Testing

Testing

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

Definitions

- A 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?

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

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

Black-box vs white-box systematic testing

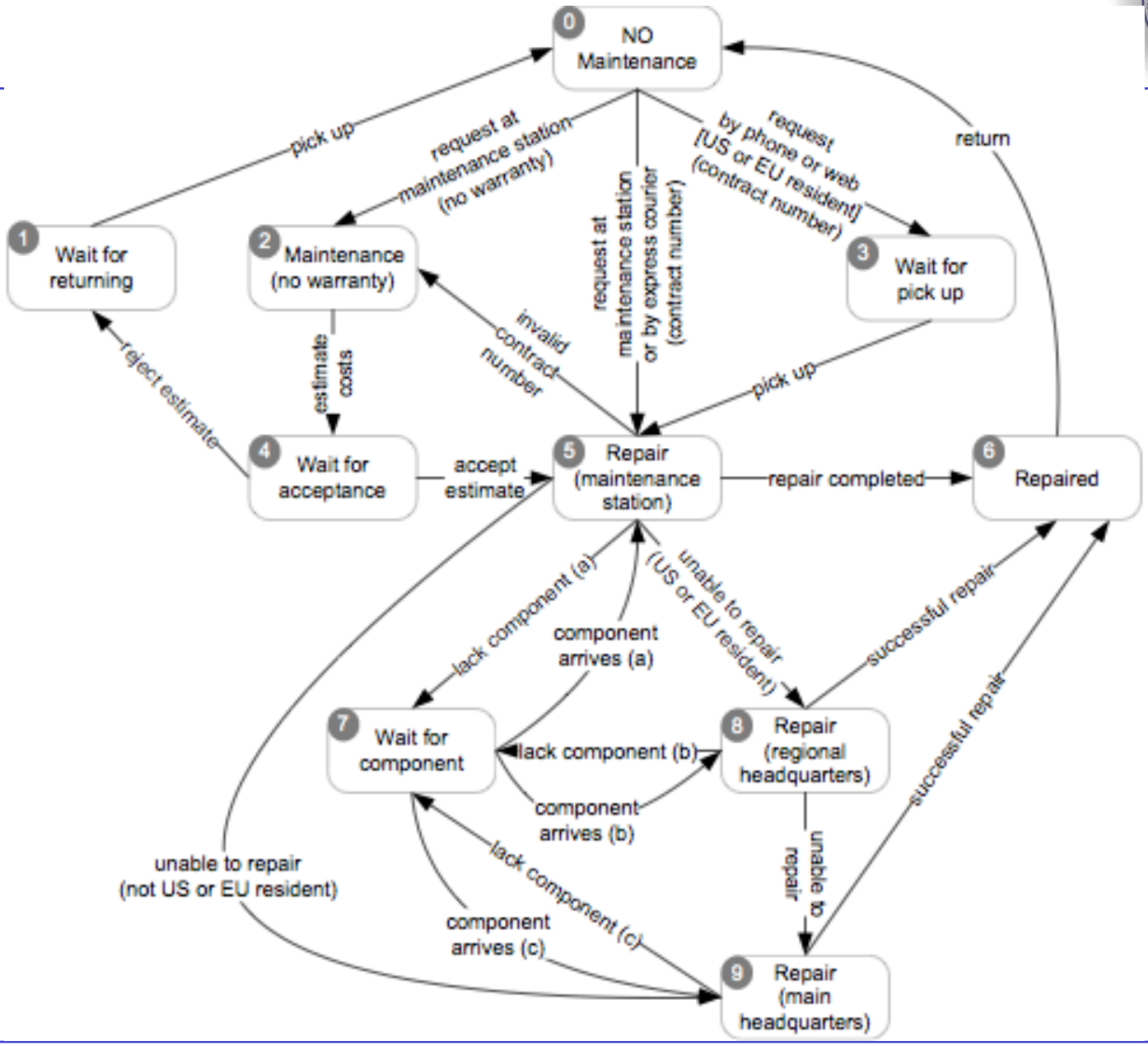
- White-box testing is suitable for unit testing
  - Covering a small portion of software is possible
- Black-box testing is suitable for integration, system and acceptance testing
  - Specs usually smaller than code
  - Specs help identifying missing functionalities in the system

An example of black-box testing: 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

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

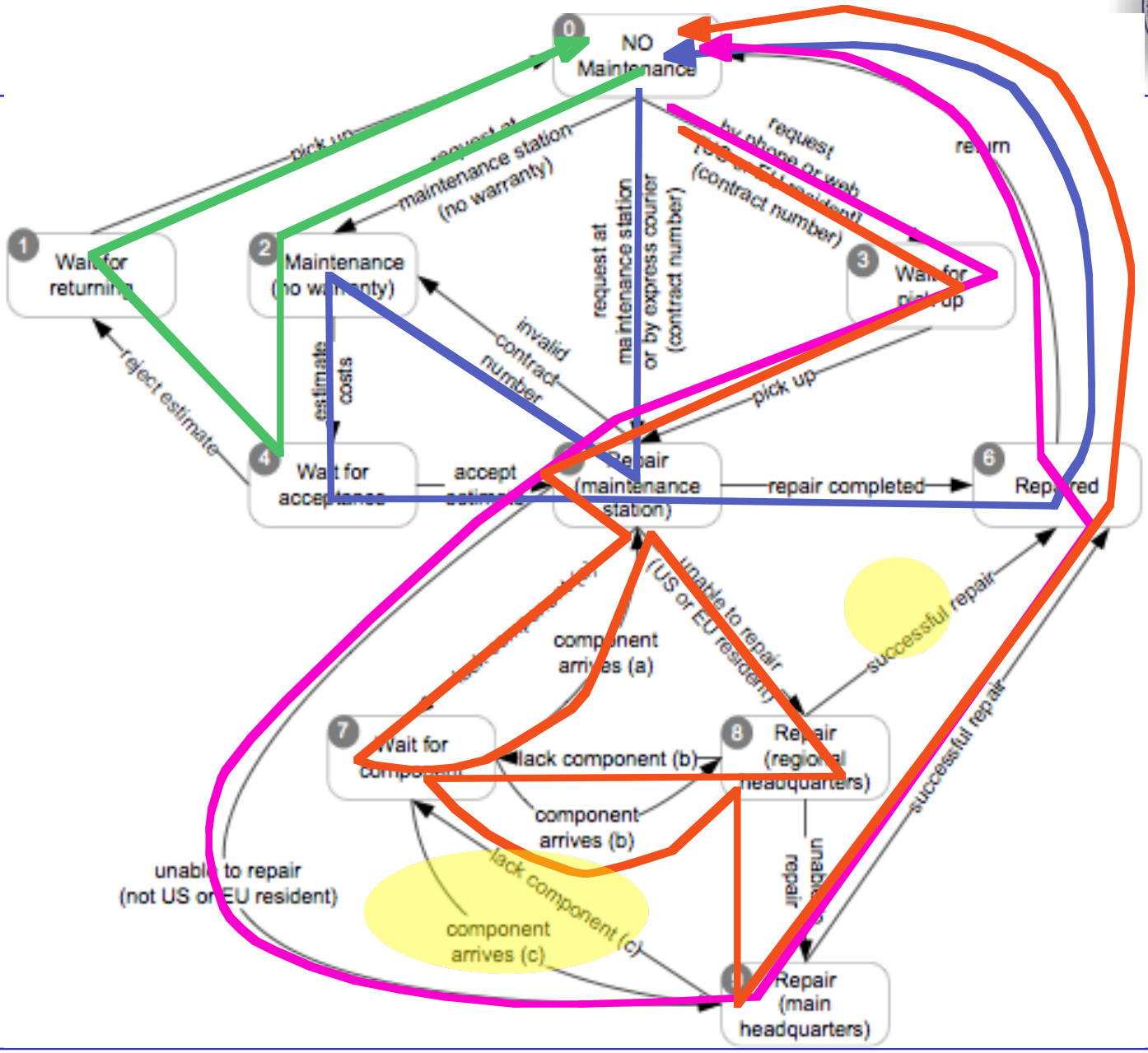

0 NO Maintenance
1 Wait for returning
2 Maintenance (no warranty)
3 Wait for pick up
4 Wait for acceptance
5 Repair (maintenance station)
6 Repaired
7 Wait for component
8 Repair (regional headquarters)
9 Repair (main headquarters)
pick up
request at maintenance station (no warranty)
request by phone or web [US or EU resident] (contract number)
return
request at maintenance station or by express courier (contract number)
reject estimate
estimate costs
invalid contract number
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)
unable to repair
successful 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

