Verification part II

(Note: verification used in a very general sense, including validation)

Credits: Michal Young
Mauro Pezzè

Analysis

- In the previous lectures we have seen
  - Inspections
  - Automated analyses on code
    - Data-flow analysis
    - Symbolic execution
  - … we have also used formal methods (in particular, Alloy) …

Testing

- Program testing can be used to show the presence of bugs, but never to show their absence. (Dijkstra 1972)

Definitions

- Test case t. Includes
  - A set of inputs for the system
  - An hypothesis on the state of the system at the time of the test case execution
  - The expected output
- Test set T
  - A set of test cases
- If P is our system
  - A test case t is successful if P(t) is correct
  - A test set T is successful if P correct for all t in T

How to select test sets?

- Random testing
  - is not effective to find flaws (it is "blind", it does not "look for bugs")

- Systematic testing:
  - Use characteristics/structure of the software artifacts (e.g., code)
  - Use information on the behavior of the system (e.g., specifications)

Test criteria

- A test criterion C identifies some test sets
- A test set T satisfies C if it is an element of C
  - Example
  - C = {<x1, x2,..., xn> | $n \geq 3 \wedge \exists\, i, j, k, (x_i<0 \wedge x_j=0 \wedge x_k>0)$}
  - <-5, 0, 22> is a test set that satisfies C
  - <-10, 2, 8, 33, 0, -19> also does
  - <1, 3, 99> does not

How do we identify test sets?

- Two main classes
  - White-box: based on the knowledge of software
    - Attempts to identify test sets that cause the execution of all instructions or of all branches or specific paths… in a system
  - Black-box: assume that the structure of software is unknown.
    - Test sets are based on the knowledge of some spec of the system
    - See later

Black-box vs white-box testing

- White-box testing is suitable for unit testing
  - Covering a small portion of software is possible

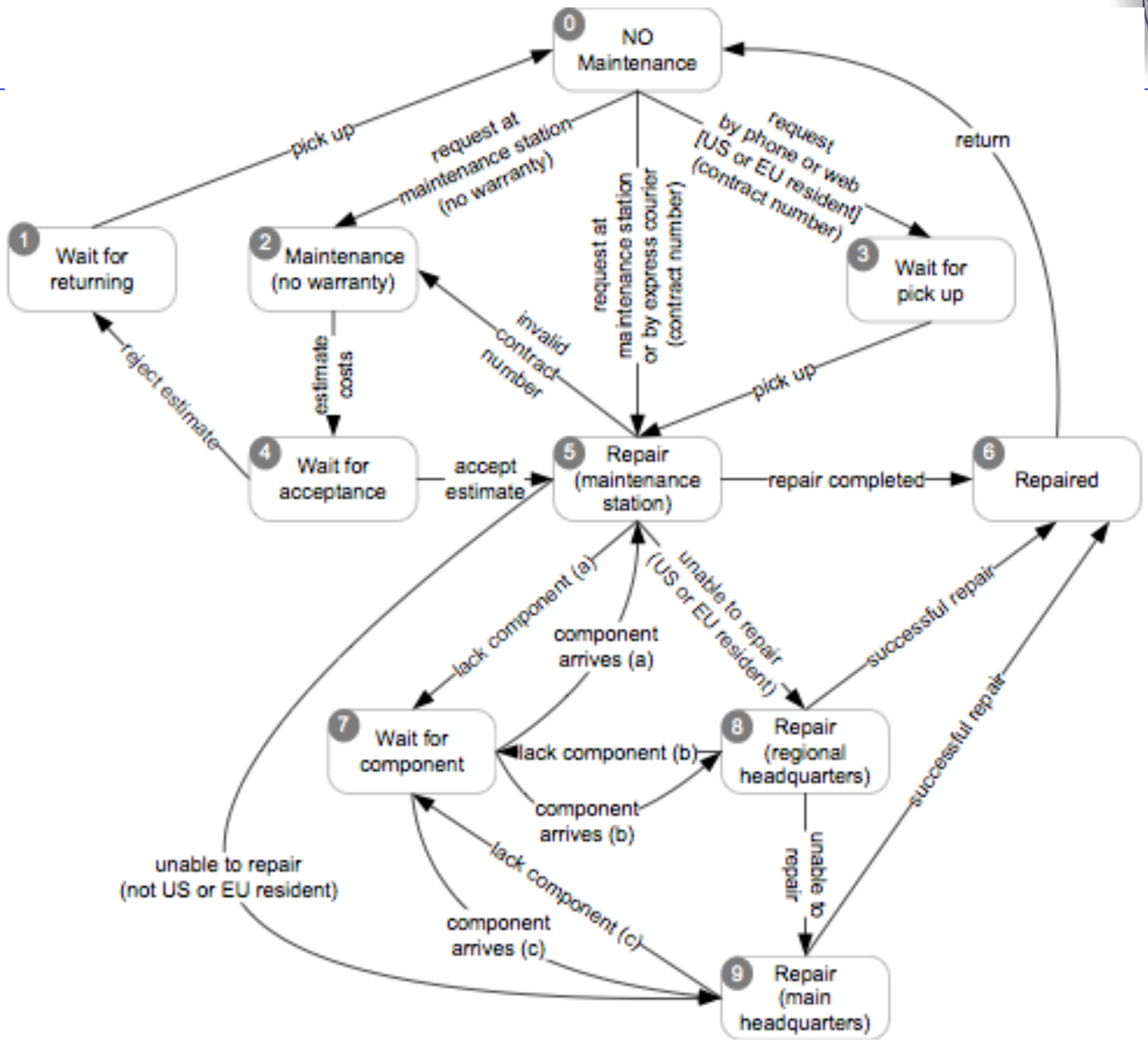
- Black-box testing is suitable for integration and acceptance testing
  - Specs usually smaller than code
  - Specs help identifying missing functionalities in the system

Black-box model-based testing

- Models used in specification or design have structure
- We can devise test cases to check actual behavior against behavior specified by the model

Example: deriving test cases from state diagrams

- The steps
  - Analyze a state diagram
  - Identify test cases
  - Check that test cases fulfill a coverage criterion
  - Execute test cases on the actual systems

