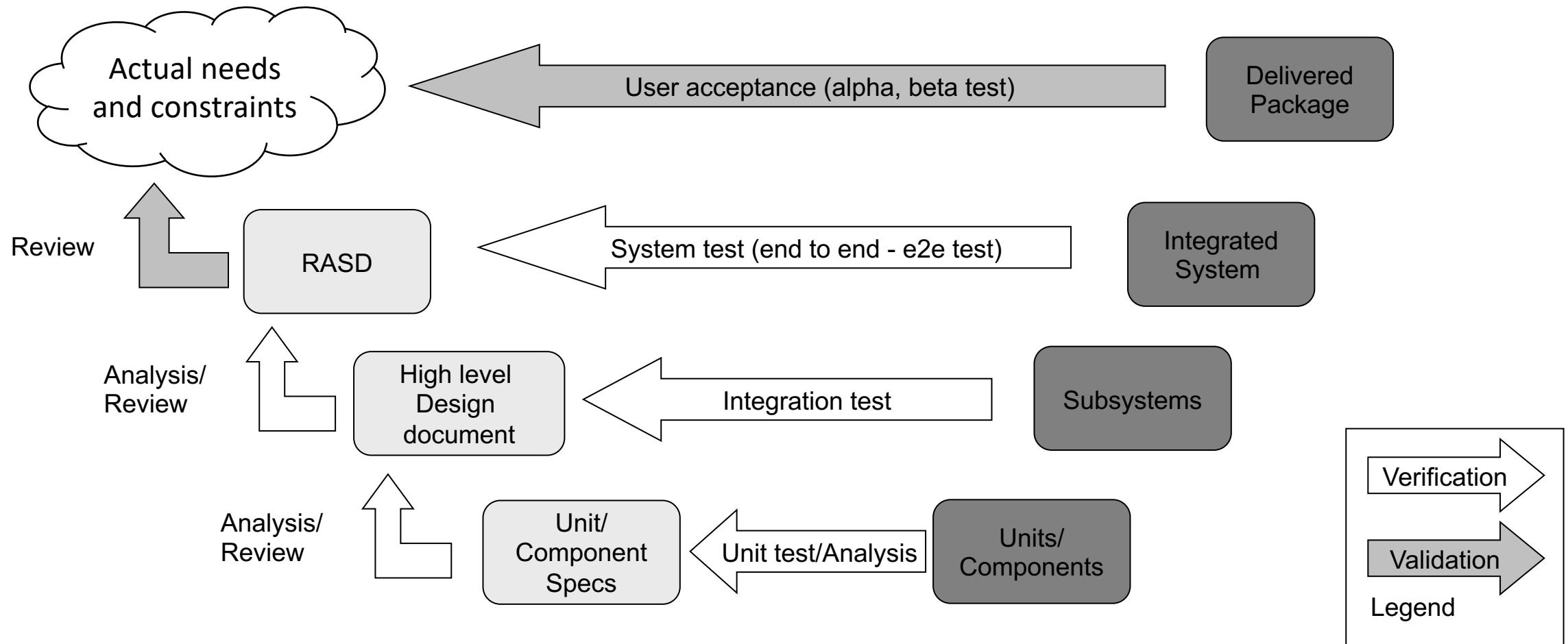


Verification & Validation

Analysis: symbolic execution

Testing: terminology, types of testing activities

Verification at which level? (V model)





Main approaches: static vs dynamic analysis

- **Static Analysis**

- Done on source code without execution
- Analysis is static but properties are dynamic

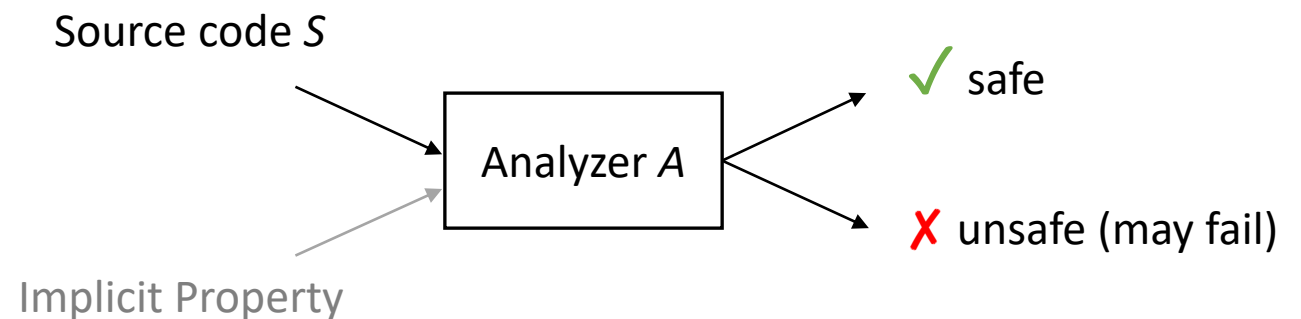
- **Testing (dynamic analysis)**

- Done by executing the sources (usually by sampling)
- Analysis of the actual behavior compared to an expected one

Static Analysis

- The very idea

- Analyzes the source code
- Each analyzer targets a fixed set of **hard-coded** (pre-defined, not custom) properties
- Completely **automatic**
- The output reports
 - **Safe** = no issues
 - **Unsafe** = potential issues



Static Analysis: properties

- Checked **properties** are often general safety properties (absence of certain conditions that may yield errors)
- Examples:
 - No **overflow** for integer variables
 - No **type errors**
 - No **null-pointer** dereferencing
 - No **out-of-bound** array accesses
 - No **race conditions**
 - No **useless assignments**
 - No **usage of undefined variables**
 - No **execution of specific paths**

Static analysis: succesful stories



[2017] “Our strategy at Uber has been to use *static* code analysis tools to prevent *null pointer exception* crashes.”

Engineering NullAway, Uber’s Open Source Tool for Detecting NullPointerExceptions on Android

<https://www.uber.com/en-IT/blog/nullaway/>



[2013] “Each month, hundreds of potential bugs identified by Facebook *Infer* are fixed [...] *before* they are [...] deployed to people’s phones.”

Facebook buys code-checking Silicon Roundabout startup Monoidics

<https://www.theguardian.com/technology/2013/jul/18/facebook-buys-monoidics>

More on Static vs Dynamic

- **Static**

- at **compile time** – before execution
- related to source **code** (or any other model of the software)
- **without execution** of the software
- on **generic (or symbolic) inputs**

- **Dynamic**

- at **runtime** – during execution
- related to software **behavior**
- **while executing** the software
- on **specific inputs**

- **Static analysis**: techniques, methods, tools used to infer properties of the dynamic behavior without explicitly running the software
 - It is a pessimistic technique

Static analysis tools

- Various tools available
- The analyses are language-specific but many tools support multiple programming languages
- The first static analysis tool was a Unix utility, Lint, developed in 1978 for C programs. From this, simple static analysis is also called **linting**
- Lists of currently available tools are available from various sources:
 - https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
 - <https://github.com/analysis-tools-dev/static-analysis>

Comparing some static analysis tools

<https://www.comparitech.com/net-admin/best-static-code-analysis-tools/>

Tool/Features	SonarQube	Checkmarx	Synopsys Coverity	Micro Focus Fortify SCA	Veracode Static Analysis	Snyk Code
Language Support	Multiple	Multiple	Multiple	Multiple	Multiple	Multiple
→ Integrations	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD
Free Trial	Yes	Yes	No	Yes	Yes	Yes
On-Premises/Cloud	Both	Both	Both	Both	Both	Cloud
Automated Scans	Yes	Yes	Yes	Yes	Yes	Yes
Compliance Reporting	Yes	Yes	Yes	Yes	Yes	Yes
Vulnerability Database	Yes	Yes	Yes	Yes	Yes	Yes
Real-Time Feedback	Yes	Yes	No	No	No	Yes



