Software Quality Assurance and Testing

Integration Testing

Outline

- Case Study - Mars Climate Orbiter
- Integration Testing
- Integration Test Strategies
- System, Acceptance, and Regression Testing
  - Acceptance Testing
  - System Testing
  - Regression Testing
- Summary

Case Study –
Mars Climate Orbiter

Case Study – Mars Climate Orbiter

- NASA's Mars Climate Orbiter
  - Launched on December 11, 1998
  - Intended to enter an orbit at 140 –150 km above Mars.
- On September 23, 1999
  - It smashed into the planet's atmosphere and was destroyed.
- Cost: $328M

Case Study – Mars Climate Orbiter

- Cause of failure
  - The software controlling the thrusters on the spacecraft used different units.
- Software modules were developed by teams in US and Europe
- Engineers failed to convert the measure of rocket thrusts
  - English unit: Pound-Force
  - Metric unit: Newton, $kg \bullet m / s^2$
  - Difference: a factor of ≈ 4.45

Integration Testing

- **Integration testing** (sometimes called **integration** and **testing**, abbreviated I&T) is the phase in software **testing** in which individual software modules are combined and tested as a group. It occurs after unit **testing** and before validation **testing**.

Objectives

- Understand the purpose of integration testing
  - Distinguish typical integration faults from faults that should be eliminated in unit testing
  - Understand the nature of integration faults and how to prevent as well as detect them
- Understand strategies for ordering construction and testing
  - Approaches to incremental assembly and testing to reduce effort and control risk

Integration vs. Unit Testing

- Unit (module) testing is a necessary foundation
  - Unit level has maximum controllability and visibility
  - Integration testing can never compensate for inadequate unit testing
- Integration testing may serve as a process check
  - If module faults are revealed in integration testing, they signal inadequate unit testing
  - If integration faults occur in interfaces between correctly implemented modules, the errors can be traced to module breakdown and interface specifications

Integration Testing

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Why do we do integration testing?

- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- Often many Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

Types of Testing

- Unit Testing:
  - Individual subsystem
  - Carried out by developers (of components)
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality
- Integration Testing:
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interface and the interplay among the subsystems

Types of Testing

- System Testing:
  - The entire system
  - Carried out by developers (testers!)
  - Goal: Determine if the system meets the requirements (functional and global)
  - Functional Testing: Test of functional requirements
  - Performance Testing: Test of non-functional requirements
- Acceptance and Installation Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client.
  - Goal: Demonstrate that the system meets customer requirements and is ready to use

What is Integration Testing?

	Unit/module test	Integration test	System test
Specification:	Module interface	Interface specs, module breakdown	Requirements specification
Visible structure:	Coding details	Modular structure (software architecture)	— none —
Scaffolding required:	Some	Often extensive	Some
Looking for faults in:	Modules	Interactions, compatibility	System functionality

Testing Level Assumptions and Objectives

Unit Assumptions	**Unit Goals**
• All other units are correct • Compiles correctly	• Correct unit function • Coverage metrics satisfied
Integration Assumptions	Integration Goals
Unit testing complete	• Interfaces correct • Correct function across units • Fault isolation support
System Assumptions	System Goals
• Integration testing complete • Tests occur at port boundary	• Correct system functions • Non-functional requirements tested • Customer satisfaction

What is Software Integration Testing?

- Testing activities that integrate software components together to form a complete system. To perform a cost-effective software integration, integration test strategy, integration test set are needed.
- Integration testing focuses on:
  - Interfaces between modules (or components)
  - Integrated functional features
  - Interacting protocols and messages
  - System architectures
- Who performs software integration:
  - Developers and test engineers
- What do you need?:
  - Integration strategy
  - Integration test environment and test suite
  - Module (or component) specifications
  - Interface and design documents

Integration Faults

- Inconsistent interpretation of parameters or values
  - Example: Mixed units (Pound/Newton) in Martian Lander
- Violations of value domains, capacity, or size limits
  - Example: Buffer overflow
- Side effects on parameters or resources
  - Example: Conflict on (unspecified) temporary file

Integration Faults

- Omitted or misunderstood functionality
  - Example: Inconsistent interpretation of web requests
- Nonfunctional properties
  - Example: Unanticipated performance issues
- Dynamic mismatches
  - Example: Incompatible polymorphic method calls

Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
static void ssl_io_filter_disable(ap filter t *f)  {
    bio_filter_in_ctx_t *inctx = f->ctx;

    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

No obvious error, but Apache leaked memory slowly (in normal use) or quickly (if exploited for a DOS attack)

Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
static void ssl_io_filter_disable(ap filter t *f)  {
    bio_filter_in_ctx_t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

The missing code is for a **structure defined and created elsewhere**, accessed through an opaque pointer.

Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
static void ssl_io_filter_disable(ap filter t *f)  {
    bio_filter_in_ctx_t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

Almost impossible to find with unit testing. (Inspection and some dynamic techniques could have found it.)

What is a software integration strategy?

- Software test strategy provides the basic strategy and guidelines to test engineers to perform software testing activities in a rational way.
- Software integration strategy usually refers to
  - an integration sequence (or order) to integrate different parts (or components) together

Integration Test Strategies

Maybe You’ve Heard …

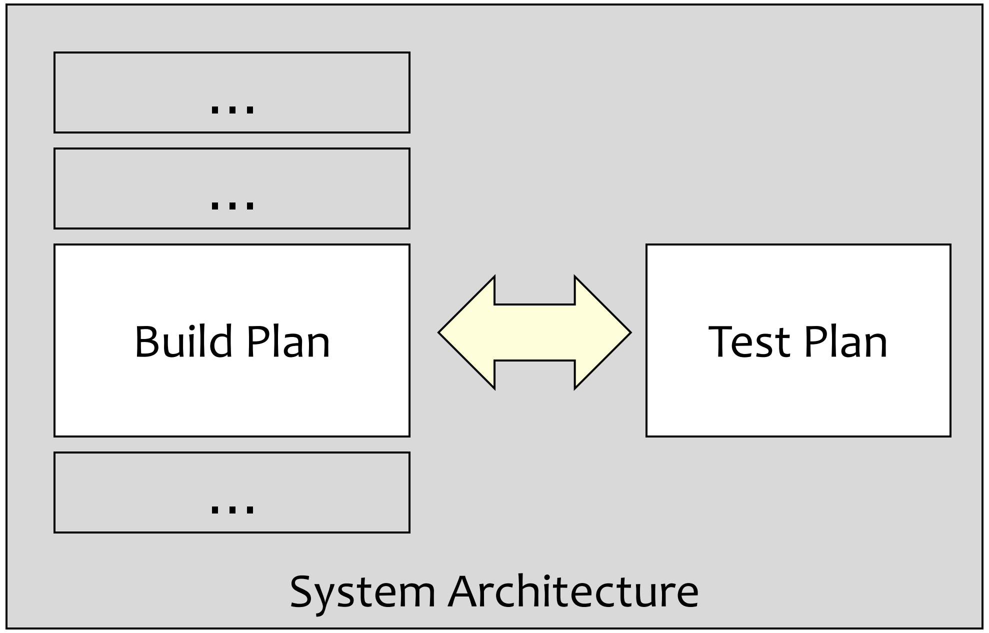
- Yes, I implemented *module A*, but I didn’t test it thoroughly yet.
- It will be tested along with *module B* when that’s ready.