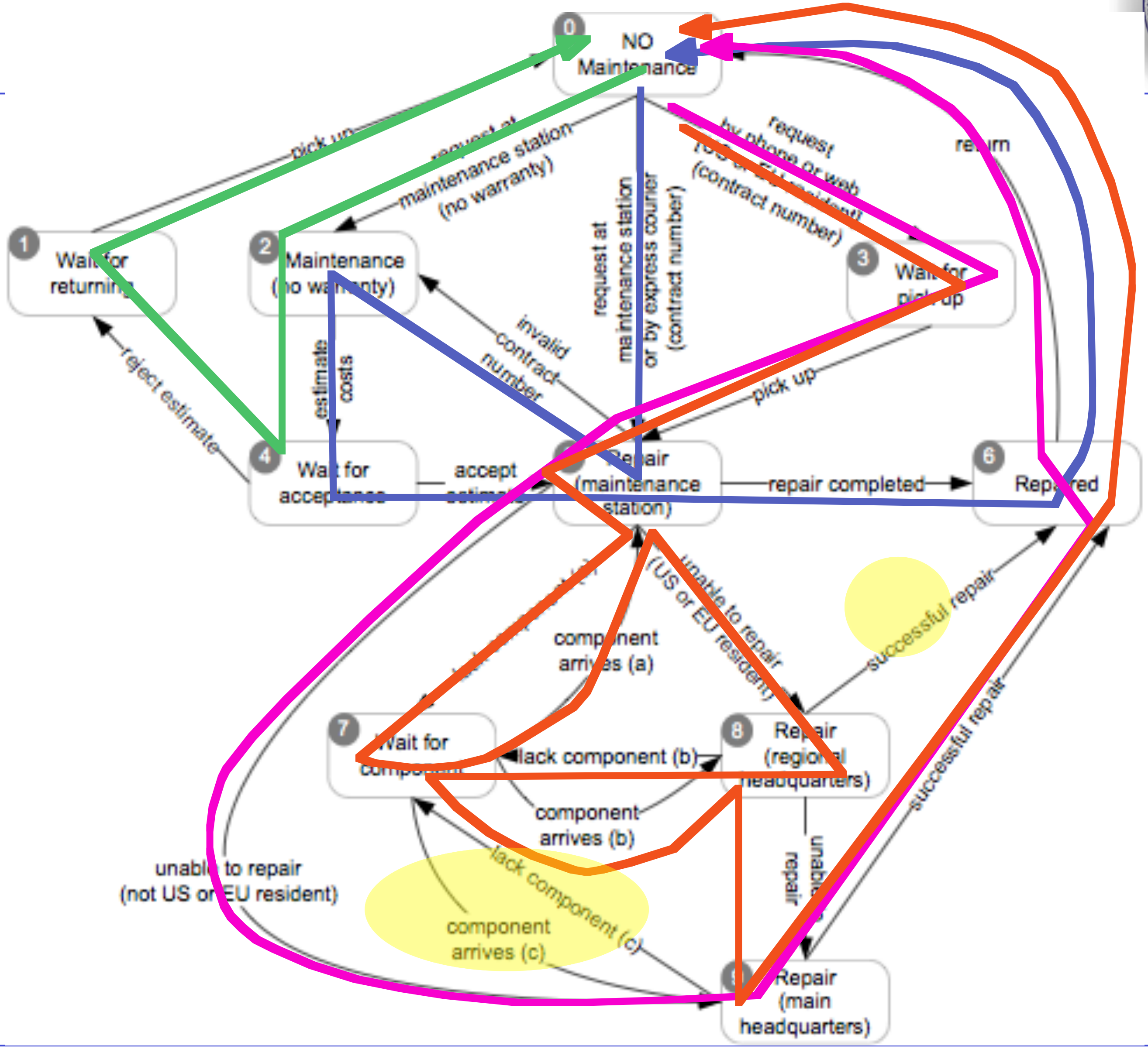

0 NO Maintenance
1 Wait for returning
2 Maintenance (no warranty)
3 Wait for pick up
4 Wait for acceptance
5 Repair (maintenance station)
6 Repaired
7 Wait for component
8 Repair (regional headquarters)
9 Repair (main headquarters)
pick up
request at maintenance station (no warranty)
request at maintenance station or by express courier (contract number)
request by phone or web [US or EU resident] (contract number)
return
invalid contract number
estimate costs
reject estimate
pick up
accept estimate
repair completed
lack component (a)
component arrives (a)
unable to repair (US or EU resident)
successful repair
lack component (b)
component arrives (b)
successful repair
unable to repair
unable to repair (not US or EU resident)
lack component (c)
component arrives (c)

A test suite

TC1	0	2	4	1	0						
TC2	0	5	2	4	5	6	0				
TC3	0	3	5	9	6	0					
TC4	0	3	5	7	5	8	7	8	9	6	0

Meaning: From state 0 to state 2 to state 4 to state 1 to state 0

Is this a thorough test suite?
How can we judge?


0
NO Maintenance
1
Wait for returning
2
Maintenance (no warranty)
3
Wait for pick up
4
Wait for acceptance
5
Repair (maintenance station)
6
Repaired
7
Wait for component
8
Repair (regional headquarters)
9
Repair (main headquarters)
pick up
request at maintenance station (no warranty)
request at maintenance station or by express courier (contract number)
request by phone or web (US or EU resident) (contract number)
return
reject estimate
estimate costs
invalid contract number
accept estimate
pick up
repair completed
lack component (a)
component arrives (a)
unable to repair (US or EU resident)
successful repair
lack component (b)
component arrives (b)
unable to repair
lack component (c)
component arrives (c)
successful repair
unable to repair (not US or EU resident)

“Covering“ finite state machines

- State coverage:
  - Every state in the model should be visited by at least one test case
- Transition coverage
  - Every transition between states should be traversed by at least one test case.
  - This is the most commonly used criterion
    - A transition can be thought of as a (precondition, postcondition) pair

Path sensitive criteria?