Example of issues report from SonarCloud/SonarQube

https://sonarcloud.io/project/issues?resolved=false&id=aws_aws-sdk-java-v2

The screenshot displays the SonarCloud web interface for the 'AWS Java SDK :: Parent' project. The left sidebar shows navigation options: Overview, Main Branch, Pull Requests (60), and Branches (2). The main content area is divided into a 'Filters' panel on the left and a list of issues on the right. The 'Filters' panel includes sections for 'Clean Code Attribute' (Consistency: 704, Intentionality: 4.9k, Adaptability: 443, Responsibility: 7), 'Software Quality' (Security: 7, Reliability: 86, Maintainability: 6k), and 'Severity' (High: 527, Medium: 1.3k). The issues list shows three items, each with a title, description, tags, and effort. The first issue is an 'Intentionality issue' titled 'Replace this if-then-else statement by a single return statement.' with a 'Maintainability' tag and 2min effort. The second issue is an 'Intentionality issue' titled 'Add a default case to this switch.' with 'Maintainability' and 'Critical' tags and 5min effort. The third issue is an 'Adaptability issue' titled 'This class has 7 parents which is greater than 5 authorized.' with a 'Maintainability' tag and 5h effort.

sonarcloud

My Projects My Issues Explore

AWS Java SDK :: Parent

Amazon Web Services > AWS Java SDK :: Parent > master

Summary **Issues** Security Hotspots Measures Code Activity

The last analysis has warnings. [See details](#)

6,078 issues 82d effort

Select issues Navigate to issue

build-tools/.../awssdk/buildtools/checkstyle/NonJavaBaseModuleCheck.java

Intentionality issue

[Replace this if-then-else statement by a single return statement.](#)

No tags

Open Matthew Miller Maintainability Code Smell Minor 2min effort • 1 year ago

build-tools/.../awssdk/buildtools/checkstyle/SdkIllegalImportCheck.java

Intentionality issue

[Add a default case to this switch.](#)

No tags

Open Not assigned Maintainability Code Smell Critical 5min effort • 2 years ago

build-tools/.../awssdk/buildtools/checkstyle/SdkPublicMethodNameCheck.java

Adaptability issue

[This class has 7 parents which is greater than 5 authorized.](#)

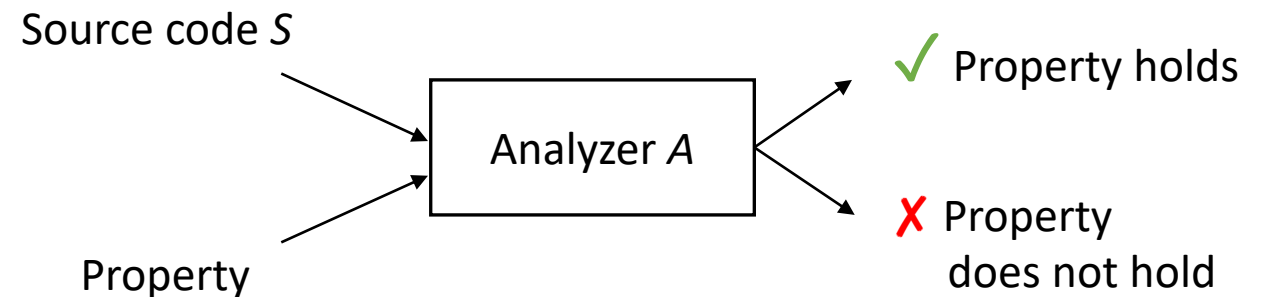
design

Open Not assigned Maintainability Code Smell Major 5h effort • 5 years ago

build-tools/.../awssdk/buildtools/findbugs/DisallowMethodCall.java

Symbolic execution

- **The very idea**
 - Analyzes **real** source code
 - Analyzes **reachability** and **path feasibility** properties
 - Is **automatic**
 - **May fail** to analyze all possible paths
 - Can be used to **support testing**



Types of checked properties

- **Reachability**: does some execution of the program reach the location l in S ?
 - Symbolic exec tries to **verify that l cannot be reached**, or alternatively **spots the condition under which l can be reached**

```
...
k:    try {
k+1:  ...
l-1:  } catch (e) {
l:    /* error */
...   }

...
l-1:  if (x < 0) {
l:    /* safe */
```

- **Path feasibility**: Is the given path p feasible?
 - Symbolic exec tries to **verify that p cannot be executed**, or alternatively **spots the condition under which p can be executed**

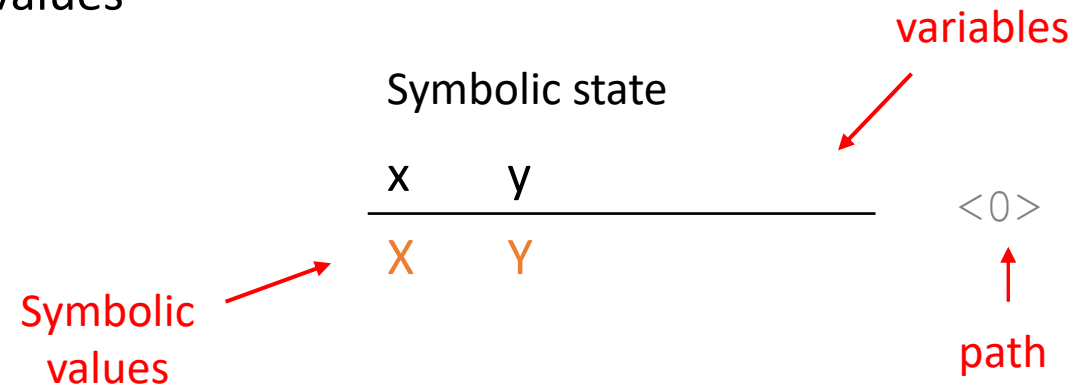
$$p = \langle 0, 1, \dots, k, \dots, n \rangle$$

Symbolic state and path condition

- Symbolic execution **executes** programs on **symbolic** values
 - **Symbolic states** keep track of the (symbolic) value of variables

Inputs are **initialized** with symbolic (generic) values

```
0: void foo(int x, int y) {  
1:   ...
```



Symbolic state and path condition

- Symbolic execution **executes** programs on **symbolic** values
 - **Symbolic states** keep track of the (symbolic) value of variables

Executing a statement **updates** the symbolic **state**

```
0: void foo(int x, int y) {  
1:   int z := x
```

Symbolic state

x	y	z	<0, 1>
X	Y	X	

Symbolic state and path condition

- Symbolic execution **executes** programs on **symbolic** values
 - **Symbolic states** keep track of the (symbolic) value of variables

Executing a branch **splits** the symbolic **state**

A **path condition** π represents the constraint of a path

```
0: void foo(int x, int y) {
1:   int z := x
2:   if (z < y)
```

Symbolic state				
Condition at point 2 true:	x	y	z	π
	X	Y	X	$X < Y$
Condition at point 2 false:	x	y	z	π
	X	Y	X	$X \geq Y$

<0, 1, 2>

Symbolic state and path condition

- Symbolic execution **executes** programs on **symbolic** values
 - **Symbolic states** keep track of the (symbolic) value of variables

The **execution continues** along **feasible paths** (path condition π is satisfiable)

```
0: void foo(int x, int y) {  
1:   int z := x  
2:   if (z < y)  
3:     z := z*2
```

Symbolic state

x	y	z	π	<0, 1, 2, 3>
X	Y	2X	X<Y	

Symbolic state and path condition

- Symbolic execution **executes** programs on **symbolic** values
 - **Symbolic states** keep track of the (symbolic) value of variables

The **execution continues** along **feasible paths** (path condition π is satisfiable)

```

0: void foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
4:   if (x < y && z >= y)

```

Condition at
point 4 true:

Symbolic state

x	y	z	π	
X	Y	2X	$X < Y$	$\langle 0, 1, 2, 3, 4 \rangle$
$X < Y \wedge 2X \geq Y$				

Condition at
point 4 false:

x	y	z	π	
X	Y	2X	$X < Y$	$\langle 0, 1, 2, 3, 4 \rangle$
$X \geq Y \vee 2X < Y$				

Final states

- Possible outcomes of symbolic execution:
 - **SAT** exit (π is satisfiable): any satisfying assignment to variables in π is an **input** that satisfies the given property in a **concrete execution**
 - **UNSAT** exit (π is not satisfiable): the given property cannot be satisfied by **any concrete execution**
- **Example:** is location 5 reachable?

```

0: int foo(int x, int y) {
1:     int z := x
2:     if (z < y)
3:         z := z*2
4:     if (x < y && z >= y)
5:         print(z) //location
6: }
```

x	y	z	π	
X	Y	2X	$X < Y$	$\langle 0, 1, 2, 3, 4, 5 \rangle$
			$X < Y \wedge 2X \geq Y$	

