COMS W4156 Advanced Software Engineering (ASE)

November 29, 2022

Agenda

1. Auth0 Demo
2. Test Order

Auth0 Demo: Daniel Um

Read the Docs

auth0 documentation

0Auth 2.0 protocol

free alternatives to auth0

Agenda

1. Auth0 Demo
2. Test Order

Test Runners

Execute a test suite (a set of test cases) in some order and report results

Usually do not stop to report when a test fails - instead run all the tests and report all the results

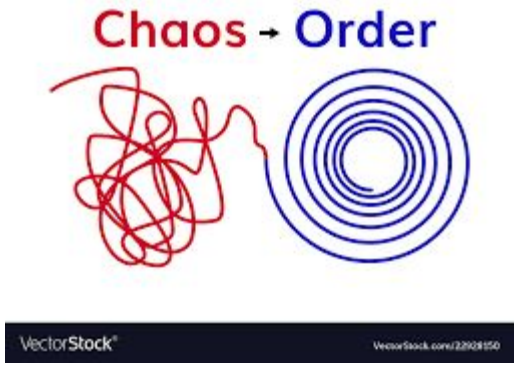
Not restricted to unit testing, test runners can run any tests where code calls other code

Test order might be built-in (e.g., alphanumeric, random) or specified by the tester

Why does test order matter?

When/Why Does Test Order Matter?

- Dependencies and Flaky Tests
- State Transition Testing
- Integration Testing



Some Tests Are “Flaky”

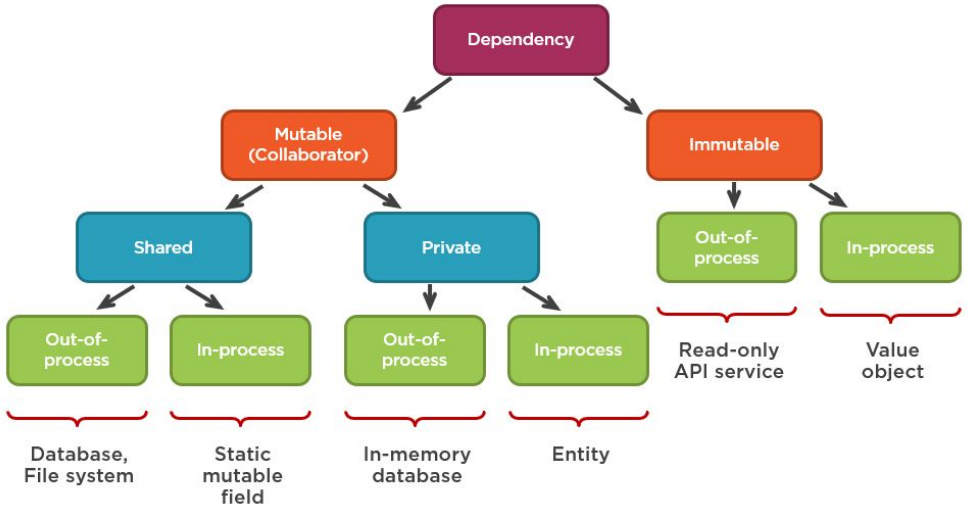
A flaky test is a test case that produces different results (pass or fail) on different test suite executions *even when no code has changed*

This can happen due to hidden dependencies among tests that cause different results for different test orders, reliance on external resources with non-deterministic latency and availability, or a problem in the test’s own code

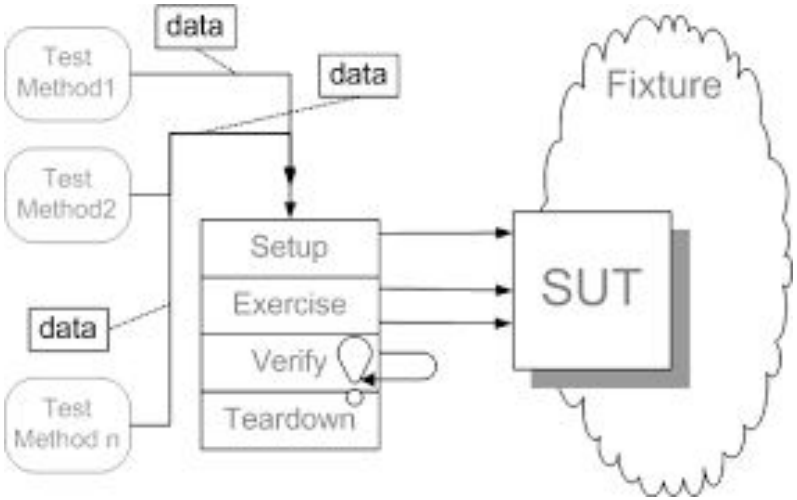
Dependencies Among Test Cases

Tests *intended* to be dependent on each other (e.g., shared state and/or one test acts as setup or teardown for another), and executed in a specified order, should be placed in the same “test class”