- Basic assumption: States fully summarize history
    - No distinction based on how we reached a state; this should be true of well-designed state machine models
- If the assumption is violated, we may distinguish paths and devise criteria to cover them
    - Single state path coverage:
        - traverse each subpath that reaches each state at most once
    - Single transition path coverage:
        - “” “” each transition at most once
    - Boundary interior loop coverage:
        - each distinct loop of the state machine must be exercised the minimum, an intermediate, and the maximum or a large number of times

Integration testing

	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

Integration Faults

- Inconsistent interpretation of parameters or values
  - Example: Mixed units (meters/yards) 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
- Omitted or misunderstood functionality
  - Example: Inconsistent interpretation of web hits
- 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;
}
```

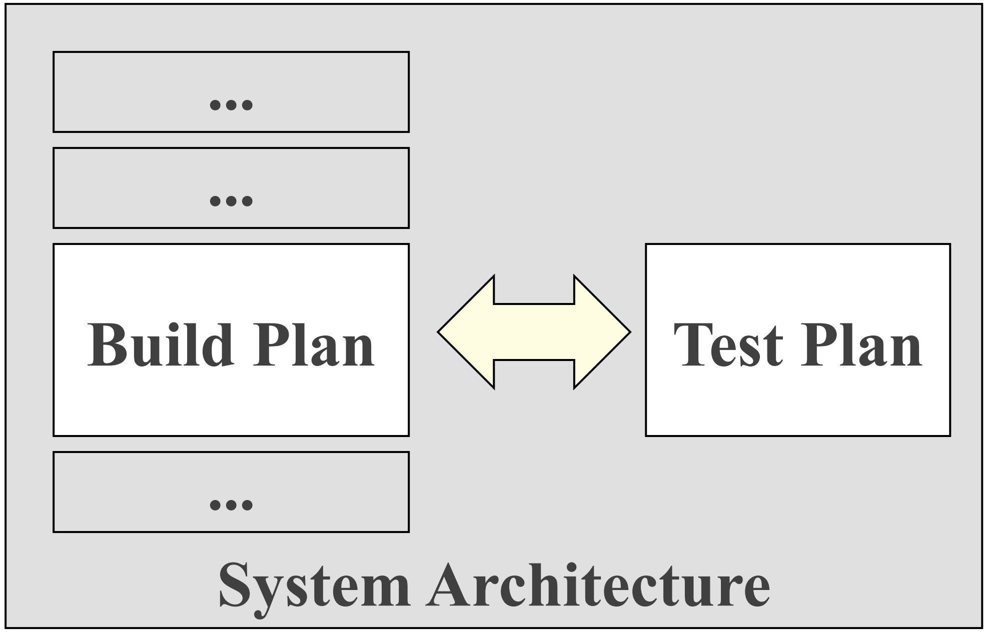
Almost impossible to find with unit testing. (Inspection and some dynamic techniques could have found it.)

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

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.

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

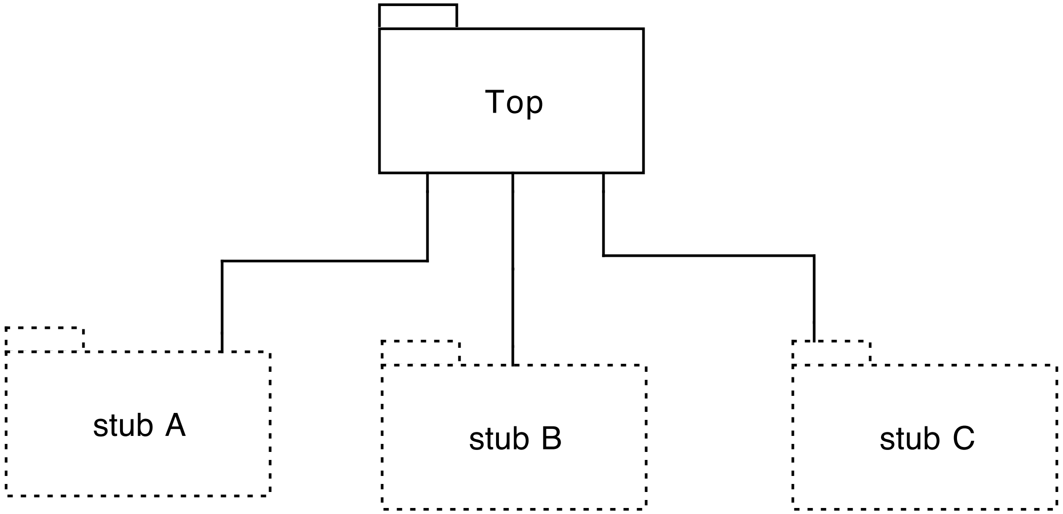
Big Bang Integration Test

- An extreme and desperate approach: Test only after integrating all modules
  - Does not require scaffolding
    - The only excuse, and a bad one
  - Minimum observability, diagnosability, efficacy, feedback
  - High cost of repair