SAT exit

Example of satisfying assignment: $X = 2, Y = 3$

Final states

- Possible outcomes of symbolic execution:
 - **SAT** exit (π is satisfiable): any satisfying assignment to variables in π is an **input** that satisfies the given property in a **concrete execution**
 - **UNSAT** exit (π is not satisfiable): the given property cannot be satisfied by **any concrete execution**
- **Example:** is path $\langle 0, 1, 2, 4, 5 \rangle$ feasible?

```

0: int foo(int x, int y) {
1:     int z := x
2:     if (z < y)
3:         z := z*2
4:     if (x < y && z >= y)
5:         print(z) //location
6: }
```

x	y	z	π	
X	Y	X	$X \geq Y$	$\langle 0, 1, 2, 4 \rangle$
			$X < Y \wedge X \geq Y$	

UNSAT exit

There is **no satisfying** assignment

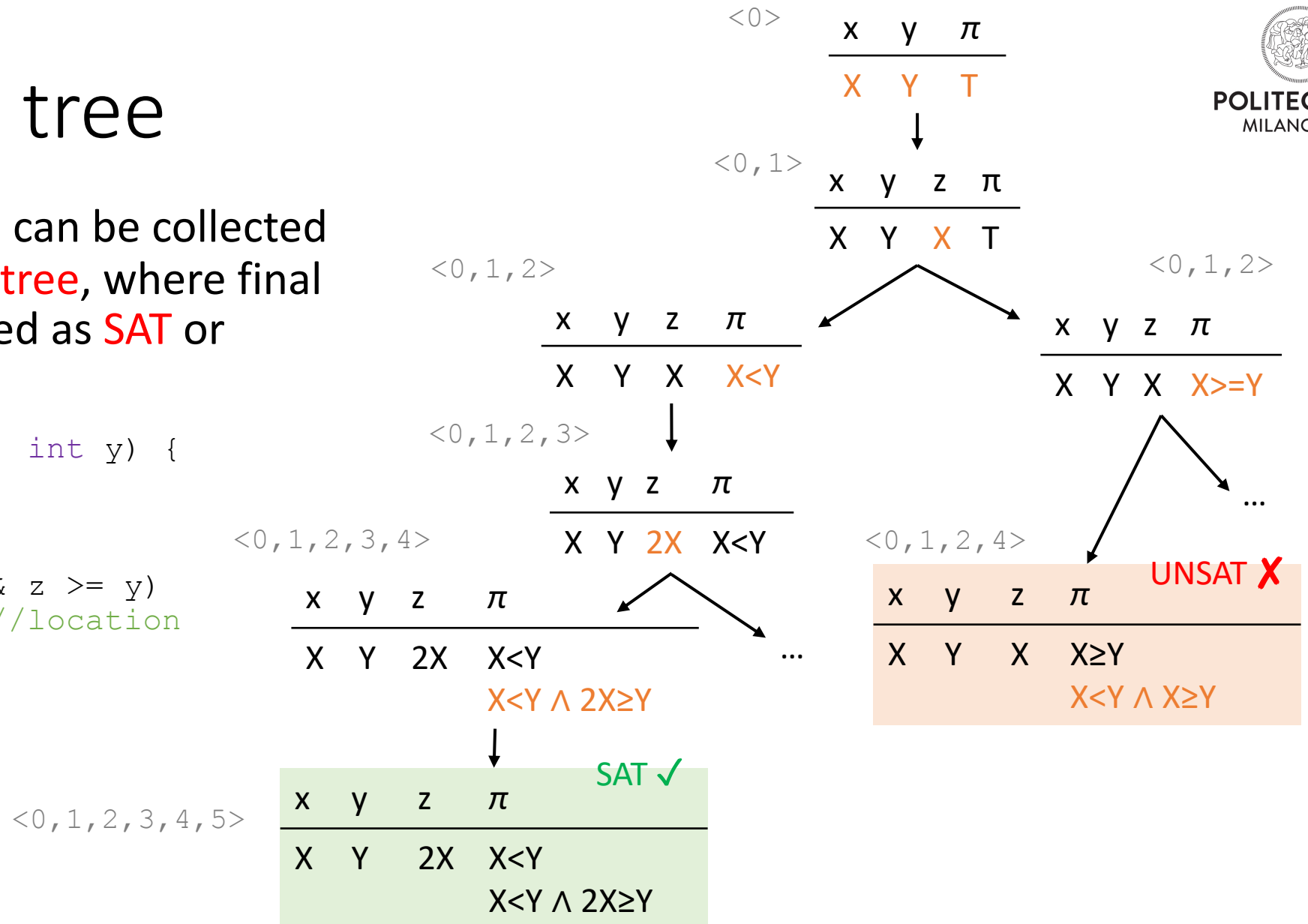
Execution tree

- Execution paths can be collected in an **execution tree**, where final states are marked as **SAT** or **UNSAT**

```

0: int foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
4:   if (x < y && z >= y)
5:     print(z) //location
6: }

```



Symbolic execution: exercise

- Consider the following program `bar`. Is the path $\langle 0, 1, 2, 3, 4, 5, 8, 2, 3, 4, 7, 8, 2, 10, 11 \rangle$ feasible?

```

0:  int bar( ) {
1:      int x := input()
2:      while (x > 0) {
3:          int y := 2*x
4:          if (x > 10)
5:              y := x - 1
6:          else
7:              x := x + 2
8:              x := x - 1
9:      }
10:  x := x - 1
11:  return x
12: }

```

$\langle 0 \rangle$	$\langle 0, 1 \rangle$	$\langle 0, 1, 2 \rangle$	$\langle 0, 1, 2, 3 \rangle$
$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$
$\quad \quad \quad \top$	$\quad \quad \quad \top$	$\quad \quad \quad \top$	$\quad \quad \quad \top$
$\langle 0, 1, 2, 3, 4 \rangle$	$\langle 0, 1, 2, 3, 4, 5 \rangle$	$\langle 0, 1, 2, 3, 4, 5, 8 \rangle$	
$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	
$\quad \quad \quad \top$	$\quad \quad \quad \top$	$\quad \quad \quad \top$	
$\langle 0, 1, 2, 3, 4, 5, 8, 2 \rangle$	$\langle 0, 1, 2, 3, 4, 5, 8, 2, 3 \rangle$	$\langle 0, 1, 2, 3, 4, 5, 8, 2, 3, 4 \rangle$	
$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	$\frac{x \quad y \quad \pi}{\quad \quad \quad}$	
$\quad \quad \quad \top$	$\quad \quad \quad \top$	$\quad \quad \quad \top$	

Symbolic execution: exercise

- Consider the following program `bar`. Is the path `<0 1 2 3 4 5 8 2 3 4 7 8 2 10 11>` feasible?

```

0:  int bar( ) {
1:      int x := input()
2:      while (x > 0) {
3:          int y := 2*x
4:          if (x > 10)
5:              y := x - 1
6:          else
7:              x := x + 2
8:              x := x - 1
9:      }
10:     x := x - 1
11:     return x
11: }

```

`<0,1,2,3,4,5,8,2,3,4>`

x	y	π
X-1	2(X-1)	X>10
		X≤11

`<0,1,2,3,4,5,8,2,3,4,7>`

x	y	π
X+1	2(X-1)	X>10
		X≤11

`<0,1,2,3,4,5,8,2,3,4,7,8>`

x	y	π
X	2(X-1)	X>10
		X≤11

`<0,1,2,3,4,5,8,2,3,4,7,8,2>`

x	y	π
X	2(X-1)	X>10
		X≤11
		X≤0

- Conclusion: path `<0,1,2,3,4,5,8,2,3,4,7,8,2,10,11>` is unfeasible

Symbolic execution: weaknesses

- It seems symbolic execution can be used to verify the correctness of any program, however...
- **Limitations**
 - Path **conditions** may be **too complex** for constraint solvers
 - Solvers are very good at checking **linear** constraints
 - It is harder for them to reason on non-linear arithmetic, bit-wise operations, string manipulation
 - **Impossible/hard to use** when number of paths to be explored is **infinite/huge**
 - **unbounded** loops give rise to **infinite** sets of paths
 - Even if set of paths is finite, checking all **loops is expensive/unfeasible in practice**
 - rule of thumb: approximate the analysis by considering 0, 1, and 2 iterations
 - There may be **external code**
 - Sources not available (e.g., pre-compiled library) → **unknown behavior** for the solver



POLITECNICO
MILANO 1863

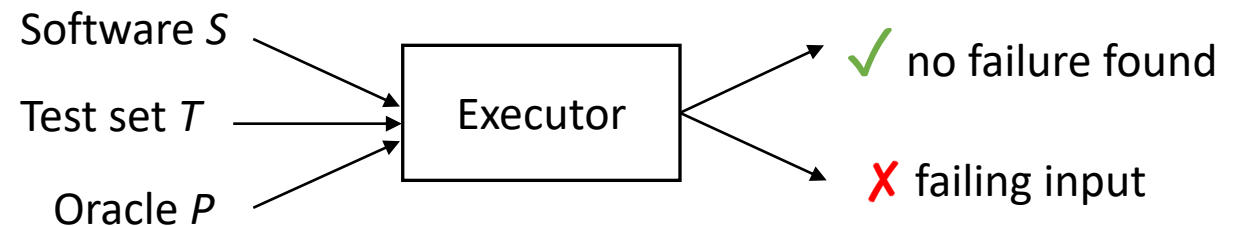
What is the goal of testing?