Application code necessarily has dependencies on other application code and often on third-party code, and tests are necessarily dependent on the application code they test, but tests (in different test classes) are supposed to be *independent* from each other - but often aren’t



Test Classes



Testing frameworks treat test classes as an atomic unit, all or nothing

The state shared within a test class might include class or global variables, test doubles and/or real resources

Setup/teardown fixtures associated with test classes construct and clean up the shared state

Hidden Dependencies Among Test Classes

Finding all dependencies among test classes is expensive: Run all subsets of the test suite in all possible orders to detect discrepancies. For n test classes, in worst case costs O(2^n), but only has to be done once - until some test class changes. Note this assumes no tests are flaky for other reasons and does not address multi-threading race conditions

Rather than try to find all dependencies, some testing tools offer an option to *prevent* dependencies from affecting test results by restarting/rebooting the runtime environment in between test classes - just in case

Extremely expensive to reinitialize before each test class: It might take 3-5ms to run a JUnit test class vs. 1.4s to restart JVM between each pair of test classes. It takes much longer to reboot Linux. But, technically, only O(n) cost for n test classes…

Why Should Test Classes Be Independent?

Some test runners intentionally run tests in a hard-to-predict order

Newer versions of test runners may run tests in a different order than older versions

Test suites finish faster when test classes run in parallel with each other

Test suites finish faster when test classes unaffected by recent (untested) changes are omitted

Example Re-Order

set up global state then run tests in sequence

do lots of tests;

test1 () { do something that changes global state; };

do lots of other tests;

test2 () { do something that uses global state; }

set up global state then run tests in sequence

do lots of tests;

test2 () { do something that uses global state; }

do lots of other tests;

test1 () { do something that changes global state; }

Example Parallel

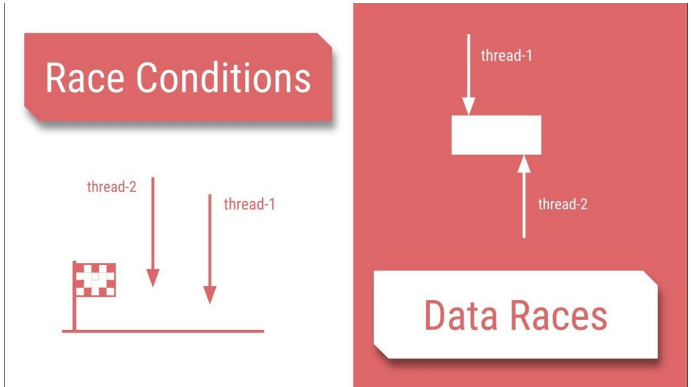
set up global state then run tests in parallel;

do lots of tests;

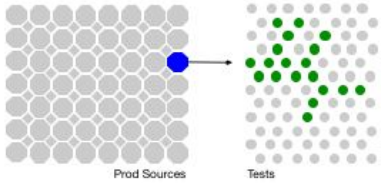
test1 () { do something that changes global state; }

do lots of other tests;

test2 () { do something that uses global state; }



How Do We Know When a Test is Affected?



If the test itself is newly added or changed, if it tests newly added or changed code, or if it accesses state accessed by newly added or changed code, then it's affected

Otherwise, use test impact analysis to determine those tests whose results might change due to recently added or changed code - and those tests whose results cannot change

The goal is to reduce the number of tests to run by identifying those most likely to detect a newly added bug - which must be in the newly added code, in an interface between the newly added code and some existing code, or in some existing code that shares state with the newly changed code

How to Eliminate Flaky Tests

Ideally, all flaky tests should be tracked down and removed, but this is hard!

