POLITECNICO
MILANO 1863

Software Engineering 2

Dynamic Analysis

Testing

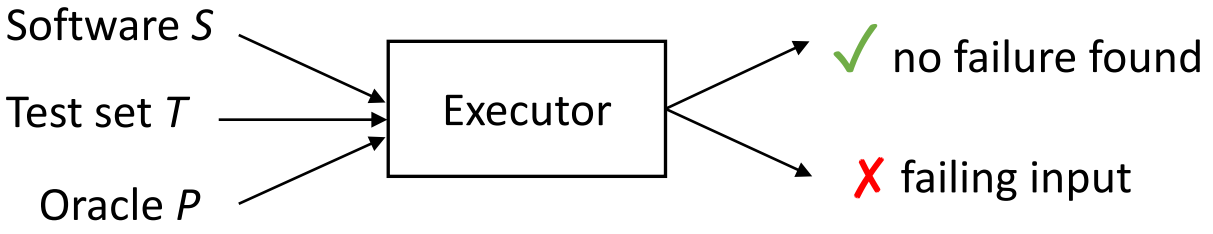
M Camilli, E Di Nitto, M Rossi

Verification & Validation

Dynamic Analysis, Testing

Dynamic analysis, aka testing

- The very idea
  - Analyzes program behavior
  - Properties are encoded as executable oracles, that represent
    - expected outputs, desired conditions (assertions)
  - It can run only finite sets of test cases → it's not exhaustive verification
  - Failures come with concrete inputs that trigger them
  - Execution is automatic (definition of test cases and oracles may not)



What is the goal of testing?

The goal of testing is
making programs fail.

Pezzè, M. and Young, M. [Software testing and analysis: process, principles, and techniques](). John Wiley & Sons, 2008. (available [for free]())

What is the goal of testing?

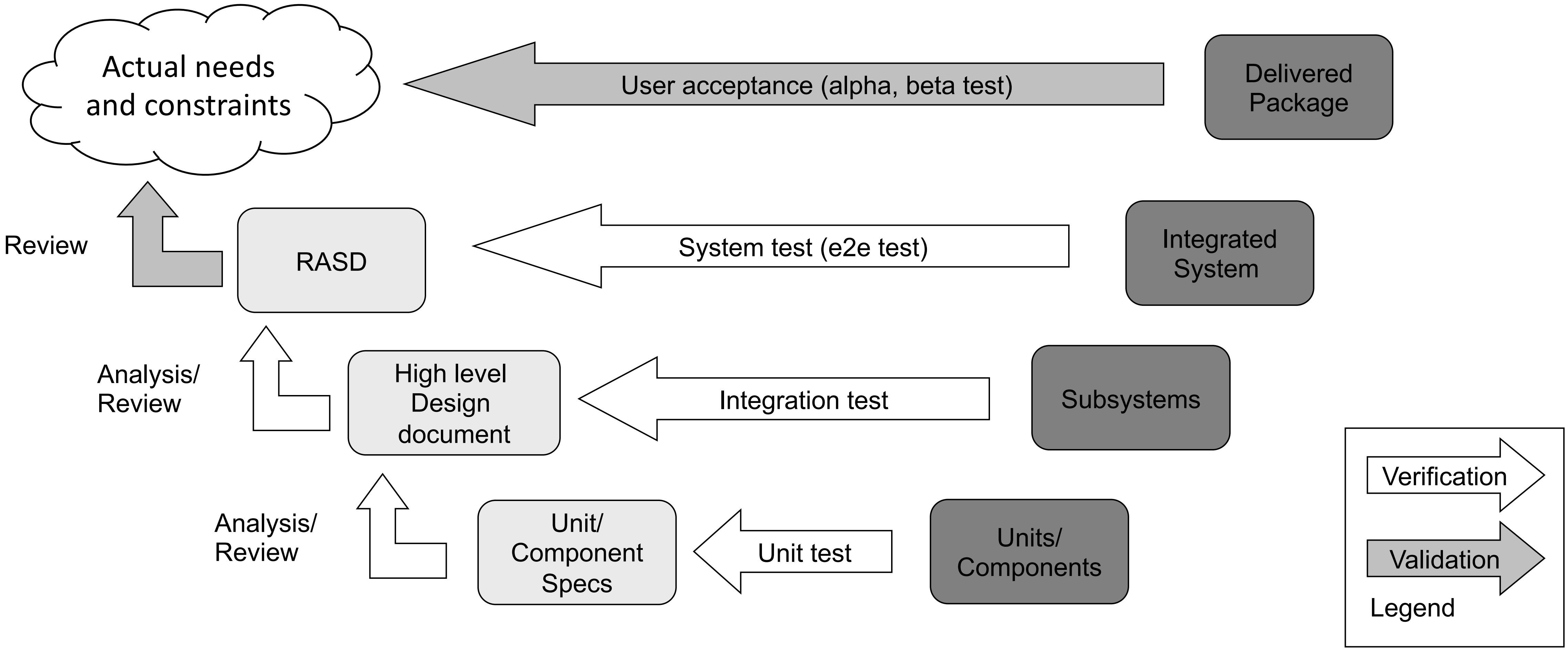
The main goal of testing is making programs fail

- Other common goals
  - Exercise different parts of a program to increase coverage
  - Make sure the interaction between components works (integration testing)
  - Support fault localization and error removal (debugging)
  - Ensure that bugs introduced in the past do not happen again (regression testing)
- Important note
  - *"Program testing can be used to show the presence of bugs, but never to show their absence!"* (Edsger W. Dijkstra)

What is a test case?

- A test case is a set of inputs, execution conditions, and a pass/fail criterion
- Running a test case typically involves
  - Setup: bring the program to an **initial state** that fulfils the execution conditions
  - Execution: **run** the program on the actual inputs
  - Teardown: **record** the output, the final state, and any **failure** determined based on the pass/fail criterion
- A test set or test suite can include multiple test cases
- A test case specification is a requirement to be satisfied by one or more actual test cases
  - Example of test case specification: “*the input must be a sentence composed of at least two words*”
  - Example of test case input: “*this is a good test case input*”