Translation …

- Yes, I implemented *module A*, but I didn't test it thoroughly yet.
- It will be tested along with *module B* when that's ready.

- I didn't think at all about the strategy for testing.
- I didn't design *module A* for testability and I didn't think about the best order to build and test modules *A* and *B*

Integration Plan & Test Plan

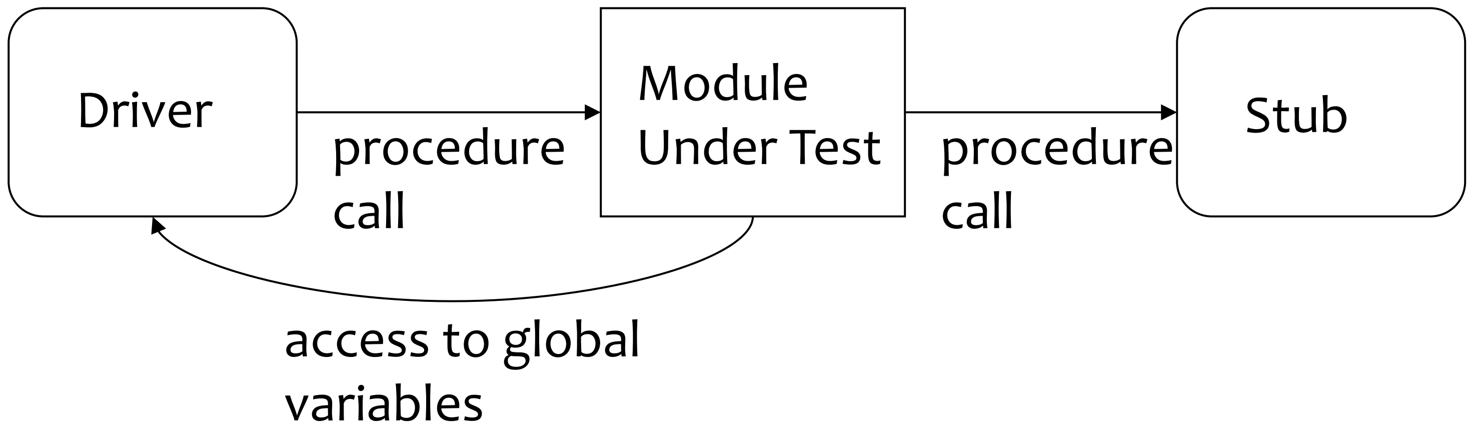


- Integration test plan drives and is driven by the project "build plan"
  - A key feature of the system architecture and project plan

Drivers and Stubs

- **Driver:** A program that calls the interface procedures of the module being tested and reports the results
  - A driver simulates a module that calls the module currently being tested
- **Stub:** A program that has the same interface as a module that is being used by the module being tested, but is simpler.
  - A stub simulates a module called by the module currently being tested

Drivers and Stubs

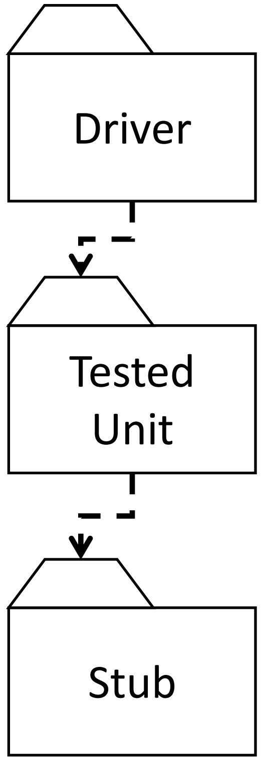


- Driver and Stub should have the same interface as the modules they replace
- Driver and Stub should be simpler than the modules they replace

