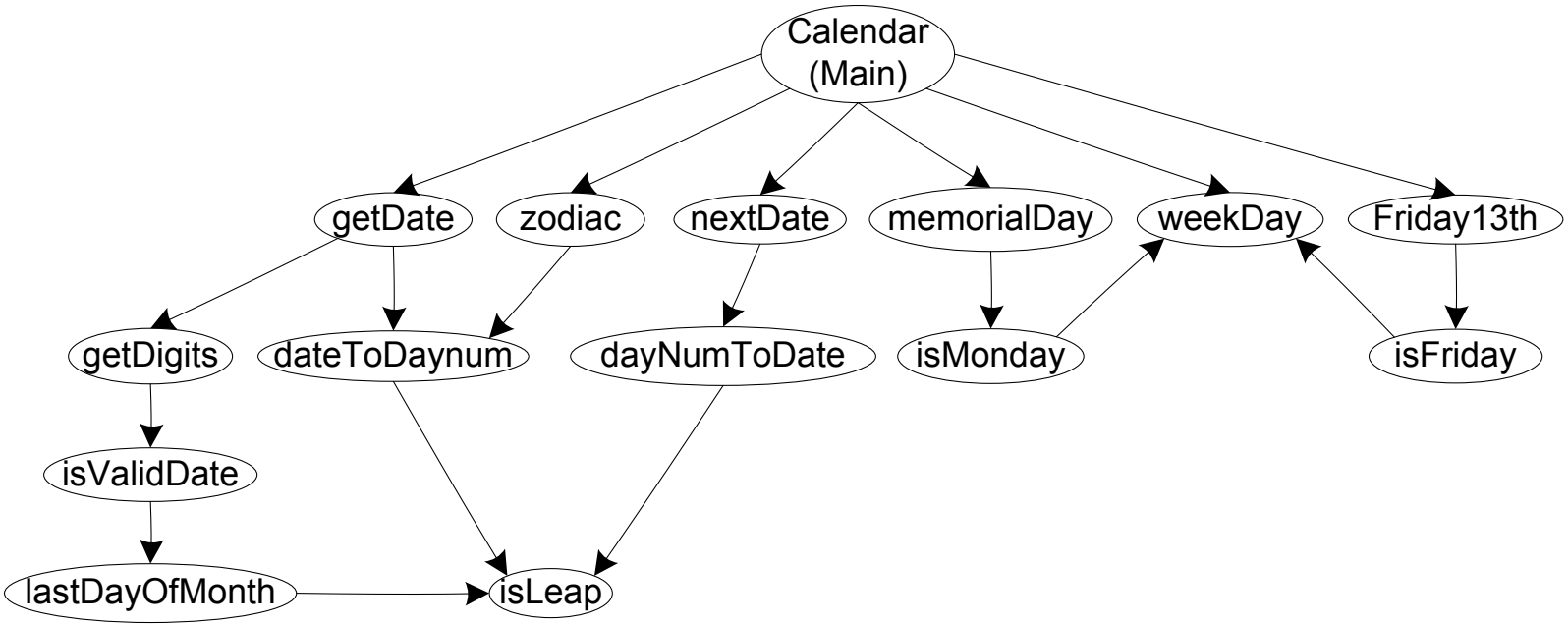
Pros and Cons of Decomposition-Based Integration

- Pros
  - intuitively clear
  - “build” with proven components
  - fault isolation varies with the number of units being integrated
- Cons
  - based on lexicographic inclusion (a purely structural consideration)
  - some branches in a functional decomposition may not correspond with actual interfaces.
  - stub and driver development can be extensive

Call Graph-Based Integration

- Definition: The *Call Graph* of a program is a directed graph in which
  - nodes are unit
  - edges correspond to actual program calls (or messages)
- Call Graph Integration avoids the possibility of impossible edges in decomposition-based integration.
- Can still use the notions of stubs and drivers.
- Can still traverse the Call Graph in a top-down or bottom-up strategy.

