Integration and Component-based Software Testing

Learning 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
- Understand special challenges and approaches for testing component-based systems

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.)

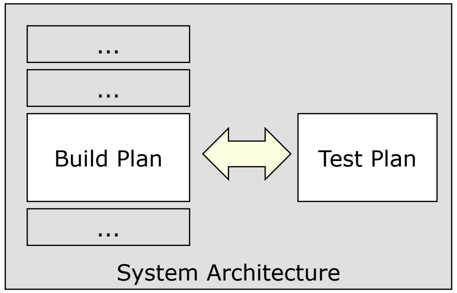
Maybe you've heard ...

- Yes, I implemented 〈module A〉, but I didn't test it thoroughly yet. It will be tested along with 〈module A〉 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 A〉 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:

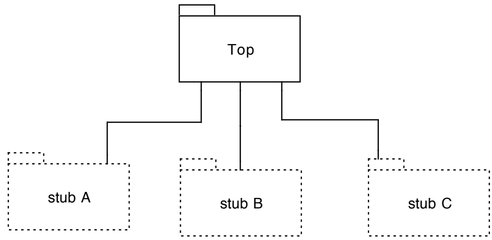
Test only after integrating all modules

+ 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

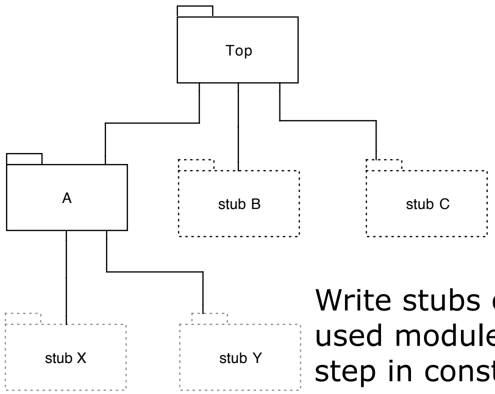
- Structural orientation:
  Modules constructed, integrated and tested based on a hierarchical project structure
  - Top-down, Bottom-up, Sandwich, Backbone
- Functional orientation:
  Modules integrated according to application characteristics or features
  - Threads, Critical module

Top down .



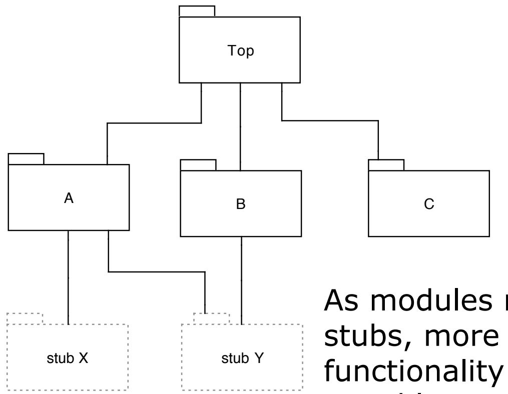
Working from the top level (in terms of "use" or "include" relation) toward the bottom.
No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.)

Top down ..



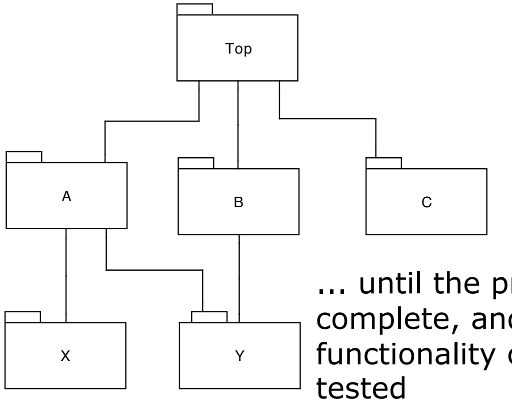
Write stubs of called or used modules at each step in construction

Top down ...



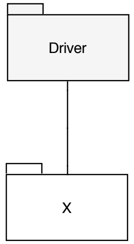
As modules replace stubs, more functionality is testable

Top down ... complete



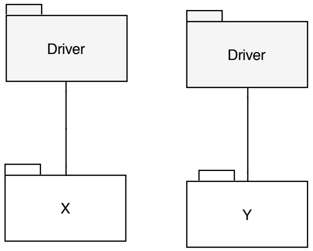
... until the program is complete, and all functionality can be tested

Bottom Up .



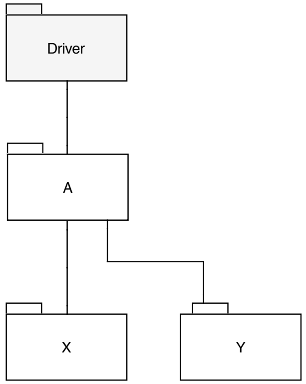
Starting at the leaves of the "uses" hierarchy, we never need stubs

Bottom Up ..



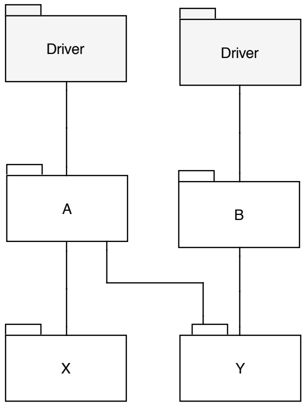
... but we must construct drivers for each module (as in unit testing) ...

Bottom Up ...

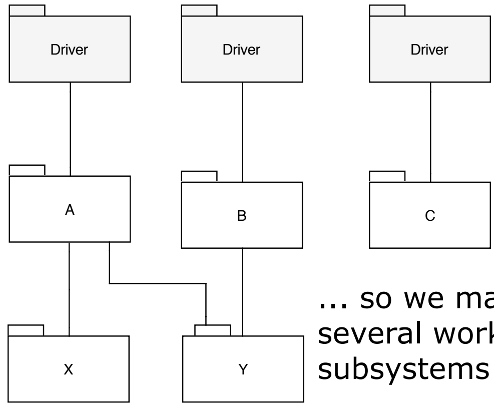


... an intermediate module replaces a driver, and needs its own driver ...