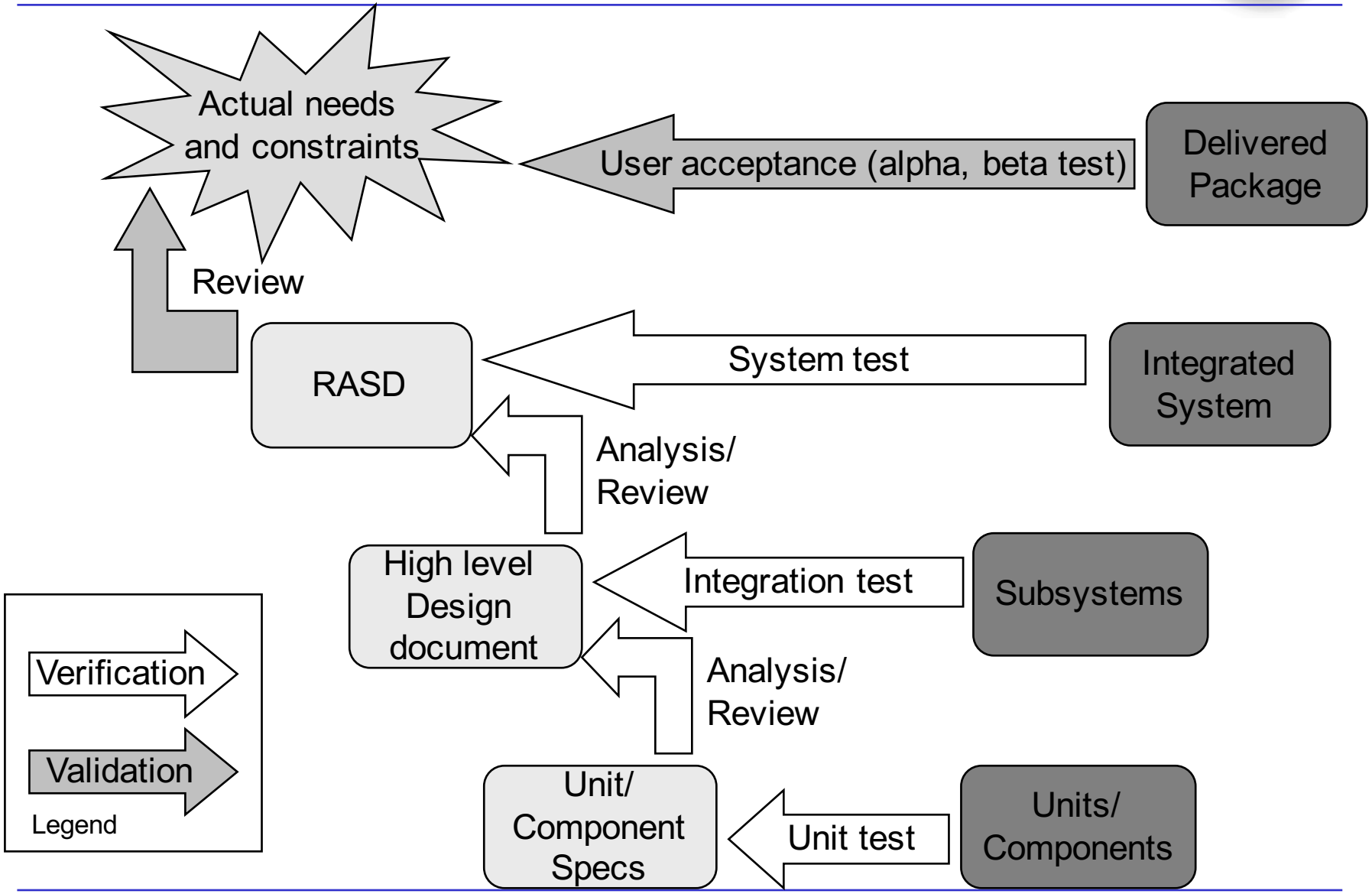
Is this a thorough test suite?
How can we judge?


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
by phone or web
(contract number)
return
request at
maintenance station
or by express courier
(contract number)
invalid
contract
number
estimate
costs
reject estimate
accept
repair completed
component
arrives (a)
unable to repair
(US or EU resident)
successful repair
lack component (b)
component
arrives (b)
unable to repair
(not US or EU resident)
lack component (c)
component
arrives (c)
repair
successful repair

"Covering" the state diagram

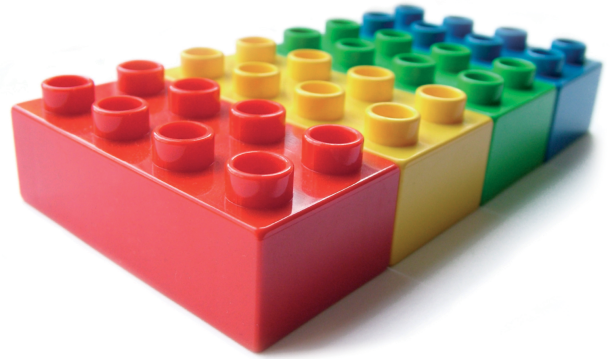
- State coverage:
  - Every state in the model should be visited by at least one test case
- Transition coverage
  - Every transition between states should be traversed by at least one test case.
  - This is the most commonly used criterion
    - A transition can be thought of as a (precondition, postcondition)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

V&V activities and software artifacts (the V model)



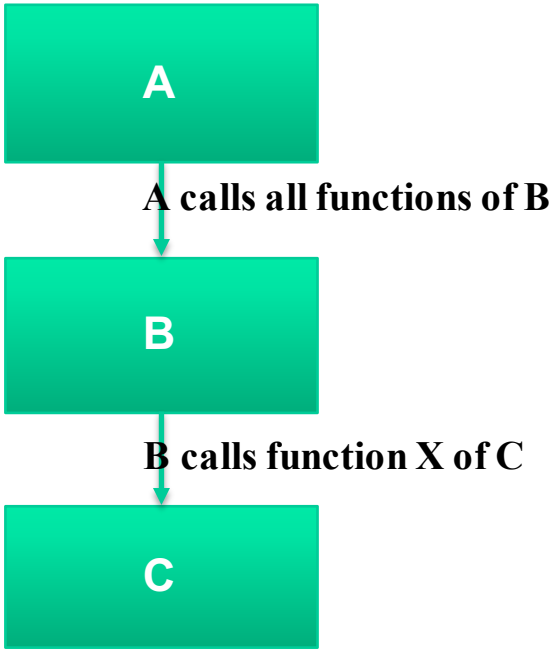
Unit Testing

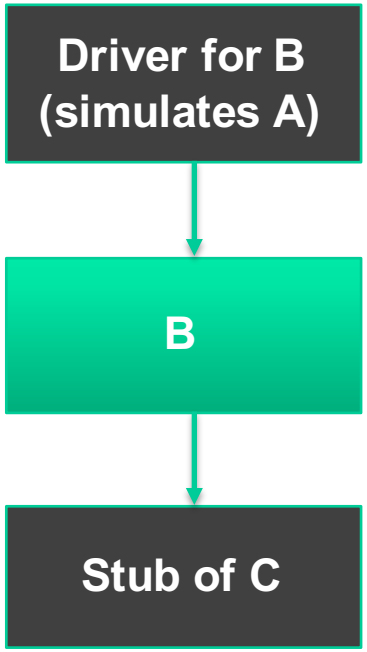
- Conducted by the same developers
- Aimed at testing sections of code
- Why unit testing?
  - Find problems early
  - Facilitates changes
  - Guides the design
  - Visual feedback