    - Recall: Cost of repairing a fault rises as a function of time between error and repair

Structural and Functional Strategies

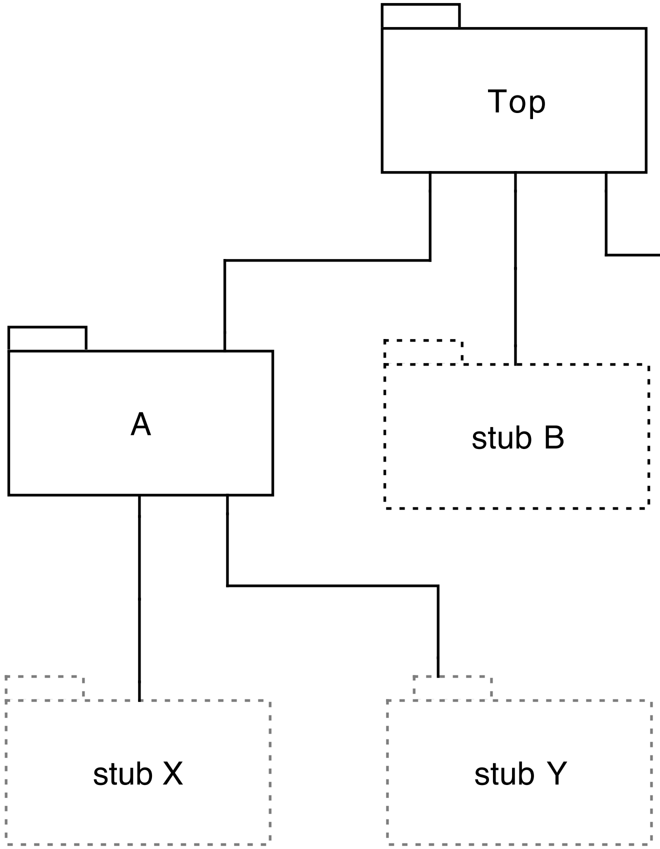
- Structural orientation:
- Modules constructed, integrated and tested based on a hierarchical project structure
  - Top-down, Bottom-up, Sandwich, Backbone
- Functional orientation:
  Modules integrated according to application characteristics or features
  - Threads, Critical module

Top down .



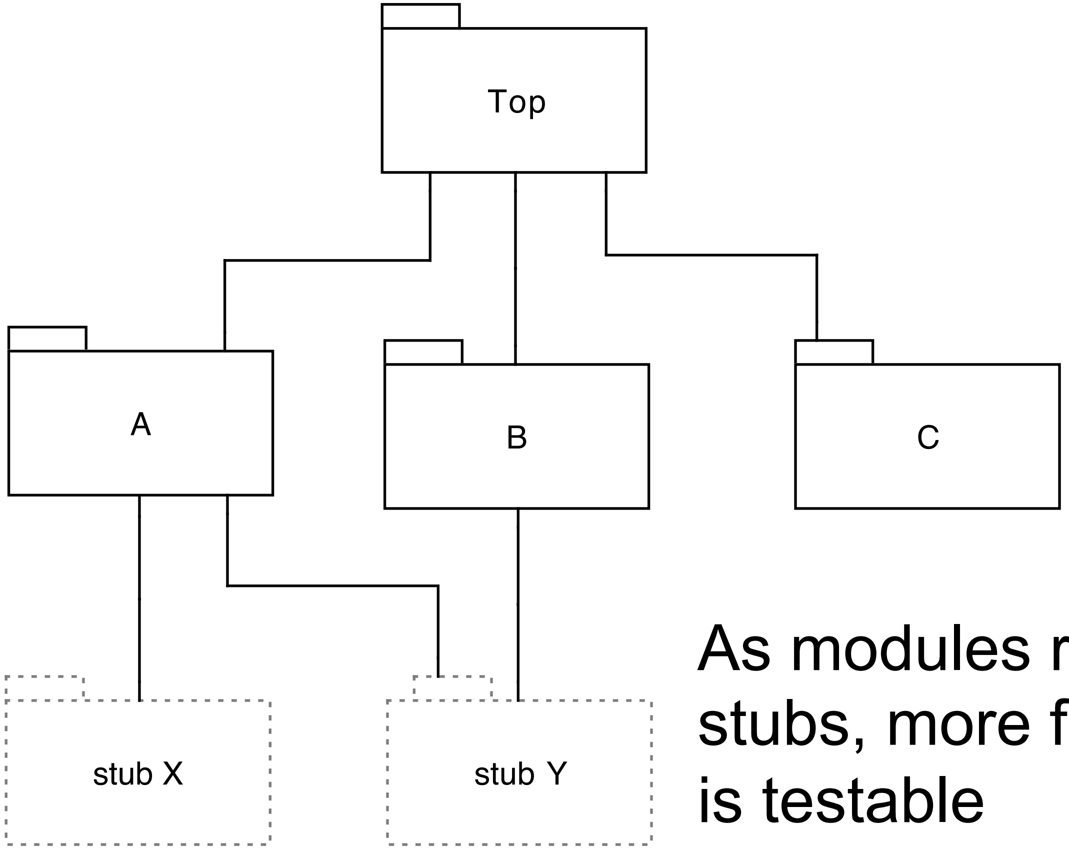
Working from the top level (in terms of “use” or “include” relation) toward the bottom.
No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.)

Top down ..



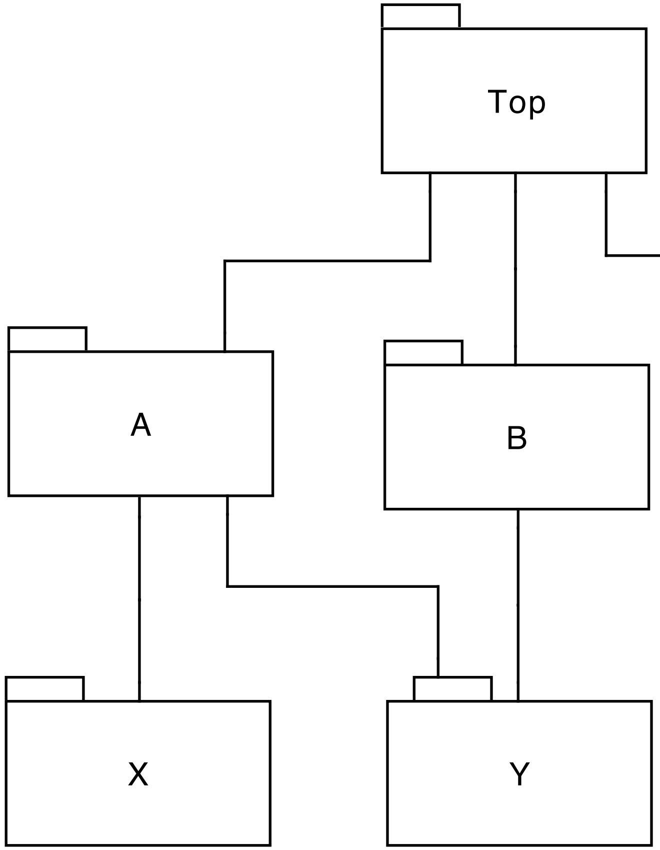
Write stubs of called or used modules at each step in construction

Top down ...



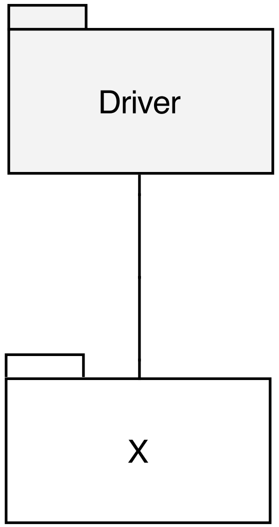
