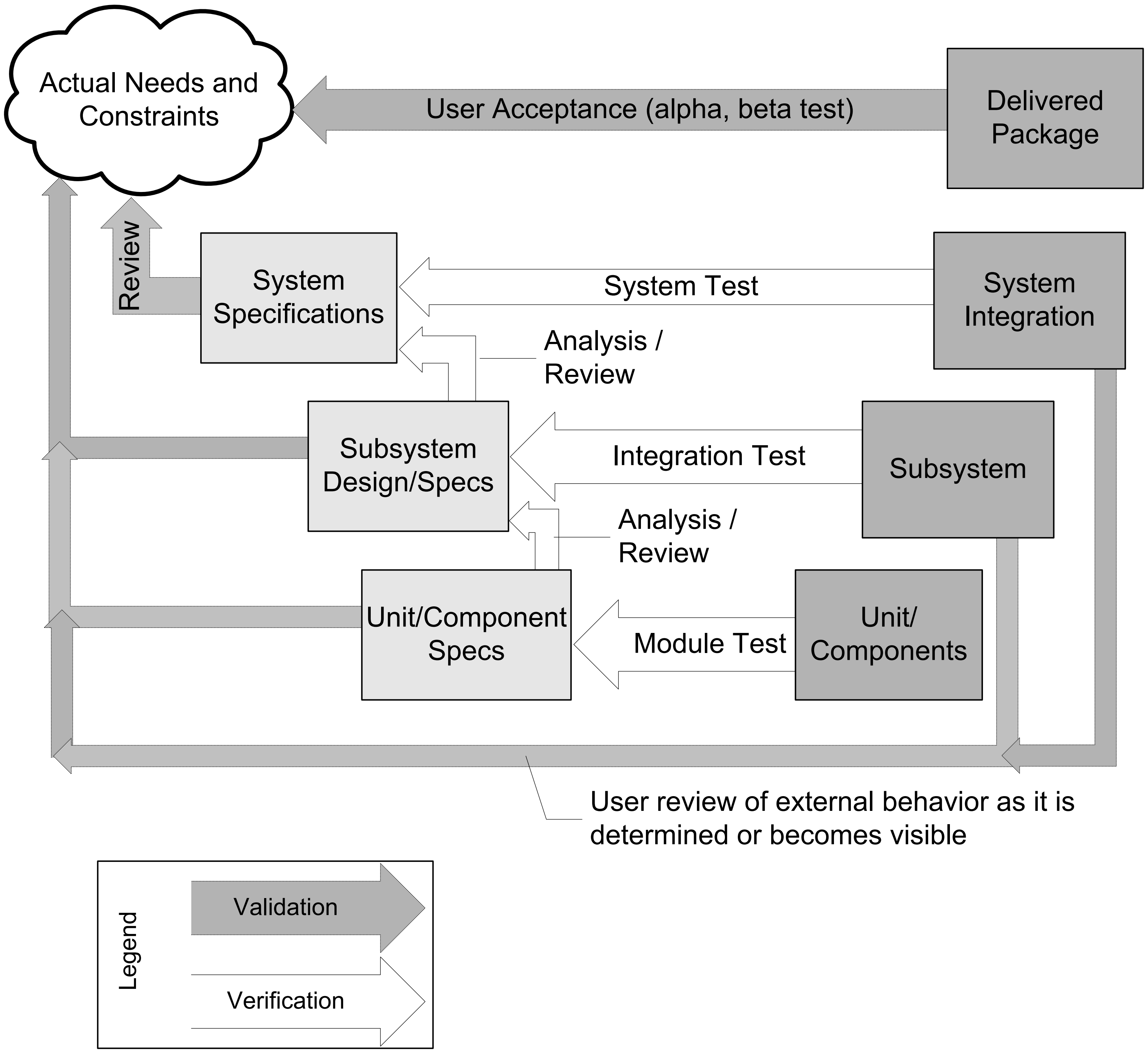
Integration, System and Regression Testing

Automated testing and verification

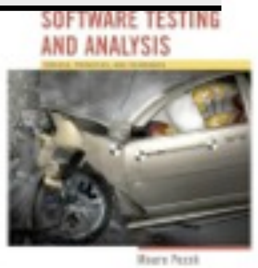
J.P.Galeotti Alessandra Gorla


Actual Needs and Constraints
User Acceptance (alpha, beta test)
Delivered Package
Review
System Specifications
System Test
System Integration
Analysis / Review
Subsystem Design/Specs
Integration Test
Subsystem
Analysis / Review
Unit/Component Specs
Module Test
Unit/ Components
User review of external behavior as it is determined or becomes visible
Legend
Validation
Verification


SOFTWARE TESTING
AND ANALYSIS

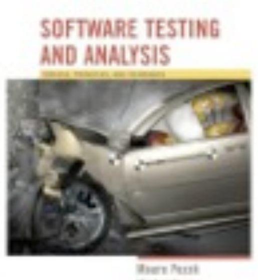
What is integration testing?

	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



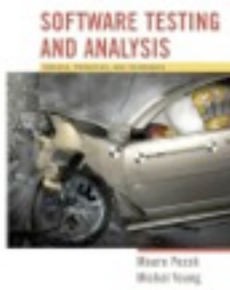
Integration versus Unit Testing

- 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 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)

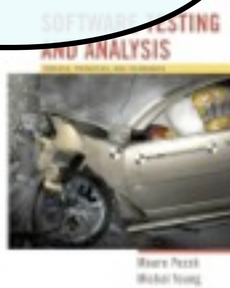
Example: A Memory Leak

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;
}
```

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.)

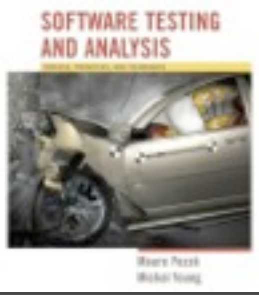
Maybe you’ve heard ...

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.

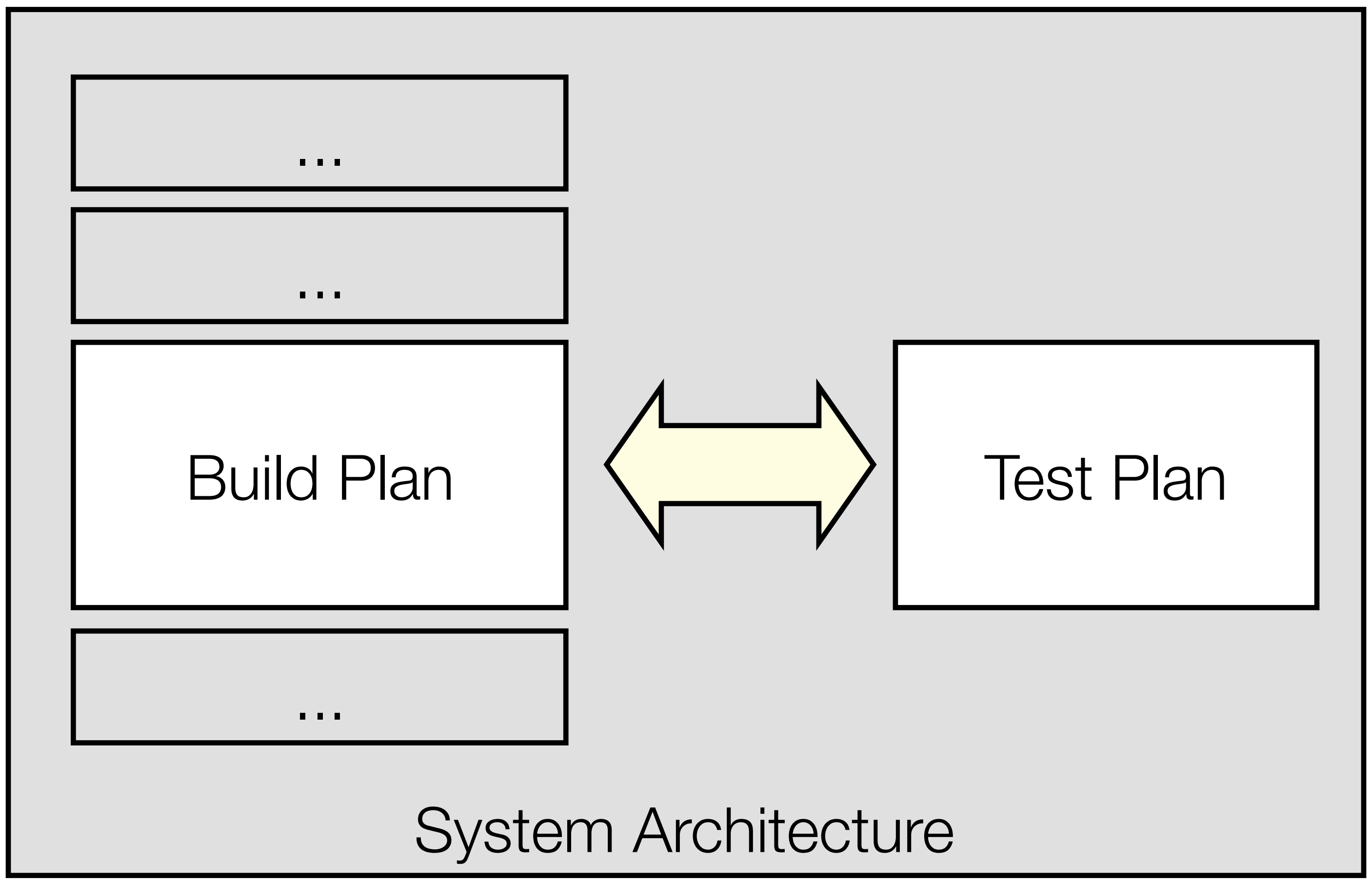
Translation...