Call Graph of the Calendar Program



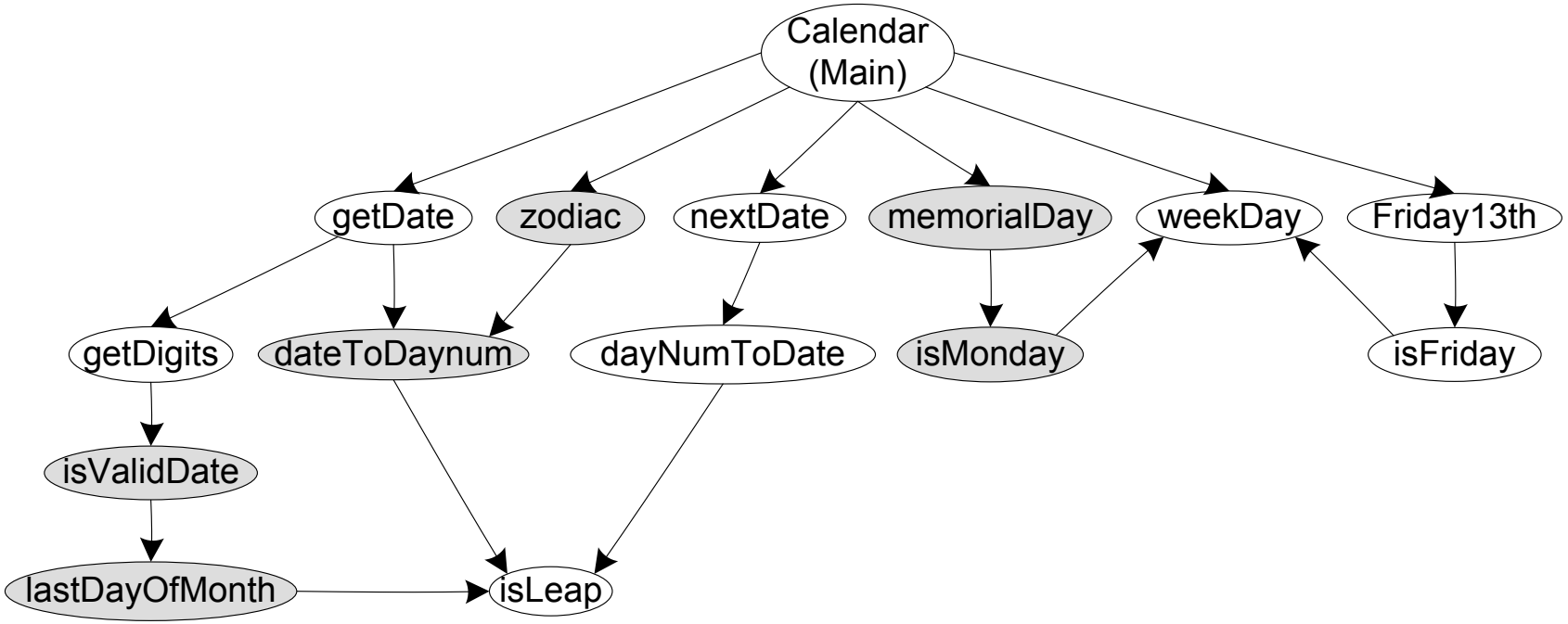
Call Graph-Based Integration (continued)

- Two strategies
  - Pair-wise integration
  - Neighborhood integration
- Degrees of nodes in the Call Graph indicate integration sessions
  - isLeap and weekDay are each used by three units
- Possible strategies
  - test high indegree nodes first, or at least,
  - pay special attention to “popular” nodes

Pair-Wise Integration

- By definition, and edge in the Call Graph refers to an interface between the units that are the endpoints of the edge.
- Every edge represents a pair of units to test.
- Still might need stubs and drivers
- Fault isolation is localized to the pair being integrated