Bottom Up ....

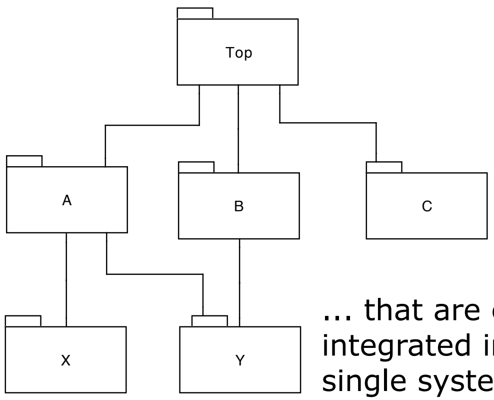


Bottom Up .....



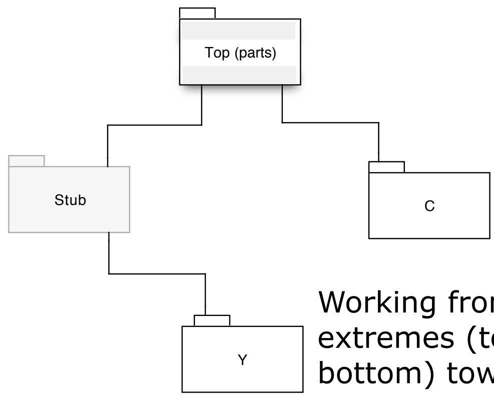
... so we may have several working subsystems ...

Bottom Up (complete)



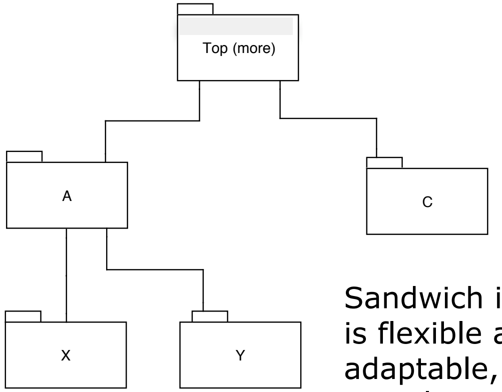
... that are eventually integrated into a single system.

Sandwich .



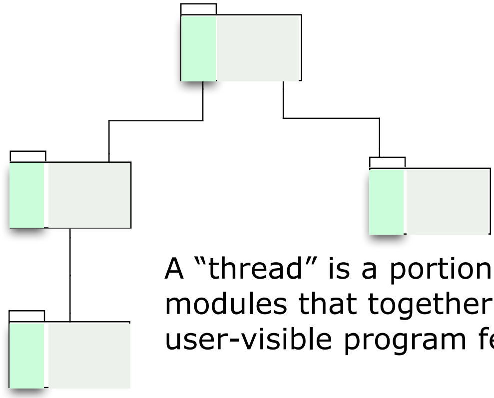
Working from the extremes (top and bottom) toward center, we may use fewer drivers and stubs

Sandwich ..



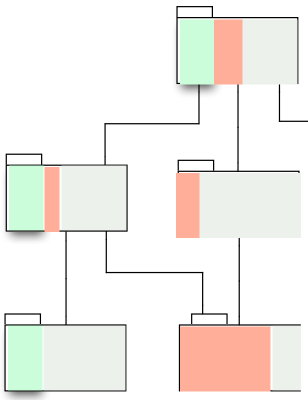
Sandwich integration is flexible and adaptable, but complex to plan

Thread ...



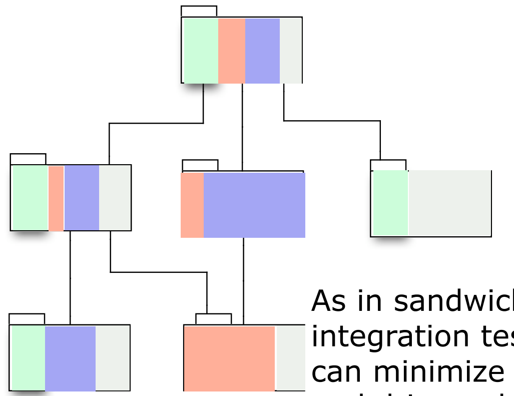
A "thread" is a portion of several modules that together provide a user-visible program feature.

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

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

Components — Related Concepts

- Framework
    - Skeleton or micro-architecture of an application
    - May be packaged and reused as a component, with "hooks" or "slots" in the interface contract
- Design patterns
    - Logical design fragments
    - Frameworks often implement patterns, but patterns are not frameworks. Frameworks are concrete, patterns are abstract
- Component-based system
    - A system composed primarily by assembling components, often "Commercial off-the-shelf" (COTS) components
    - Usually includes application-specific "glue code"

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

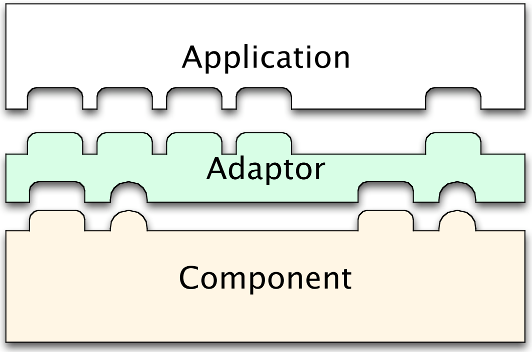
Testing a Component: Producer View

- First: Thorough unit and subsystem testing
  - Includes thorough functional testing based on application program interface (API)
  - Rule of thumb: 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
  - Risk high 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