The goal of testing is
making programs fail.

Testing, aka dynamic analysis

- 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 **main** goal of testing is **making programs fail**

- **Other common goals**

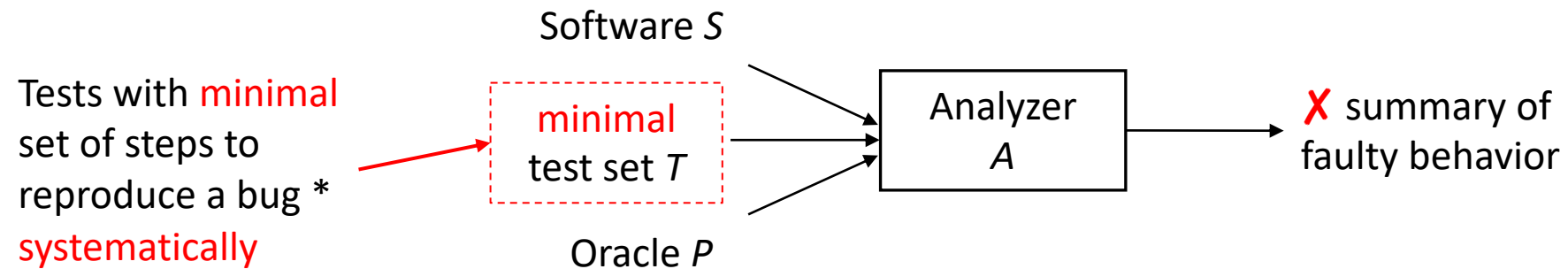
- Exercise different parts of a program to increase coverage
- Make sure the interaction between components works (integration testing)
- Support fault localization and error removal (debugging)
- Ensure that bugs introduced in the past do not happen again (regression testing)

- **Important note**

- *“Program testing can be used to show the presence of bugs, but never to show their absence!” (Edsger W. Dijkstra)*

Debugging

- Systematic approaches to fault localization + error removal
 - Testing output is often used to support **debugging**



* sometimes, also tests that do not trigger a failure but are similar to the failure-inducing ones

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”*

Unit testing

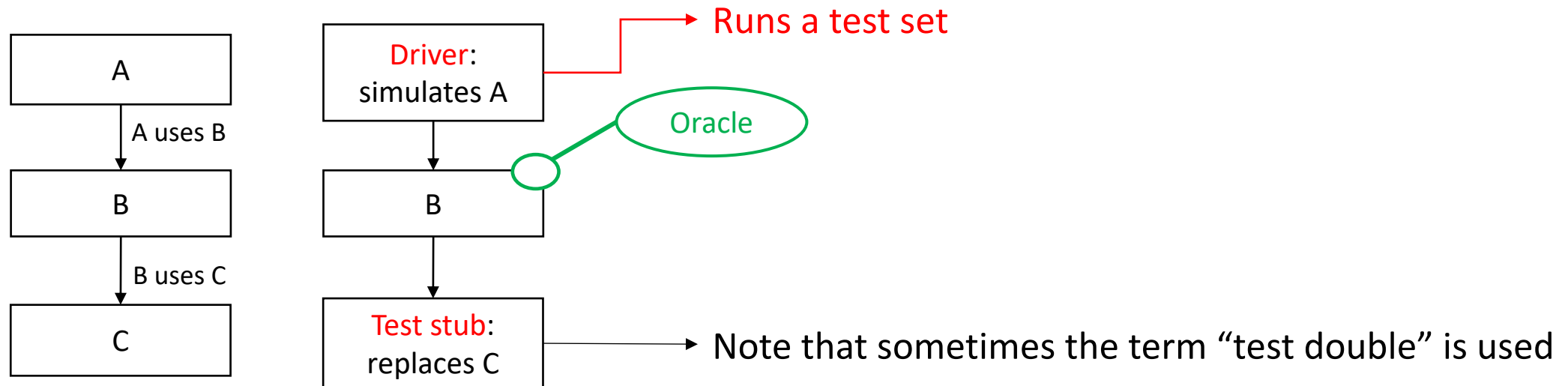
- Conducted by the developers
- Aimed at **testing small pieces** (units) of code in **isolation**
 - The notion of "unit" typically depends on the programming language (e.g., class, method, function, procedure)
- Why unit testing?
 - Find problems early

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

- 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
 - Concurrency-specific problems (next lecture)

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?

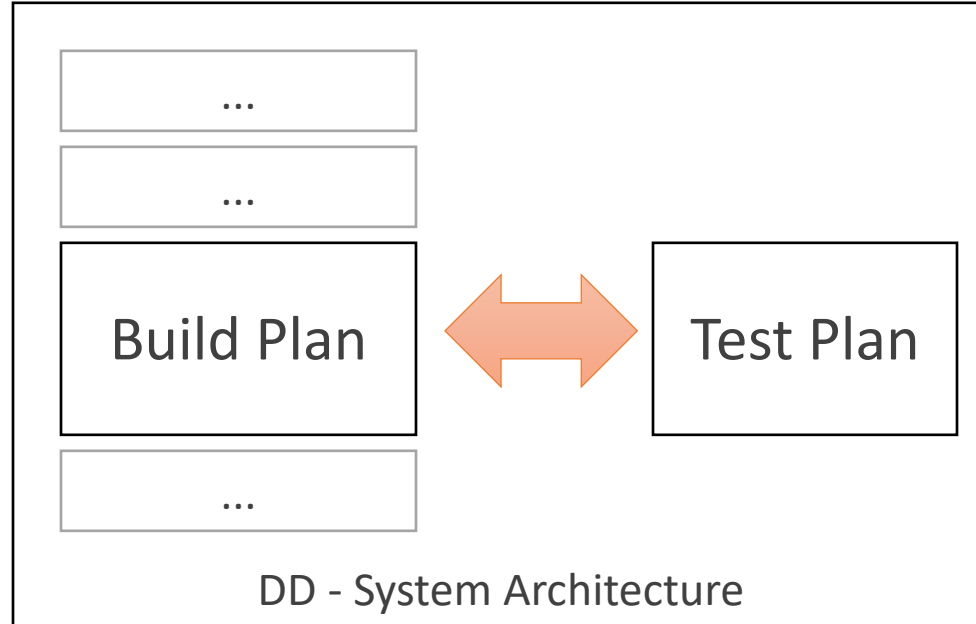
```
static void ssl_io_filter_disable(ap_filter_t *f) {  
    bio_filter_in_ctx_t *inctx = f->ctx;  
  
    inctx->ssl = NULL;  
    inctx->filter_ctx->pssl = NULL;  
}
```