Unit testing and scaffolding

- The problem: components may not work in isolation
- Thus, we need to simulate components that are missing.
  - Example: B is used by A and uses C. We want to unit test B





Oracle: checks that, after test execution, the system is in a correct state and the output is correct

Integration Testing

- Aimed at exercising
  - interfaces
  - modules interactions
- How
  - Incrementally (facilitates bug tracking)
  - Big bang (better to avoid it)

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: Remember the 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;
}
```

Impossible to find it with unit testing.
Inspection and some dynamic techniques can find it.

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

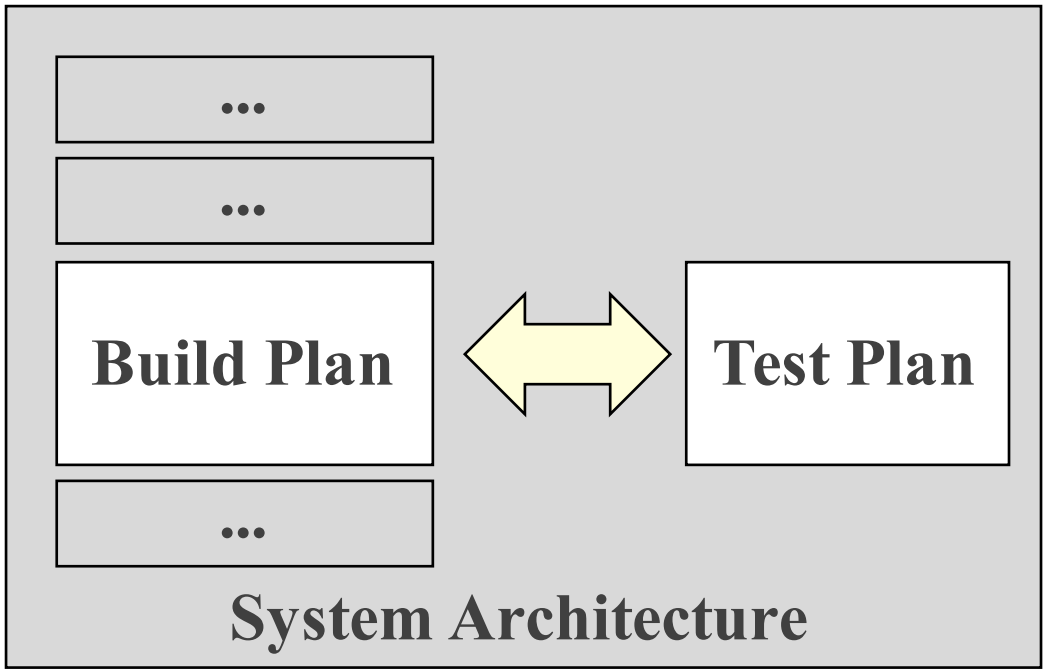
Example: Remember the memory leak?

- 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;
}
```

If the interface of the module where f is defined offers SSL_free and clearly specifies how it should be used, the problem could be found during integration testing

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

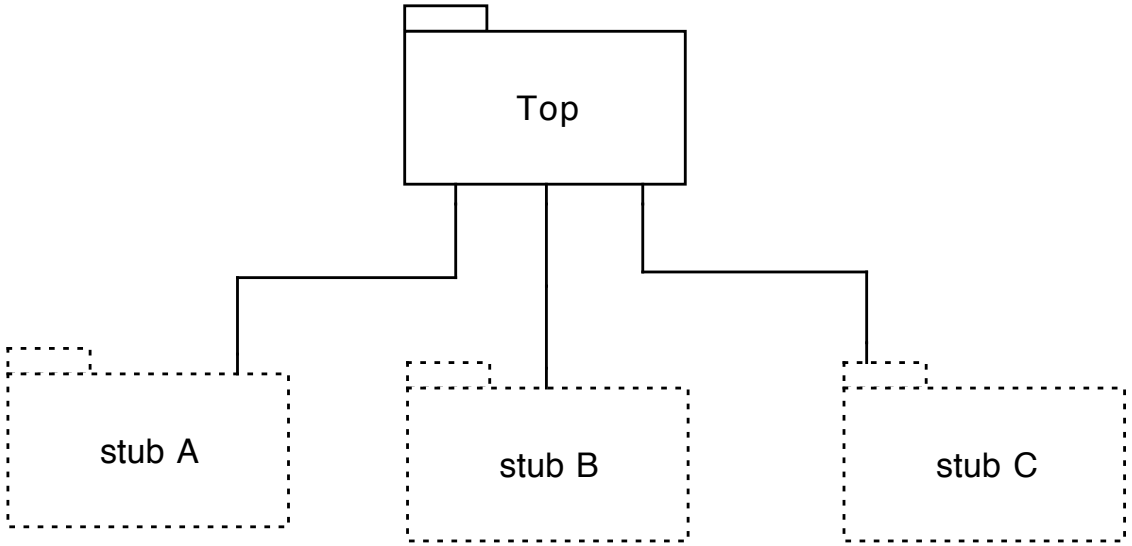
Big Bang Integration Test

- An extreme and desperate approach: Test only after integrating all modules
  - Does not require stubs/drivers/oracles
    - 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