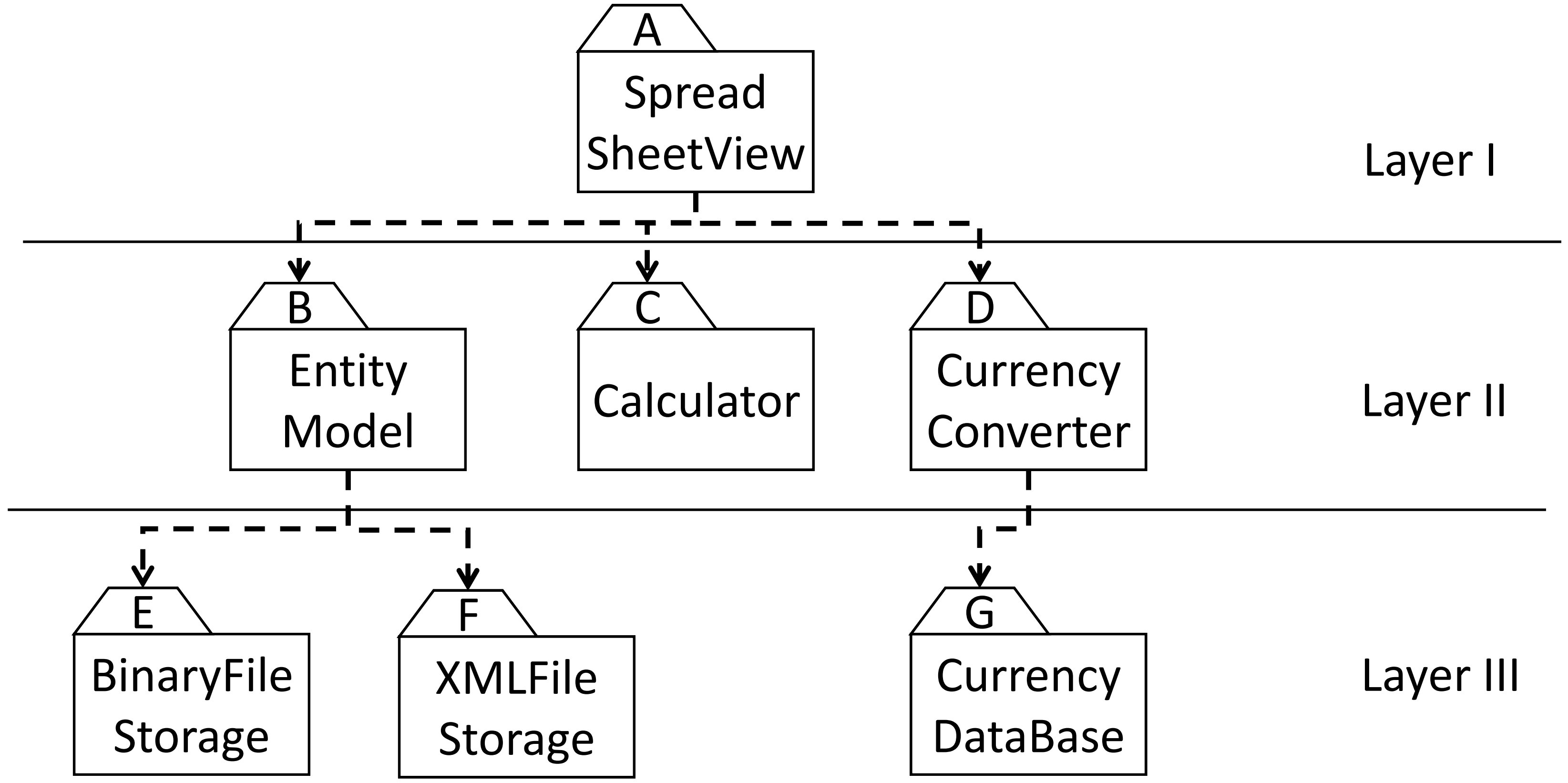
Stubs and drivers

- Driver:
  - A component, that calls the **`TestedUnit`**
  - Controls the test cases

- Stub:
  - A component, the **`TestedUnit`** depends on
  - Partial implementation
  - Returns fake values.



Example: A 3-Layer-Design (Spreadsheet)



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

Basis of Integration Testing Strategies

- Functional Decomposition applies best to procedural code
- Call Graph
  - applies to both procedural and object-oriented code

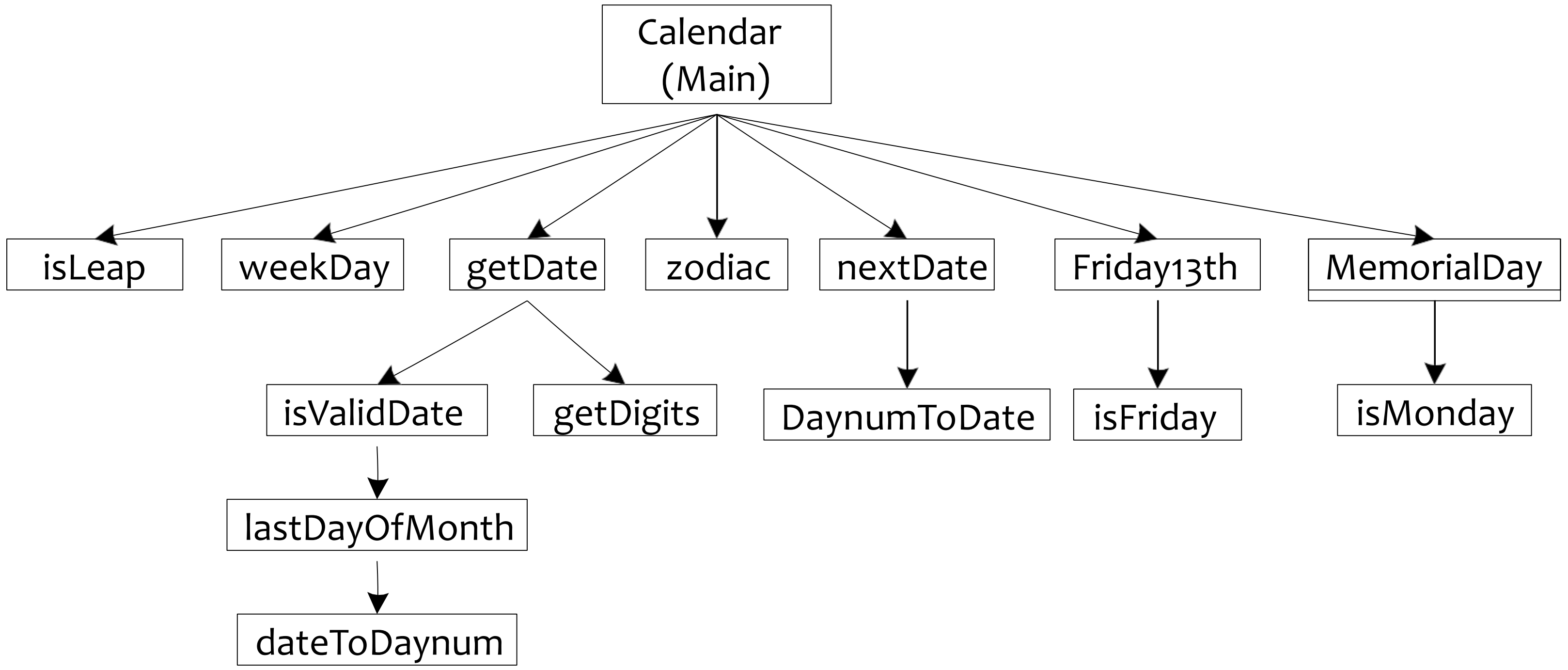
Example—Calendar Program

- Date in the form mm, dd, yyyy
- Calendar functions
  - the date of the next day (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