Write test assertions to accommodate application non-determinism, e.g., Kotest

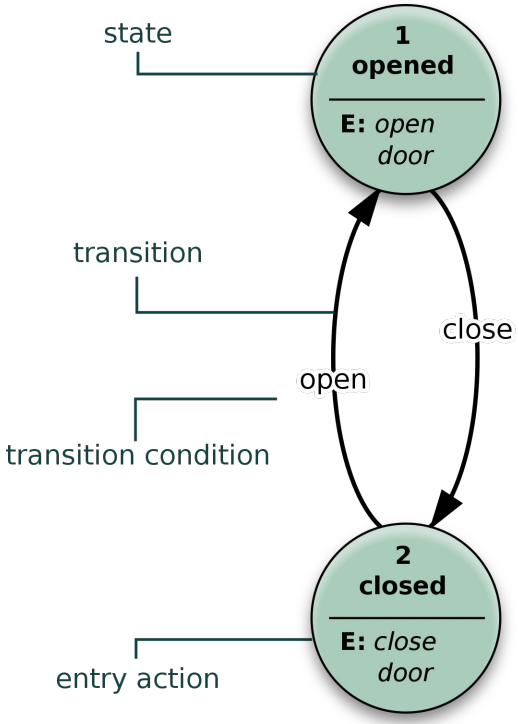
Use record/replay and time-travel debugging, e.g., rr, replay, to identify root causes of non-determinism

Remove all sources of non-determinism that can be removed, e.g., use re-initialized test doubles for all external resources except during explicit integration/system testing with those external resources and during end-to-end testing - when flaky tests are unavoidable

When/Why Does Test Order Matter?

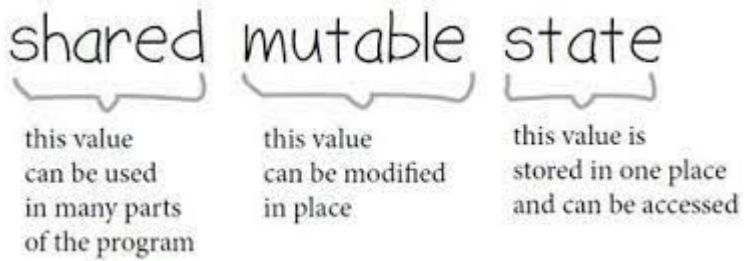
- Dependencies and Flaky Tests
- State Transition Testing
- Integration Testing

State Transition Testing



Most often applied to OO classes where methods share state, typically object fields and class static fields

Also applicable to non-OO languages when state is scoped to modular units that can be tested in isolation from rest of program