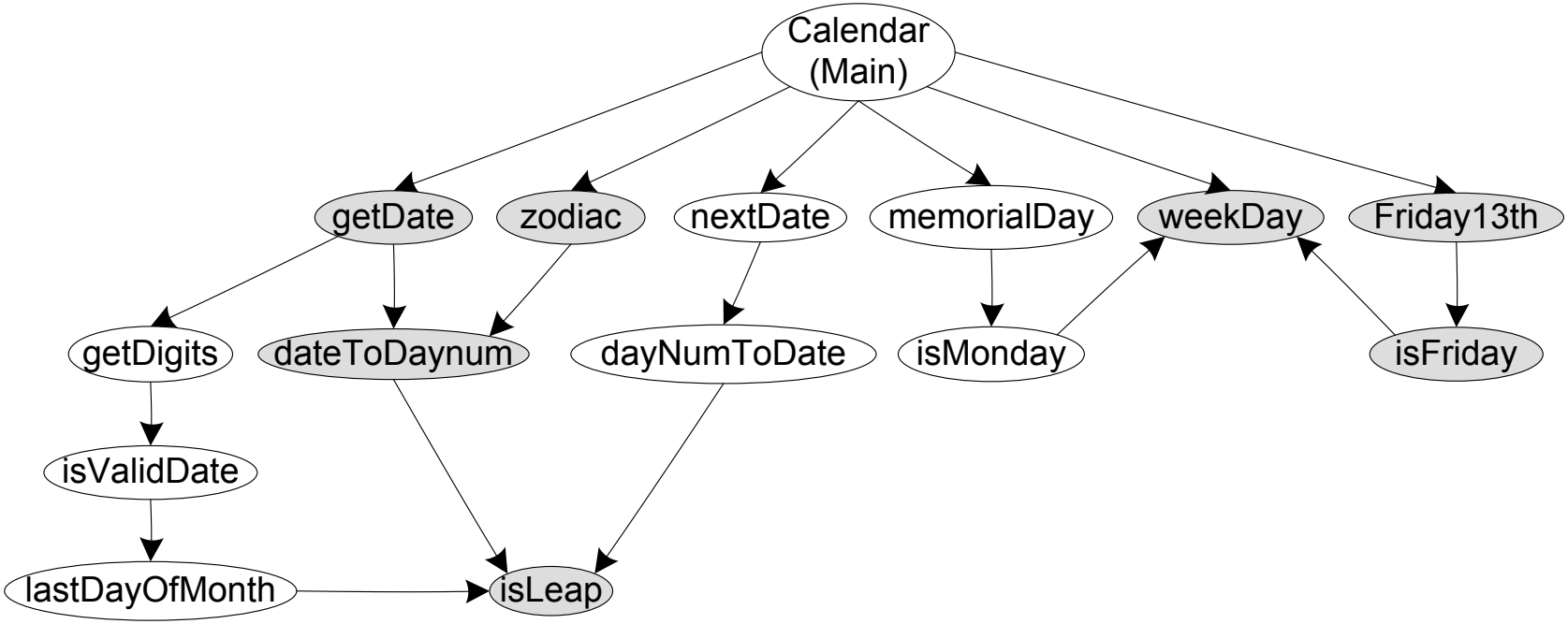
Three Pairs for Pair-Wise Integration



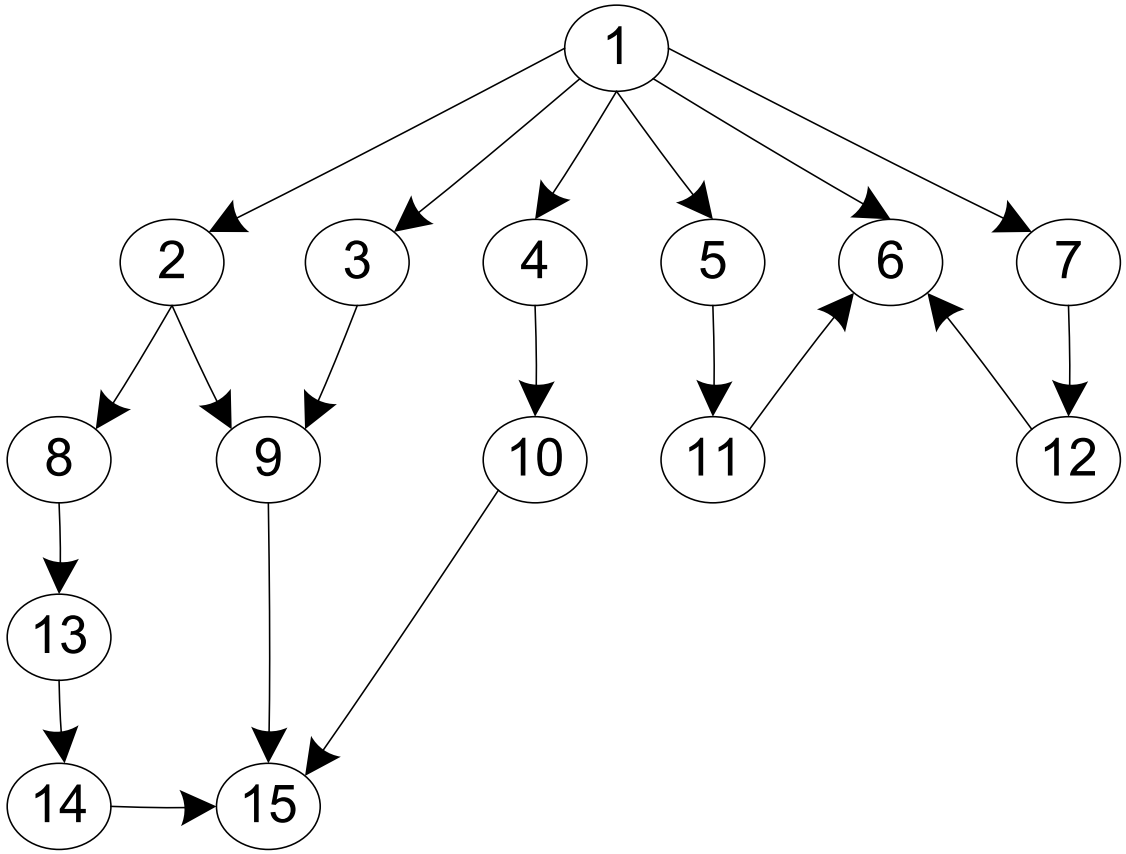
Neighborhood Integration

- The neighborhood (or radius 1) of a node in a graph is the set of nodes that are one edge away from the given node.
- This can be extended to larger sets by choosing larger values for the radius.
- Stub and driver effort is reduced.

Two Neighborhoods (radius = 1)



Calendar Call Graph with Numbered Nodes



Neighborhoods in the Calendar Program Call Graph			
Node	**Unit name**	**Predecessors**	**Successors**
1	Calendar (Main)	(none)	2, 3, 4, 5, 6, 7
2	getDate	1	8, 9
3	zodiac	1	9
4	nextDate	1	10
5	memorialDay	1	11
6	weekday	1, 11, 12	(none)
