Carnegie Mellon University Qatar

Software Testing

17-313 Fall 2023

S3D

Software and Societal Systems Department

Learning goals

- Identify the scope and limitations of software testing
- Appreciate software testing as a methodology to use automation in improving software quality
- Describe the benefits of using continuous integration and deployment (CI/CD)
- Measure the quality of software tests and define test adequacy criteria
- Enumerate different levels of testing such as unit testing, integration testing, system testing, and testing in production
- Describe the principles of test-driven development
- Outline design principles for writing good tests
- Recognize and avoid testing anti-patterns

What is testing good for?

- What is testing?
  - Execution of code on sample inputs in a controlled environment
- Principle goals:
  - Validation: program meets requirements, including quality attributes.
  - Defect testing: reveal failures.

What makes a good test?

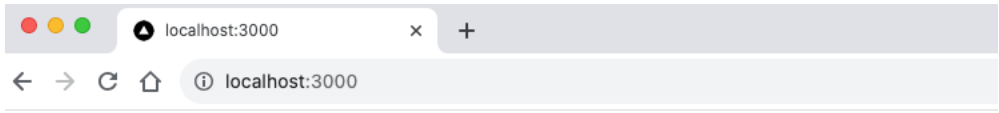
Why write tests at all?

Why write tests at all?

- [Low bar] Ensure that our software meets requirements, is correct, etc.
- Preventing bugs or quality degradations from being accidentally introduced in the future → **Regression Testing**
- Helps uncover unexpected behaviors that can't be identified by reading source code
- Increased confidence in changes ("will I break the internet with this commit?")
- Bridges the gap between a declarative view of the system (i.e., requirements) and an imperative view (i.e., implementation) by means of redundancy.
- Tests are executable documentation; increases code maintainability
- Forces writing testable code <-> checks software design

Where should we run the tests?

- **Unit testing**
  - Code level, E.g. is a function implemented correctly?
  - Does not require setting up a complex environment
- **Integration testing**
  - Do components interact correctly? E.g. a feature that cuts across client and server.
  - Usually requires some environment setup, but can abstract/mock out other components that are not being tested (e.g. network)
- **System testing**
  - **Validating the whole system end-to-end (E2E)**
  - Requires complete deployment in a staging area, but fake data
- **Testing in production**
  - **Real data but more risks**



Welcome!!

Please enter your query in the box below:

Sort the numbers: 1, 2, 1

1, 1, 2

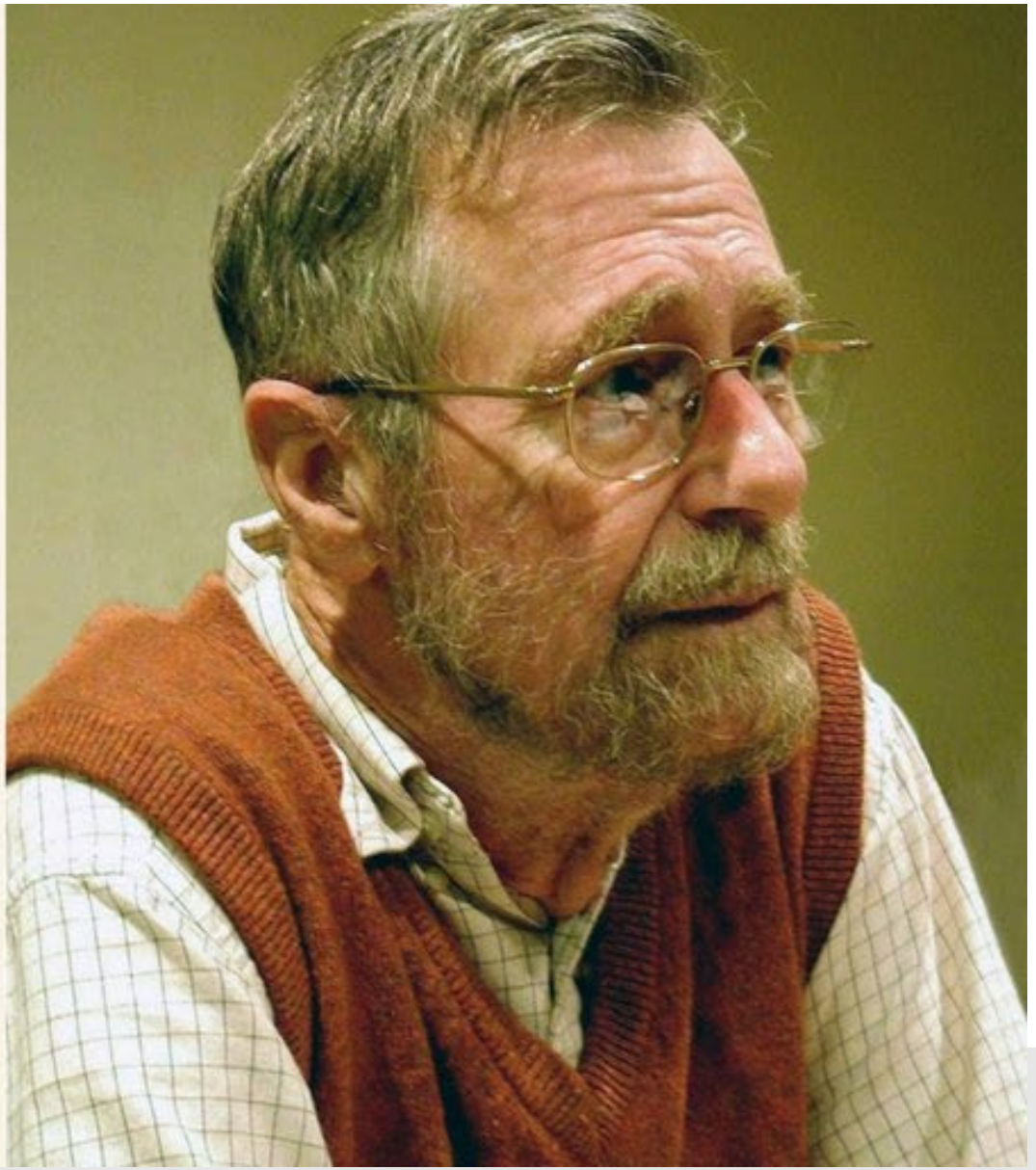
What are the limitations of testing?
(What does testing not achieve?)