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



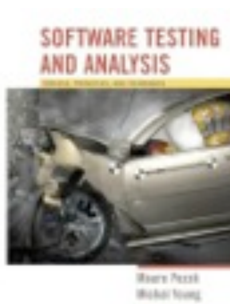
- Integration test plan drives and is driven by the project "build plan"
  - A key feature of the system architecture and project plan

Big Bang Integration Test

An extreme and desperate approach:

Test only after integrating all modules

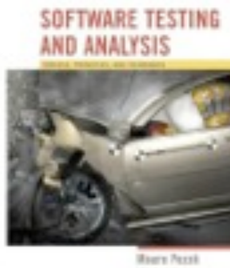
+ Does not require scaffolding
    - The only excuse, and a bad one
- Minimum observability, diagnosability, efficacy, feedback
- High cost of repair
    - Recall: Cost of repairing a fault rises as a function of *time between error and repair*



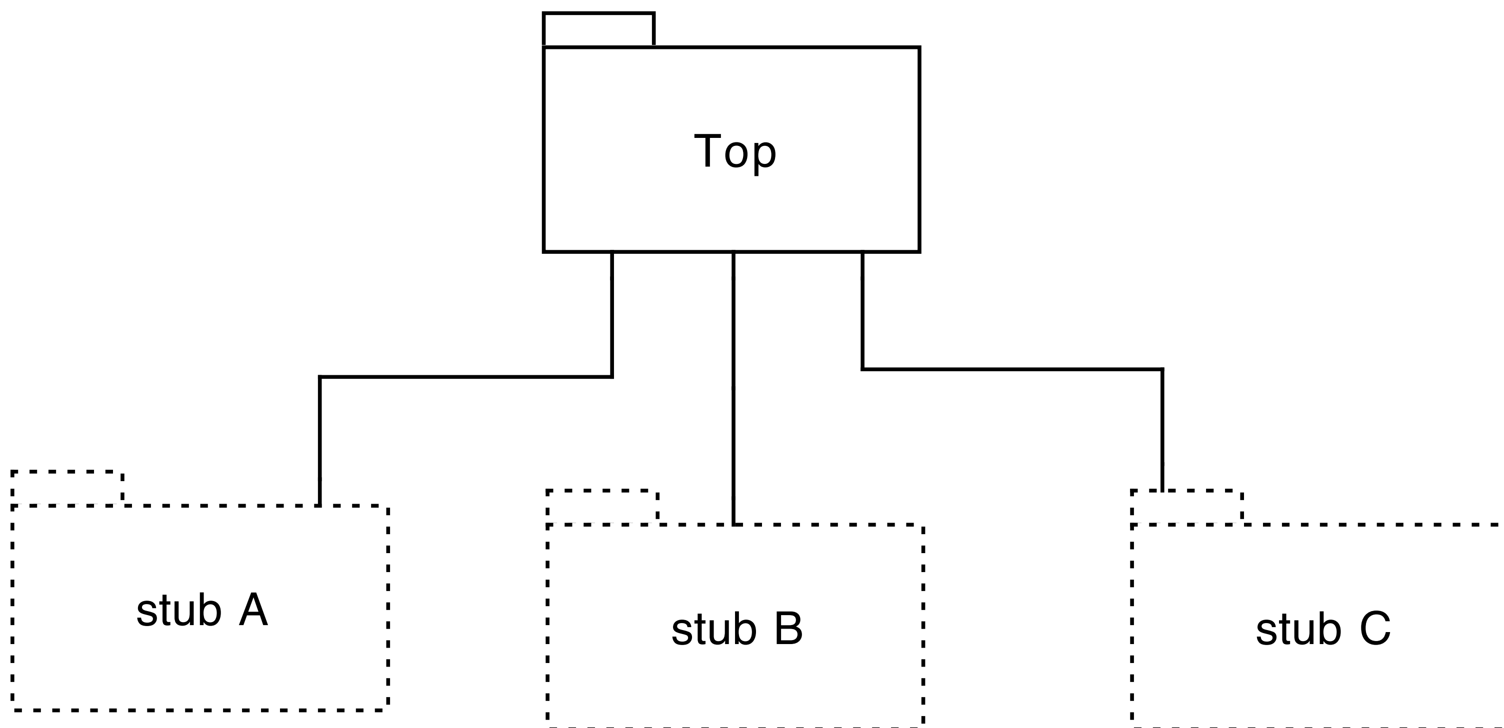
Structural and Functional Strategies

- **Structural orientation**:
  Modules constructed, integrated and tested based on a hierarchical project structure
  - Top-down, Bottom-up, Sandwich

- **Functional orientation**:
  Modules integrated according to application characteristics or features
  - Threads, Critical module

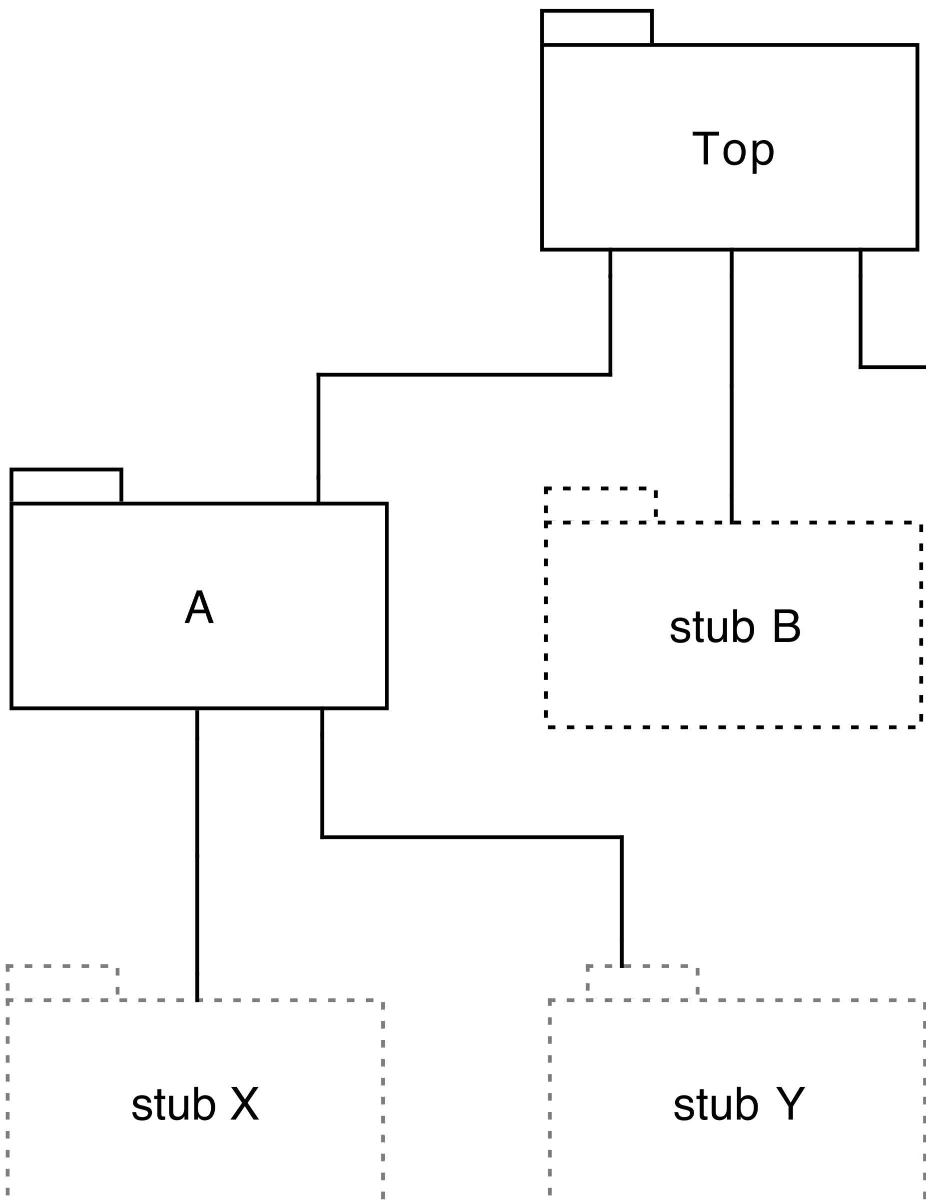


Top down .



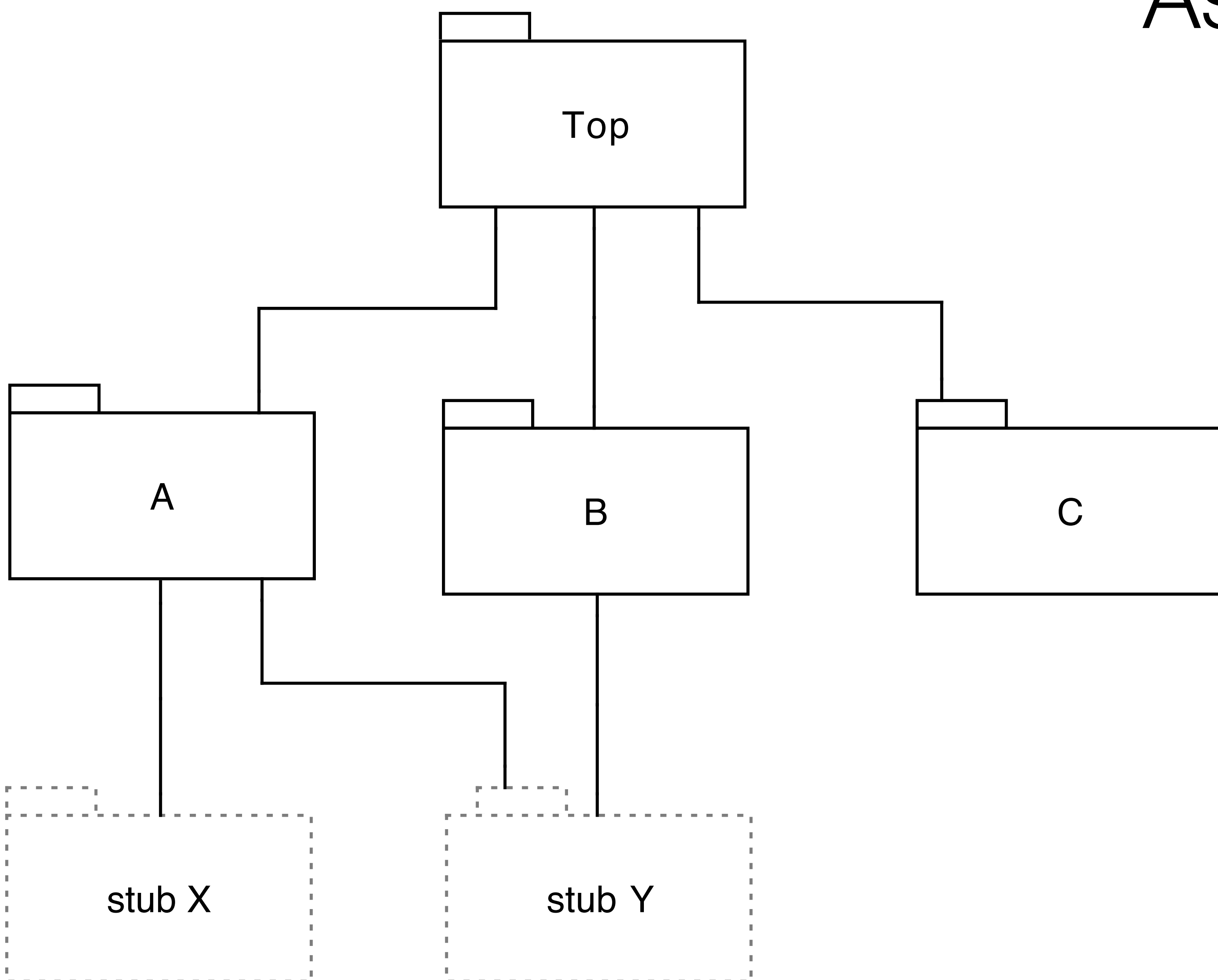
Working from the top level (in terms of “use” or “include” relation) toward the bottom.
No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.)

Top down ..