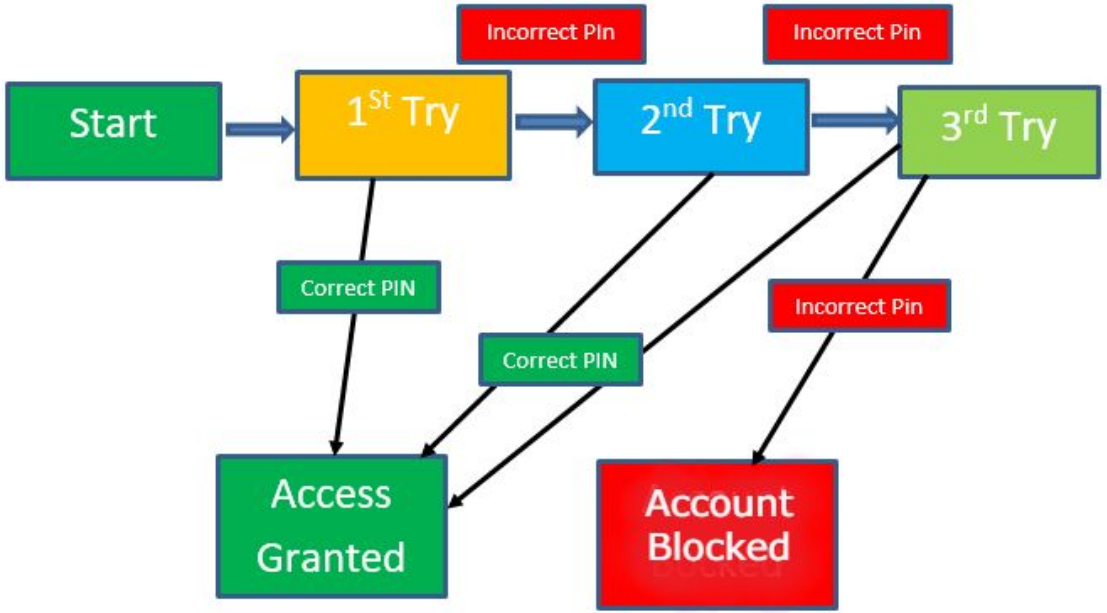


Test All Transitions in the State Diagram

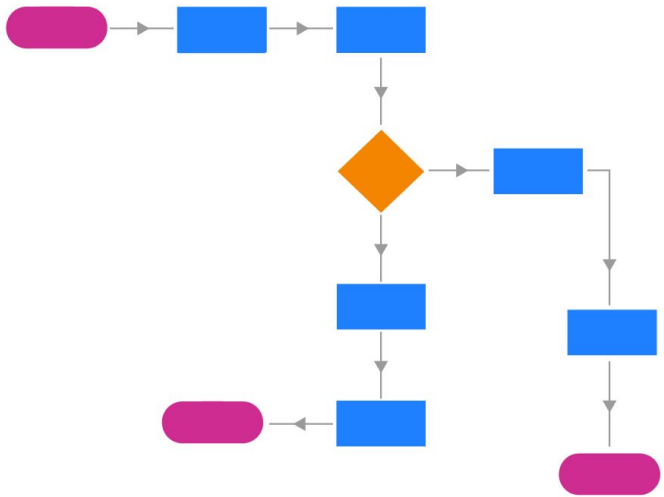
Draw “state diagram” (finite state automaton) to visualize the possible states and their transitions

Choose combinations of methods and inputs to those methods that exercise all the transitions between states

Does not scale to large systems with many states and/or loops among states



State Transition Testing Also Applies to (Some) API Testing



For many (not all) APIs, the appropriate ordering among API calls follows state transitions within the API implementation

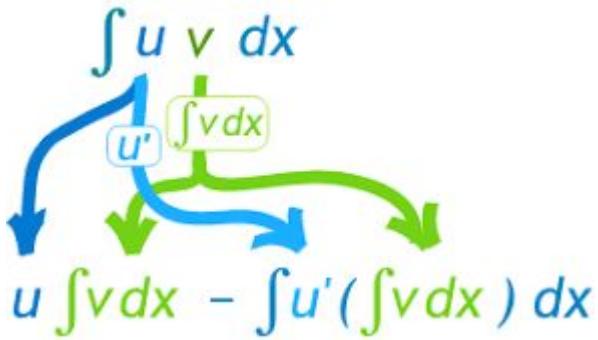
Choose combinations of methods and inputs to those methods that exercise all possible transitions between states

But the API state diagram may be implicit - not known to developer/tester of client code

Or the API may consist of unrelated utility functions that share little or no state

When/Why Does Test Order Matter?

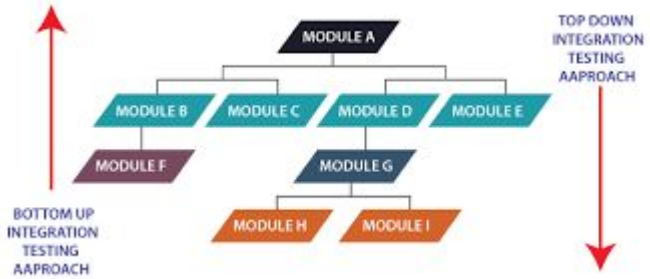
- Dependencies and Flaky Tests
- State Transition Testing
- Integration Testing

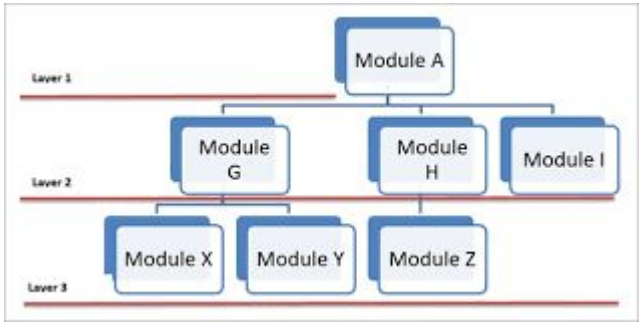


Traditional Integration Testing Orders

Developers **add to** test suite incrementally, integrating components and assemblies of components in some order