As modules replace stubs, more functionality is testable

Top down ... complete



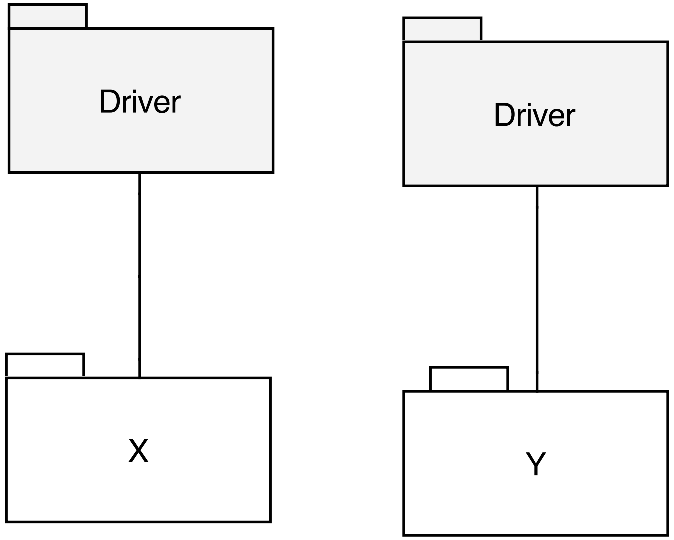
... until the program is complete, and all functionality can be tested

Bottom Up .



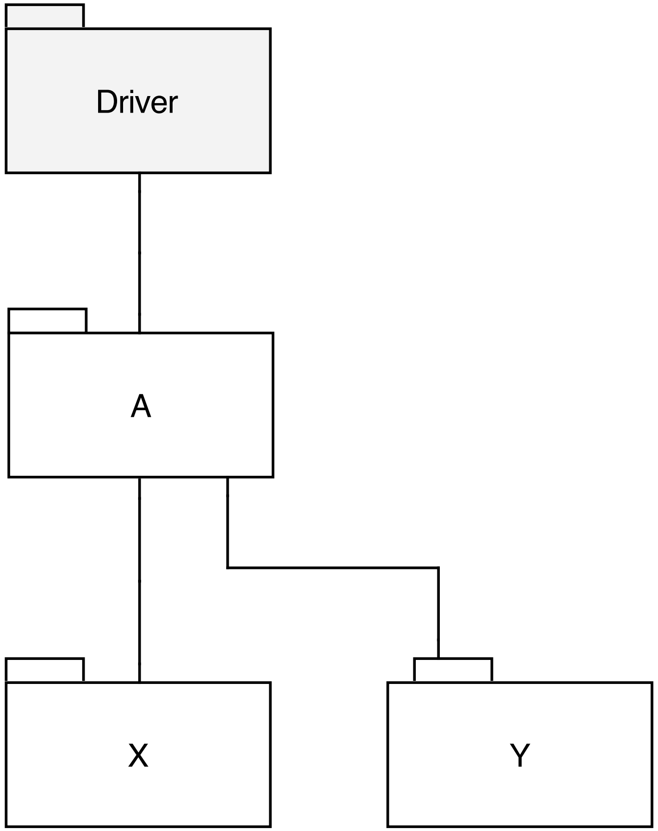
Starting at the leaves of the "uses" hierarchy, we never need stubs

Bottom Up ..



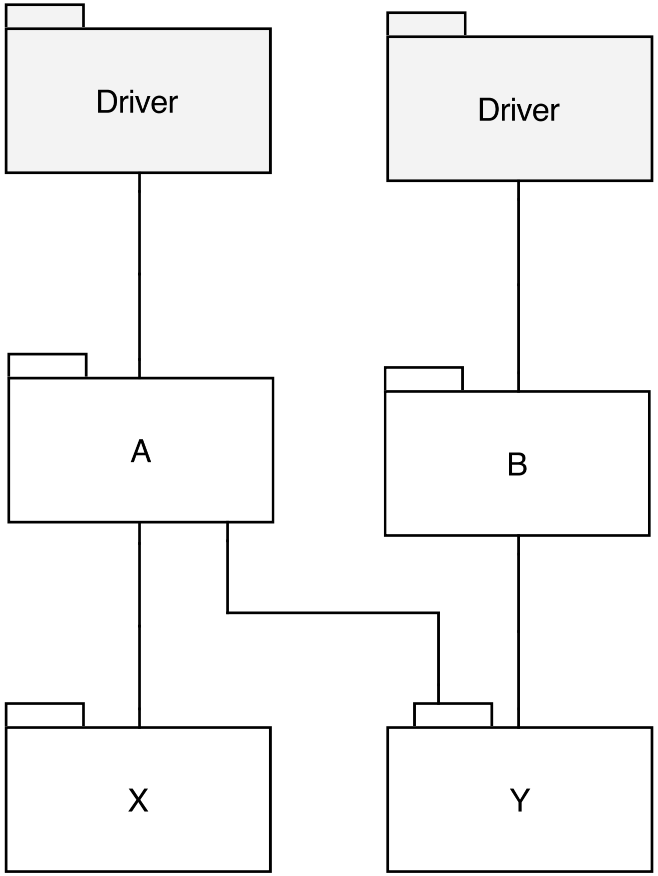
... but we must construct drivers for each module (as in unit testing) ...

Bottom Up ...

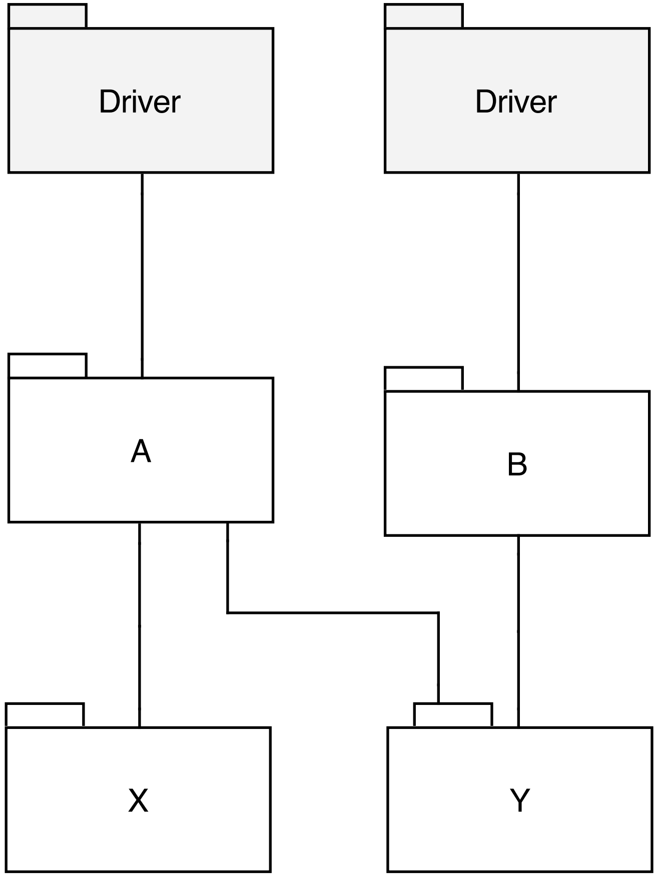


... an intermediate module replaces a driver, and needs its own driver ...

Bottom Up ....

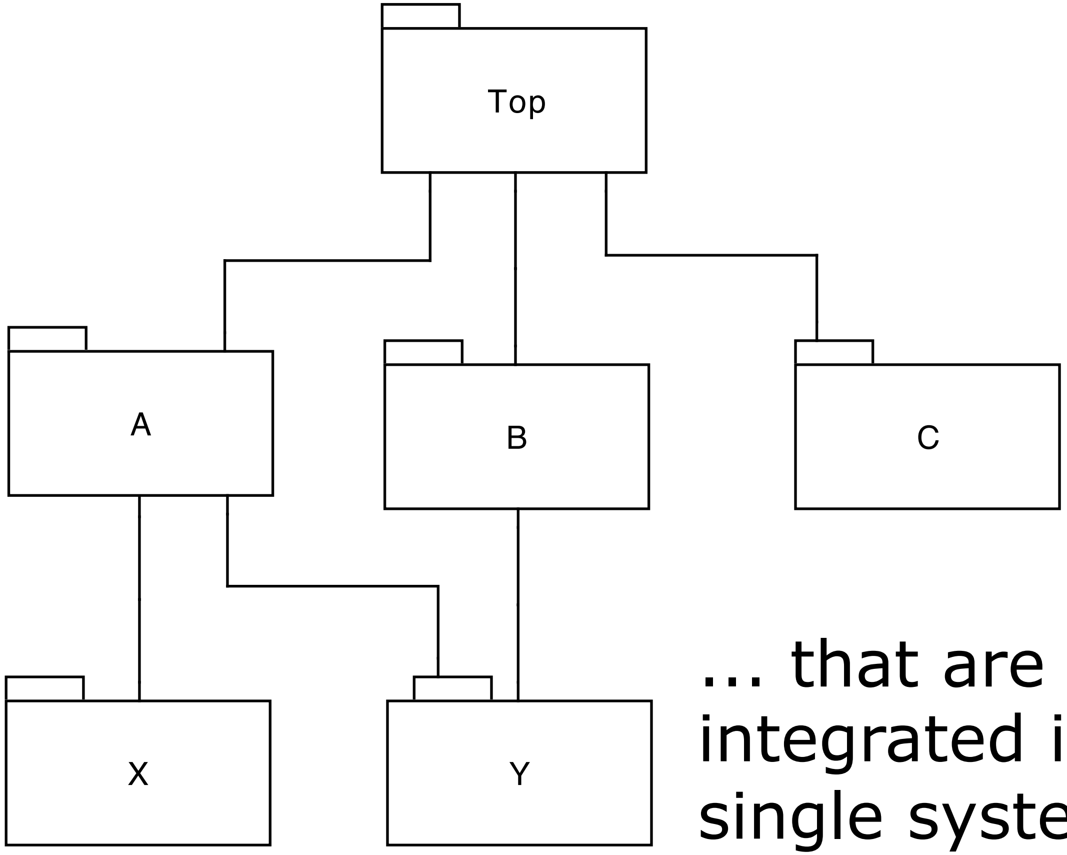


Bottom Up .....



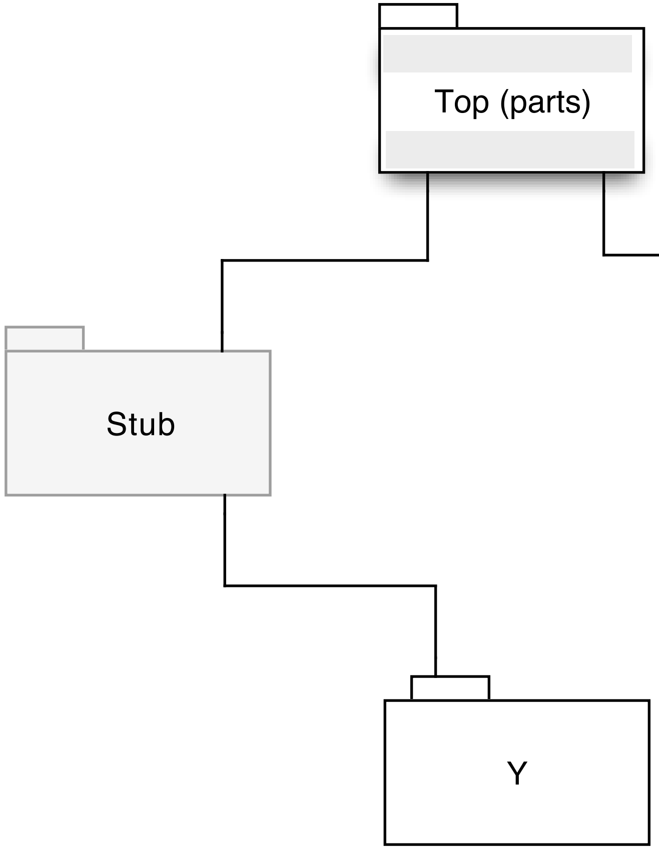
... so we may have several working subsystems ...

Bottom Up (complete)



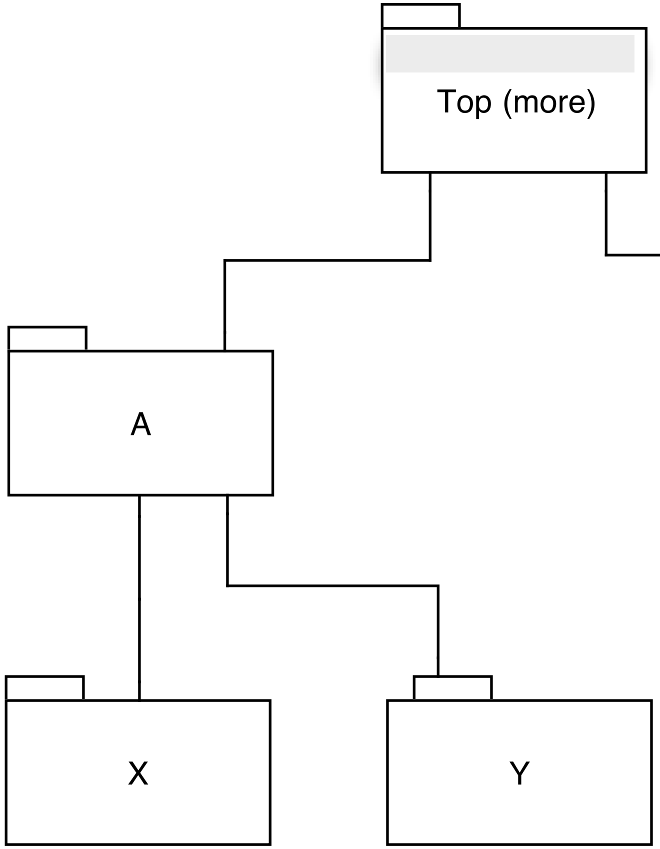