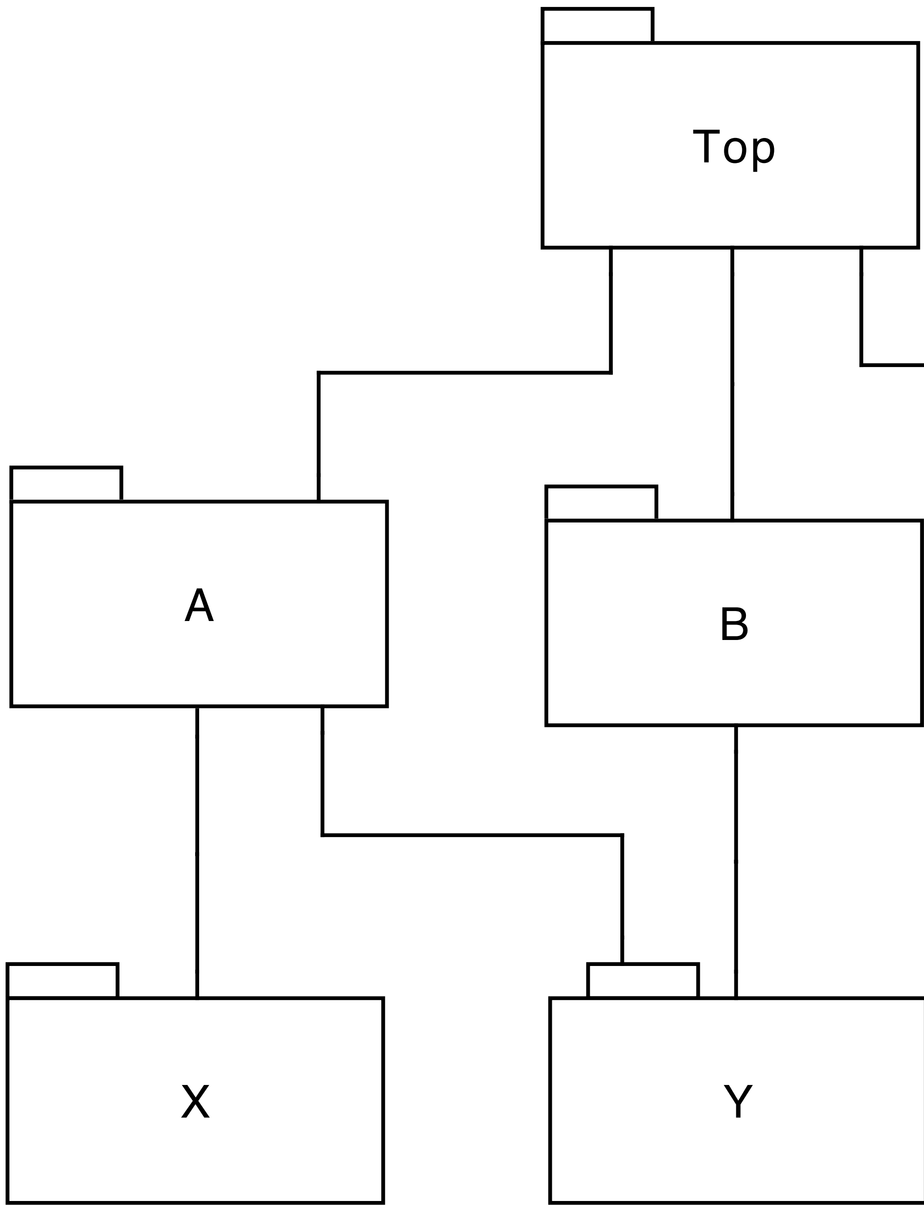
Write stubs of called or used modules at each step in construction

Top down ...



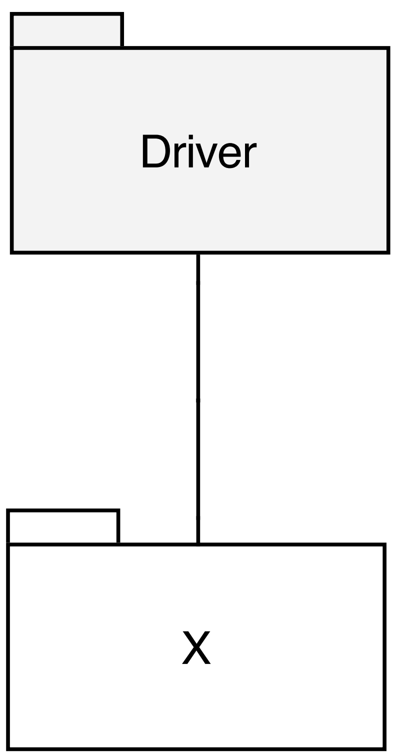
As modules replace stubs, more functionality is testable

Top down ... complete

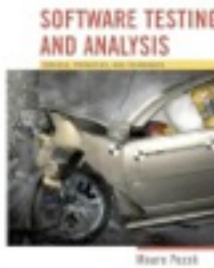


... until the program is complete, and all functionality can be tested

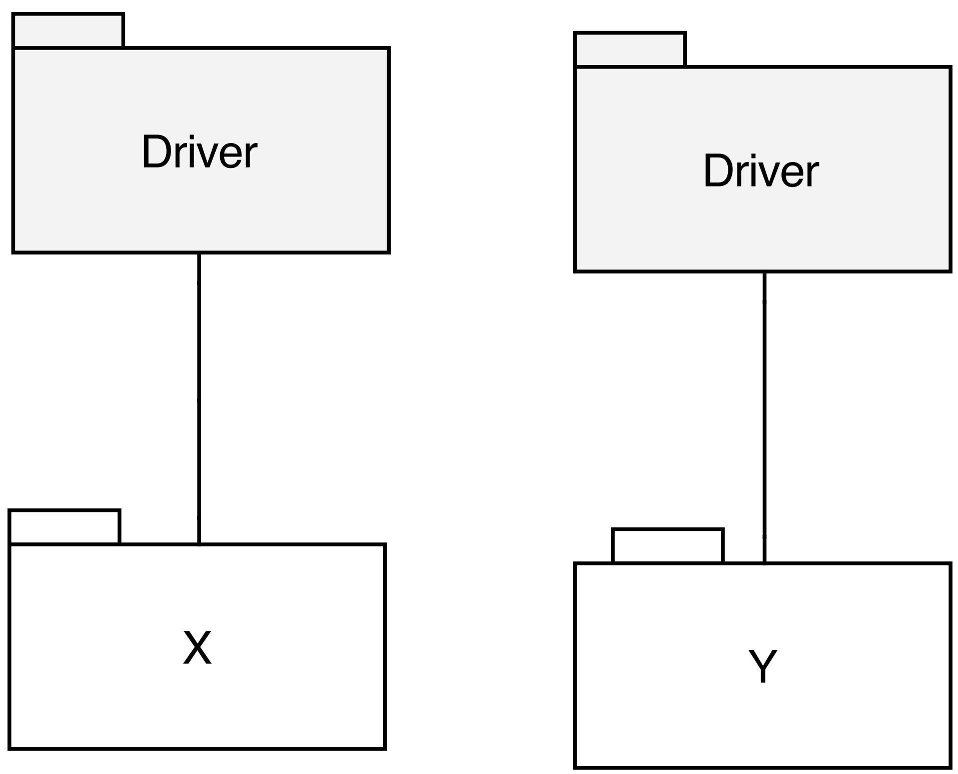
Bottom Up .



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

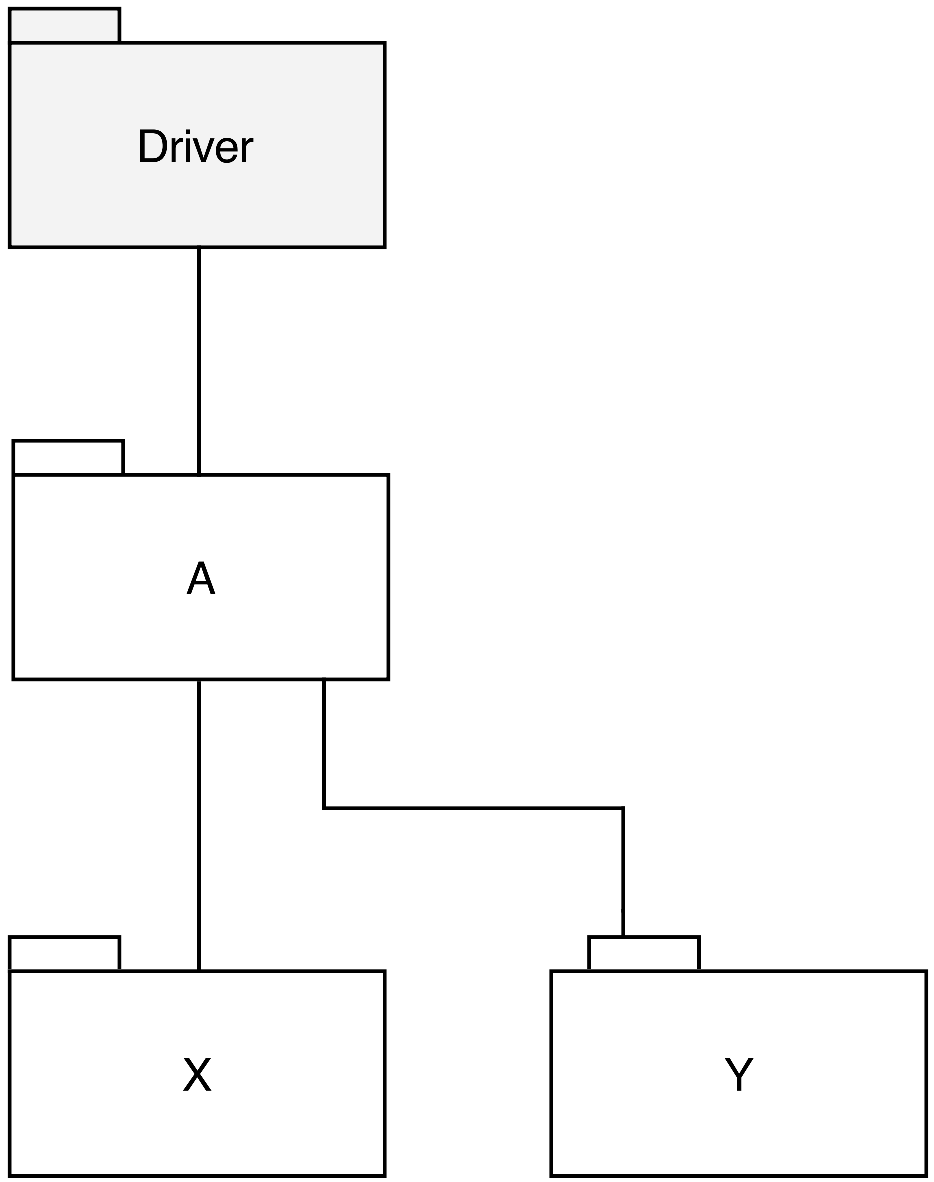


Bottom Up ..

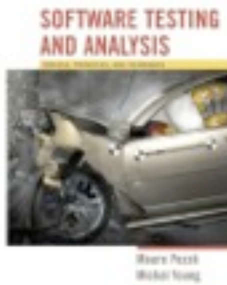


... but we must construct drivers for each module (as in unit testing) ...