Incremental integration test

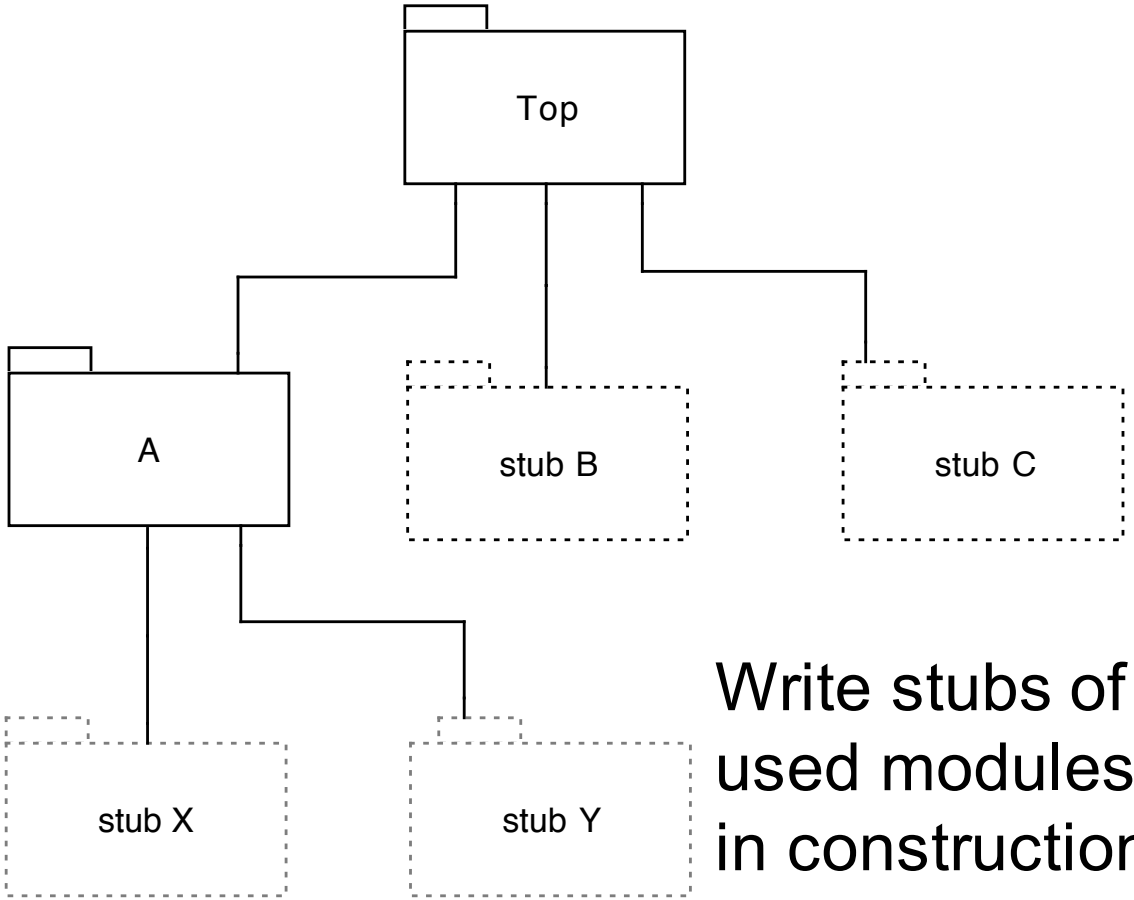
- It occurs while components are released
- E.g., as soon as a first version of components A and B is released, we integrate them and test the integration
- Then, if tests pass, we integrate also the first version of C that has been released in the meanwhile
  - And we test such new integration
- Typical strategies for incremental integration and testing
  - Based on the hierarchical structure of the system
    - Top down
    - Bottom up

Top down .



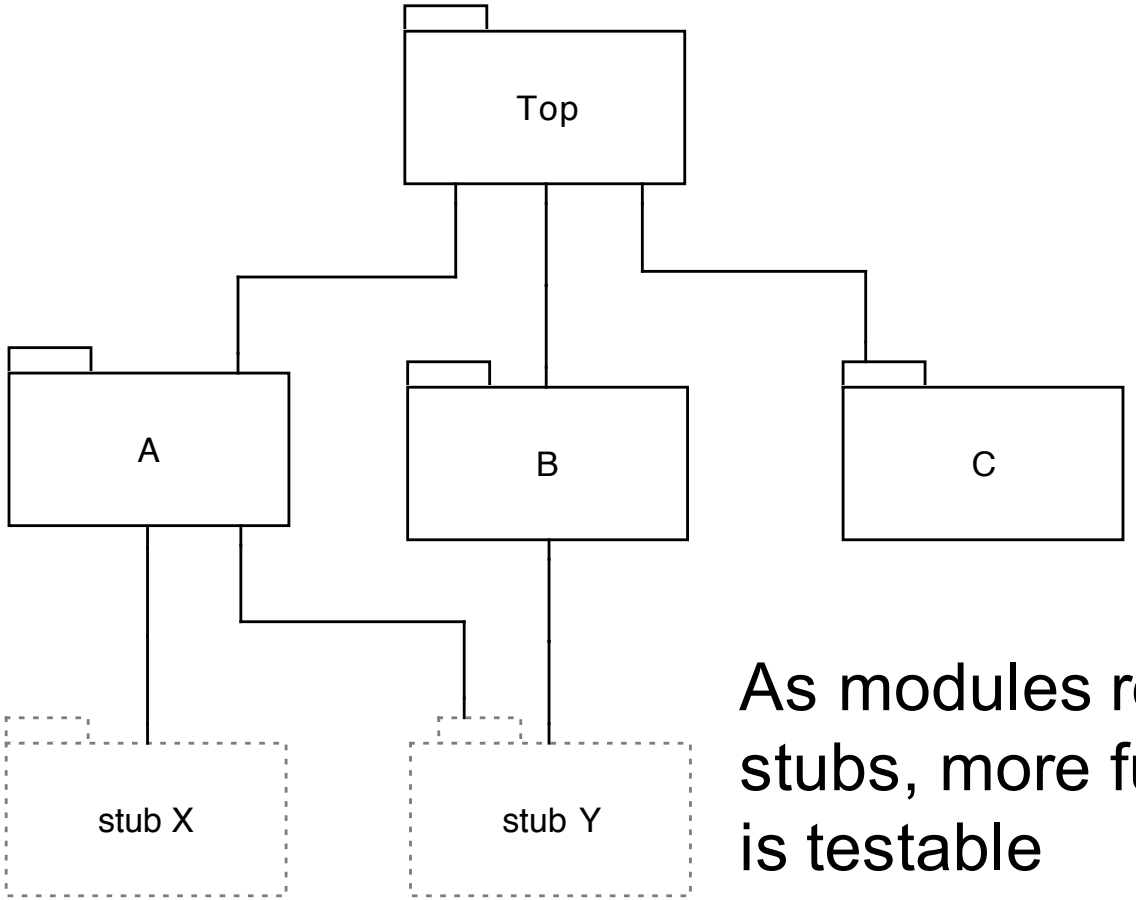
Working from the top level (in terms of “use” or “include” relation) toward the bottom.
No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.)

Top down ..



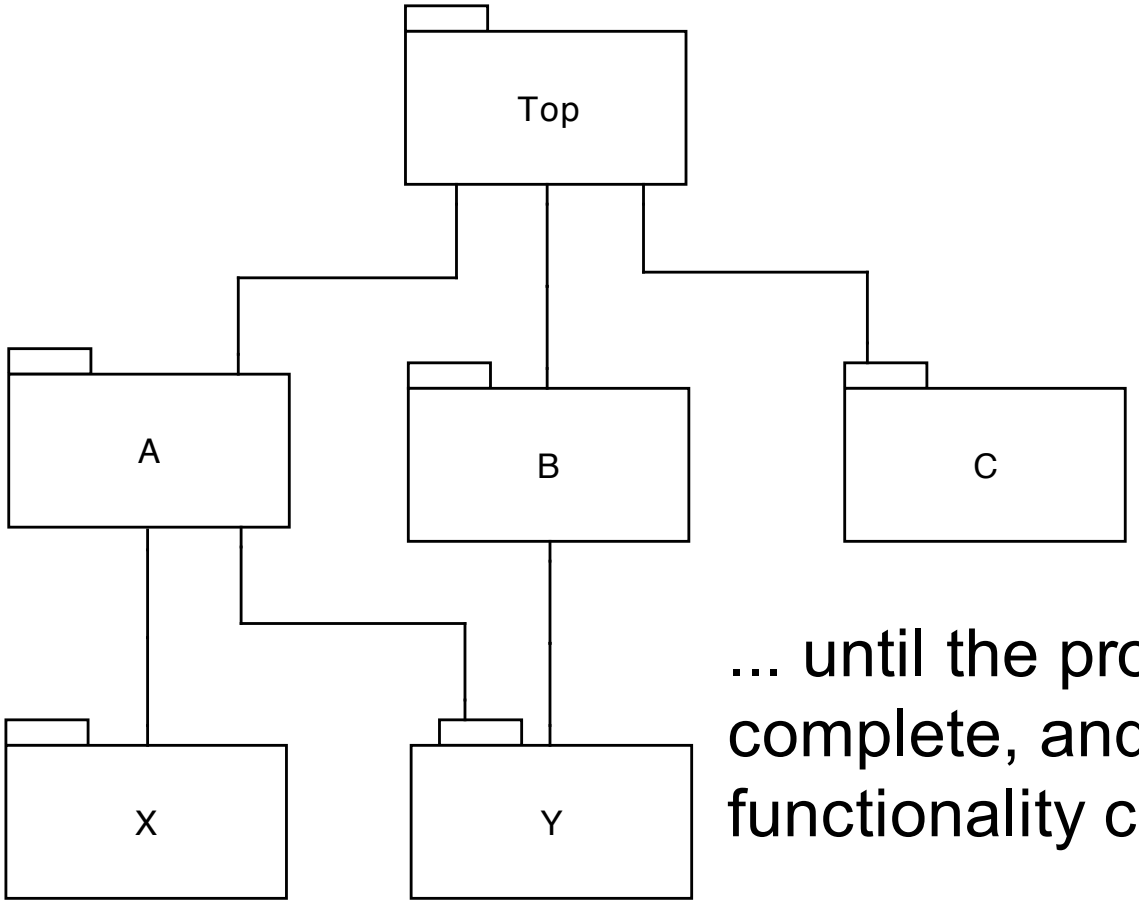
Write stubs of called or used modules at each step in construction