7	Friday13th	1	12
8	getDigits	2	13
9	dateToDayNum	3	15
10	dayNumToDate	4	15
11	isMonday	5	6
12	isFriday	7	6
13	isValidDate	8	14
14	lastDayOfMonth	13	15
15	isLeap	9, 10, 14	(none)

Path-Based Integration

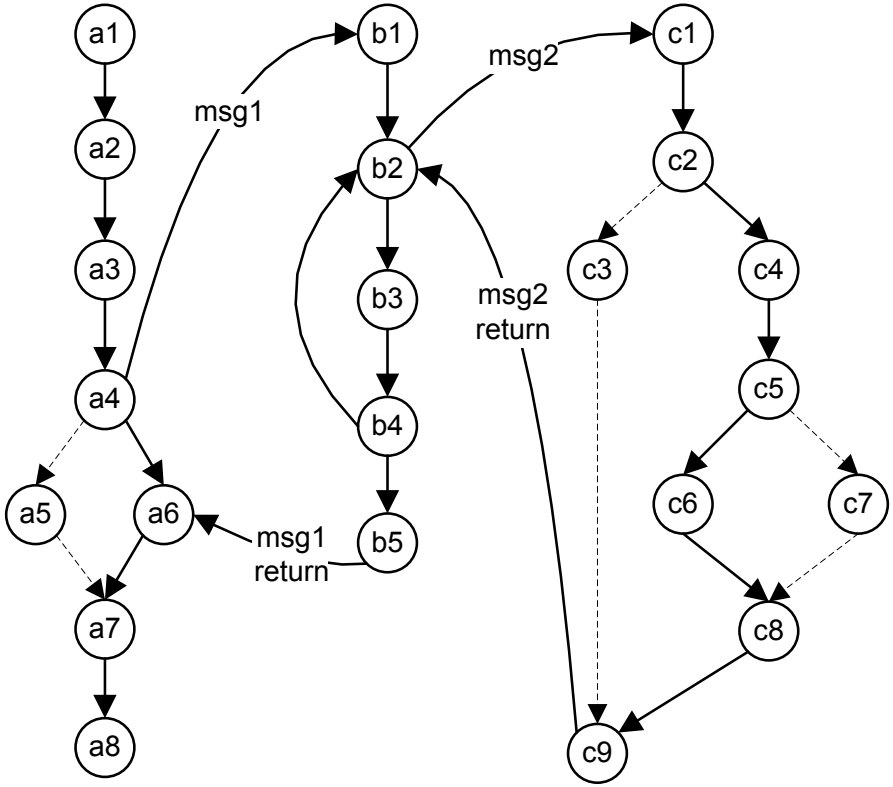
- Wanted: an integration testing level construct similar to DD-Paths for unit testing...
  - extend the symbiosis of spec-based and code-based testing to the integration level
  - greater emphasis on behavioral threads
  - shift emphasis from interface testing to interactions (co-functions) among units
- Need some new definitions
  - source and sink nodes in a program graph
  - module (unit) execution path
  - generalized message
  - MM-Path