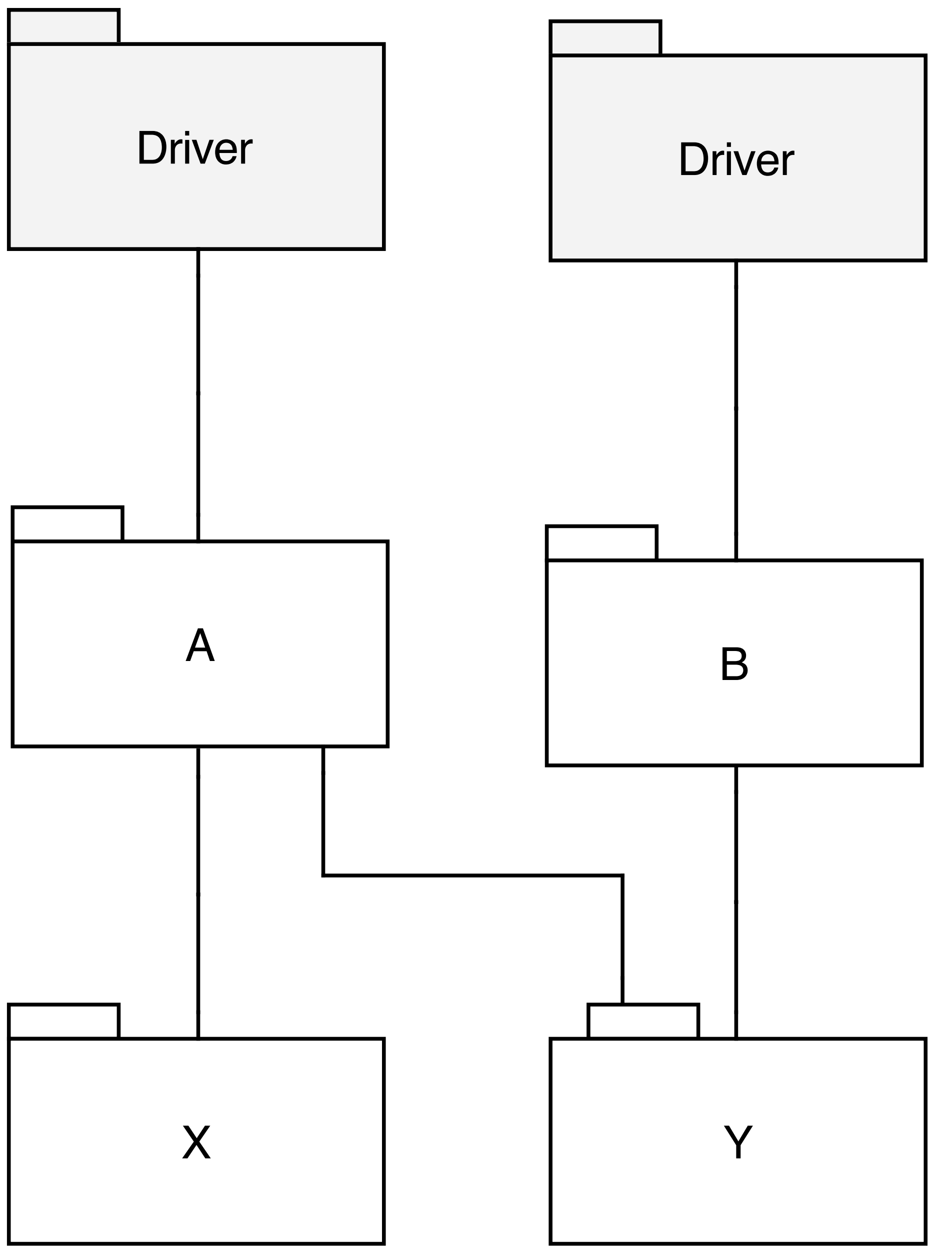
Bottom Up ...



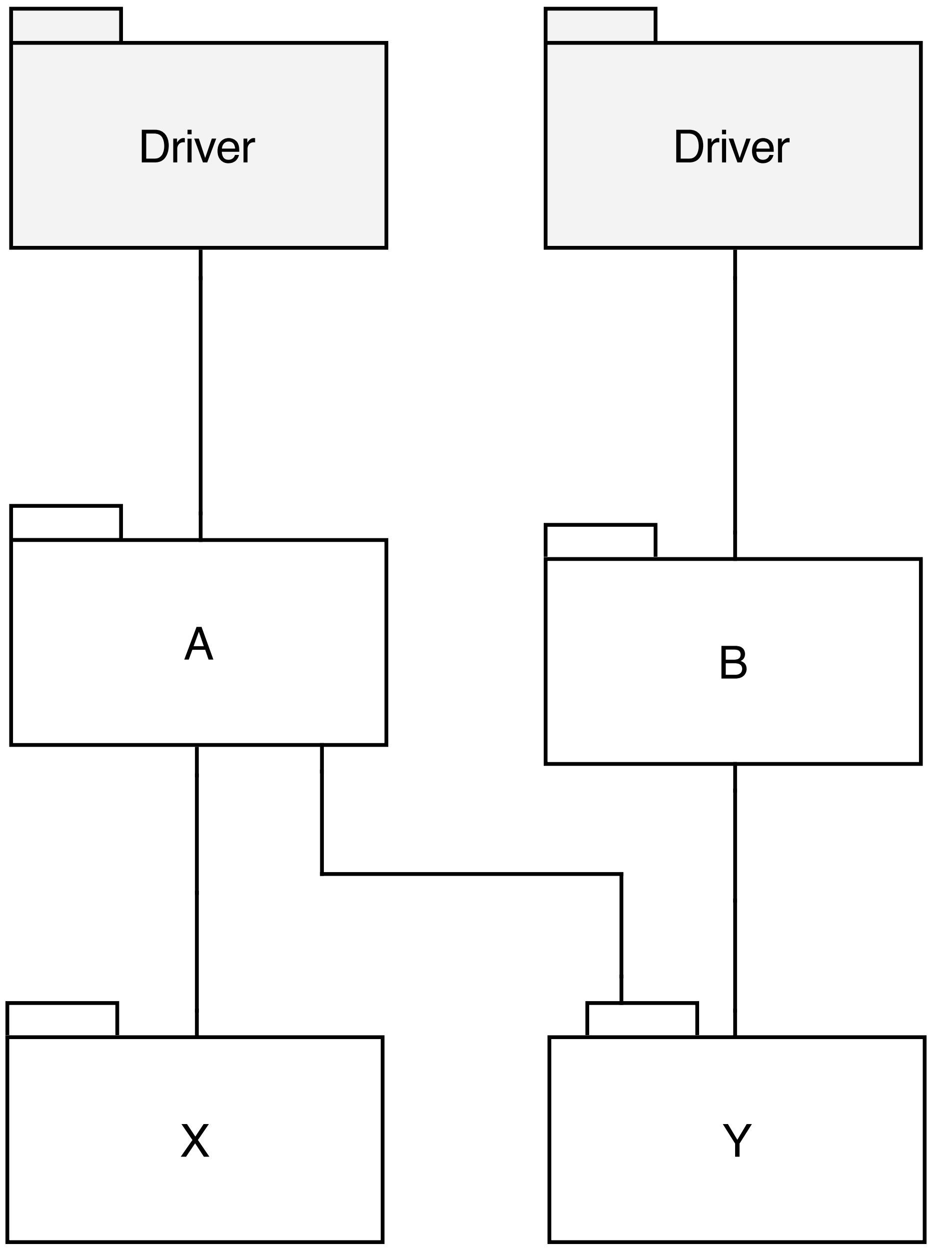
... an intermediate module replaces a driver, and needs its own driver ...



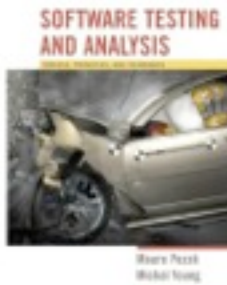
Bottom Up ....



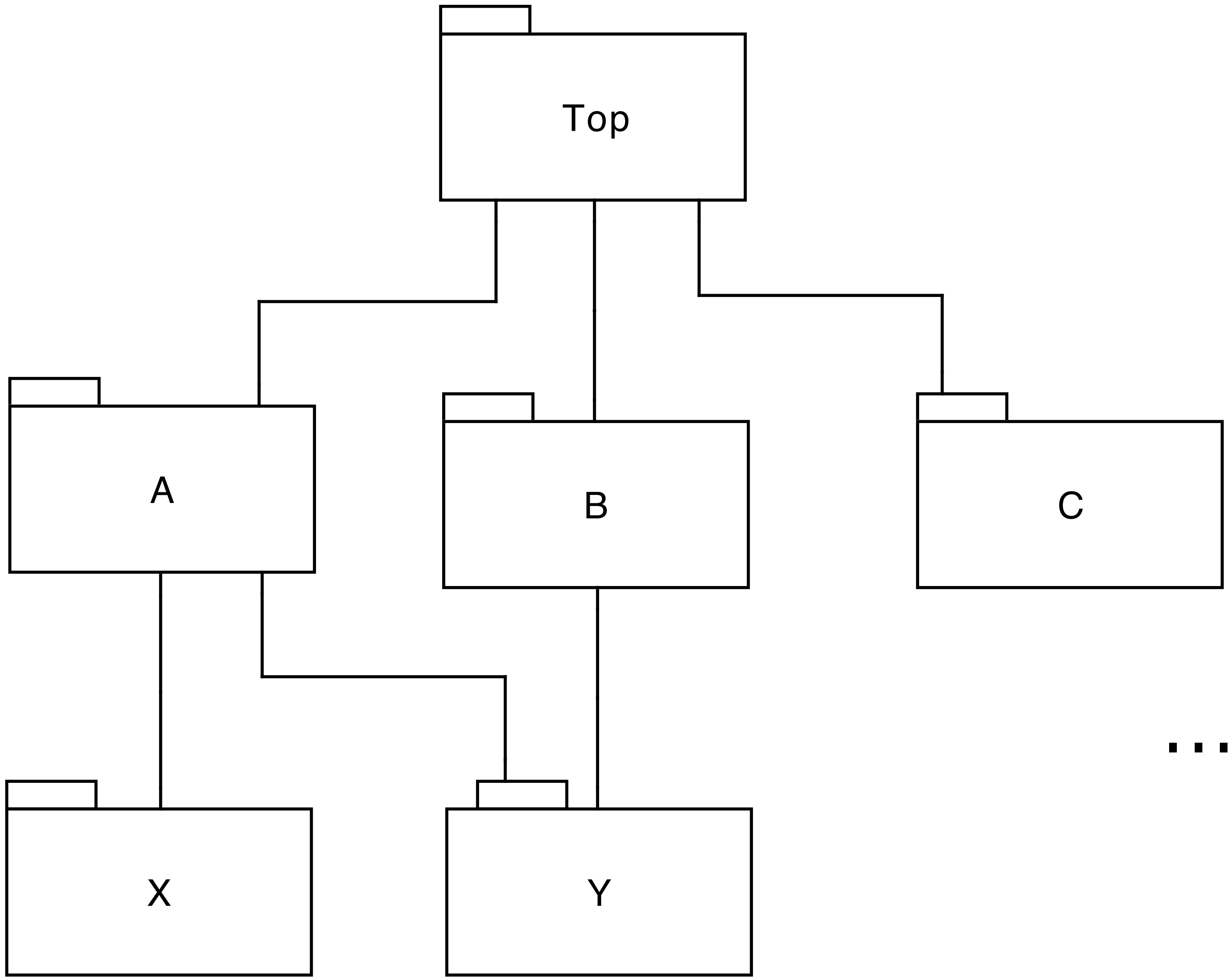
Bottom Up .....



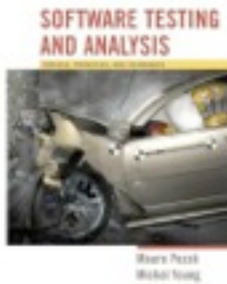
... so we may have several working subsystems ...