Top down …



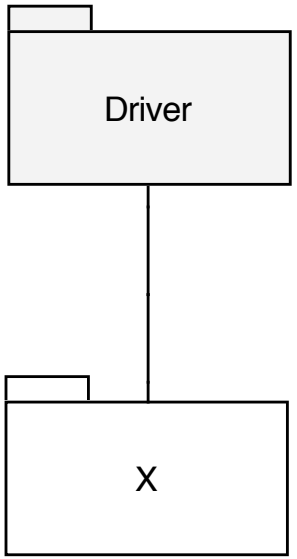
As modules replace stubs, more functionality is testable

Top down … complete



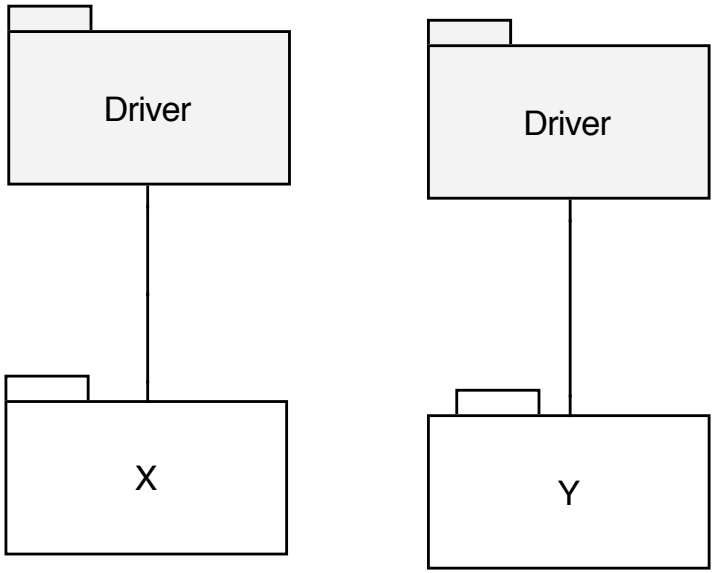
… until the program is complete, and all functionality can be tested

Bottom Up .



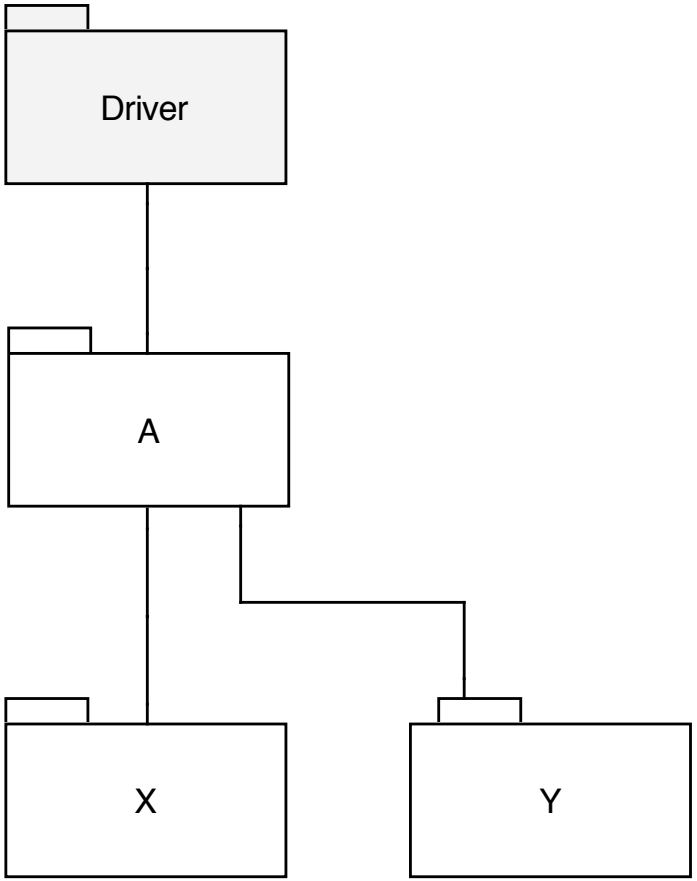
Starting at the leaves of the "uses" hierarchy, we never need stubs

Bottom Up ..



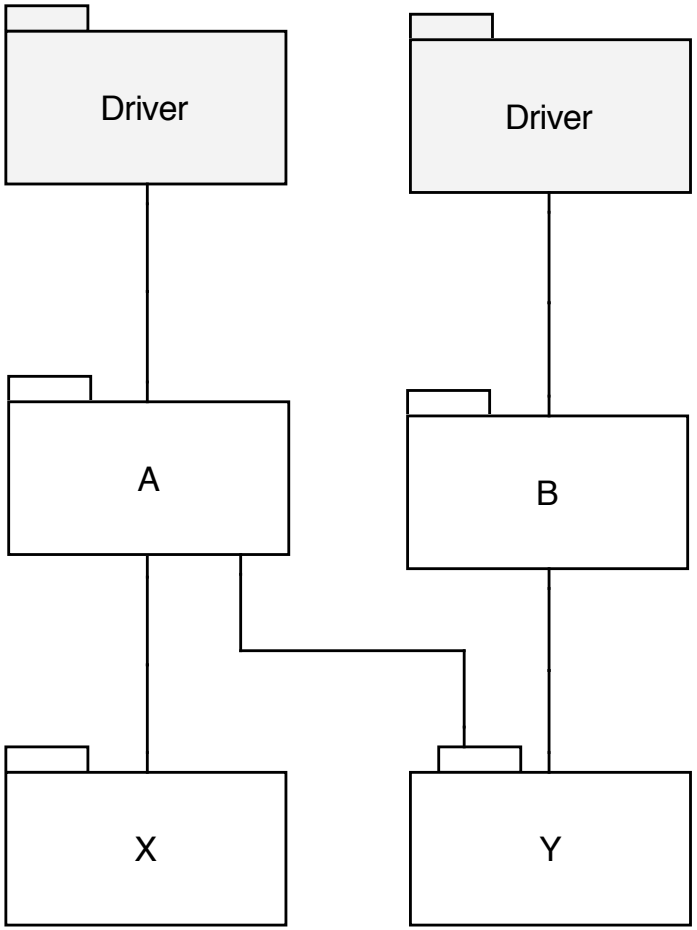
... but we must construct drivers for each module (as in unit testing) ...