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

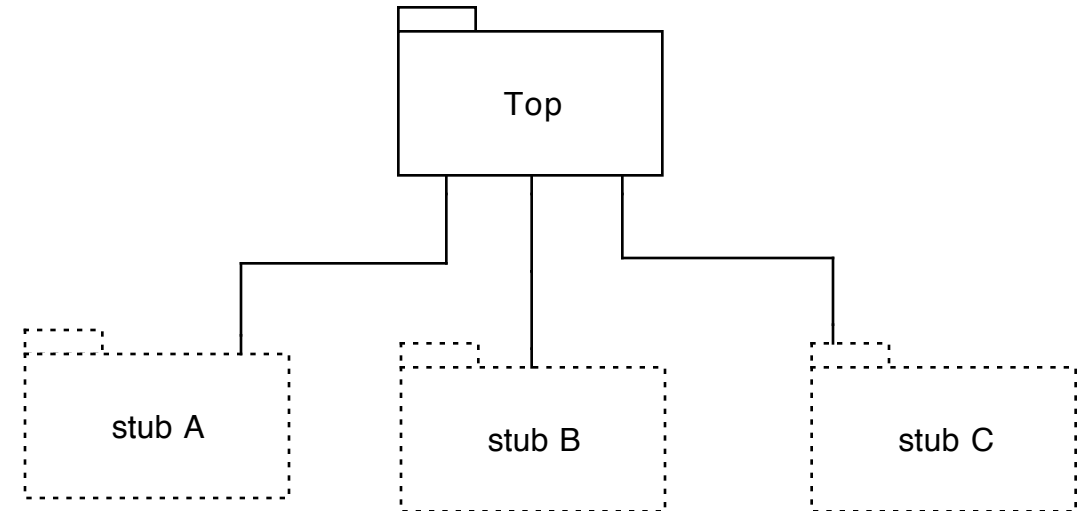
- **Big bang**: test only after integrating all modules together (not even a real strategy)
 - **Pros**
 - Does not require stubs, requires less drivers/oracles
 - **Cons**
 - Minimum observability, fault localization/diagnosability, efficacy, feedback
 - High cost of repair
 - Recall: Cost of repairing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

Integration testing: strategies

- **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

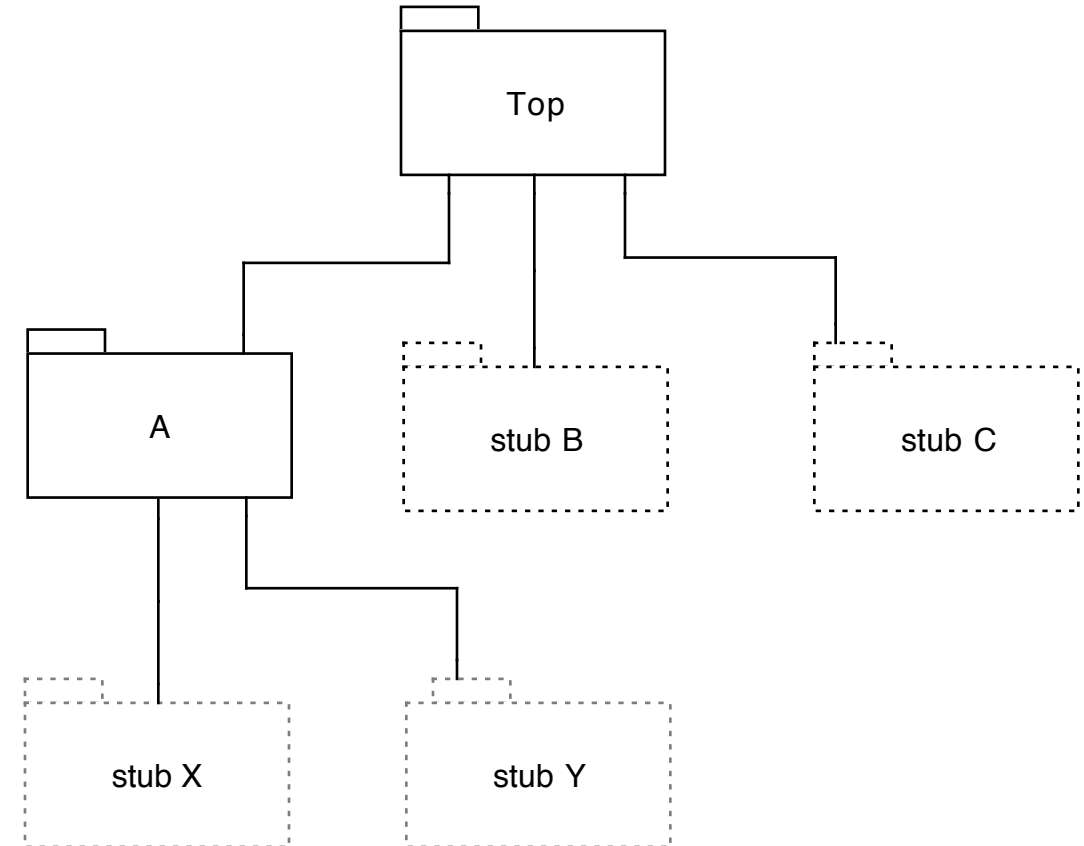
Integration testing: top-down

- **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



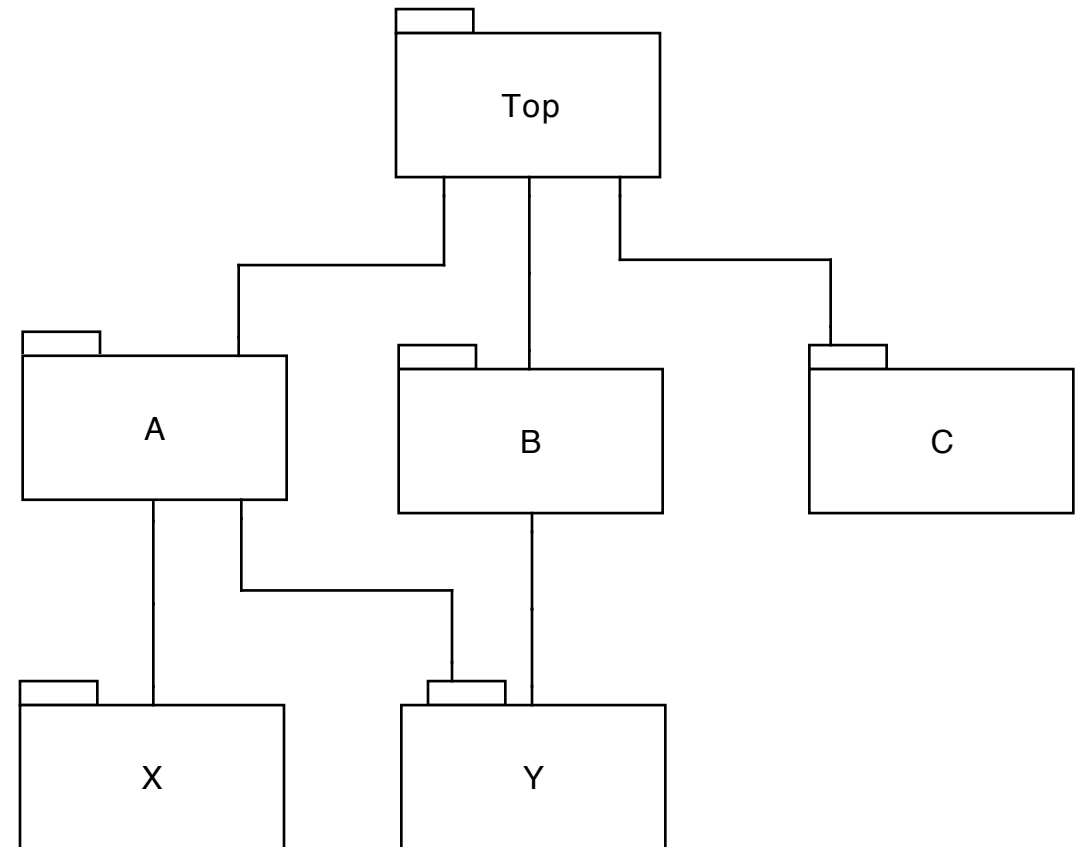
Integration testing: top-down

- **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



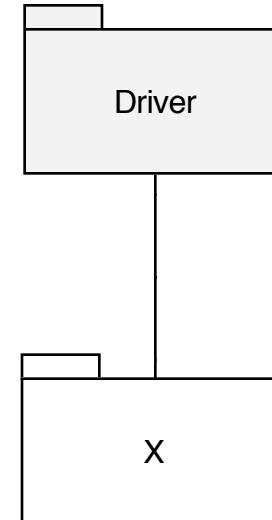
Integration testing: top-down

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



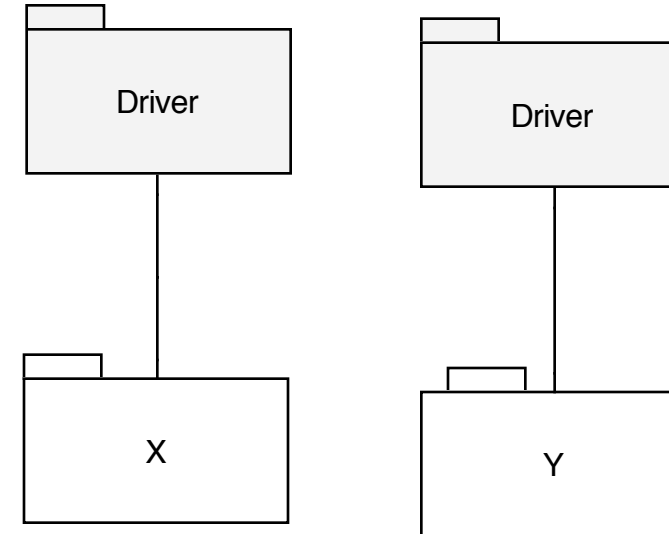
Integration testing: Bottom-up

- **Bottom-up strategy**
 - Starting from the leaves of the “uses” hierarchy
 - Does not need stubs