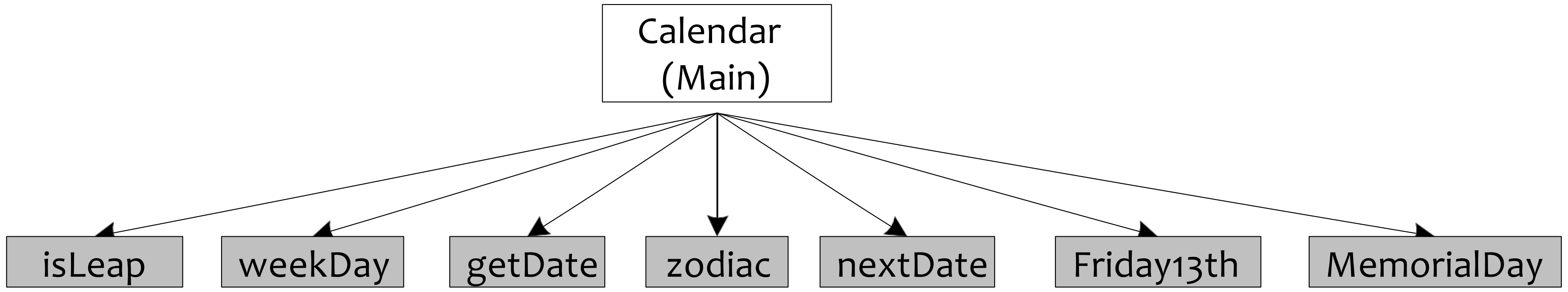
Calendar Program Units

```
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



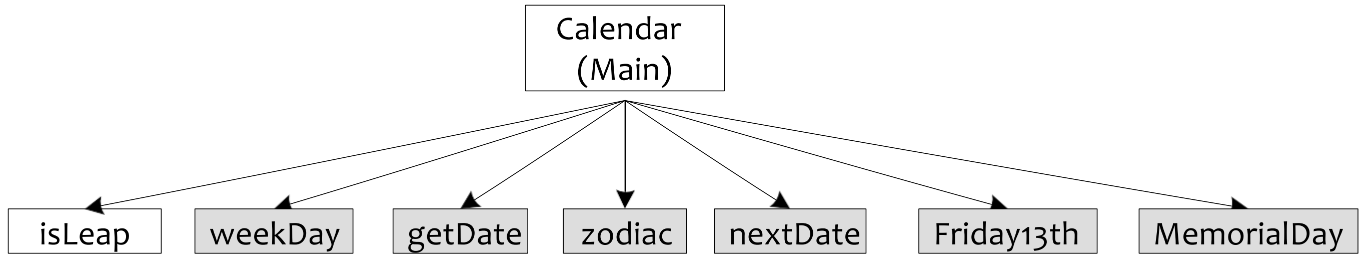
“Grey” units are stubs that return the correct values when referenced.
This level checks the main program logic.

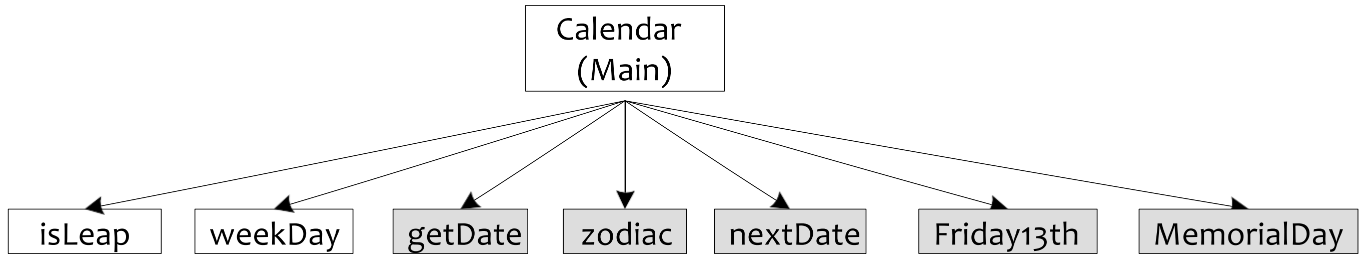
weekDayStub

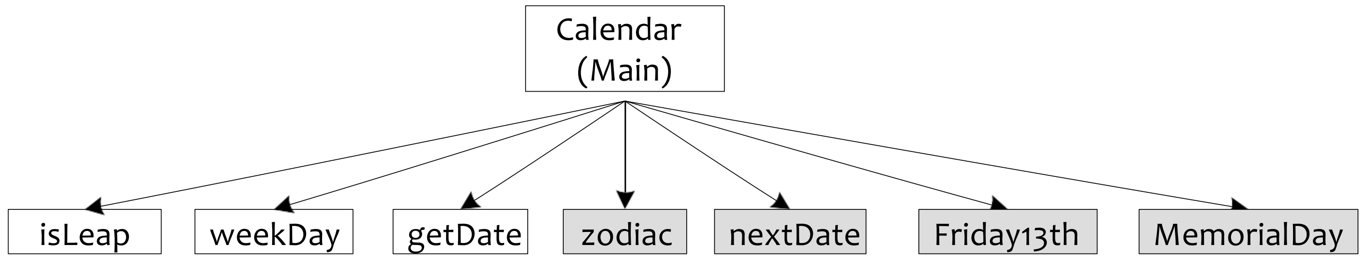
```
Procedure weekDayStub(mm, dd, yyyy, dayName) If
((mm = 10) AND (dd = 28) AND (yyyy = 2013))
        Then dayName = “Monday”
EndIf
.
.
.
If ((mm = 10) AND (dd = 30) AND (yyyy = 2013))
        Then dayName = “Wednesday”
EndIf
```

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

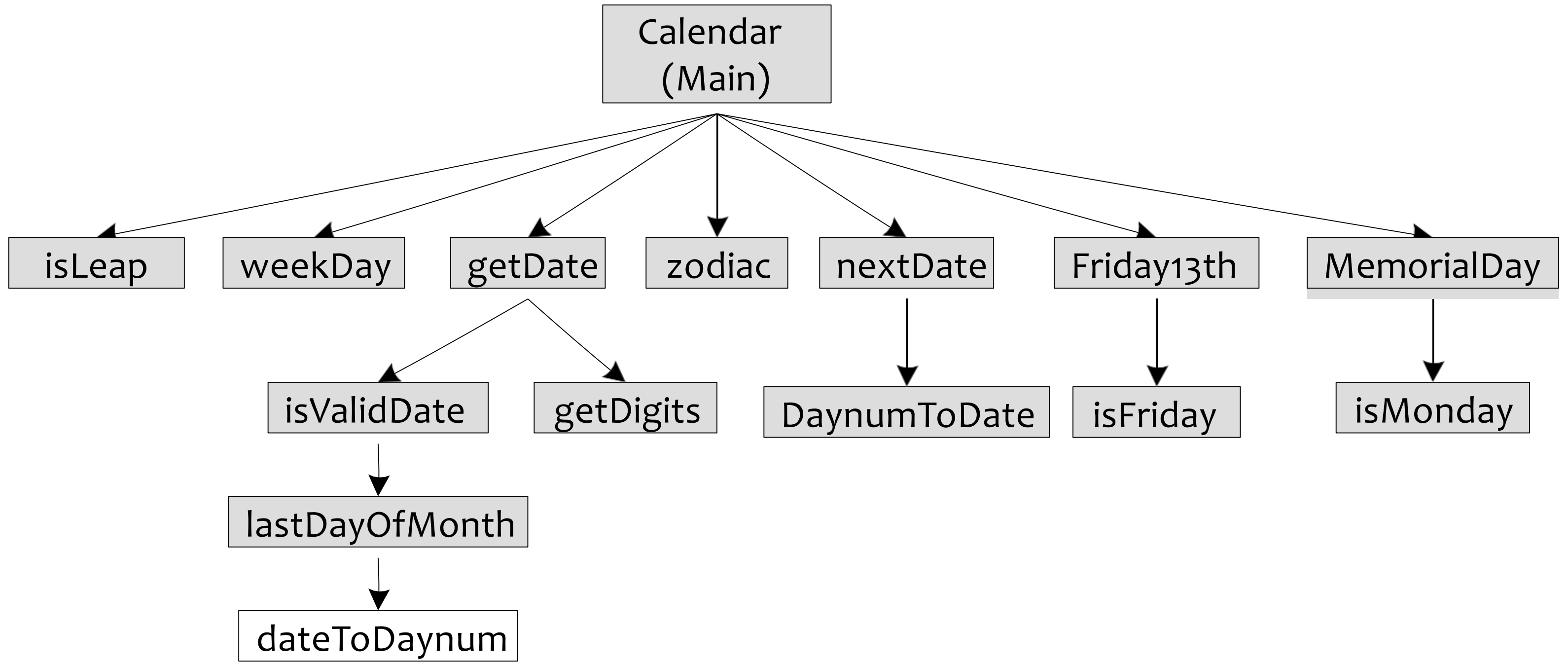
Bottom-Up Integration Mechanism

- 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.

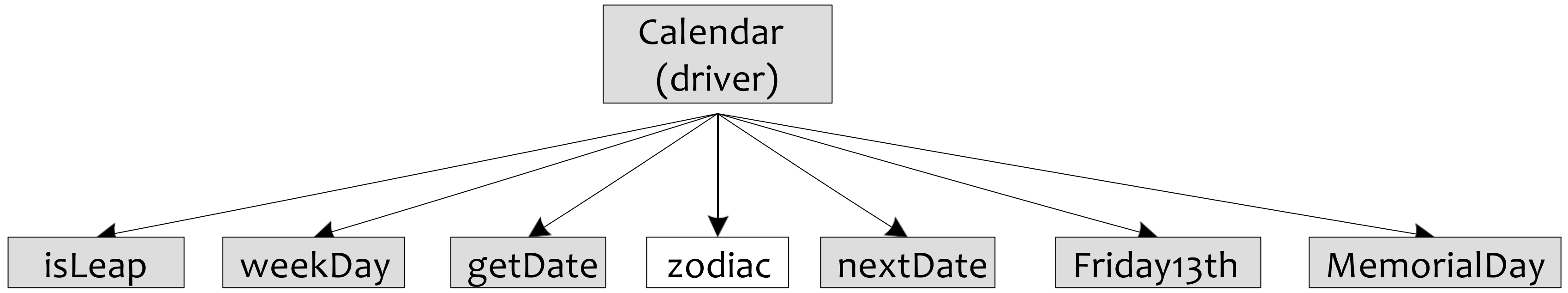