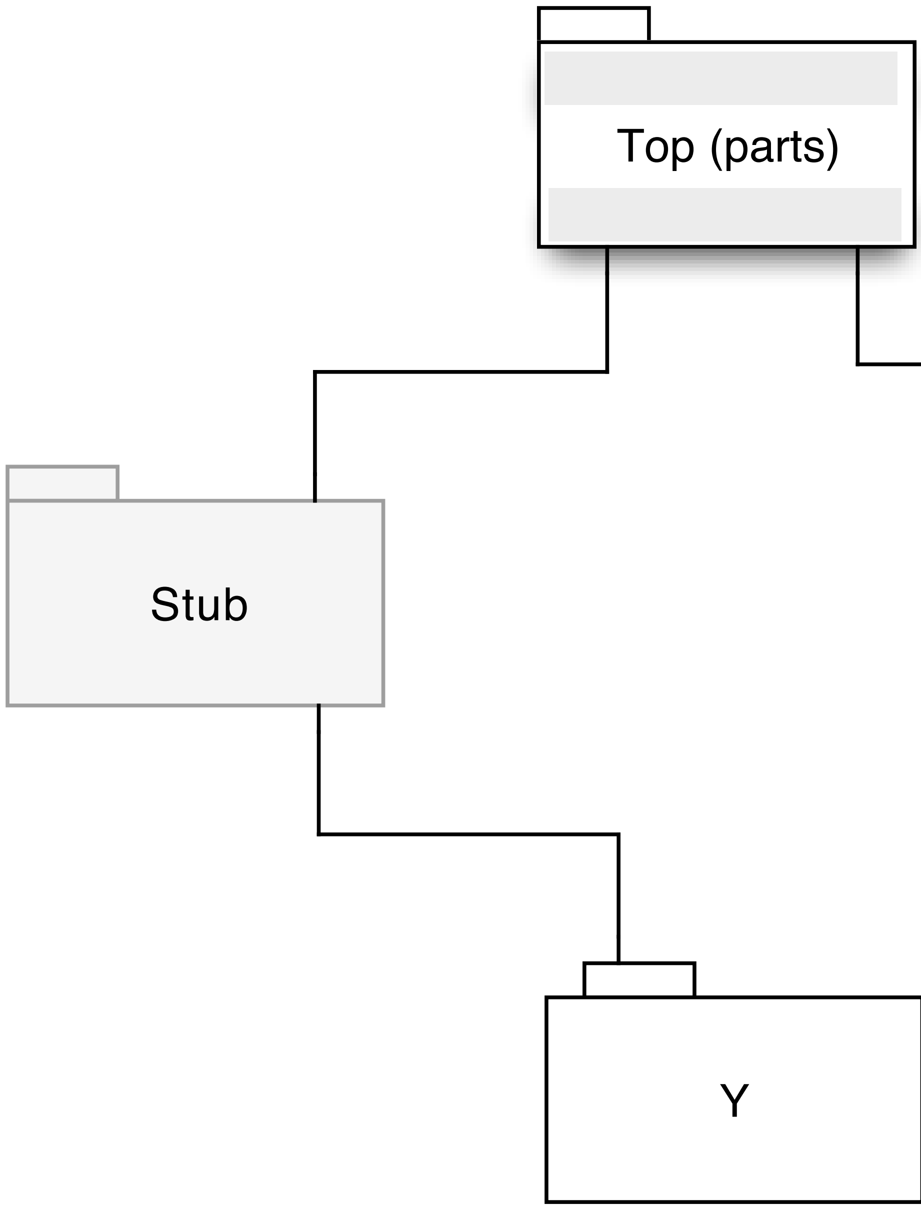
Bottom Up (complete)



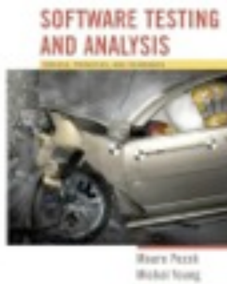
... that are eventually integrated into a single system.



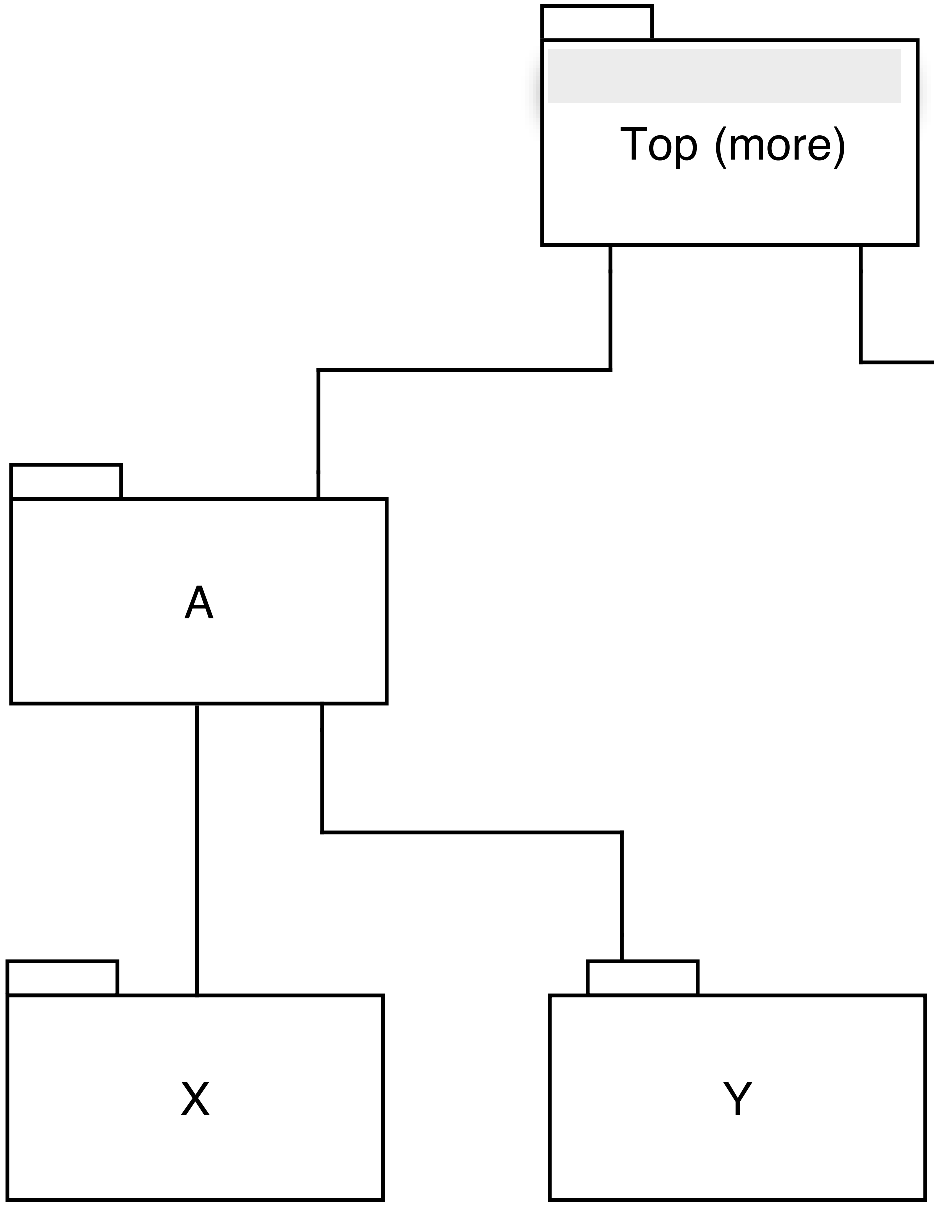
Sandwich .



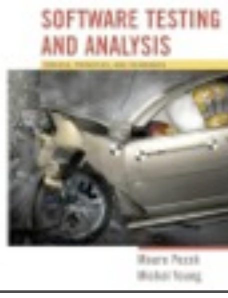
Working from the extremes (top and bottom) toward center, we may use fewer drivers and stubs



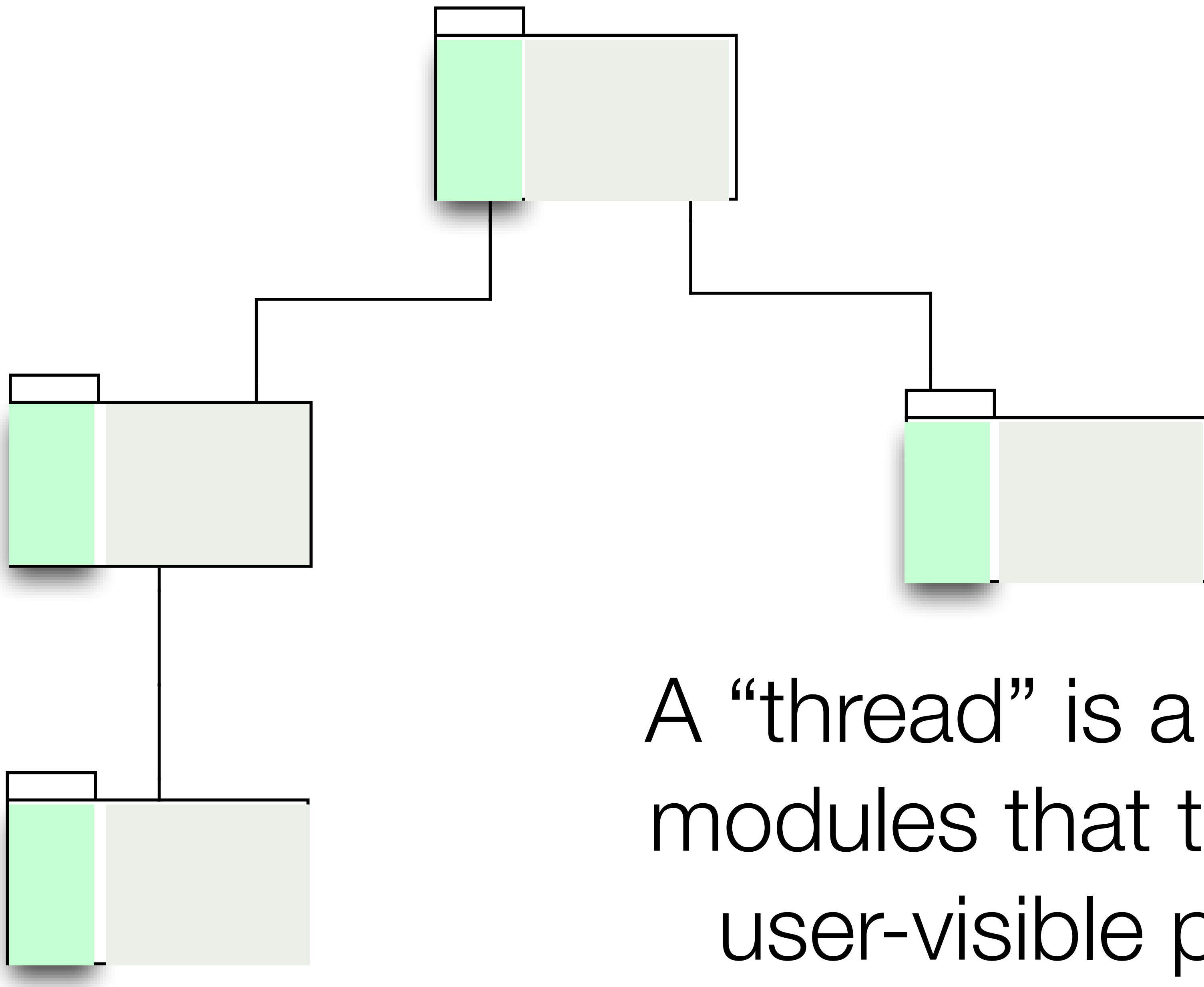
Sandwich ..