Bottom Up …

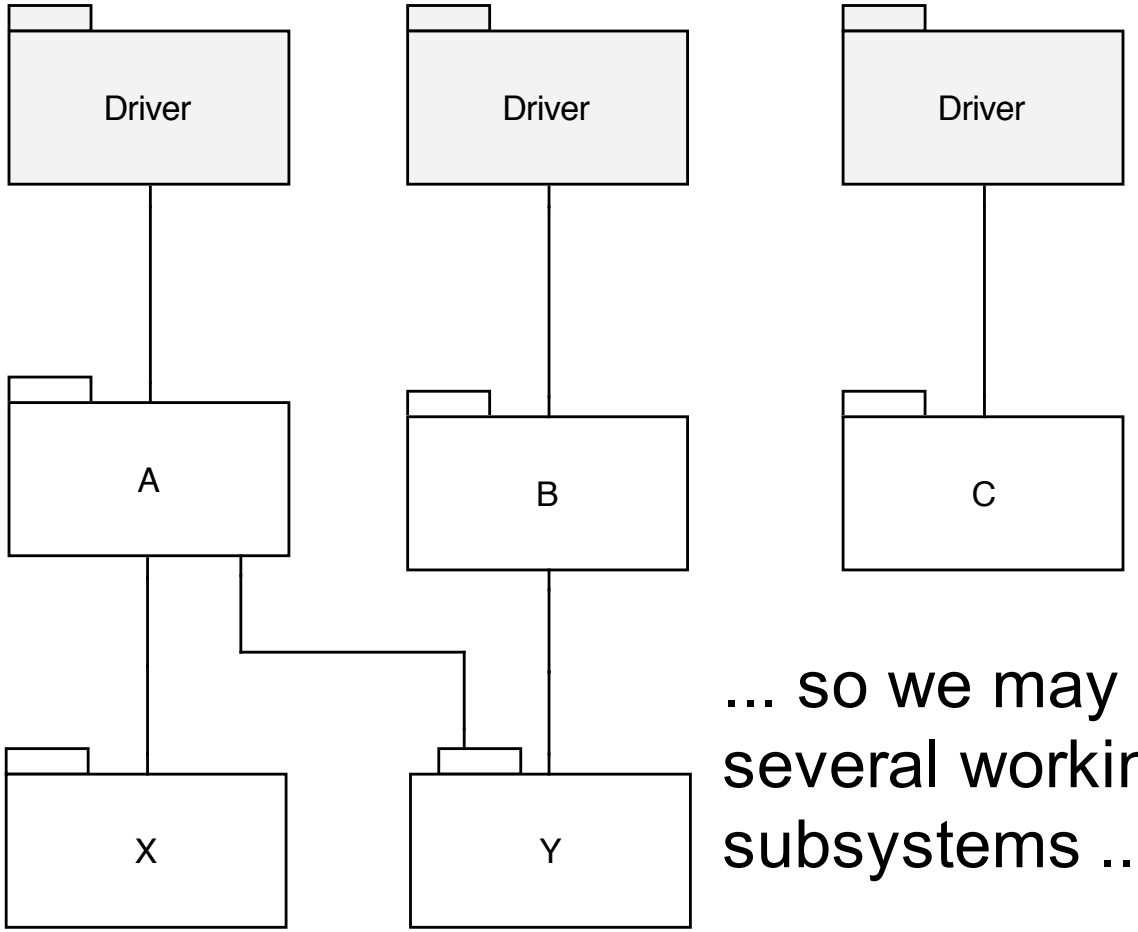


… an intermediate module replaces a driver, and needs its own driver …

Bottom Up ....

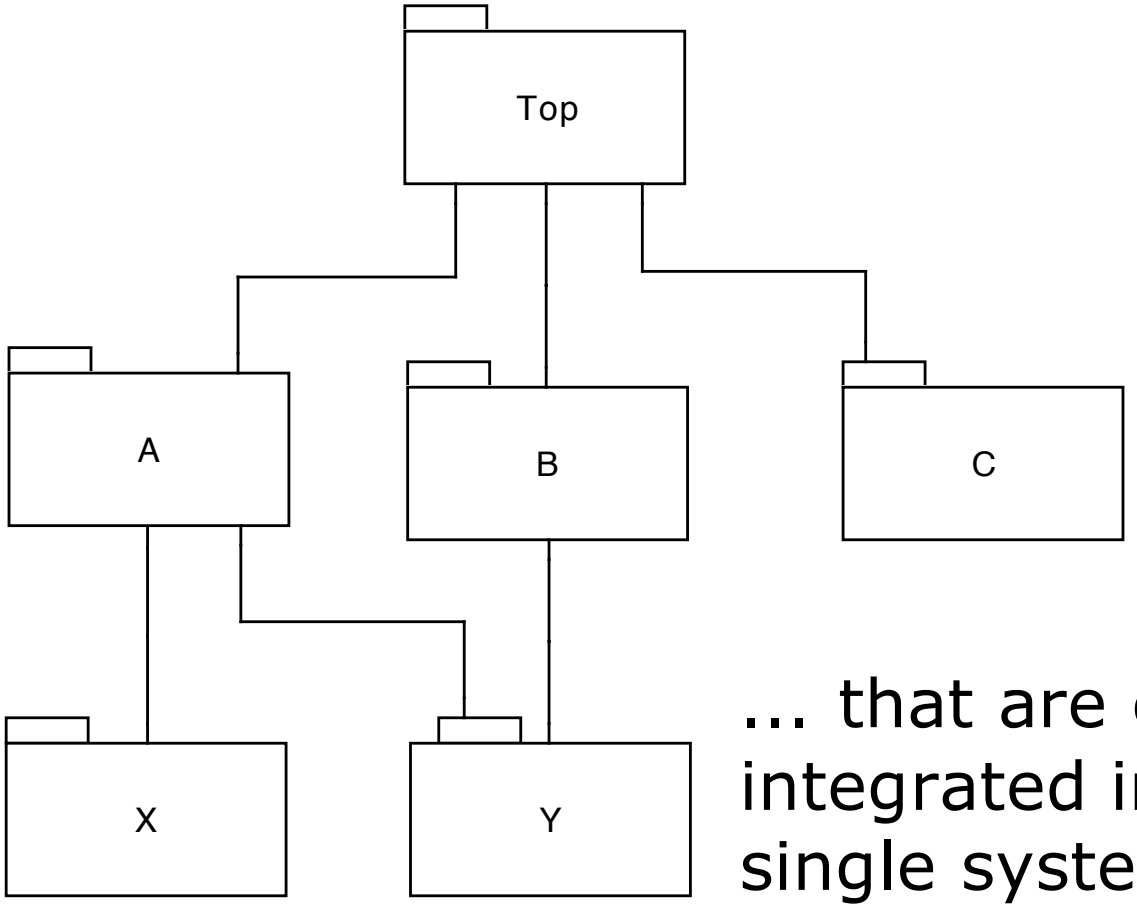


Bottom Up .....