Top-Down and Bottom-Up Integration

- 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 in some redundancy.

Starting Point of Bottom-Up Integration



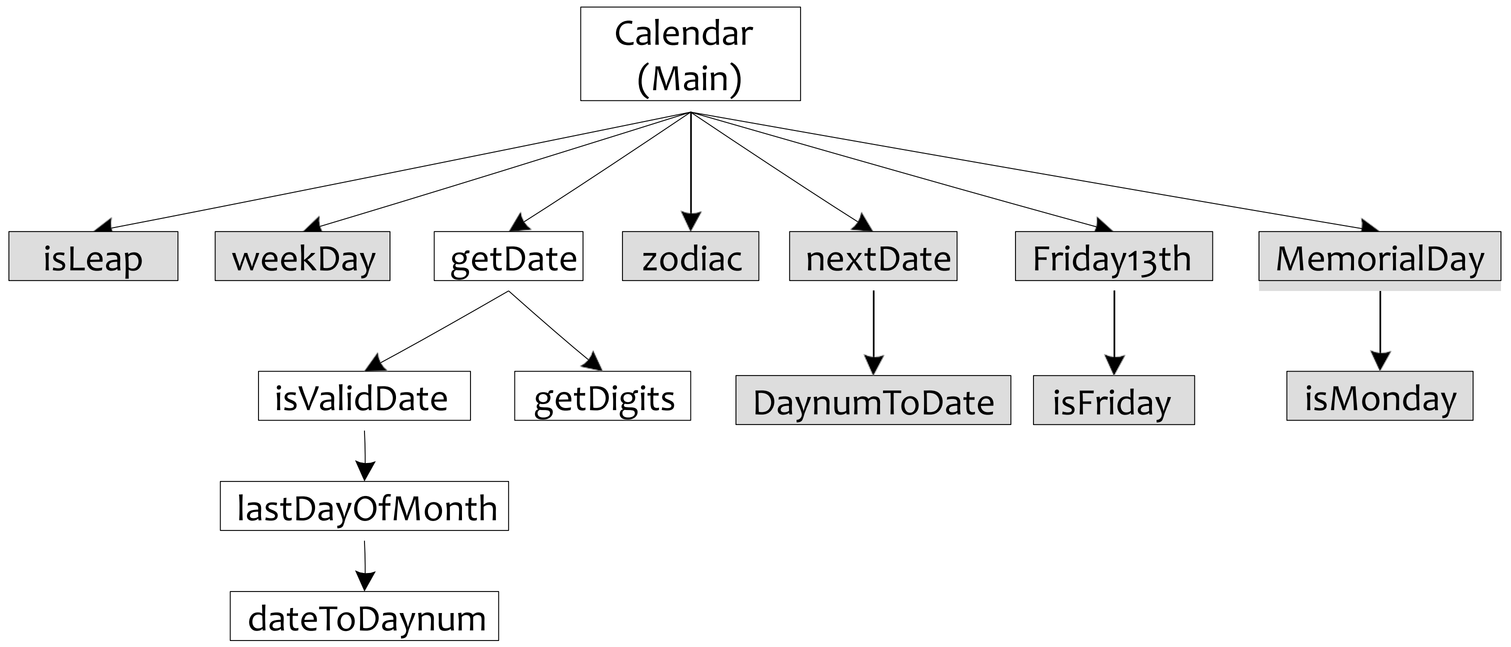
Bottom-Up Integration of Zodiac



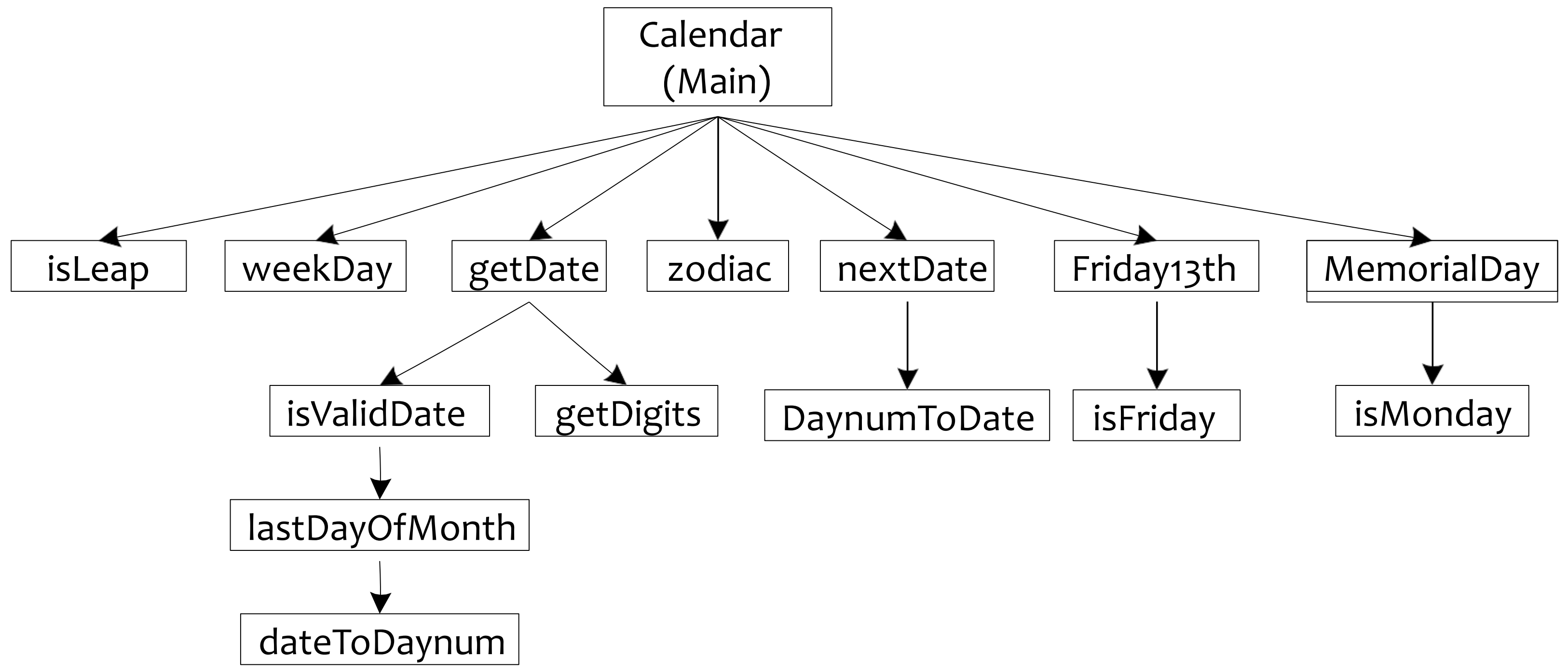
Sandwich Integration

- 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
- This is the practice in an agile environment with a daily run of the project to that point.

Top-down Approach Vs Bottom-up Approach

Basis for Comparison	Top-down Approach	Bottom-up Approach
Basic	Breaks the massive problem into smaller sub-problems	Solves the fundamental low-level problem and integrates them into a larger one
Process	Submodules are solitarily analyzed	Examine what data is to be encapsulated, and implies the concept of information hiding
Redundancy	Contain redundant information	Redundancy can be eliminated