Sandwich integration is flexible and adaptable, but complex to plan

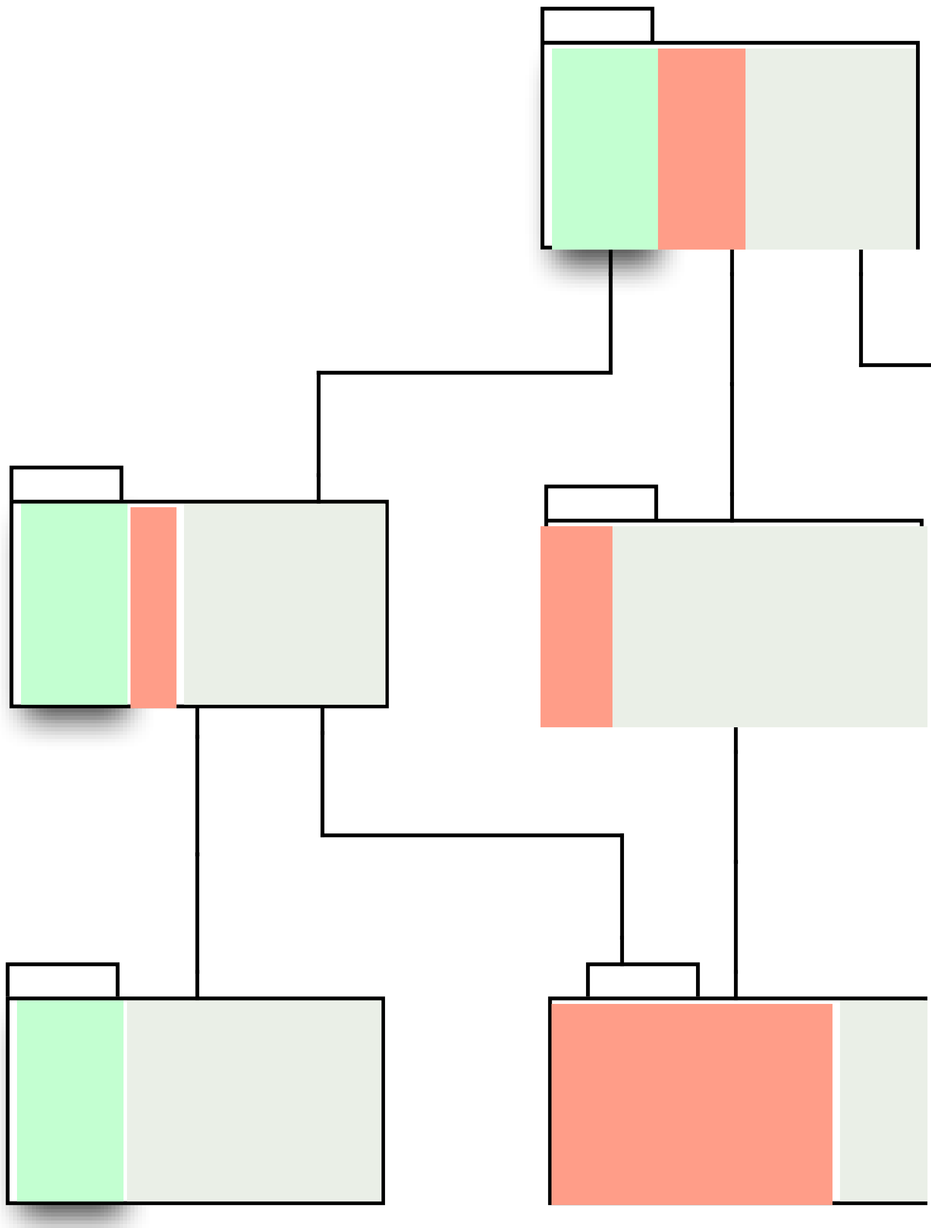


Thread ...



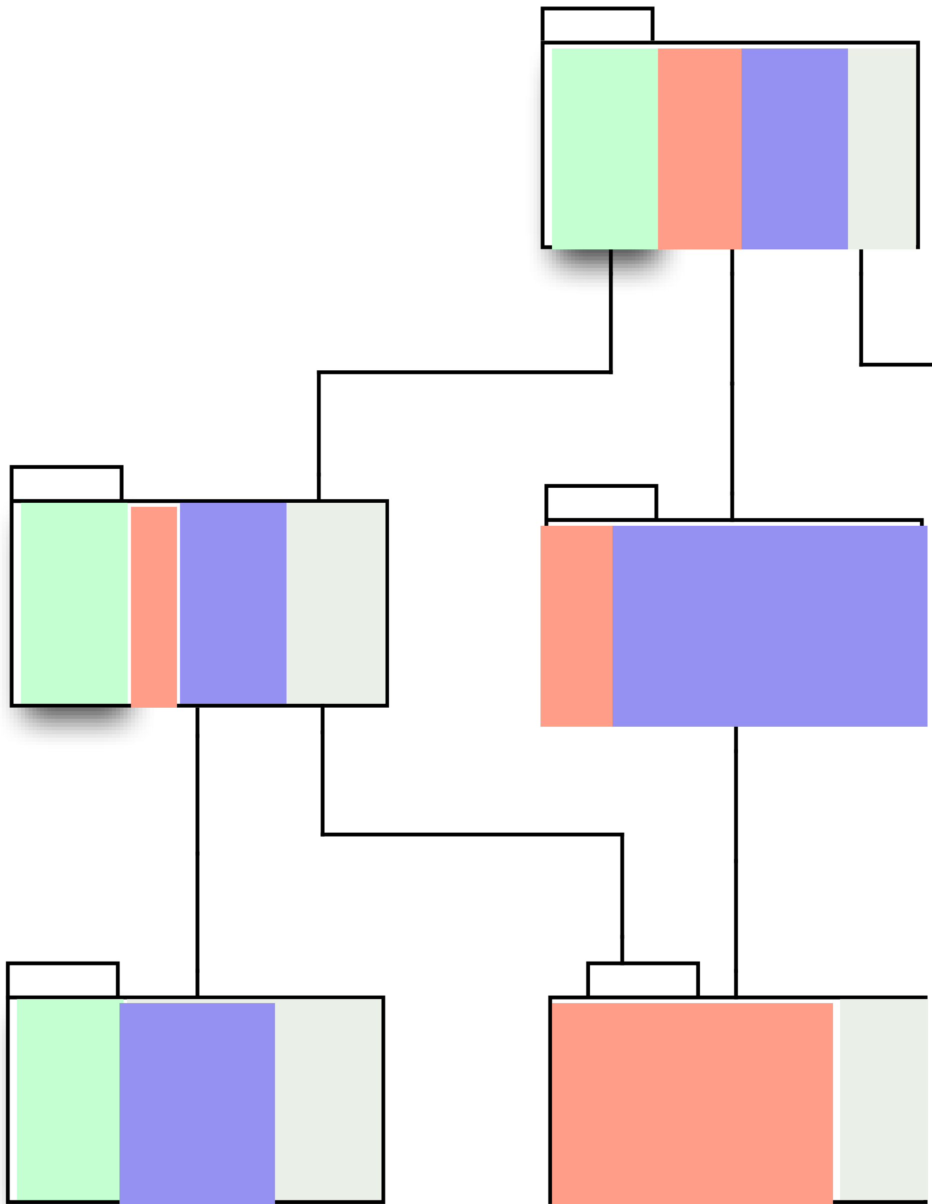
A “thread” is a portion of several modules that together provide a user-visible program feature.

Thread ...



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

Thread ...

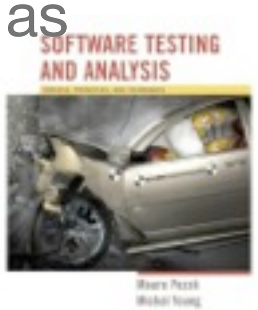


As in sandwich integration testing, we can minimize stubs and drivers, but the integration plan may be complex

Critical Modules

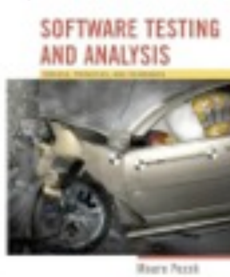
Strategy: Start with riskiest modules

- 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



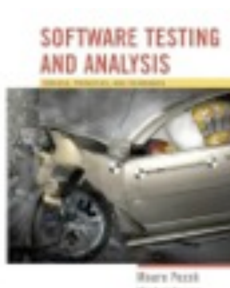
Choosing a Strategy

- 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



Working Definition of **Component**

- <u>Reusable</u> unit of <u>deployment</u> and <u>composition</u>
  - Deployed and integrated multiple times
  - Integrated by different teams (usually)
    - Component producer is distinct from component user
- Characterized by an *interface* or *contract*
  - Describes access points, parameters, and all functional and non-functional behavior and conditions for using the component
  - No other access (e.g., source code) is usually available
- Often larger grain than objects or packages
  - Example: A complete database system may be a component



Component Interface Contracts

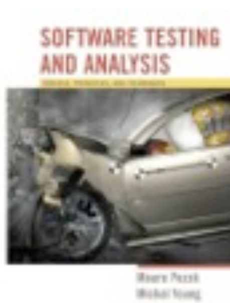
- Application programming interface (API) is distinct from implementation
  - Example: DOM interface for XML is distinct from many possible implementations, from different sources
- Interface includes *everything* that must be known to use the component
  - More than just method signatures, exceptions, etc
  - May include non-functional characteristics like performance, capacity, security
  - May include dependence on other components

Challenges in Testing Components

- The component builder's challenge:
  - Impossible to know all the ways a component may be used
  - Difficult to recognize and specify all potentially important properties and dependencies
- The component user's challenge:
  - No visibility "inside" the component
  - Often difficult to judge suitability for a particular use and context

Testing a Component: Producer View

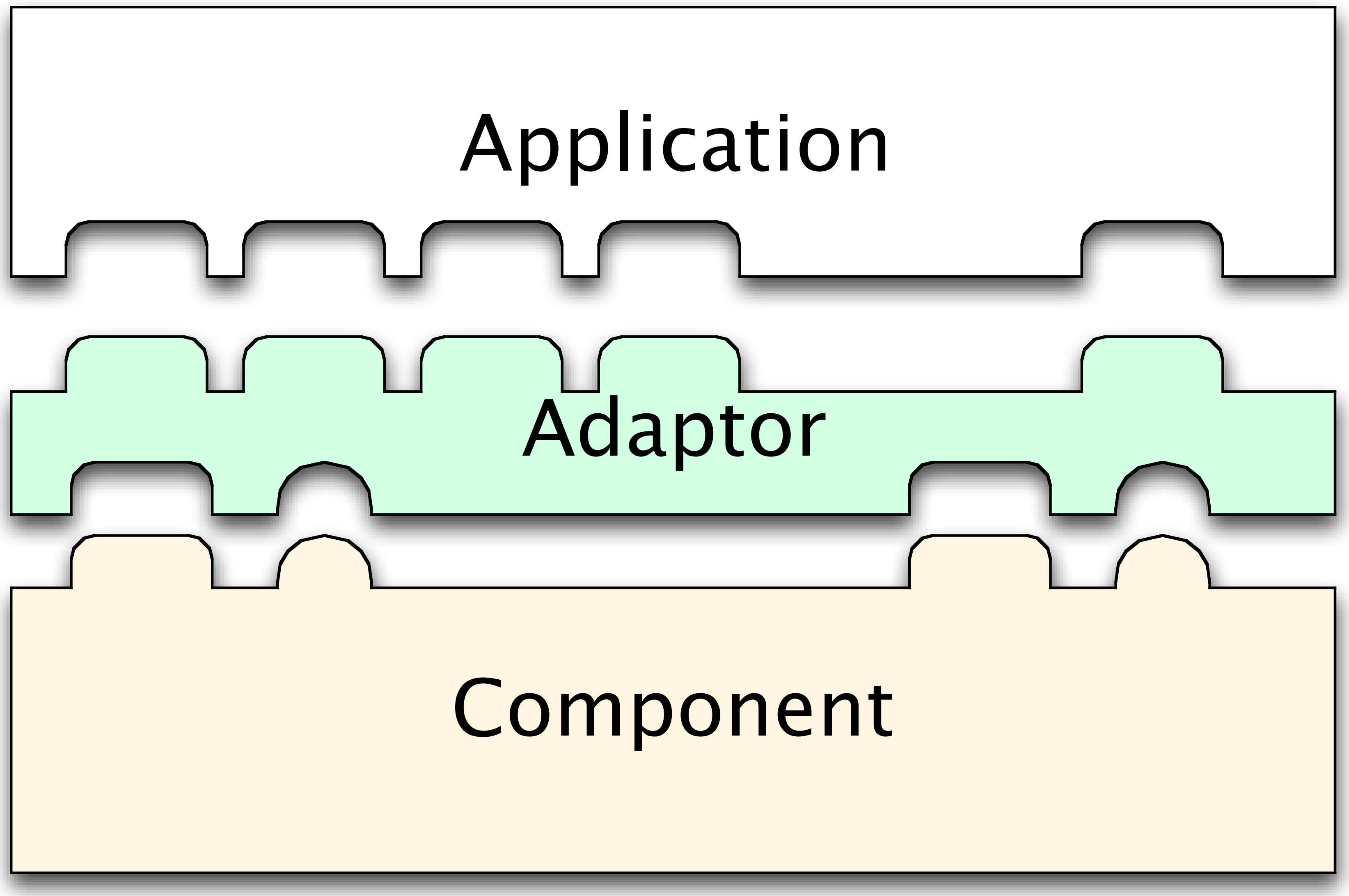
- First: Thorough **unit and subsystem** testing
  - Includes thorough functional testing based on application program interface (API)
  - Reusable component requires at least *twice the effort in design*, *implementation*, and *testing* as a subsystem constructed for a single use (often more)
- Second: Thorough **acceptance** testing
  - Based on scenarios of expected use
  - Includes stress and capacity testing
    - Find and document the limits of applicability



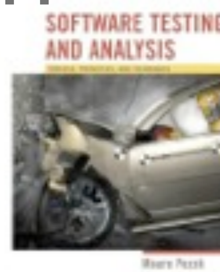
Testing a Component: User View

- Not primarily to find faults in the component
- Major question: Is the component suitable for *this* application?