These are not a conventional “test runner” orders, until last phase when set of integration tests is complete and the test runner executes all of them in priority order





Traditional Integration Testing Orders

Developers add to test suite incrementally, not really a “test runner” order

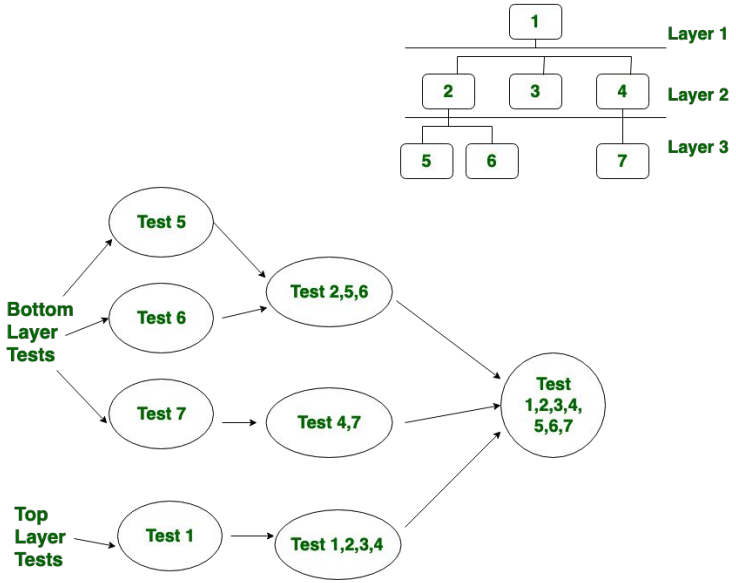
TOP DOWN INTEGRATION TESTING	BOTTOM UP INTEGRATION TESTING
Top Down Integration testing is an approach of Integration testing in which integration takes place from top to bottom, meaning system integration begins with top level modules.	Bottom Up Integration testing is an approach to Integration testing in which integration takes place from bottom to top, meaning system integration begins with lowest level modules.
The higher level modules are tested first, then the lower level modules are tested, and then the modules are integrated accordingly.	The lower level modules are tested first, then the higher level modules are tested, and then the modules are integrated accordingly.
Stubs/mocks are used to simulate a submodule if the invoked submodule is not developed or integrated yet, working as a temporary replacement.	Drivers are used to simulate the main module if the main module is not developed or integrated yet, working as a temporary replacement.

Traditional Integration Testing Orders continued

TOP DOWN INTEGRATION TESTING	BOTTOM UP INTEGRATION TESTING
Most beneficial if significant defects occur toward the top of the program.	Most beneficial if crucial flaws are encountered towards the bottom of the program.
The high-level modules are designed/created/tested first, then lower-level submodules/subroutines are called from them.	Low-level modules are designed/created/tested first, then these submodules/subroutines are called from the high-level modules.
Commonly implemented in procedure-oriented programming languages.	Commonly implemented in object-oriented programming languages.
The complexity of this testing is simple.	The complexity of this testing is complex and highly data intensive.
Works on big to small components.	Works on small to big components.
Stub/mock modules must be produced.	Driver modules must be produced.

Hybrid (“Sandwich”) Integration Testing

- Combines both top down and bottom up strategies.
- Basically viewed as 3 layers:
  - (i) Main target layer
  - (ii) A layer above the target layer
  - (iii) A layer below the target layer
- Testing mainly focused on main target layer, selected on the basis of system characteristics and code structures.
- Tries to minimize the number of stubs and drivers when there are more than 3 layers.



Sandwich Integration Testing

Advantages:

- Used in very large projects with sub-projects.
- Allows parallel testing.
- Time saving approach.
- Achieves more coverage with same stubs.

Disadvantages:

- Very costly.
- Cannot be used for systems that have a lot of interdependencies between different modules.
- Needs more stubs and drivers.

Upcoming Assignments

Second Iteration due yesterday, but you can continue working on it

Second Iteration demo due December 5 (next Monday)

Second Individual Assessment from 12:01am Tuesday December 6 through 11:59pm Friday December 9

Demo Day Monday December 19 10am to 4pm

Coming Soon

Thursday:

Debugging

Mutation Testing

Next Week:

Refactoring demo

Test Oracles, Metamorphic Testing

Test Generation, Fuzzing, Symbolic Execution

Ask Me Anything