... that are eventually integrated into a single system.

Sandwich .



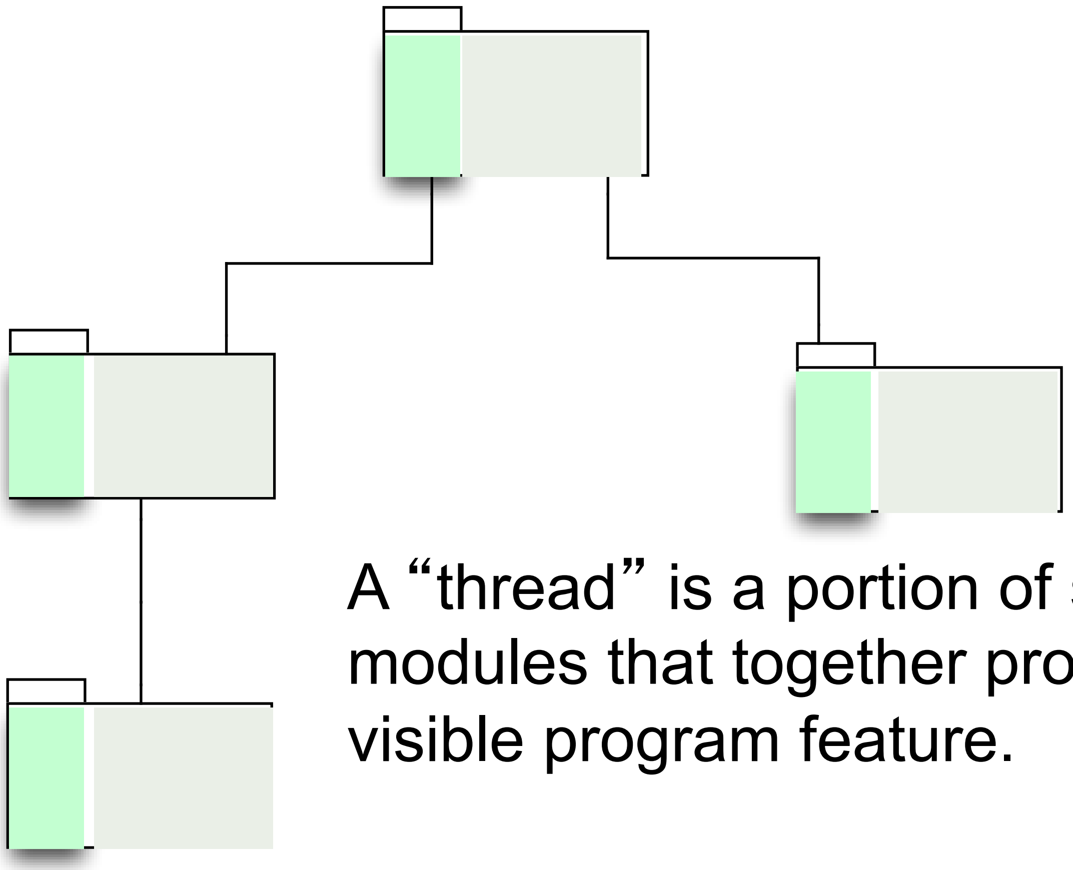
Working from the extremes (top and bottom) toward center, we may use fewer drivers and stubs

Sandwich ..



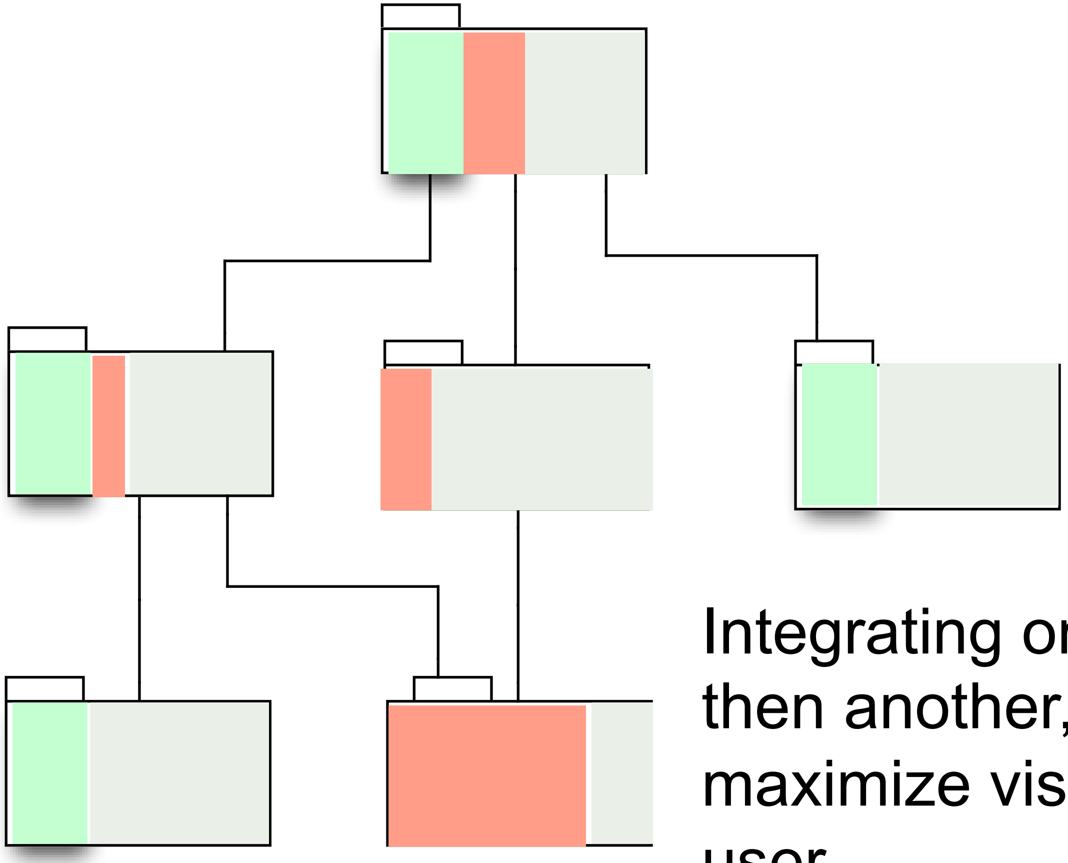
Sandwich integration is flexible and adaptable, but complex to plan

Thread ...



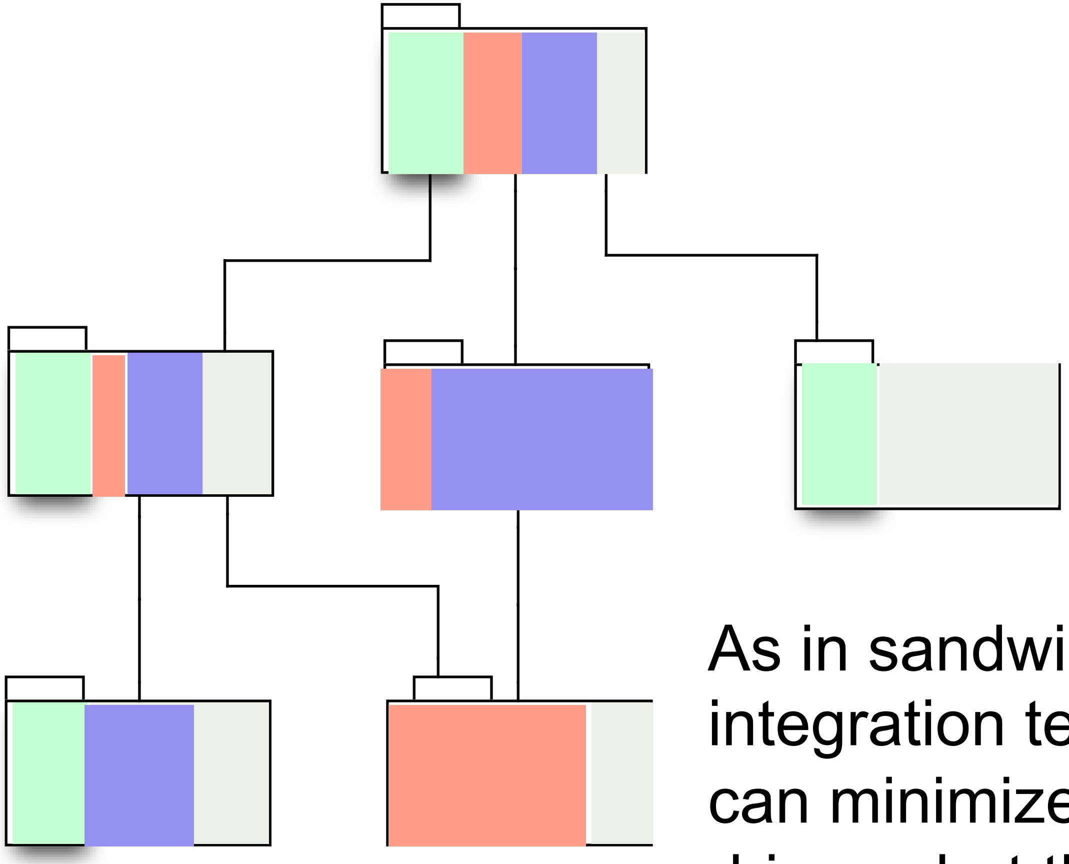
A "thread" is a portion of several modules that together provide a user-visible program feature.

Thread …



Integrating one thread, then another, etc., we maximize visibility for the user

Thread …



As in sandwich integration testing, we can minimize stubs and drivers, but the integration plan may be complex

Critical Modules

- Strategy: Start with riskiest modules
  - Risk assessment is necessary first step
  - May include technical risks (is X feasible?), process risks (is schedule for X realistic?), other risks
- May resemble thread or sandwich process in tactics for flexible build order
  - E.g., constructing parts of one module to test functionality in another
- Key point is risk-oriented process
  - Integration testing as a risk-reduction activity, designed to deliver any bad news as early as possible

Choosing a Strategy

- Functional strategies require more planning
  - Structural strategies (bottom up, top down, sandwich) are simpler
  - But thread and critical modules testing provide better process visibility, especially in complex systems
- Possible to combine
  - Top-down, bottom-up, or sandwich are reasonable for relatively small components and subsystems
  - Combinations of thread and critical modules integration testing are often preferred for larger subsystems