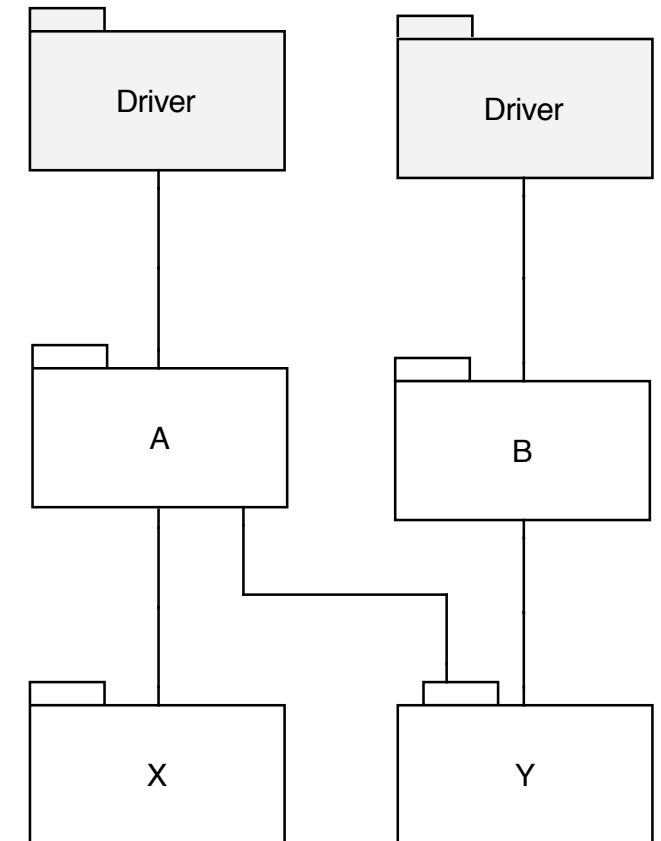
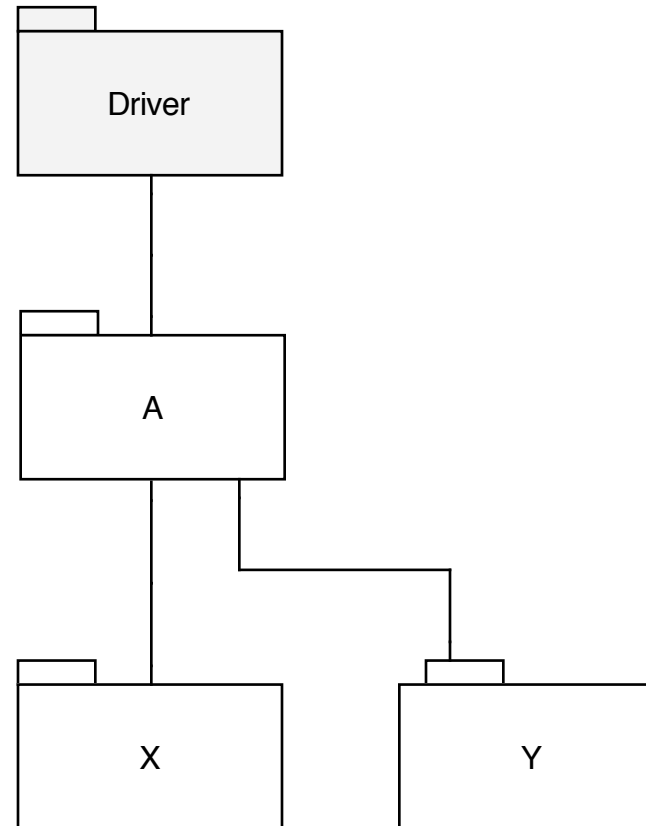
Integration testing: Bottom-up

- **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)