The V model and multiple types of testing



Unit testing

POLITECNICO
MILANO 1863

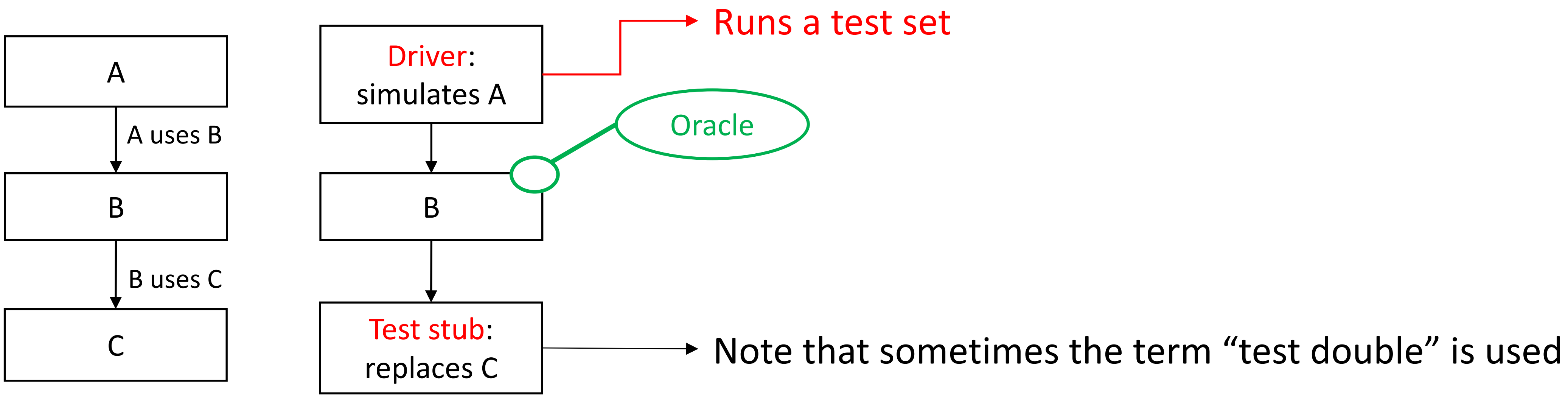
- Conducted by the developers
- Aimed at testing small pieces (units) of code in isolation
  - The notion of "unit" typically depends on the programming language (e.g., class, method, function, procedure)
- Why unit testing?
  - Find problems early
  - Guide the design
  - Increase coverage

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	221	84%	2970/3513	81%	859/1060	1.727
junit.extensions	6	82%	52/63	87%	7/8	1.25
junit.framework	17	76%	399/525	90%	139/154	1.605
junit.runner	3	49%	77/155	41%	23/56	2.225
junit.textui	2	76%	99/130	76%	23/30	1.686
org.junit	14	85%	196/230	75%	68/90	1.655
org.junit.experimental	2	91%	21/23	83%	5/6	1.5
org.junit.experimental.categories	5	100%	67/67	100%	44/44	3.357
org.junit.experimental.max	8	85%	92/108	86%	26/30	1.969
org.junit.experimental.results	6	92%	37/40	87%	7/8	1.222
org.junit.experimental.runners	1	100%	2/2	N/A	N/A	1

Unit testing and scaffolding

- The problem of testing in isolation: units may depend on other units
- We need to simulate missing units
  - e.g., we want to unit test B



Integration testing

POLITECNICO
MILANO 1863

- Aimed at exercising interfaces and components' interaction
- Faults discovered by integration testing
  - Inconsistent interpretation of parameters
    - e.g., mixed units (meters/yards) in Mars Climate Orbiter
  - Violations of assumptions about domains
    - e.g., buffer overflow
  - Side effects on parameters or resources
    - e.g., conflict on (unspecified) temporary file
  - Nonfunctional properties
    - e.g., unanticipated performance issues

An example of integration error

- Apache web server, version 2.0.48
- Code fragment for reacting to normal Web page requests that arrived on the secure (https) server port
- Which problem do we have here?