New and Extended Definitions

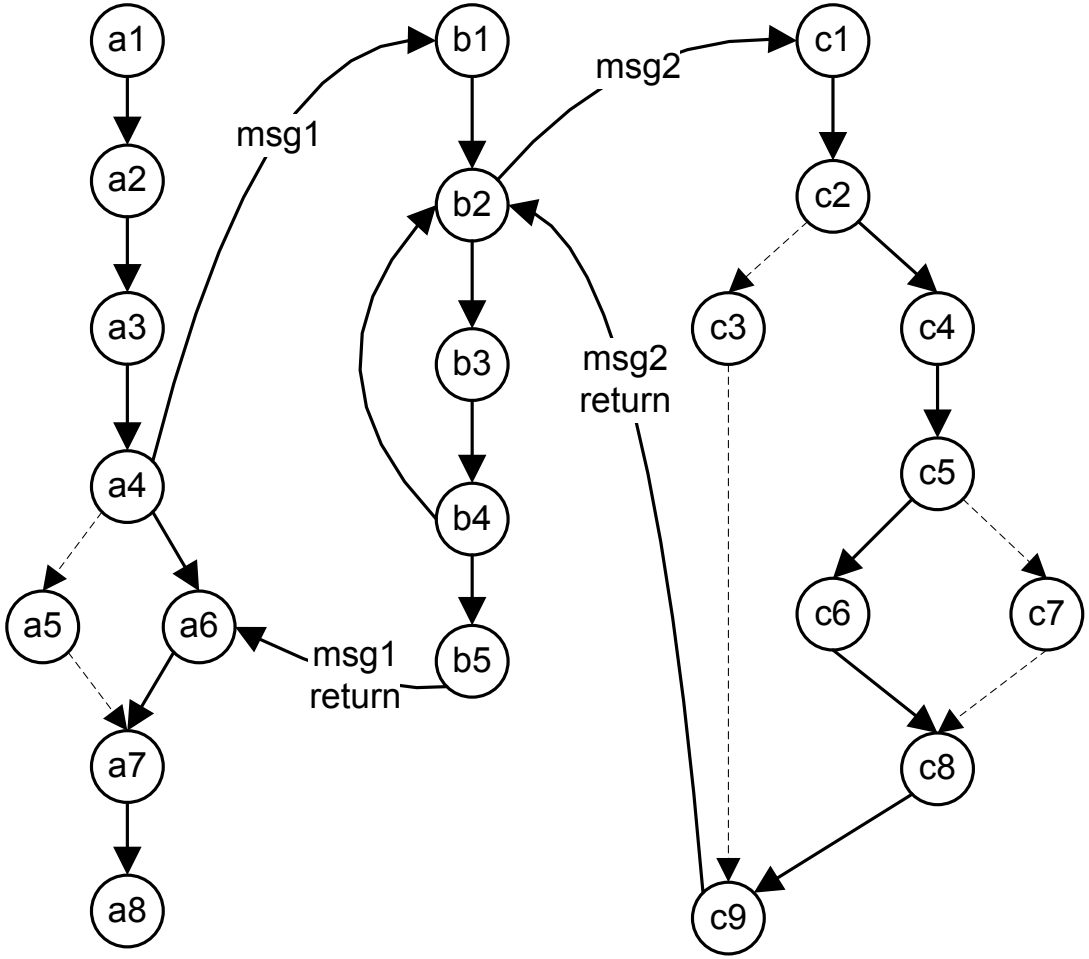
- A *source node* in a program is a statement fragment at which program execution begins or resumes.
- A *sink node* in a unit is a statement fragment at which program execution terminates.
- A *module execution path* is a sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes.
- A *message* is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.