Programming languages	Structure/procedural oriented programming languages	Object-oriented programming languages
Mainly used in	Module documentation, test case creation, code implementation and debugging	Testing

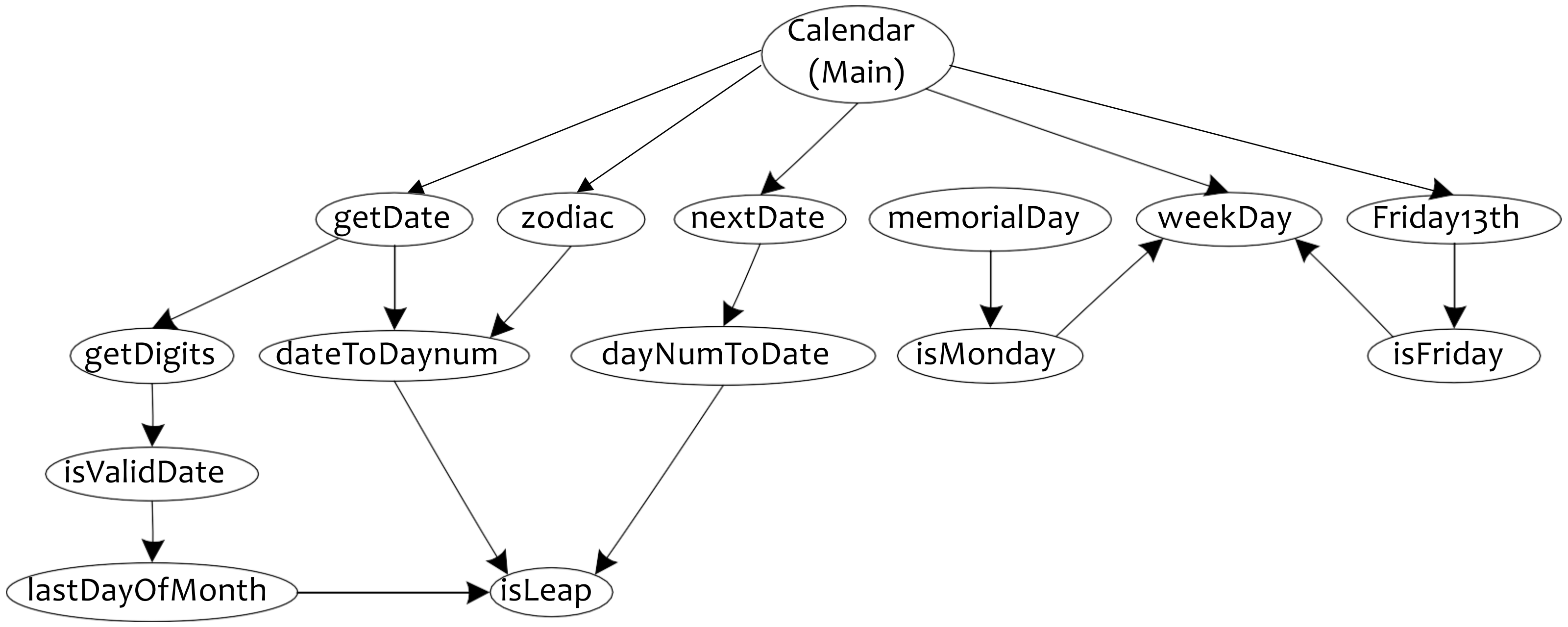
Pros and Cons of Decomposition-Based Integration

- 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



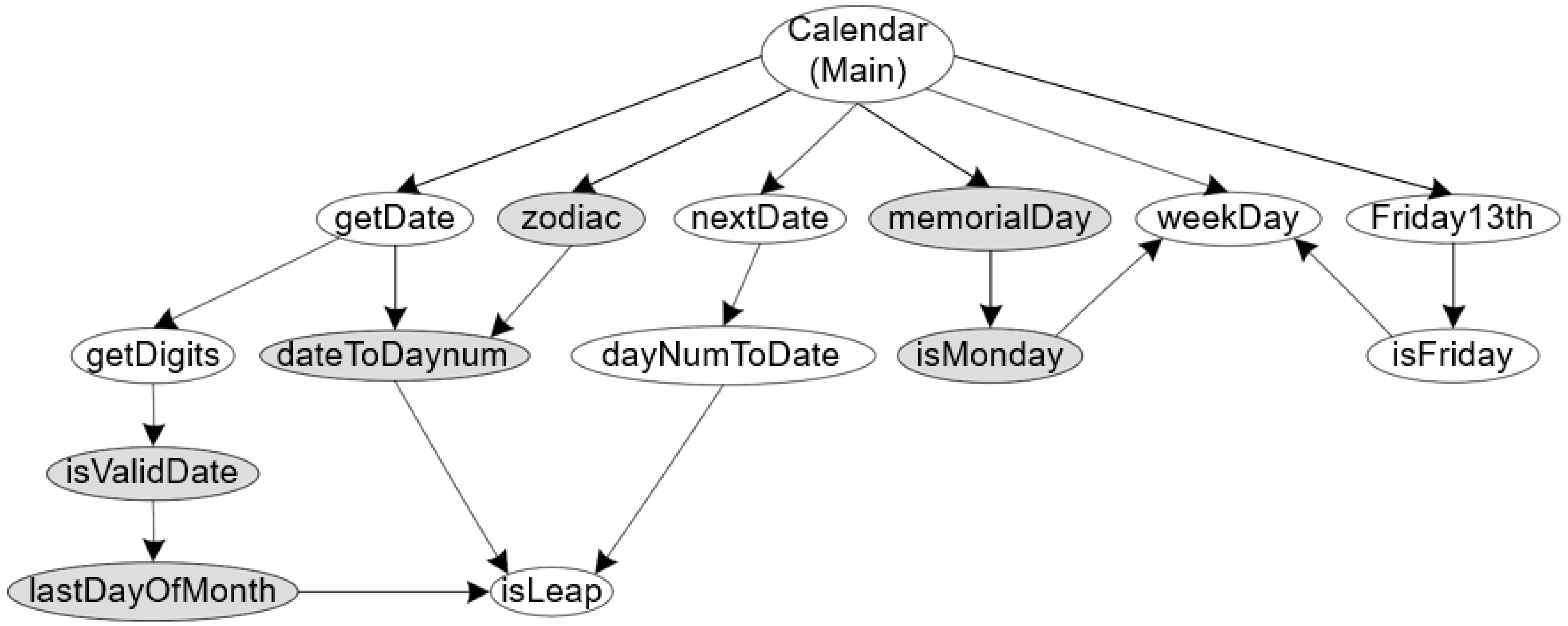
Call Graph-Based Integration (continued)

- 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, an 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

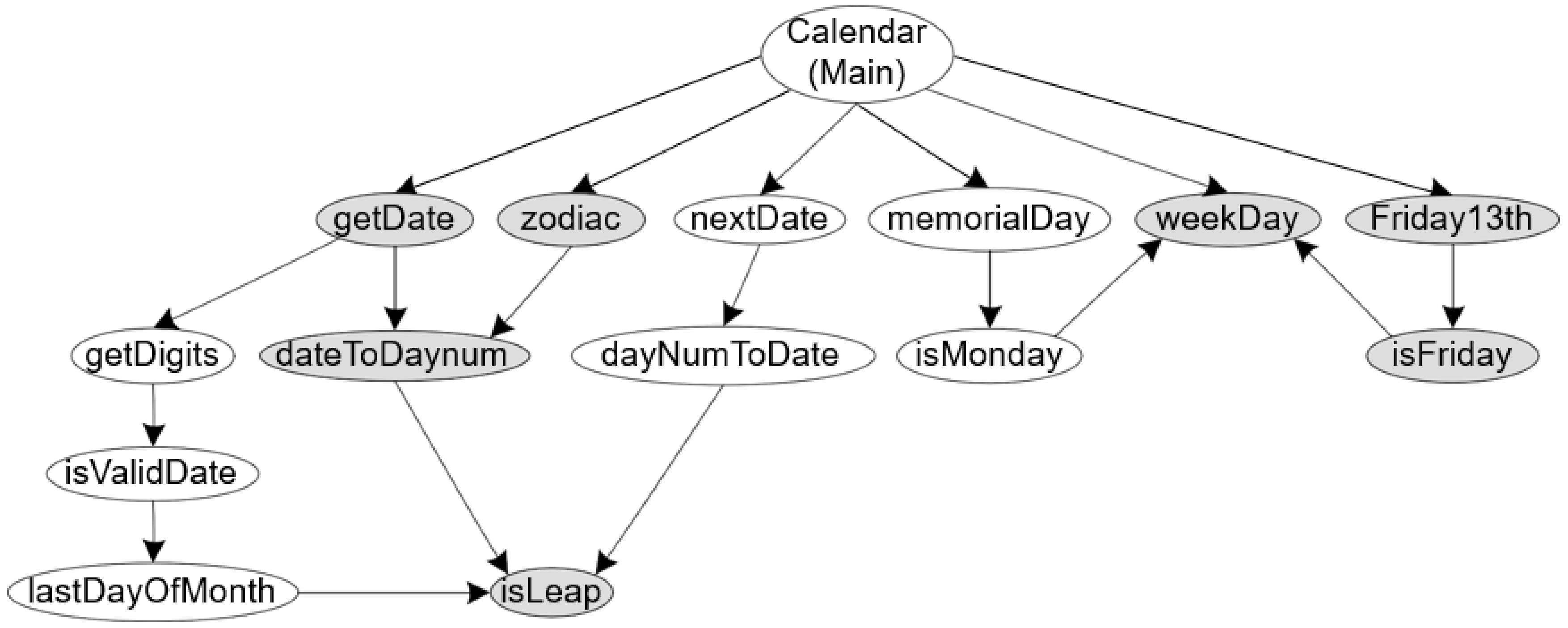
Three Pairs for Pair-Wise Integration



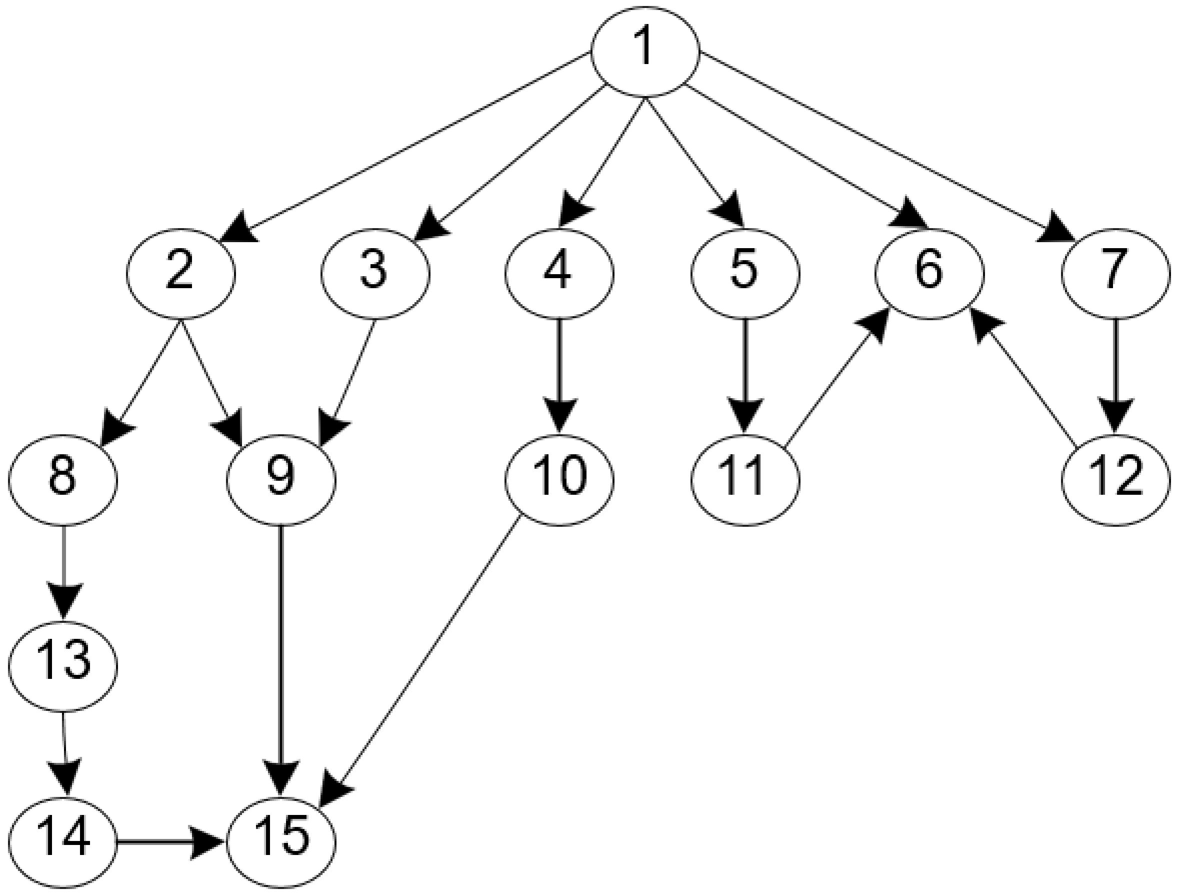
Neighborhood Integration

- 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)

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

Continuous Testing