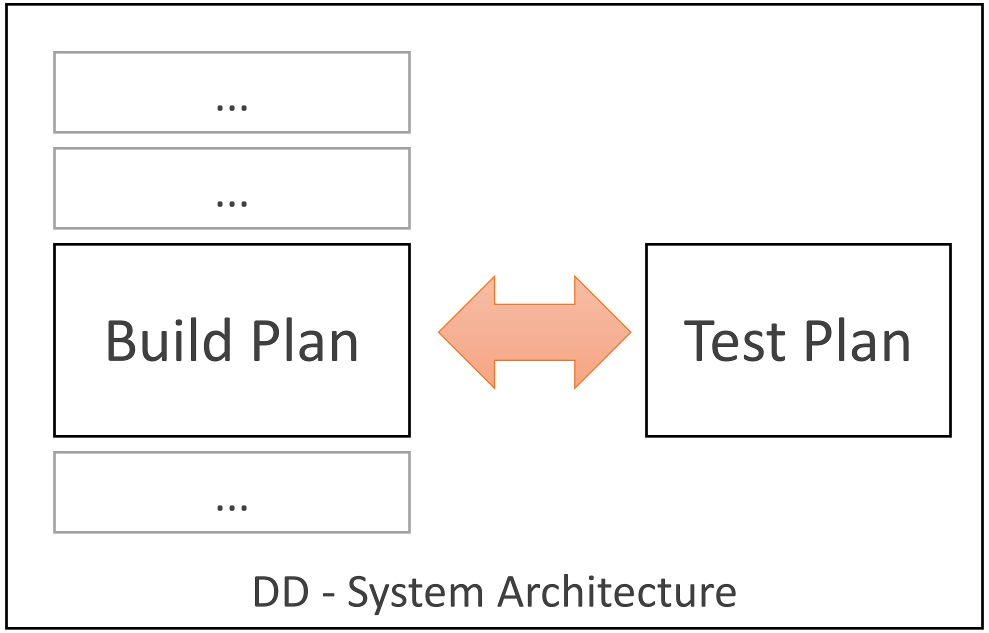
```
static void ssl_io_filter_disable(ap_filter_t *f) {
  bio_filter_in_ctx_t *inctx = f->ctx;

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

An example of integration error

- Repair applied in version 2.0.49

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

Integration and test plan



- Typically defined by the Design Document
- Build plan = defines the order of the implementation
- Test plan = defines how to carry out integration testing
  - Must be consistent with the build plan!

Integration testing: strategies

- Big bang: test only after integrating all modules together (not even a real strategy)
  - Pros
    - Does not require stubs, requires less drivers/oracles
  - Cons
    - Minimum observability, fault localization/diagnosability, efficacy, feedback
    - High cost of repair
      - Recall: Cost of repairing a fault increases as a function of time between the introduction of an error in the code and repair

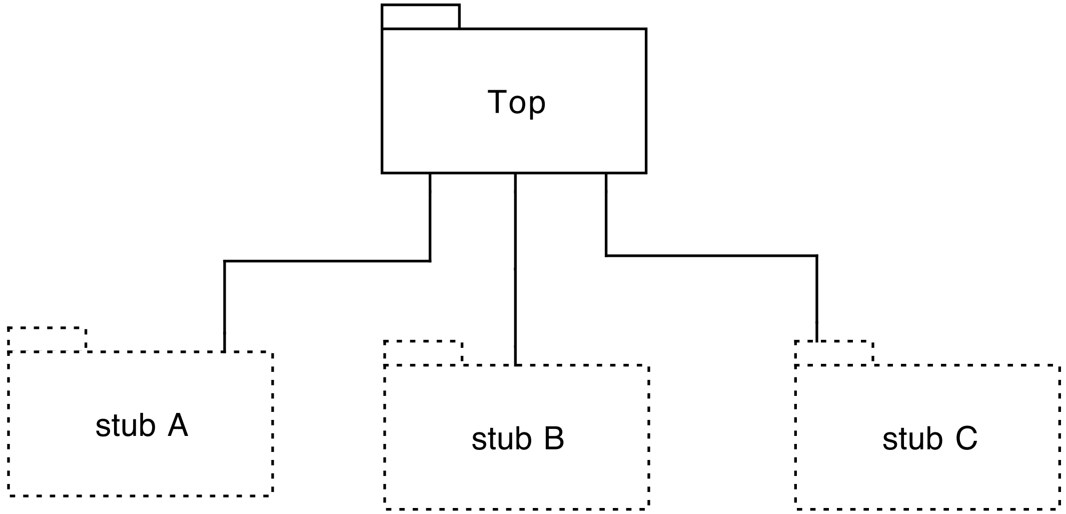
Integration testing: strategies

POLITECNICO
MILANO 1863

- Iterative and incremental strategies