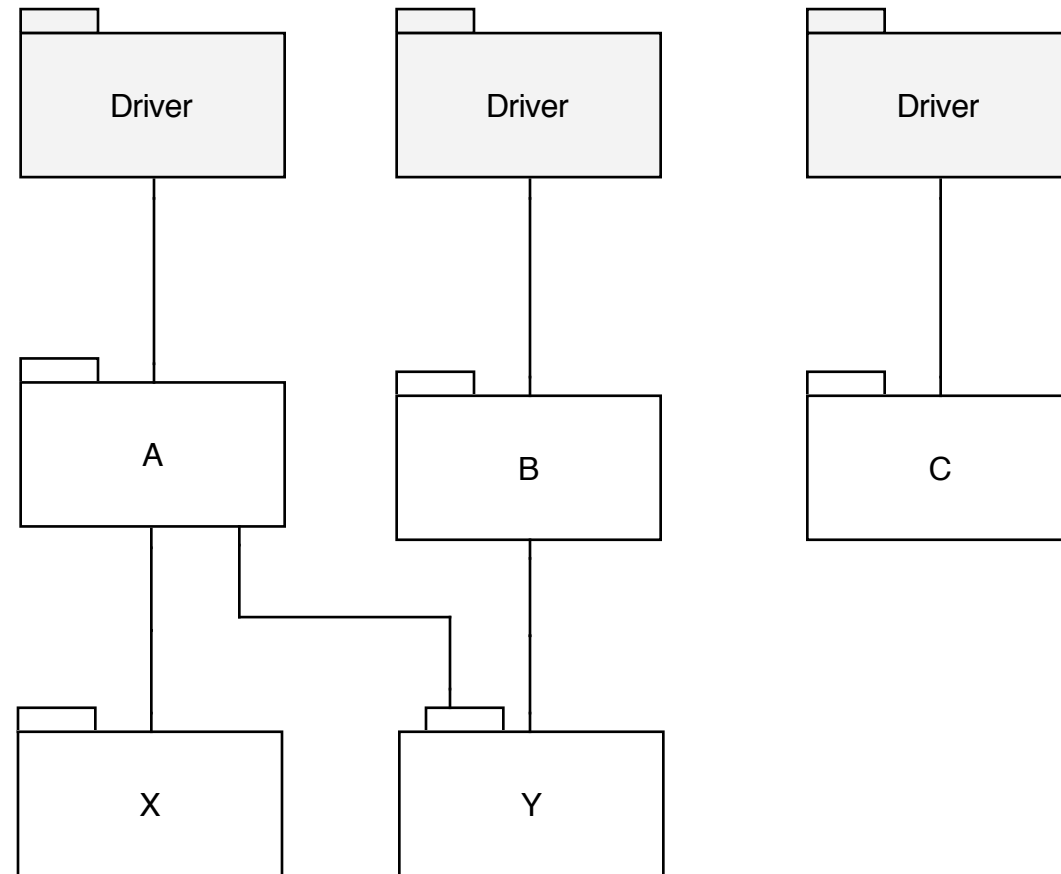
Integration testing: Bottom-up

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



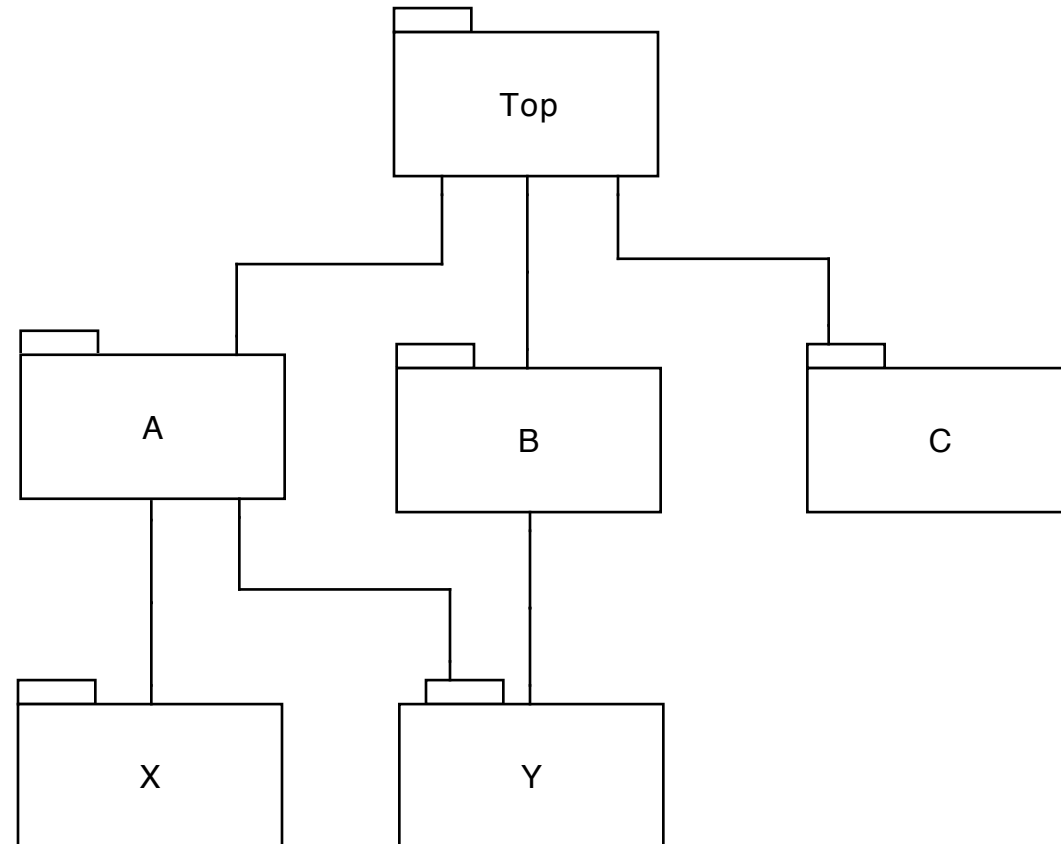
Integration testing: Bottom-up

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



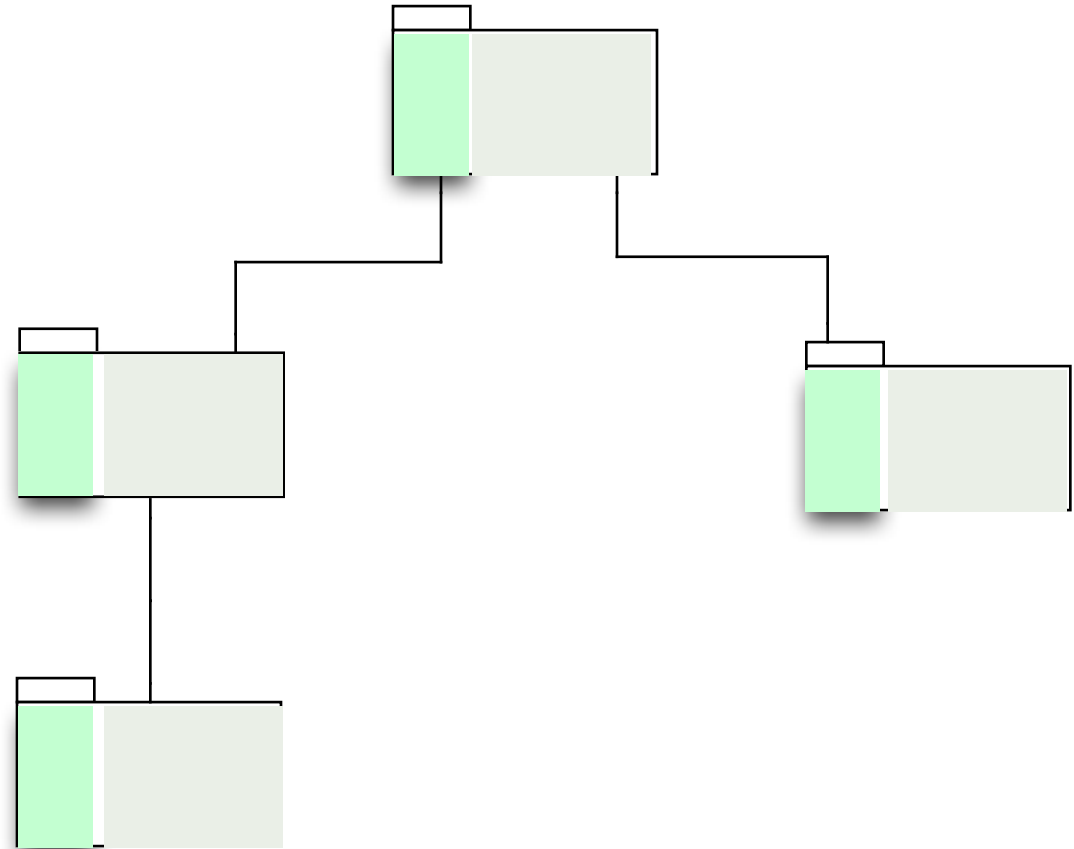
Integration testing: Bottom-up

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



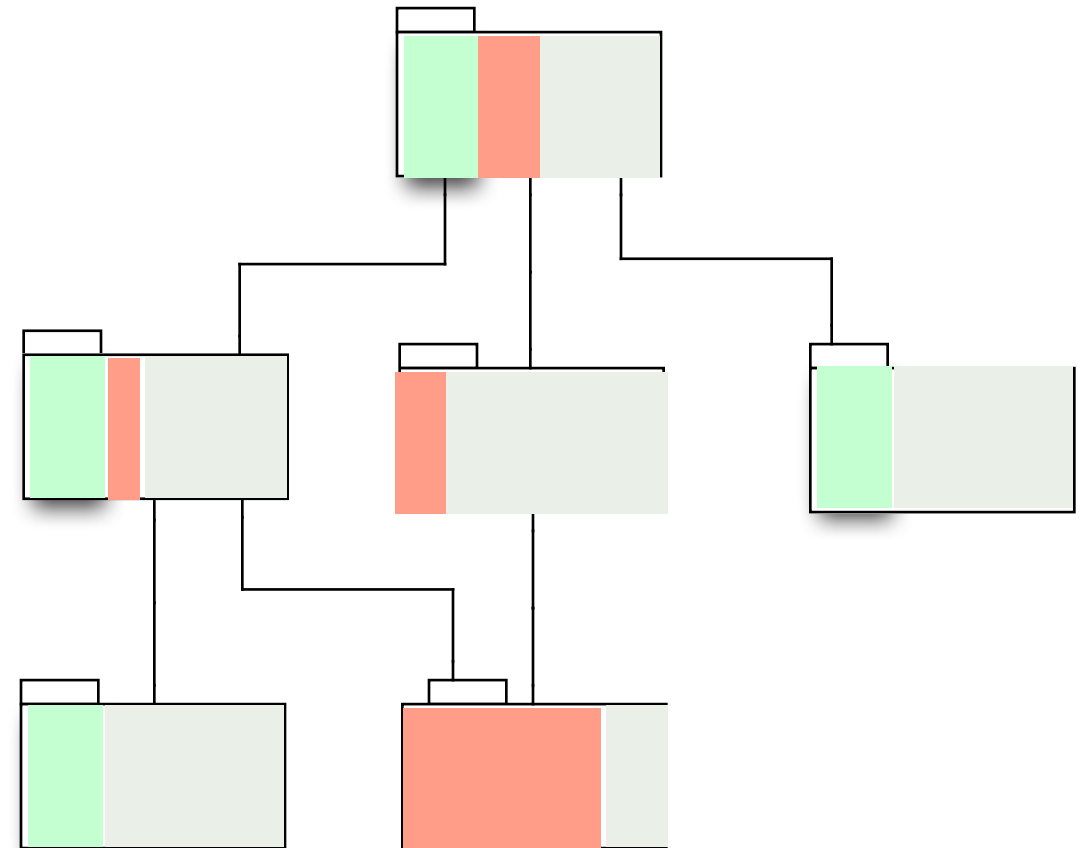
Integration testing: Threads

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



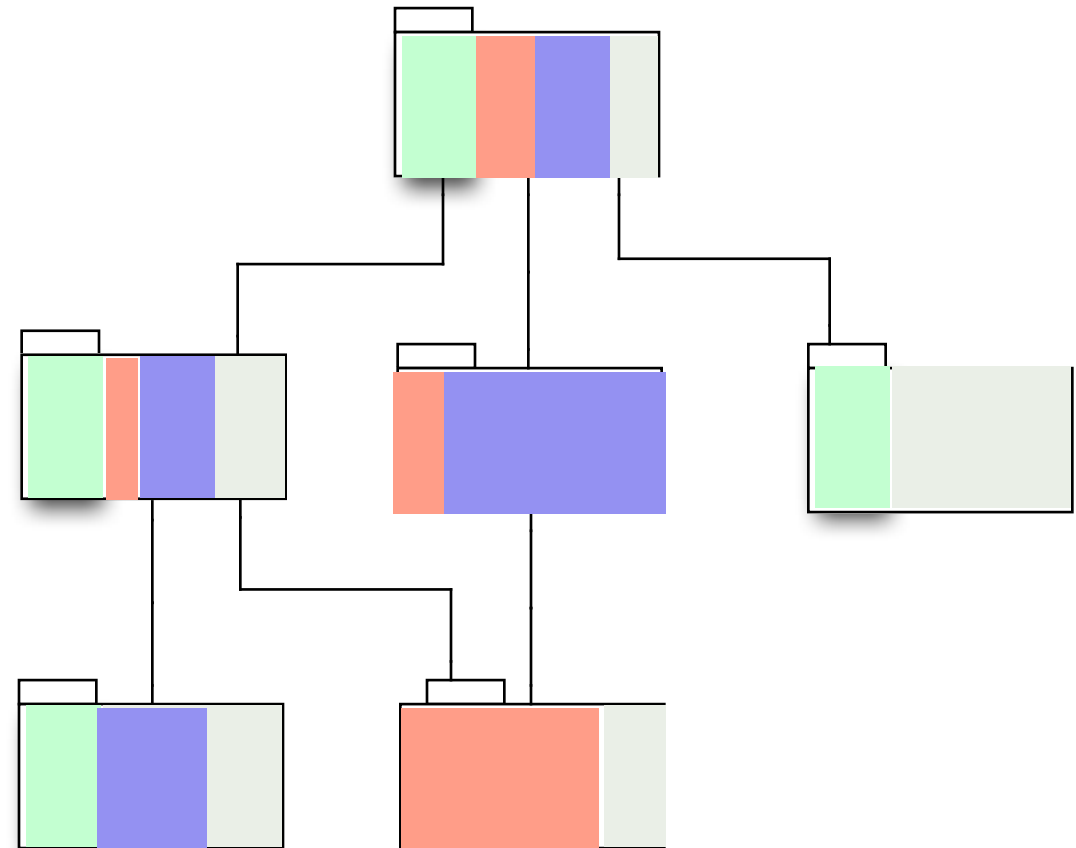
Integration testing: Threads

- 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

- 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

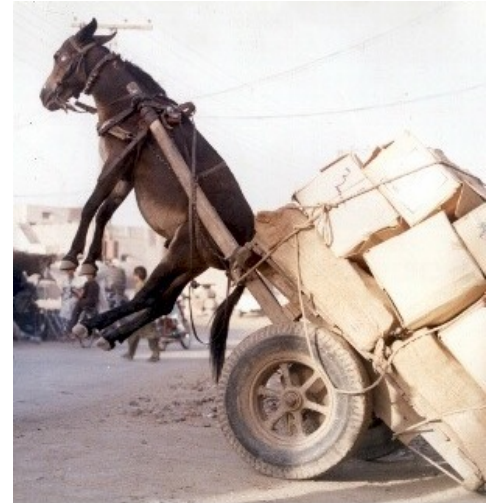
- **How**

- Trying to break the system under test by overwhelming its resources or by reducing resources