MM-Path Definition and Example

- An *MM-Path* is an interleaved sequence of module execution paths and messages.
- An MM-Path across three units:



MM-Path Across Three Units



Exercise: List the source and sink nodes in the previous example.

Unit	Source Nodes	Sink Nodes
A		
B		
C		

Details of the Example MM-Path

The node sequence in the example MM-Path is:

<a1, a2, a3, a4>

message msg1

<b1, b2>

message msg2

<c1, c2, c4, c5, c6, c8, c9>

msg2 return

<b3, b4, (b2, b3, b4)*, b5>

msg1 return

<a6, a7, a8>

Note: the (b2, b3, b4)* is the Kleene Star notation for repeated traversal of the loop.

About MM-Paths

- Message quiescence: in a non-trivial MM-Path, there is always (at least one) a point at which no further messages are sent.
- In the example MM-Path, unit C is the point of message quiescence.
- In a data-driven program (such as NextDate), MM-Paths begin (and end) in the main program.
- In an event program (such as the Saturn Windshield Wiper), MM-Paths begin with the unit that senses an input event and end in the method that produces the corresponding output event.

Some Sequences...

- A DD-Path is a sequence of source statements.
- A unit (module) execution path is a sequence of DD-Paths.
- An MM-Path is a sequence of unit (module) execution paths.
- A (system level) thread is a sequence of MM-Paths.
- Taken together, these sequences are a strong "unifying factor" across levels of testing. They can lead to the possibility of trade-offs among levels of testing (not explored here).

Cyclomatic Complexity for MM-Paths?

- The next slide shows potential MM-Paths in two examples.
- The solid edges in thesecond slide shows actual MM-Paths.
- Possibilities for their cyclomatic complexity is explored in the next few slides.
- Questions
  - cyclomatic complexity is ALWAYS understood as a static construct/quantity
  - BUT MM-Paths are dynamic by definition and design

Cyclomatic Complexity

- The formula for strongly connected directed graphs applies, because messages always return to their sending unit.
- We always have exactly one connected region, so we use V(G) = edges – nodes + 1.
- What are nodes?
  - ordinary program graph nodes?
  - unit (module) execution paths?
  - a full unit?
- what are edges?
  - ordinary program graph edges?
  - messages?
  - Both ordinary program graph edges and messages?
  - what about message return edges?

Applying These Choices to the Three Unit Example...

- $V_1(G)$ = both ordinary program graph edges and messages– ordinary program graph nodes+ 1.
- $V_2(G)$ = messages - module execution paths + 1
- $V_3(G)$ = messages - full units + 1
- $V_1(G) = 27 – 22 + 1 = 6$
- $V_2(G) = 4 – 7 + 1 = -2$
- $V_3(G) = 4 – 3 + 1 = 2$

- $V_2(G)$ won't work.
- Next: apply $V_1(G)$ and $V_3(G)$ to the 4 unit example (Integration NextDate)

Applying $V_1(G)$ and $V_3(G)$ to Integration NextDate...

- $V_1(G) = 75 - 52 + 1 = 24$
- $V_3(G) = 6 - 4 + 1 = 3$
- Still doesn't "seem" right...
  - According to $V_1(G)$, integration NextDate has four times the complexity of the three unit example
  - According to $V_2(G)$, integration NextDate has only 50% more complexity than the three unit example
- BUT, recalling the static nature of cyclomatic complexity, consider $V_4(G)$ is the sum of the individual unit complexities plus the additional messages. $V_4(G) = 21 + 4 = 25$, which lines up pretty well with $V_1(G)$.
- (to be continued in Chapter 16)

Comparison of Integration Testing Strategies

Strategy Basis	Ability to test interfaces	Ability to test co-functionality	Fault isolation and resolution
Functional Decomposition	acceptable, but can be deceptive (phantom edges)	limited to pairs of units	good to a faulty unit
Call Graph	acceptable	limited to pairs of units	good to a faulty unit
MM-Path	excellent	complete	excellent to a faulty unit execution path