  - run as soon as components are released (not just at the end)
  - Hierarchical: based on the hierarchical structure of the system
    - Top-down
    - Bottom-up
  - Threads: a portion of several modules that offers a user-visible function
  - Critical modules

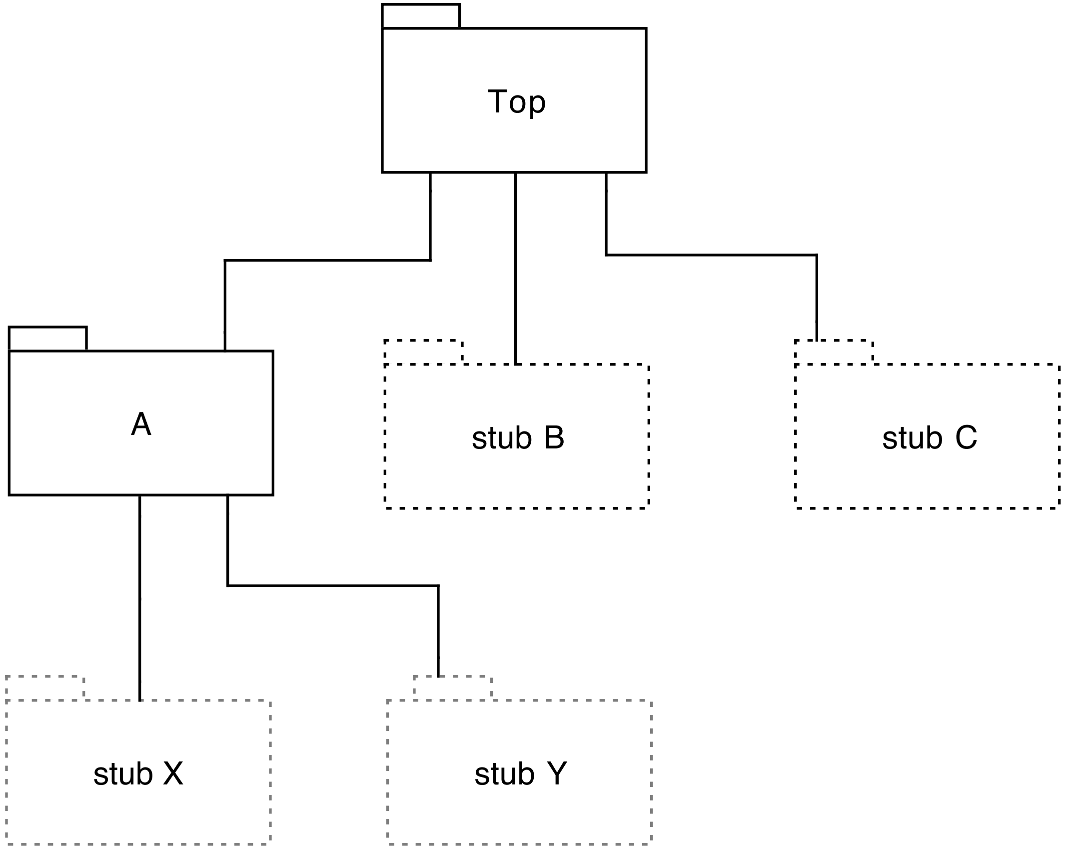
Integration testing: top-down

- Top-down strategy
  - Working from the top level (in terms of “use” or “include” relation) toward the bottom
  - Driver uses the top-level interfaces (e.g., CLI, REST APIs)
  - We need stubs of used modules at each step of the process



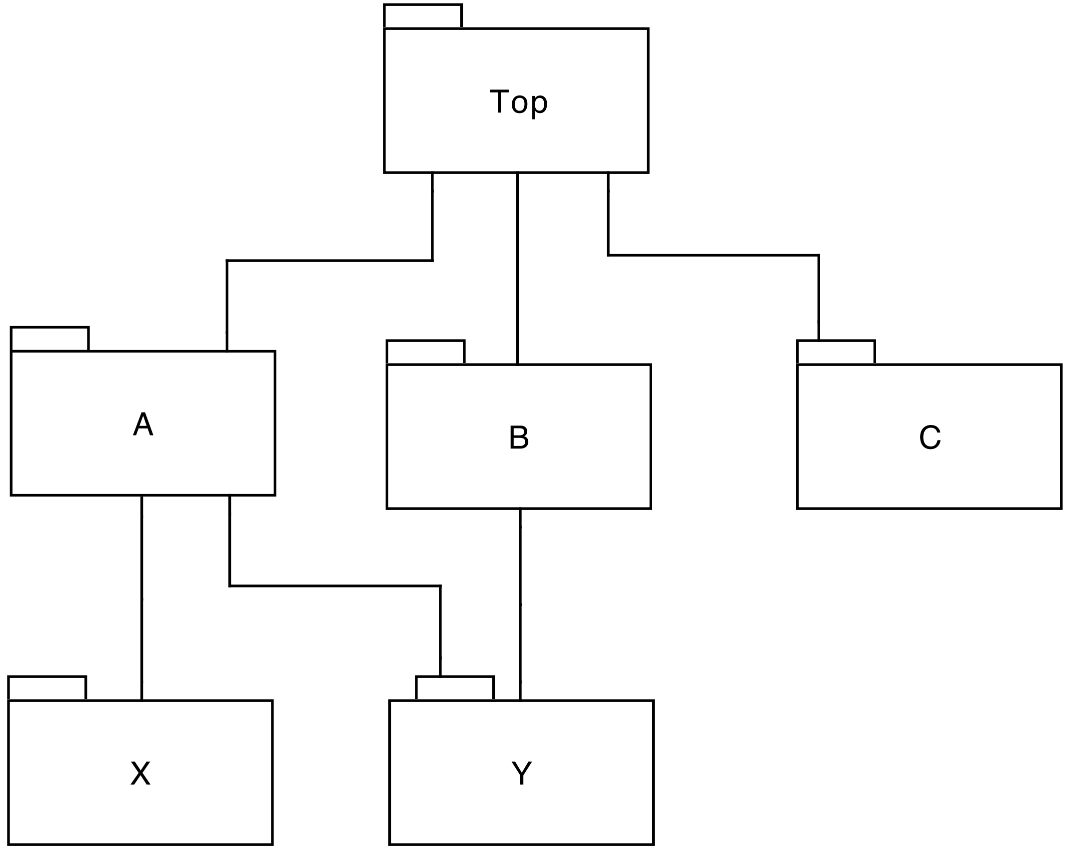
Integration testing: top-down

- Top-down strategy
  - As modules are ready (following the build plan) more functionality is testable
  - We replace some stubs and we need other stubs for lower levels



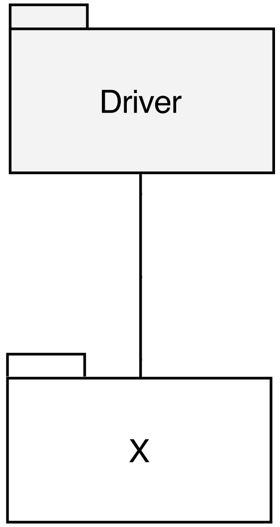
Integration testing: top-down

- Top-down strategy
  - When all modules are incorporated, the whole functionality can be tested



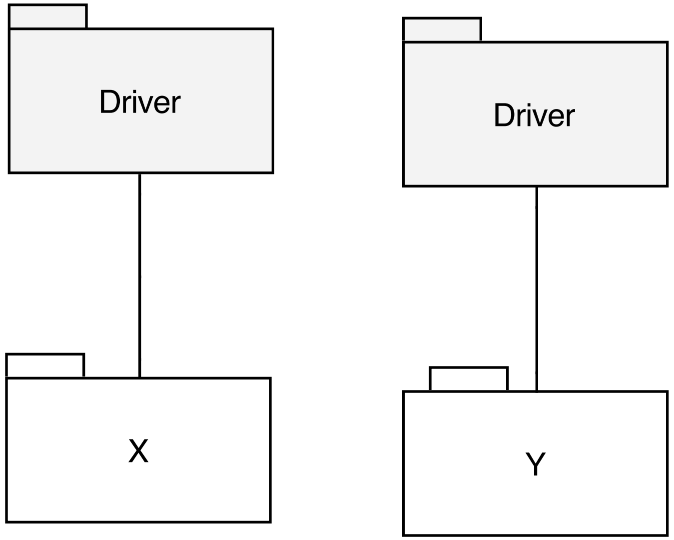
Integration testing: Bottom-up

- Bottom-up strategy
  - Starting from the leaves of the "uses" hierarchy