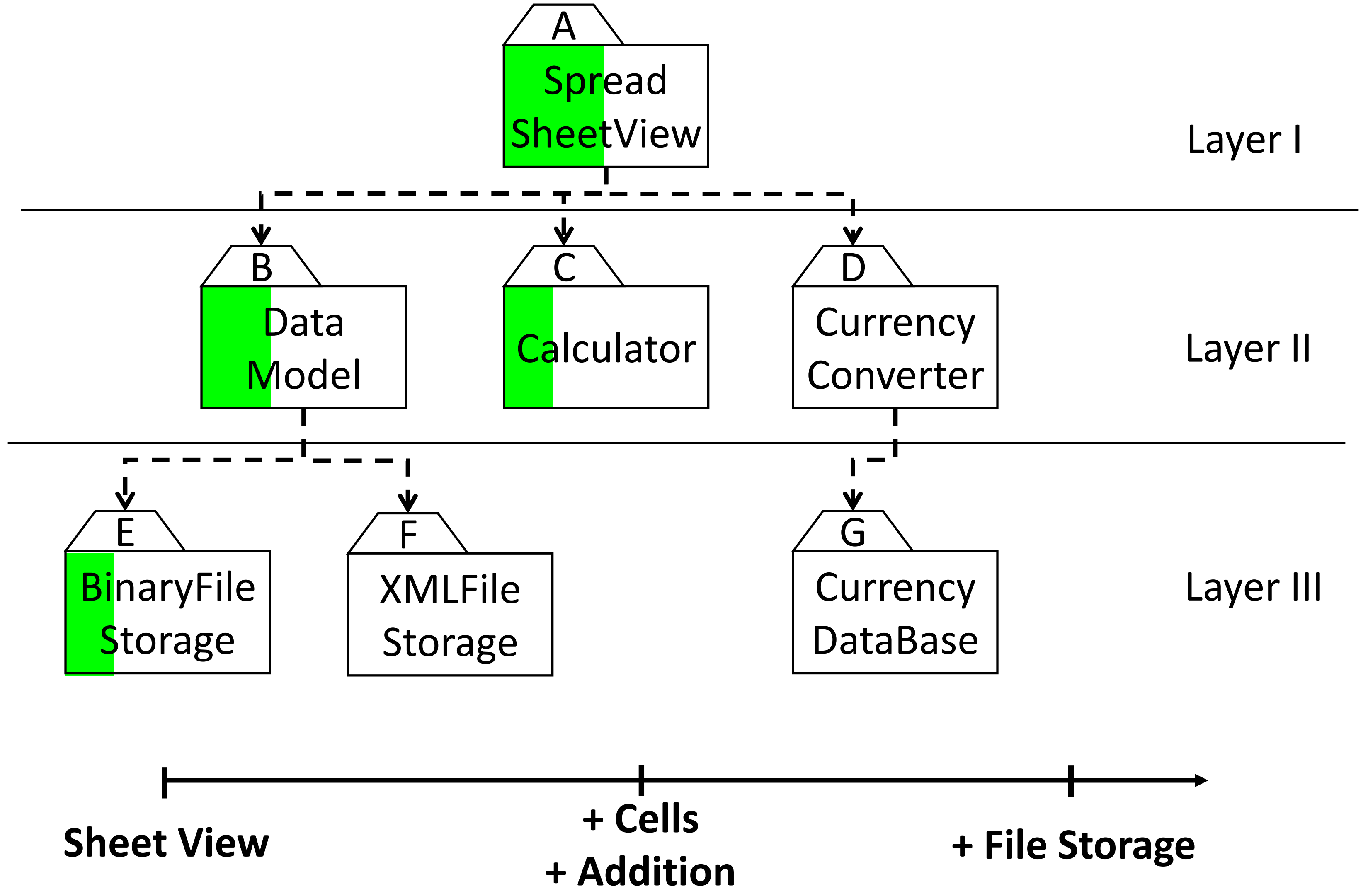
- Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day.

- Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

Continuous Testing

- Continuous build:
  - Build from day one
  - Test from day one
  - Integrate from day one
  - System is always runnable
- Requires integrated tool support:
  - Continuous build server
  - Automated tests with high coverage
  - Tool supported refactoring
  - Software configuration management
  - Issue tracking

Continuous Testing Strategy



Which Integration Strategy should you use?

- Factors to consider
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns

Which Integration Strategy should you use?

- Bottom up approach
  - Good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - Top-level components are usually important and cannot be neglected up to the end of testing
  - Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult

Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements*:* Define test cases that exercise all use cases with the selected component
4. Test subsystem decomposition*:* Define test cases that exercise all dependencies
5. Test non-functional requirements: Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify failures* with the (current) component *configuration*.

Summary

- Integration testing focuses on interactions
  - Must be built on foundation of thorough unit testing
  - Integration faults often traceable to incomplete or misunderstood interface specifications
    - Prefer prevention to detection, and make detection easier by imposing design constraints
- Strategies tied to project build order
  - Order construction, integration, and testing to reduce cost or risk