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

Bottom Up (complete)

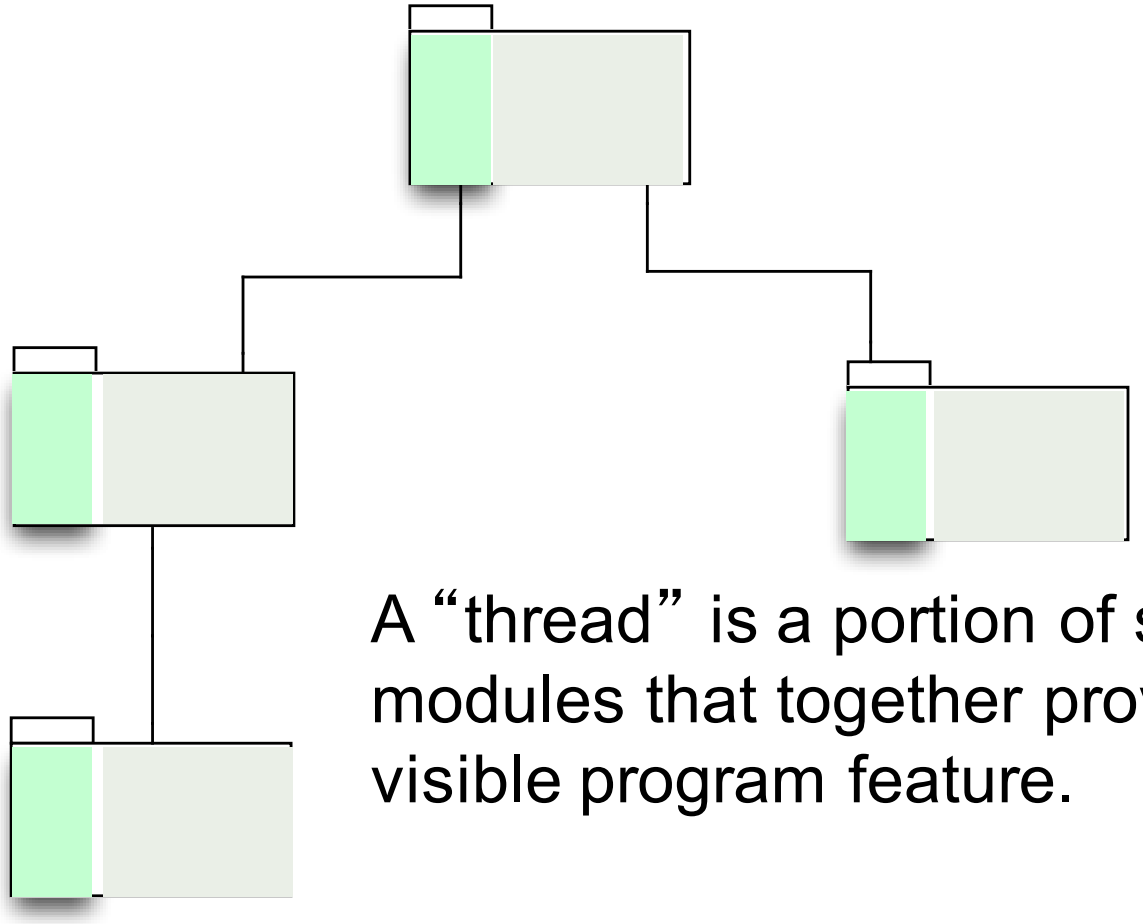


... that are eventually integrated into a single system.

Other integration and testing strategies

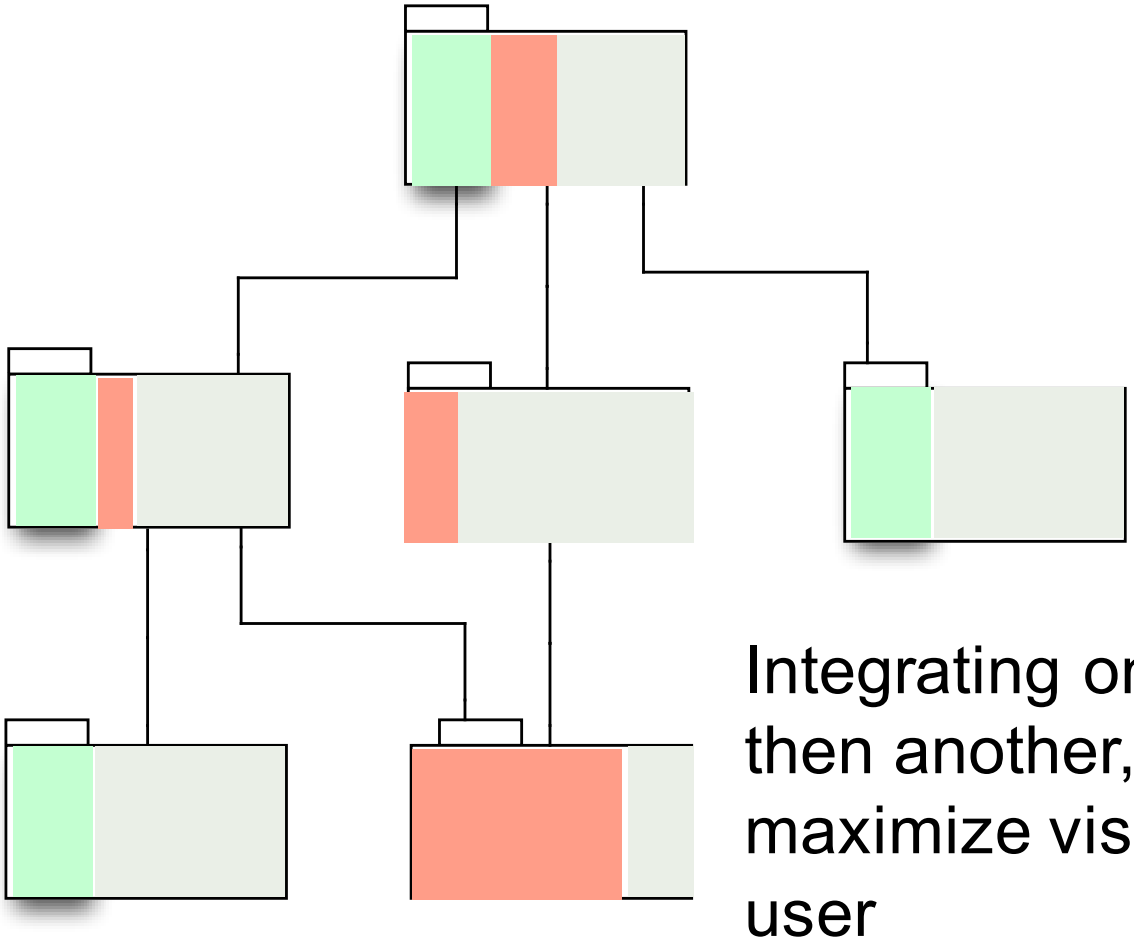
- Threads: a portion of several modules that offers a user-visible function
- Critical modules

Thread ...