  - Does not need stubs



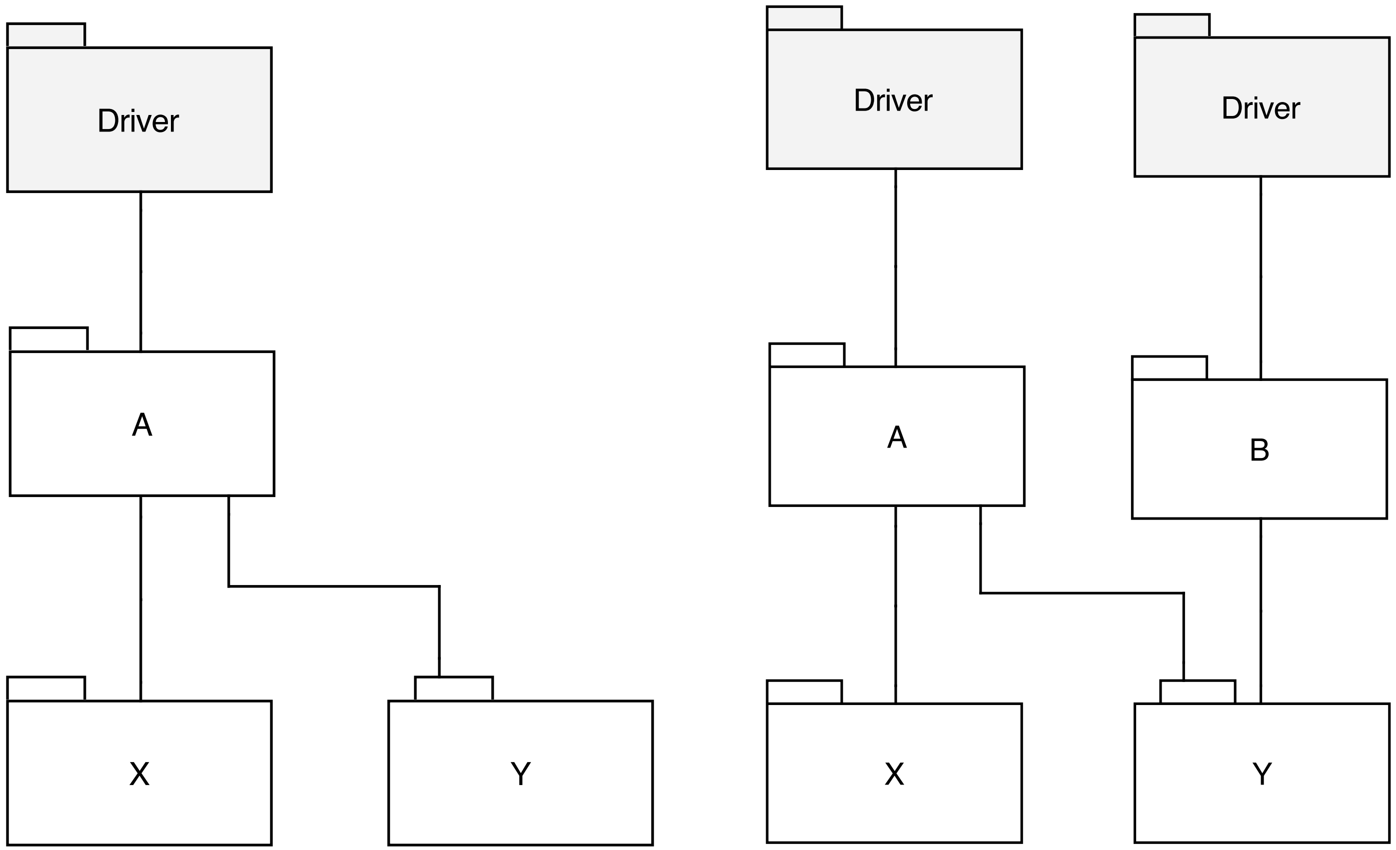
Integration testing: Bottom-up

- Bottom-up strategy
  - Starting from the leaves of the "uses" hierarchy
  - Does not need stubs
  - Typically requires more drivers: one for each module (as in unit testing)



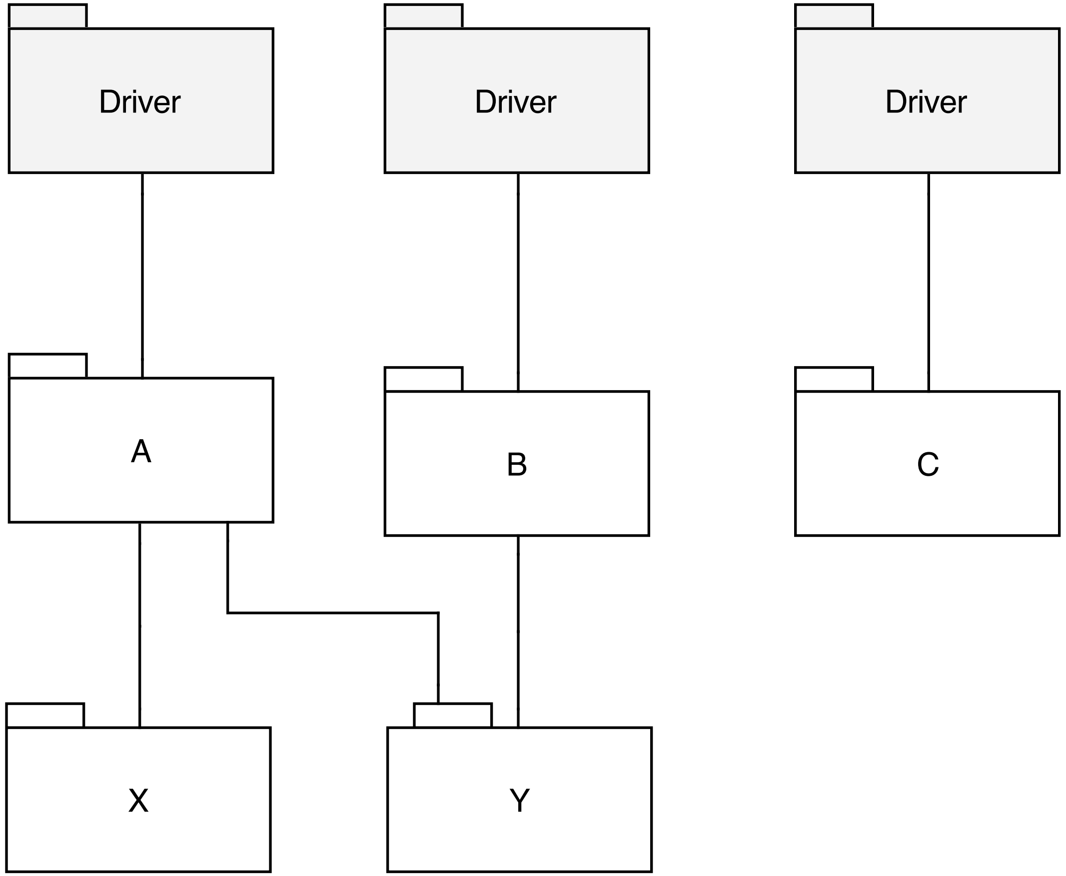
Integration testing: Bottom-up

- Bottom-up strategy
  - Newly developed module may replace an existing driver
  - New modules require new drivers



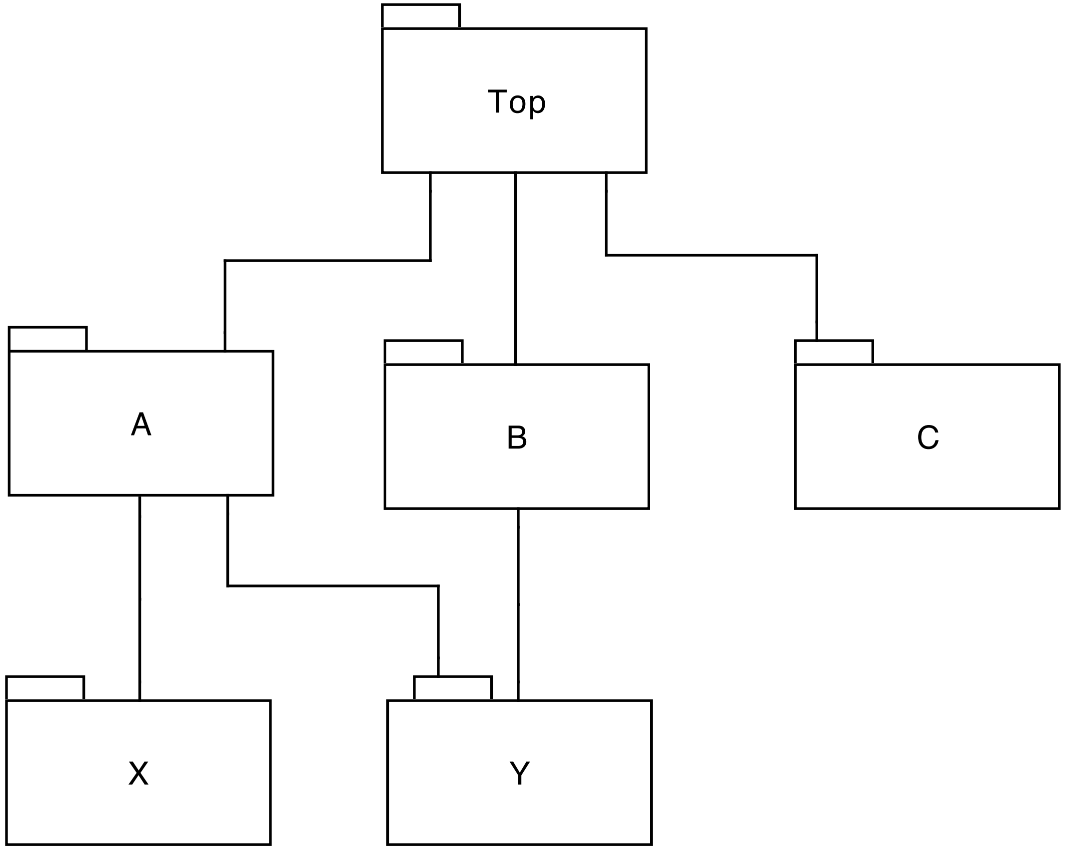
Integration testing: Bottom-up

- Bottom-up strategy
  - It may create several working subsystems



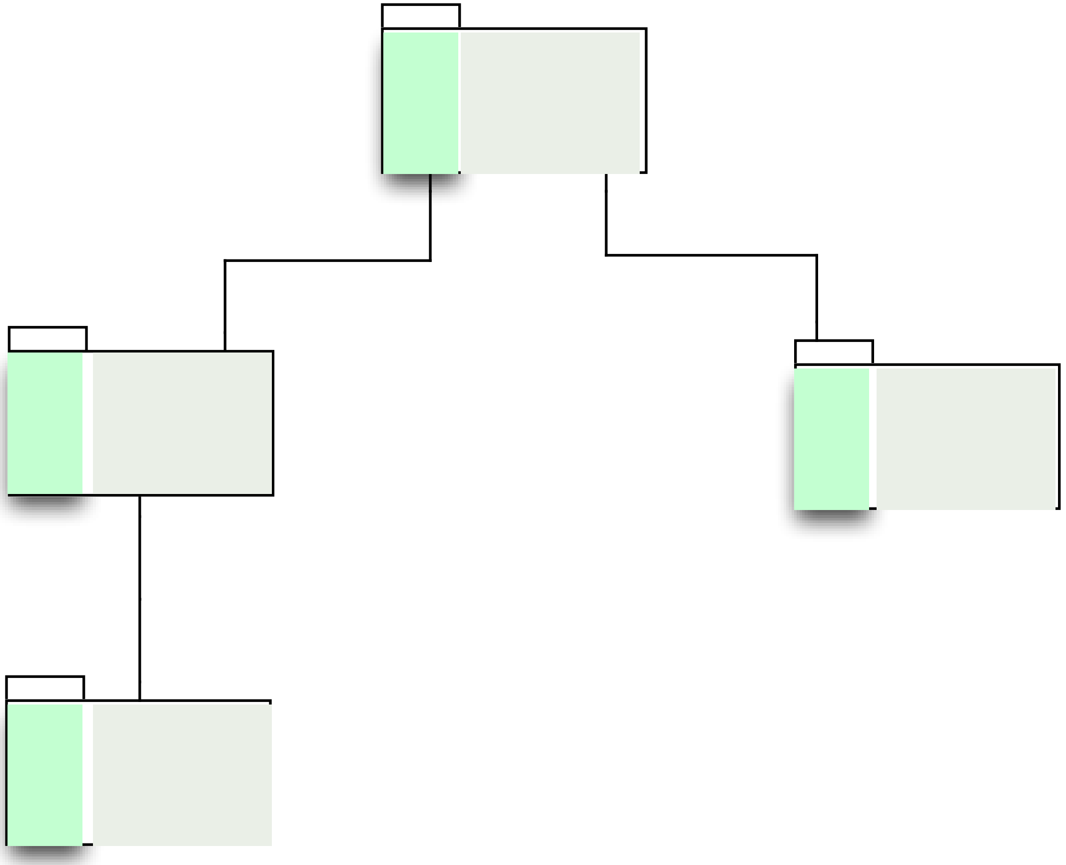
Integration testing: Bottom-up

- Bottom-up strategy
  - Working subsystems are eventually integrated into the final one



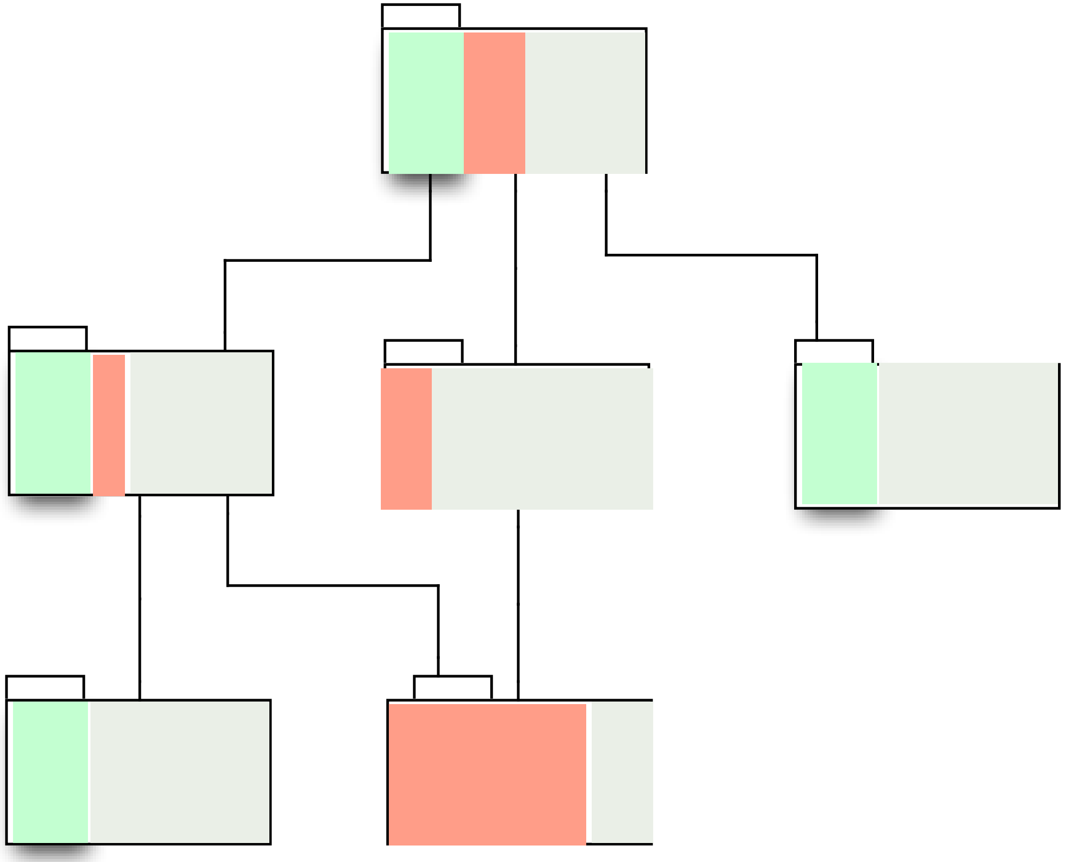
Integration testing: Threads

- Thread strategy
  - A thread is a portion of several modules that, together, provide a user-visible program feature



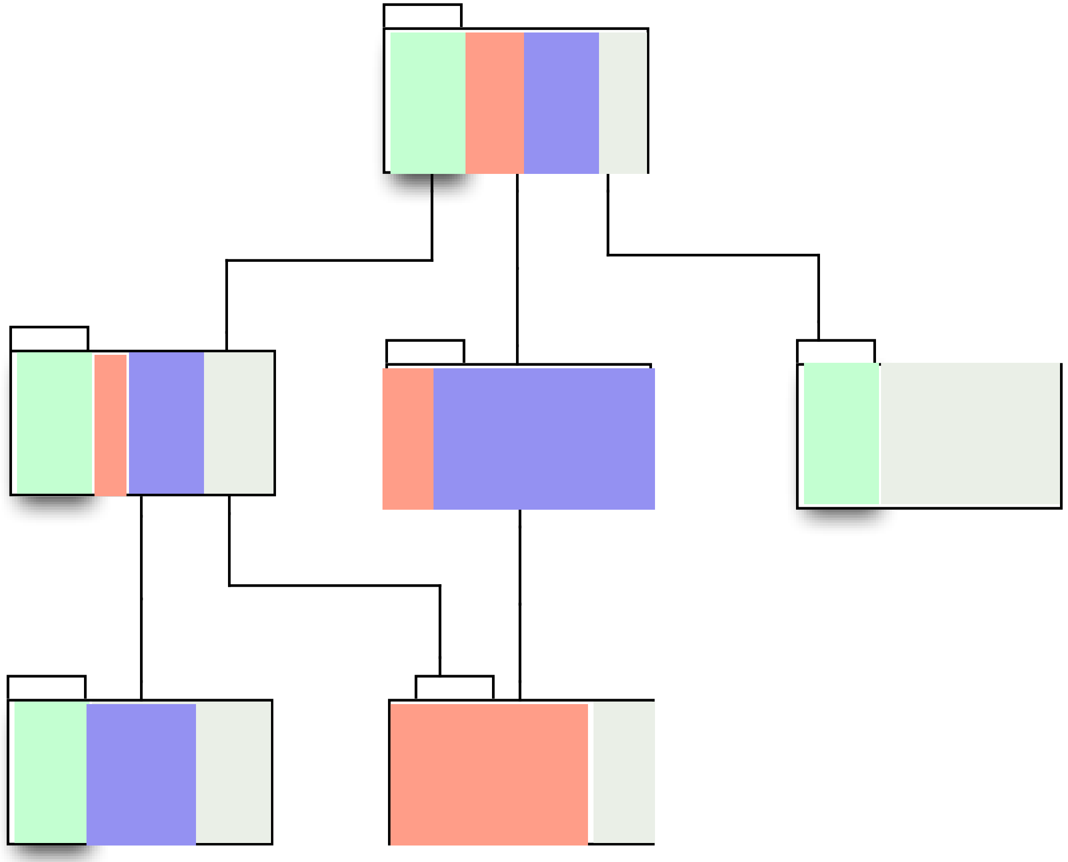
Integration testing: Threads

- Thread strategy
  - Integrating by thread maximizes visible progress for users (or other stakeholders)



