Modules/components (ideally micro-services!) at various layers — low (no dependence on others), middle (depend and are depended on), high (depend, but are not depended on) — note that multiple such top layers should be pretty normal, e.g. UIs and APIs.

By "test" I always mean an automated test…
How do we test it?

- ancient way: white-box, black-box
- too-close modern way:
  - unit-tests: white-boxy, dev-focused
  - integration-tests: end-to-end
  - maybe: human-in-loop QA

QA = Quality Assurance (use a different term than "test", it's TOTALLY different!!!). Integration and unit tests (being automated) can be used to "gate" continuous integration, QA really can't.

Best way: Layers

- unit-tests (fast, run all the time)
- strictly on internal logic
- mock out "every" dependency
- fast (almost) "above all"
- higher-layer tests
- pattern language: match the CODE layers!

If running automatically in the background, fast allows lower latency for feedback; if not, fast means you'll run unit tests more often, integrate more easily w/faster int.tests (case study: Python 3.6.1rc1 — https://www.mail-archive.com/python-committers@python.org/msg04246.html); either way, faster tests -> higher productivity

The fundamental things apply... e.g:

- all tests must be reproducible
- if any randomness, force a seed
- if depends on current time of day, day of week, etc etc, must fake time
- test-driven approach to fixing bugs

...and many other excellent, necessary qualities of automated tests, layered or not
7  

Testing DBa

8  

How to test DBa

- pure unit-test: mock out DB
- fine if we understand DB 100%
- 2nd layer int-test: fake (emulated) DB
- local, controlled, maybe in-mem
- including semantic constraints!

9  

What's a semantic constraint?

- e.g.: "after con.close() no other call to con's methods, else RuntimeError"
- a fake must emulate this behavior
- a mock won't unless you KNOW about it (still worth it, to help future maintainers)

A fake may also add constraints such as "DB size < 23 MB" — for test uses, should be OK
from unittest import mock  # also in later slides

# for pure unit-test:
with mock.patch.object(dba, 'db', autospec=True) as fdb:
    # prepare fdb's side effects under test
    fdb.connect.cursor.side_effect = ...

    # body of tests

# for 2nd layer integration-test:
fdb = fake_db.Fake(...params...)
with mock.patch.object(dba, 'db', new=fdb):
    # populate fdb for the test
    fdb.connect().cursor().execute(...)

    # body of tests

# for full integration-test:
# start and populate real db for the test
# body of tests

"Body of tests" constant and reusable; difference among test layers is in the preparation for running the "body of tests" (optionally also in extra checks afterwards for mocks [or other spies wrapping fakes or actuals] — but, beware white-box-itis!-

Mocks aren't Fakes

...nor other kinds of test doubles (see https://martinfowler.com/articles/mocksArentStubs.html)

dummy, fake, stub, spy, mock,...

a key issue: who owns/maintains
Mocks vs Fakes

- **mock**: flexibly simulate anything depending on tests' calls on it
- **spies**: lets you check calls, args, ...
- **fake**: fast, limited emulation of a specific component/module

Both mock and fakes should also, on request, emulate error situations (e.g. "CPU on fire":-) to check that your code handles such disasters gracefully (almost impossible to check w/o simulation!).

Testing MLA

MLA = middle layer

How to test MLA

- **pure unit-test**: mock out LL1, LL2
  - fastest, mostly fine (team ownership)
- **mid-layer int-test**: actual LL1, LL2
  - if fast enough (check w/ timeit)...
  - don't need pure unit-test (less work)
# for pure unit-test:
with mock.patch.object(mla,'ll1',autospec=True) as f1,
    mock.patch.object(mla,'ll2',autospec=True) as f2:
    # prepare f1's & f2's side effects under test
    # body of tests
# for 2nd layer integration-test:
# prepare ll1's and ll2's for the test
# body of tests
# no further integration-tests in this specific case

Testing HLβ

How to test HLβ

- pure unit-test: mock out MLB, MLC
- 2nd layer unit-tests: actual MLB, MLC
  - mock DBa, DNSa; mock or real LL2
- 3rd layer i-t: act. MLB, LL2, DBa, DNSa
  - mock or fake DB, DNS
  - (pick subset, else, combinatorial explosion!)

tradeoffs: mocks may be faster; but, mostly: ownership of test-double, thus, how much detailed/precise understanding of corner cases is needed
# code for this single example test:
with mock.patch.object(hbb,'mlb',autospec=True) as fb:
    # prepare fb's side effects under test
    fdb = fake_db.Fake(...params...)
with mock.patch.object(dba,'db',new=fdb):
    # populate fdb for the test
    fdb.connect().cursor().execute(...)
with mock.patch.object(dnsa,'dns',autospec=True) as fd:
    # prepare fd's side effects under test

Use mock, fake, or actual module?

- mock: fastest, least accurate
- actual: least work, if fast enough
- design it to be primeable for speed
- fake: best if there (thorough, deep, fast)
- coding a good fake is a lot of work

Dependency injection and other such patterns can help you make a module "primeable for speed" (but: beware turning module into a fake of itself and/or cutting off paths from tests!)
Check complexity
- e.g., external "DNS" module — what kind of DNS records do you need?
- often: essentially just A records, DNS -> IP
- trivial to mock or fake
- maybe: CNAME, HINFO, MX, NS, PTR, SOA, TXT, ...
- needs careful fake (mock VERY hard to make and keep correct and complete!)

Load-test in layers
- actual elapsed-time measurements need end-to-end code paths
- BUT: with intermediate tests you can get \( t + n \cdot (e.s.'s \text{ SLAs}) \) time as \( t + n \cdot (e.s.'s \text{ SLAs}) \)

"Body of tests" for load-testing
- NOT the same as for other tests
- take correctness for granted
- exercise performance-critical bits
- usually best to separate for easier elapsed-time measurement

22 DNS = Domain Naming System; once mostly a simple "my.host.com" -> 22.33.44.55 mapping, now (over?-)grown into much richer functionality (e.g., TXT records to validate ownership)

23 SLA = "Service Level Agreement" (e.g.: "90% of queries answered in < 33 msec")
In a sense, it's automatically a test-first situation (since tests must exist before refactoring).

Tests & logging
- Unit-tests must be fast
  - Check only what can be checked rapidly
- For everything else:
  - Log/snapshot status in detail
  - Run batch/background jobs to check
  - Batch sanity checks on logs/snaps: good idea
  - Including non-testing production runs!

Q? A!
http://www.aleax.it/oscon17.pdf