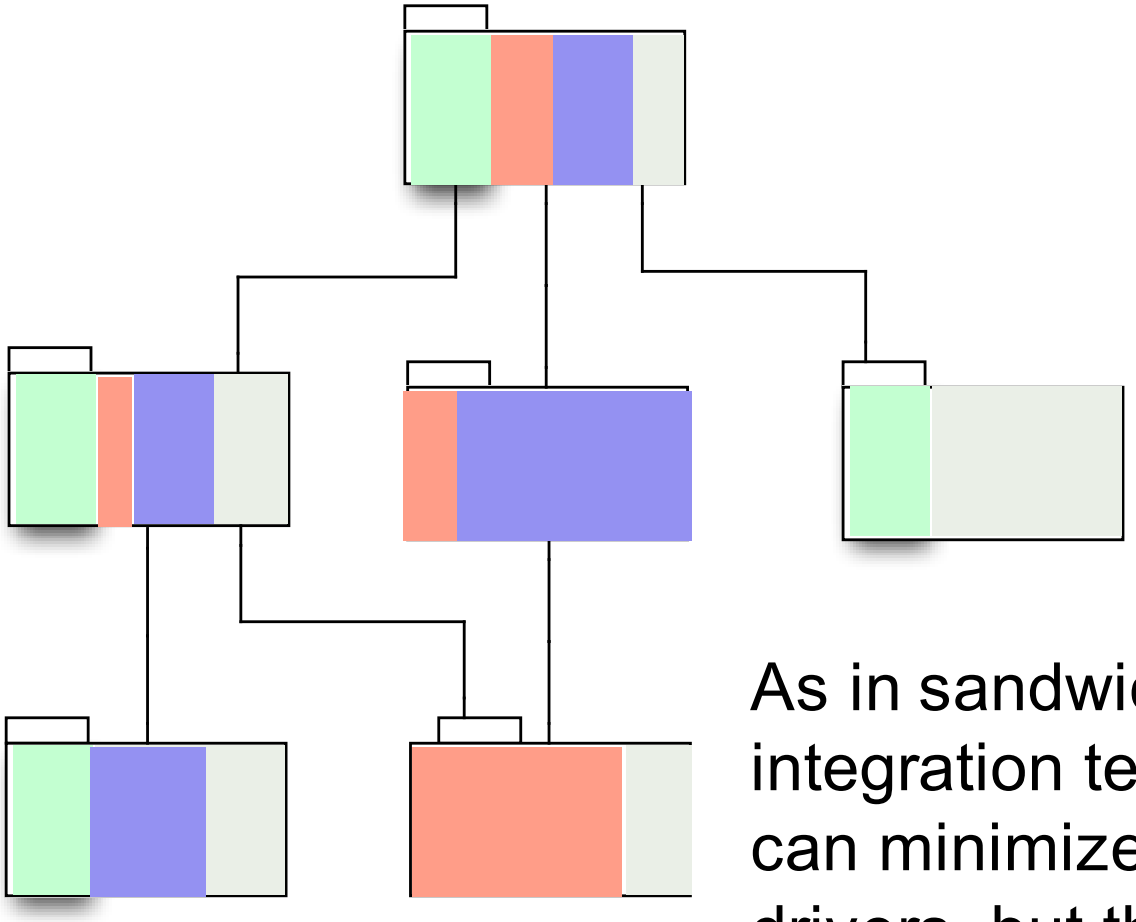
A “thread” is a portion of several modules that together provide a user-visible program feature.

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

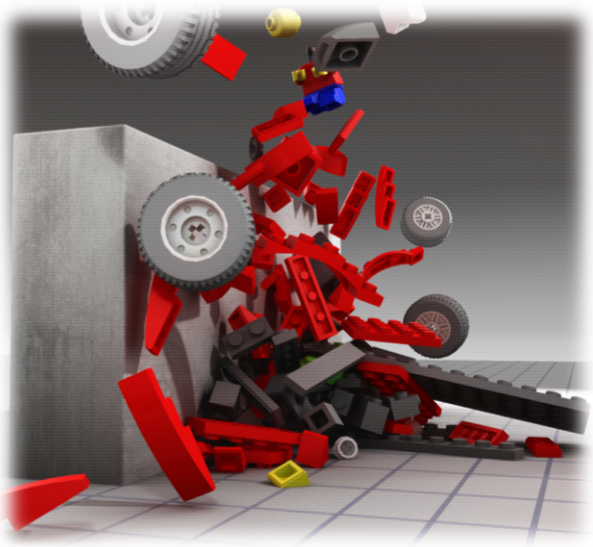
- Strategy: Start with riskiest modules
  - Risk assessment is necessary first step
  - May include technical risks (is X feasible?), process risks (is schedule for X realistic?), other risks
- May resemble thread or sandwich process in tactics for flexible build order
  - E.g., constructing parts of 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 and top down) are simpler
  - But thread and critical modules testing provide better process visibility, especially in complex systems
- Possible to combine
  - Top-down and bottom-up are reasonable for relatively small components and subsystems
  - Combinations of thread and critical modules integration testing are often preferred for larger subsystems

System Testing

- Conducted on a complete integrated system
- Independent teams (black box)
- Functional and non-functional requirements
- Testing environment should be as close as possible to production environment



Types of System Testing: Performance Testing

- Purpose
  - Identify bottlenecks affecting response time, utilization, throughput (whitebox)
  - Benchmarking, *i.e.*, establish a performance baseline and possibly compare it with different versions of the same product or a different competitive product (blackbox)
- Prerequisites
  - Expected workload
  - Acceptable performance
- Identifying
  - Inefficient algorithms
  - Query optimizations
  - Hardware/Network issues

Types of System Testing: Load Testing

- Purpose
  - Expose bugs such as memory leaks, mismanagement of memory, buffer overflows
  - Identify upper limits of components
- How
  - Increase the load until threshold
  - Load the system with the maximum load it can operate for a long period

Types of System Testing: Stress Testing

Purpose

- Make sure that the system recovers gracefully after failure

How

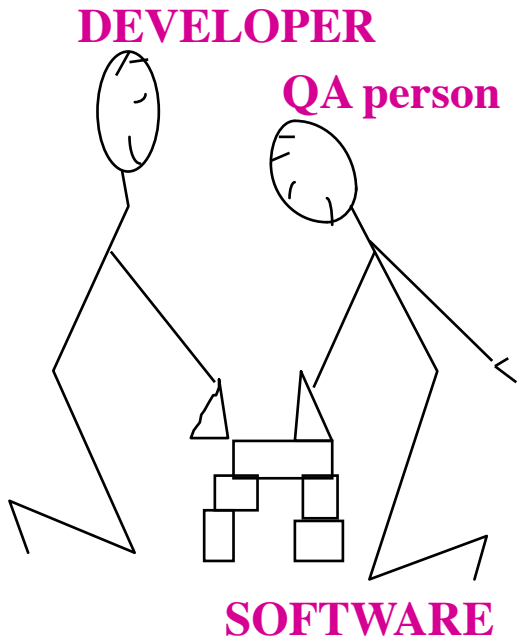
- Trying to break the system under test by overwhelming its resources or by taking resources away from it

Examples

- Double the baseline number for concurrent users/HTTP connections
- Randomly shut down and restart ports on the network switches/routers that connects servers

Conclusion

- Verification and validation take place in most phases of development process
- Many techniques available
- A lot of possibilities for automation
- We need to avoid the mood: yes, I'm great in my work so I do not need verification/validation
- We need to plan for the V&V process carefully
- We need to pay attention to the cohesion of the team



References

- C. Ghezzi, M. Jazayeri, D. Mandrioli *Fundamentals of Software Engineering*, 2nd Edition, Prentice Hall, 2003 (Italian translation Pearson Ed. 2004)
- M. Pezzè, M. Young *Software Testing and Analysis: Process, Principles, and Techniques*. To be published. (A draft version will be available on the course site)
- Michael Fagan, Advances in Software Inspections, *IEEE Transactions on Software Engineering*, July 1986 www.mfagan.com/aisi1986.pdf
- Nasa, "Software Formal Inspections Guidebook," Office of Safety and Mission Assurance, NASA-GB-A302 approved August 1993 (http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt)
- Check list for inspections of Java code by Christopher Fox (available on the course web site)
- A tool for supporting code review (among other things) http://phabricator.org

References

- Code reviews for teams too busy to review code (video) https://www.youtube.com/watch?v=1m3eRFeCInY
- Bacchelli, Bird, Expectations, Outcomes, and Challenges Of Modern Code Review http://research.microsoft.com/pubs/180283/ICSE%202013-codereview.pdf