Integration testing: Threads

- Thread strategy
  - Reduces drivers and stubs
  - Integration plan is typically more complex



Integration testing: critical modules

- Critical modules strategy
  - Start with modules having highest risk
    - Risk assessment is necessary first step
    - May include technical risks (is X feasible?), process risks (is schedule for X realistic?)
    - May resemble thread process with specific priority
  - Key point is risk-oriented process
    - Integration & testing as a risk-reduction activity, designed to deliver any bad news as early as possible

Integration testing: choosing a strategy

POLITECNICO MILANO 1863

- Structural strategies (bottom up and top down) are simpler
- Thread and critical modules strategies provide better external visibility on progress (especially in complex systems)
- Possible to combine different strategies
  - 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
  - Note: we can also combine threads and top-down/bottom-up

System (e2e) testing

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

System (e2e) testing: common types

- Functional testing
  - **Purpose**
    - Check whether the software meets the functional requirements
  - **How**
    - Use the software as described by use cases in the RASD, check whether requirements are fulfilled
- Performance testing
  - **Purpose**
    - Detect bottlenecks affecting response time, utilization, throughput
    - Detect inefficient algorithms
    - Detect hardware/network issues
    - Identify optimization possibilities
  - **How**
    - Load system with expected workload
    - Measure and compare acceptable performance

System (e2e) testing: common types

- Load testing
  - **Purpose**
    - Expose bugs such as memory leaks, mismanagement of memory, buffer overflows
    - Identify upper limits of components
    - Compare alternative architectural options
  - **How**
    - Test the system at increasing workload until it can support it
    - Load the system for a long period

  - Remember this piece of code?

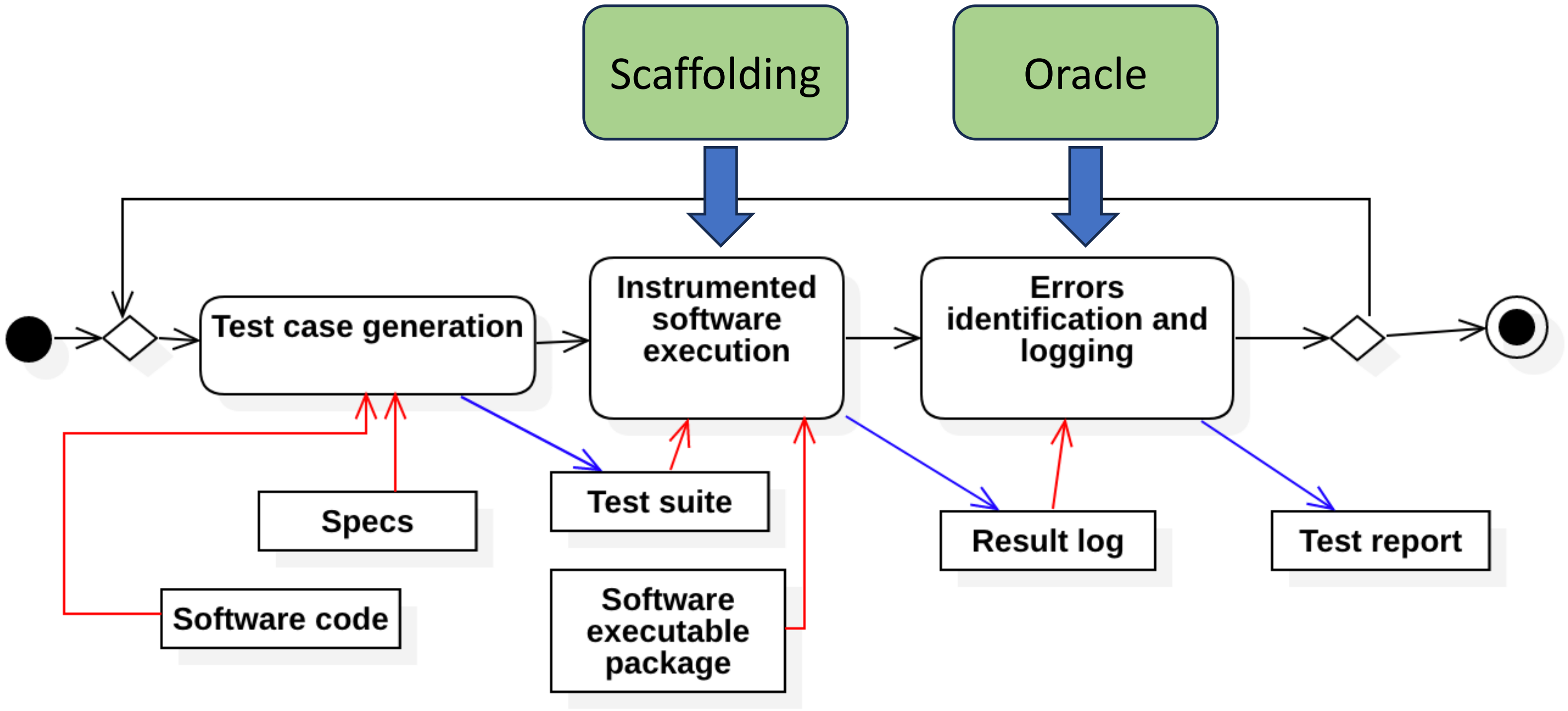
```
static void ssl_io_filter_disable(ap_filter_t *f){
    bio_filter_in_ctx_t *inctx = f->ctx;
    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