  - Primary risk is not fitting the application context:
    - Unanticipated dependence or interactions with environment
    - Performance or capacity limits
    - Missing functionality, misunderstood API
  - High risk when using component for first time
- Reducing risk: Trial integration early
  - Often worthwhile to build driver to test model scenarios, long before actual integration

Adapting and Testing a Component

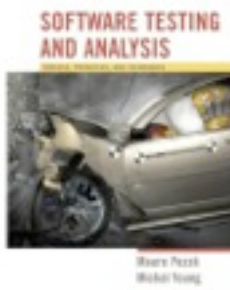


Applications often access components through an adaptor, which can also be used by a test driver



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
- Reusable components require special care
  - For component builder, and for component user



System, Acceptance, and Regression Testing

	System	Acceptance	Regression
Test for ...	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by ...	Development test group	Test group with users	Development test group
	Verification	*Validation*	Verification

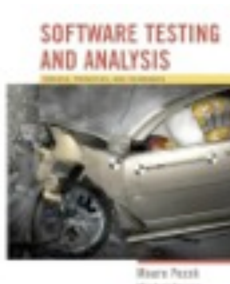
System Testing

- Key characteristics:
  - Comprehensive (the whole system, the whole spec)
  - Based on specification of observable behavior

    Verification against a requirements specification, not validation, and not opinions

  - Independent of design and implementation

    Independence: Avoid repeating software design errors in system test design

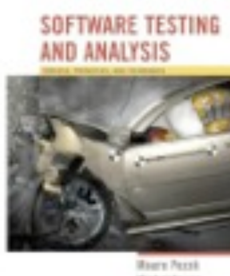


Independent V&V

- *One strategy for maximizing independence:* System (and acceptance) test performed by a different organization
  - Organizationally isolated from developers (no pressure to say "ok")
  - Sometimes outsourced to another company or agency
    - Especially for critical systems
    - Outsourcing for independent judgment, not to save money
    - May be *additional* system test, not replacing internal V&V
  - Not all outsourced testing is IV&V
    - Not *independent* if controlled by development organization

Independence without changing staff

- If the development organization controls system testing ...
  - Perfect independence may be unattainable, but we can reduce undue influence
- Develop system test cases early
  - As part of requirements specification, before major design decisions have been made
    - Agile “test first” and conventional “V model” are both examples of designing system test cases before designing the implementation
    - An opportunity for “design for test”: Structure system for critical system testing early in project

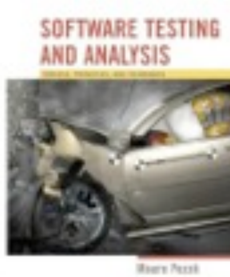


Incremental System Testing

- System tests are often used to measure progress
  - System test suite covers all features and scenarios of use
  - As project progresses, the system passes more and more system tests
- Assumes a “threaded” incremental build plan: Features exposed at top level as they are developed

Global Properties

- Some system properties are inherently global
  - Performance, latency, robustness, ...
  - Early and incremental testing is still necessary, but provide only estimates
- A major focus of system testing
  - The only opportunity to verify global properties against actual system specifications
  - Especially to find unanticipated effects, e.g., an unexpected performance bottleneck

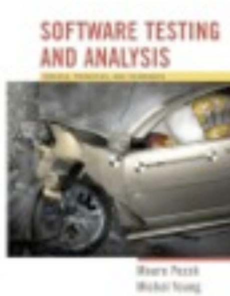


Context-Dependent Properties

- Beyond system-global: Some properties depend on the system context and use
  - Example: Performance properties depend on environment and configuration
  - Example: Privacy depends both on system and how it is used
    - Medical records system must protect against unauthorized use, and authorization must be provided only as needed
  - Example: Security depends on threat profiles
    - And threats change!

Establishing an Operational Envelope

- When a property (e.g., performance or real-time response) is parameterized by use ...
  - requests per second, size of database, ...
- Extensive stress testing is required
  - varying parameters within the envelope, near the bounds, and beyond
- Goal: A well-understood model of how the property varies with the parameter
  - How sensitive is the property to the parameter?
  - Where is the “edge of the envelope”?
  - What can we expect when the envelope is exceeded?



Stress Testing

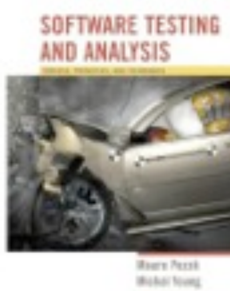
- Often requires extensive simulation of the execution environment
    - With systematic variation: What happens when we push the parameters? What if the number of users or requests is 10 times more, or 1000 times more?
- Often requires more resources (human and machine) than typical test cases
    - Separate from regular feature tests
    - Run less often, with more manual control
    - Diagnose deviations from expectation
        - Which may include difficult debugging of latent faults!

Estimating Dependability

- Measuring quality, not searching for faults
  - Fundamentally different goal than systematic testing
- Quantitative dependability goals are statistical
  - Reliability
  - Availability
  - Mean time to failure
  - ...
- Requires valid statistical samples from *operational profile*
  - Fundamentally different from systematic testing

Statistical Sampling

- We need a valid *operational profile* (model)
  - Sometimes from an older version of the system
  - Sometimes from operational environment (e.g., for an embedded controller)
  - *Sensitivity testing* reveals which parameters are most important, and which can be rough guesses
- And a clear, precise definition of what is being measured
  - Failure rate? Per session, per hour, per operation?
- And many, many random samples
  - Especially for high reliability measures

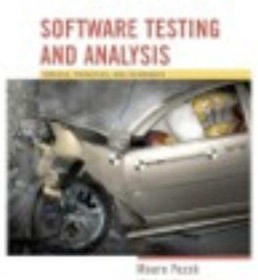


Is Statistical Testing Worthwhile?

- Necessary for ...
  - Critical systems (safety critical, infrastructure, ...)
- But difficult or impossible when ...
  - Operational profile is unavailable or just a guess
    - Often for new functionality involving human interaction
      - But we may factor critical functions from overall use to obtain a good model of only the critical properties
  - Reliability requirement is very high
    - Required sample size (number of test cases) might require years of test execution
    - Ultra-reliability can seldom be demonstrated by testing

Process-based Measures

- Less rigorous than statistical testing
  - Based on similarity with prior projects
- System testing process
  - Expected history of bugs found and resolved
- Alpha, beta testing
  - Alpha testing: Real users, controlled environment
  - Beta testing: Real users, real (uncontrolled) environment
  - May statistically sample users rather than uses
  - Expected history of bug reports



Usability

- A usable product
  - is quickly learned
  - allows users to work efficiently
  - is pleasant to use
- Objective criteria
  - Time and number of operations to perform a task
  - Frequency of user error
    - blame user errors on the product!
- Plus overall, subjective satisfaction

Verifying Usability

- Usability rests ultimately on testing with real users — validation, not verification
  - Preferably in the usability lab, by usability experts
- But we can *factor* usability testing for process visibility — validation *and verification* throughout the project
  - Validation establishes criteria to be verified by testing, analysis, and inspection

Factoring Usability Testing

Validation (usability lab)

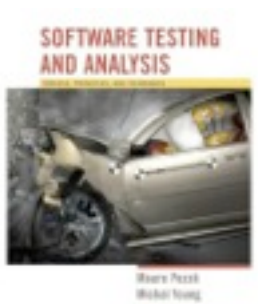
- Usability testing establishes usability check-lists
  - Guidelines applicable across a product line or domain
- Early usability testing evaluates “cardboard prototype” or mock-up
  - Produces interface design

Verification (developers, testers)

- Inspection applies usability check-lists to specification and design
- Behavior objectively verified (e.g., tested) against interface design

Varieties of Usability Test

- **Exploratory** testing
  - Investigate mental model of users
  - Performed early to guide interface design
- **Comparison** testing
  - Evaluate options (specific interface design choices)
  - Observe (and measure) interactions with alternative interaction patterns
- **Usability** validation testing
  - Assess overall usability (quantitative and qualitative)
  - Includes measurement: error rate, time to complete

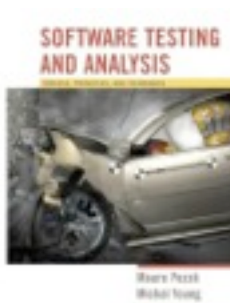


Typical Usability Test Protocol

- Select *representative sample* of user groups
  - Typically 3-5 users from each of 1-4 groups
  - Questionnaires verify group membership
- Ask users to perform a representative sequence of tasks
- Observe without interference (no helping!)
  - The hardest thing for developers is to *not help*. Professional usability testers use one-way mirrors.
- Measure (clicks, eye movement, time, ...) and follow up with questionnai

Accessibility Testing

- Check usability by people with disabilities
  - Blind and low vision, deaf, color-blind, ...
- Use accessibility guidelines
  - Direct usability testing with all relevant groups is usually impractical; checking compliance to guidelines is practical and often reveals problems
- Example: W3C Web Content Accessibility Guidelines
  - Parts can be checked automatically
  - but manual check is still required
    - e.g., is the “alt” tag of the image meaningful?

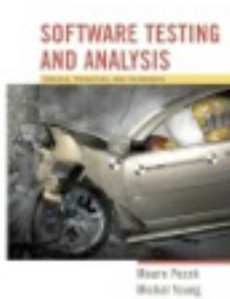


Regression

- Yesterday it worked, today it doesn't
  - I was fixing X, and accidentally broke Y
  - That bug was fixed, but now it's back
- Tests must be re-run after any change
  - Adding new features
  - Changing, adapting software to new conditions
  - Fixing other bugs
- Regression testing can be a major cost of software maintenance
  - Sometimes much more than making the change

Basic Problems of Regression Test

- Maintaining test suite
  - If I change feature X, how many test cases must be revised because they use feature X?
  - Which test cases should be removed or replaced? Which test cases should be added?