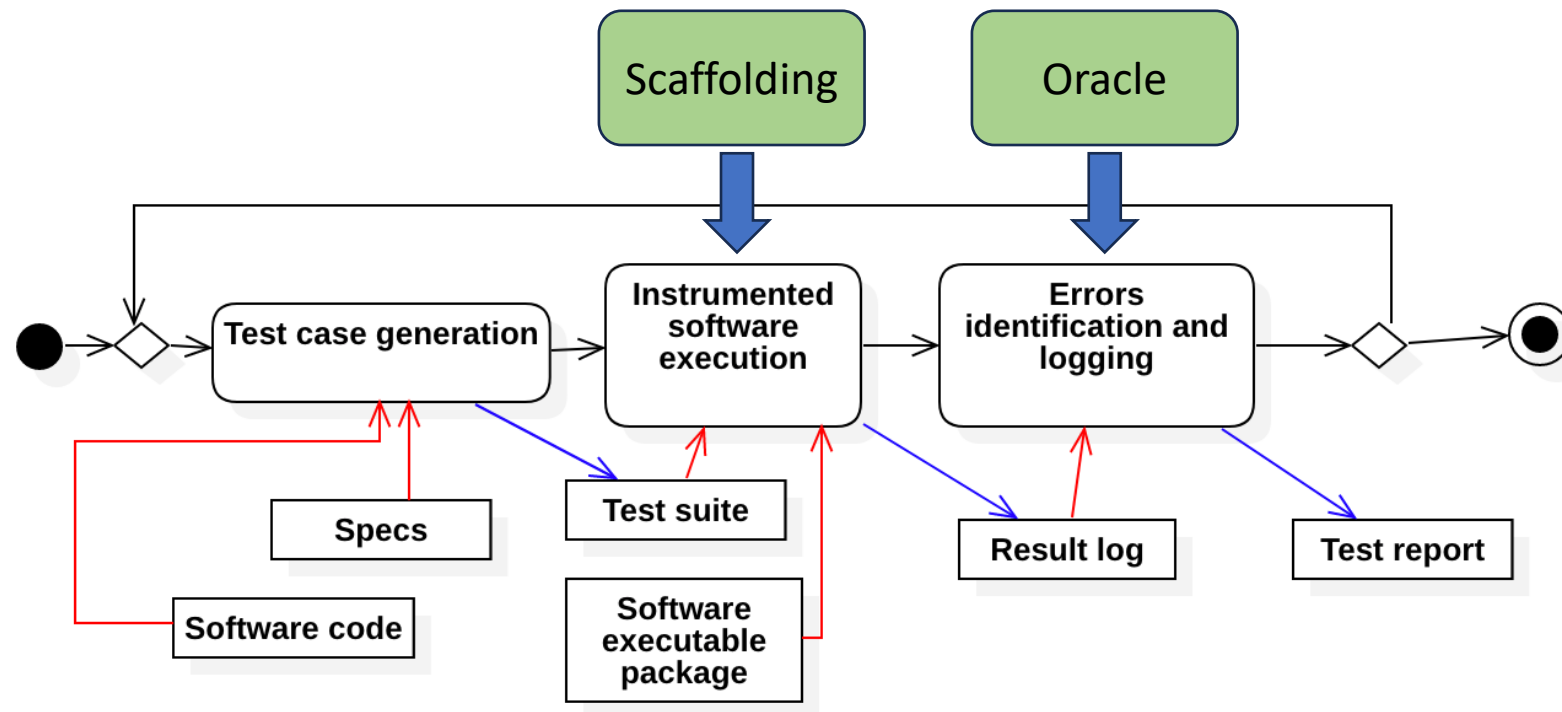
- **Examples**

- 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



References

- Patrick Thomson, Static Analysis: An Introduction. ACMQueue 2021
<https://queue.acm.org/detail.cfm?id=3487021>
- Nichols, W. R., Jr. 2020. The cost and benefits of static analysis during development.arXiv:2003.03001;
<https://ui.adsabs.harvard.edu/abs/2020arXiv200303001N>
- James C. King. (IBM Thomas J. Watson Research Center) 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.
<https://doi.org/10.1145/360248.360252>
- Static analysis services <https://sonarcloud.io/> <https://www.sonarqube.org/>
- Carlo A. Furia. Material for the Software Analysis course.
<https://github.com/bugcounting/software-analysis/tree/master>



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>