System (e2e) testing: common types

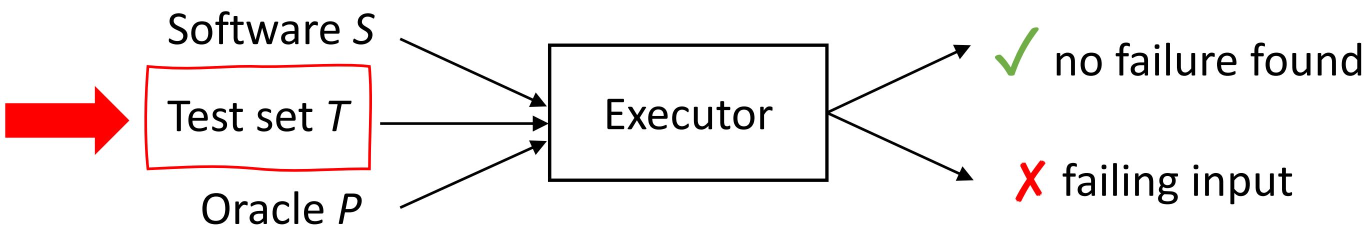
- Stress testing
  - **Purpose**
    - Make sure that the system recovers gracefully after failure
  - **How**
    - Trying to break the system under test by overwhelming its resources or by reducing resources
  - **Examples**
    - Double the baseline number for concurrent users/HTTP connections
    - Randomly shut down and restart ports on the network switches/routers that connect servers
  - See also **Chaos engineering** (e.g., https://netflix.github.io/chaosmonkey/)

Testing workflow

POLITECNICO
MILANO 1863



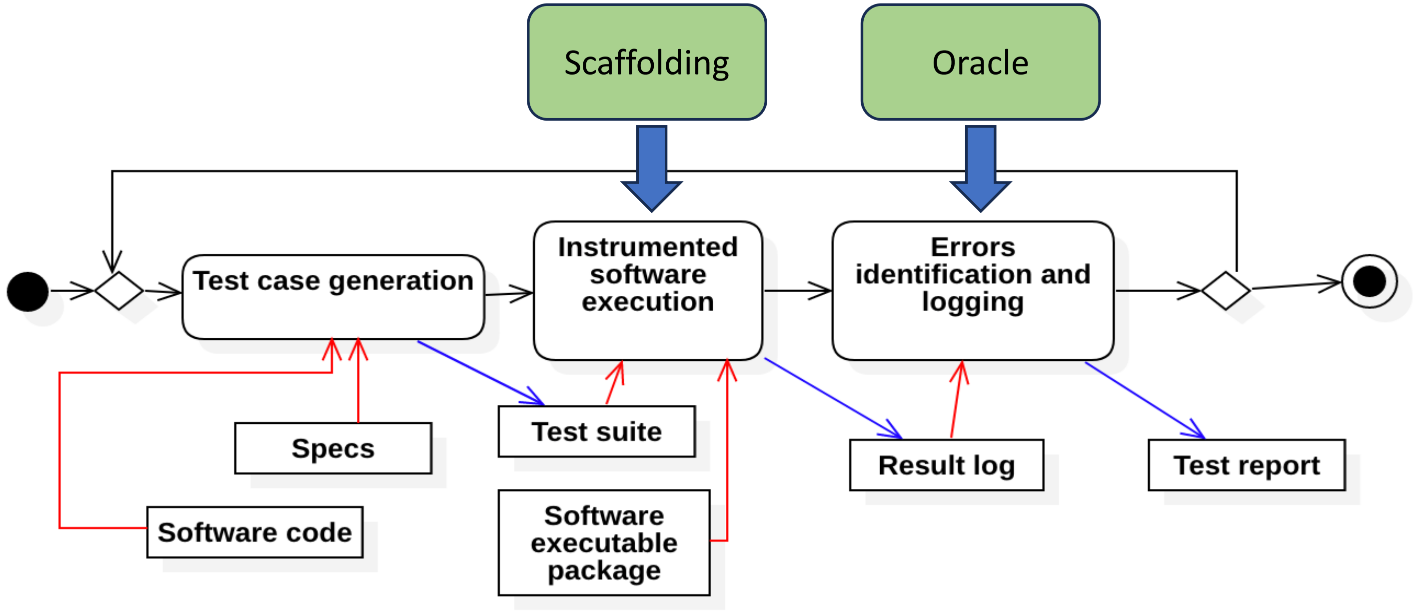
Test case generation



- Purpose: define *good quality* test sets
  - Showing a high probability of finding errors
  - Able to cover an acceptable amount of cases
  - Sustainable (we cannot run tests forewer…)

Test case generation

- Test cases can be generated in a black box or white box manner
  - **White box**: generation is based on code characteristics
  - **Black box**: generation is based on specs characteristics



Test case generation

- Test cases can be defined manually
- Test cases can be automatically generated (automated testing)
  - Combinatorial testing = enumerate all possible inputs following some policy (e.g., smaller to larger).. not in this course!
  - Concolic execution = pseudo-random generation of inputs guided by symbolic path properties
  - Fuzz testing (fuzzing) = pseudo-random generation of inputs including invalid, unexpected inputs
  - Search-based testing = explores the space of valid inputs looking for those that improve some metrics (e.g., coverage, diversity, failure inducing capability)

References

- Pezzè, M. and Young, M. Software testing and analysis: process, principles, and techniques. John Wiley & Sons, 2008. Available for free from here https://ix.cs.uoregon.edu/~michal/book/free.php