- Cost of re-testing
  - Often proportional to product size, not change size
  - Big problem if testing requires manual effort
    - Possible problem even for automated testing, when the test suite and test execution time grows beyond a few hours

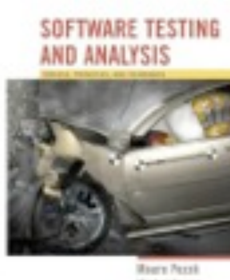


Test Case Maintenance

- Some maintenance is inevitable
  - If feature X has changed, test cases for feature X will require updating
- Some maintenance should be avoided
  - Example: Trivial changes to user interface or file format should not invalidate large numbers of test cases
- Test suites should be modular!
  - Avoid unnecessary dependence
  - *Generating* concrete test cases from test case specifications can help

Obsolete and Redundant

- Obsolete: A test case that is not longer valid
  - Tests features that have been modified, substituted, or removed
  - Should be removed from the test suite
- Redundant: A test case that does not differ significantly from others
  - Unlikely to find a fault missed by similar test cases
  - Has some cost in re-execution
  - Has some (maybe more) cost in human effort to maintain
  - May or may not be removed, depending on costs

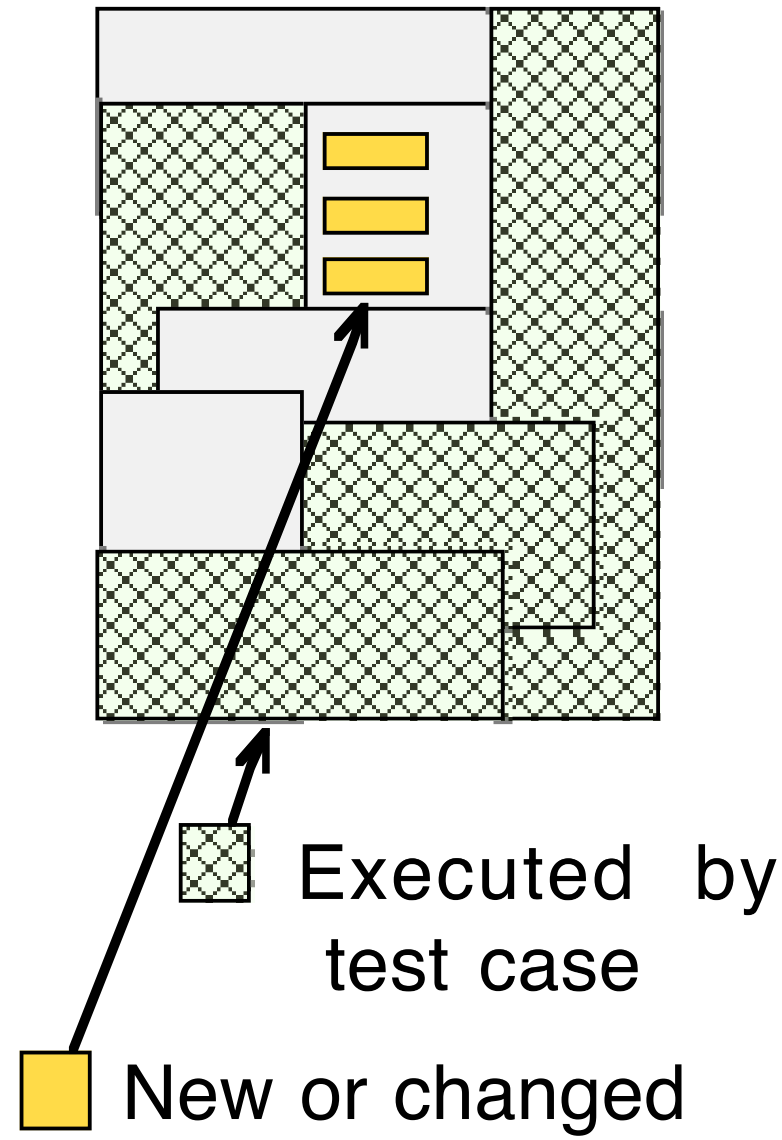


Selecting and Prioritizing Regression Test Cases

- Should we re-run the whole regression test suite? If so, in what order?
  - Maybe you don't care. If you can re-rerun everything automatically over lunch break, do it.
  - Sometimes you do care ...
- Selection matters when
  - Test cases are expensive to execute
    - Because they require special equipment, or long run-times, or cannot be fully automated
- Prioritization matters when
  - A very large test suite cannot be executed every day

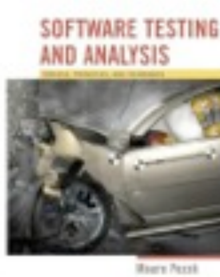
Code-based Regression Test Selection

- Observation: A test case can't find a fault in code it doesn't execute
  - In a large system, many parts of the code are untouched by many test cases
- So: Only execute test cases that execute changed or new code

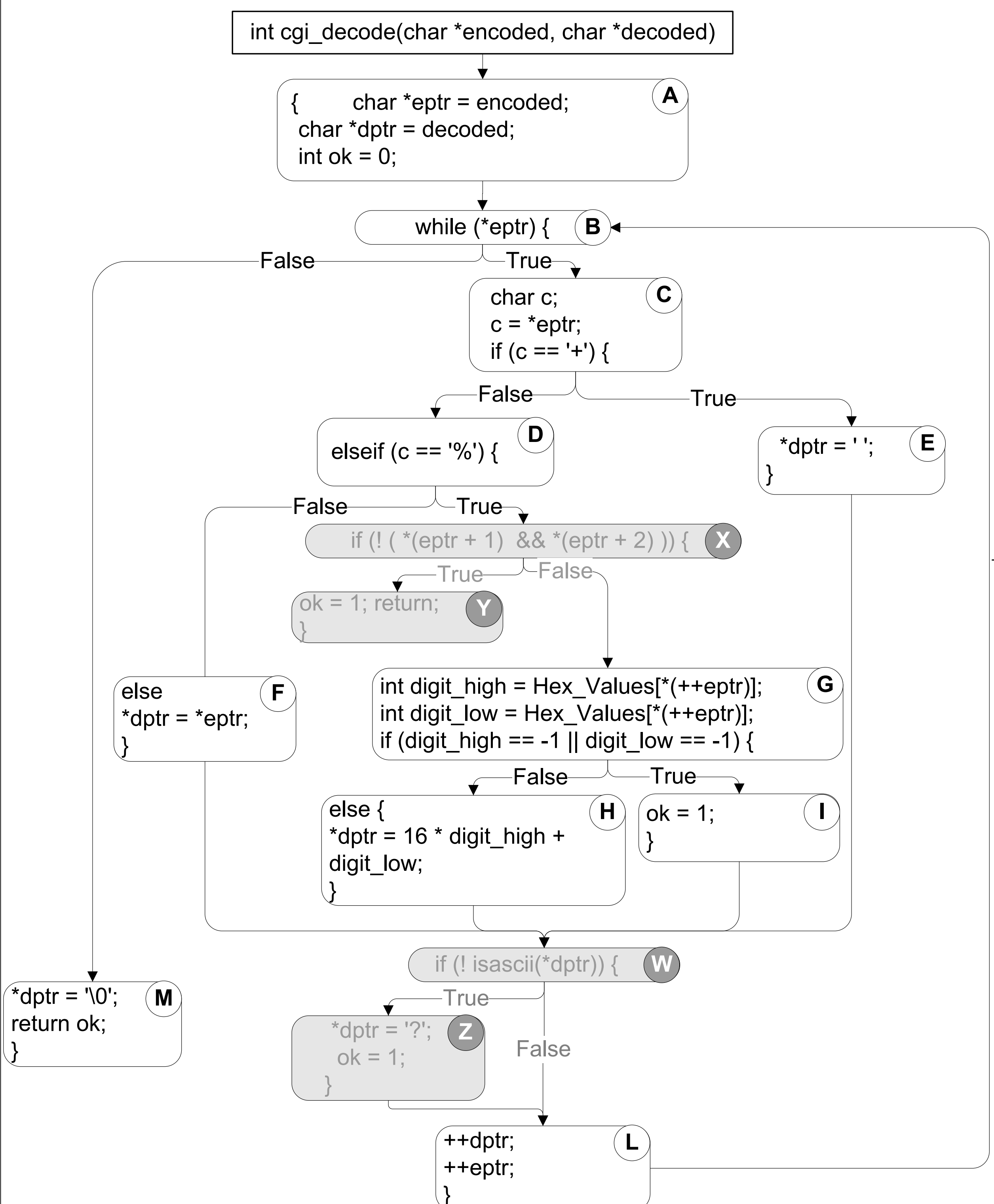


Control-flow and Data-flow Regression Test Selection

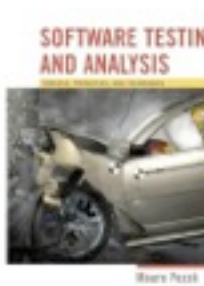
- Same basic idea as code-based selection
  - Re-run test cases only if they include changed elements
  - Elements may be modified control flow nodes and edges, or definition-use (DU) pairs in data flow
- To automate selection:
  - Tools record elements touched by each test case
    - Stored in database of regression test cases
  - Tools note changes in program
  - Check test-case database for overlap



Test Case Selection based on control flow



Id	Test case	Path
TC1	“ ”	A B M
TC2	“test+case%1Dadequacy”	A B C D F L ... B M
TC3	“adequate+test%0Dexecution%7U”	A B C D F L ... B M
TC4	“%3D”	A B C D G H L B M
TC5	“%A”	A B C D G I L B M
TC6	“a+b”	A B C D F L B C E L B C D F L B M
TC7	“test”	A B C D F L B C D F L B C D F L B C D F L B M
TC8	“+%0D+%4J”	A B C E L B C D G I L ... B M
TC9	“first+test%9Ktest%K9”	A B C D F L ... B M

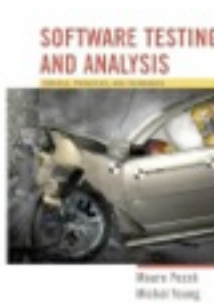


Specification-based Regression Test Selection

- Like code-based and structural regression test case selection
    - Pick test cases that test new and changed functionality
- Difference: No guarantee of independence
    - A test case that isn't "for" changed or added feature X might find a bug in feature X anyway
- Typical approach: Specification-based prioritization
    - Execute all test cases, but start with those that related to changed and added features

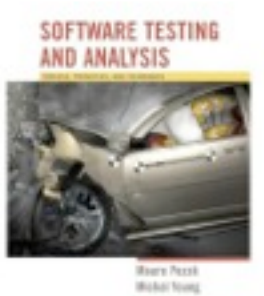
Test Case Selection based on specifications

Case	Too small	Ship where	Ship method	Cust type	Pay method	Same addr	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	–	–	–	–
TC-3	Yes	–	–	–	–	–	–
TC-4	No	Dom	Air	–	–	–	–
TC-5	No	Int	Land	–	–	–	–
TC-6	No	–	–	Edu	Inv	–	–
TC-7	No	–	–	–	CC	Yes	–
TC-8	No	–	–	–	CC	–	No (abort)
TC-9	No	–	–	–	CC	–	No (no abort)



Prioritized Rotating Selection

- Basic idea:
    - Execute all test cases, eventually
    - Execute some sooner than others
- Possible priority schemes:
    - Round robin: Priority to least-recently-run test cases
    - Track record: Priority to test cases that have detected faults before
        - They probably execute code with a high fault density
    - Structural: Priority for executing elements that have not been recently executed
        - Can be coarse-grained: Features, methods, files, ...



Summary

- System testing is verification
  - System consistent with specification?
  - Especially for global properties (performance, reliability)
- Acceptance testing is validation
  - Includes user testing and checks for usability
- Usability and accessibility require both
  - Usability testing establishes objective criteria to verify throughout development
- Regression testing repeated after each change
  - After initial delivery, as software evolves