Program testing can be used to show the presence of bugs, but never to show their absence!

”

EDSGER W. DIJKSTRA

Notes On Structured Programming, 1970



Limitations of Testing

- Testing doesn't really give any formal assurances
- Writing tests is hard, time-consuming
- Knowing if your tests are good enough is not obvious
- Executing tests can be expensive, especially as software complexity and configuration space grows
  - Full test suite for a single large app can take several days to run

What can we run (automated) tests for?
(Software Quality attributes)

What can we not (easily) test for?
(Software Quality attributes)

Activity: Identify two qualities that are testable and two that would be hard to test for

- Functionality
- Performance
- Scalability
- Security
- Extensibility
- Usability
- Reliability
- Availability
- Maintainability
- Safety
- Fairness
- Portability
- Regulatory compliance

Test Oracles

- ”Oracles” are mechanisms that tell you when program execution seems abnormal or unexpected
- E.g. assert, segfault, exception
- Other examples: performance threshold, memory footprint, address sanitizer

Test Oracles

- Obvious in some applications (e.g. “sort()”) but more challenging in others (e.g. “encrypt()” or UI-based tests)

- Lack of good oracles can limit the scalability of testing. Easy to generate lots of input data, but not easy to validate if output (or other program behavior) is correct.

- Fortunately, we have some tricks.

Differential Testing

- If you have two implementations of the same specification, then their output should match on all inputs.
  - E.g. `mergeSort(x).equals(bubbleSort(x))` → should always be true
  - Special case of a property test, with a free oracle.

- If a differential test fails, at least one of the two implementations is wrong.
  - But which one?
  - If you have N > 2 implementations, run them all and compare. Majority wins (the odd one out is buggy).

- Differential testing works well when testing programs that implement standard specifications such as compilers, browsers, SQL engines, XML/JSON parsers, media players, etc.
  - Not feasible in general e.g. for CMU's custom grad application system.

Regression Testing

- Differential testing through time (or *versions*, say V1 and V2).
- Assuming V1 and V2 don't add a new feature or fix a known bug, then f(x) in V1 should give the same result as f(x) in V2.
- *Key Idea*: Assume the current version is correct. Run program on current version and log output. Compare all future versions to that output.

When should we test?
(And where should we run the tests?)

Test Driven Development (TDD)

- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test
- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid unneeded code
  - Higher product quality
  - Higher test suite quality
  - Higher overall productivity

Common bar for contributions

Chromium

- **Changes should include corresponding tests.** Automated testing is at the heart of how we move forward as a project. All changes should include corresponding tests so we can ensure that there is good coverage for code and that future changes will be less likely to regress functionality. Protect your code with tests!

Firefox

Testing Policy

Everything that lands in mozilla-central includes automated tests by default. Every commit has tests that cover every major piece of functionality and expected input conditions.

Docker

Conventions

Fork the repo and make changes on your fork in a feature branch:

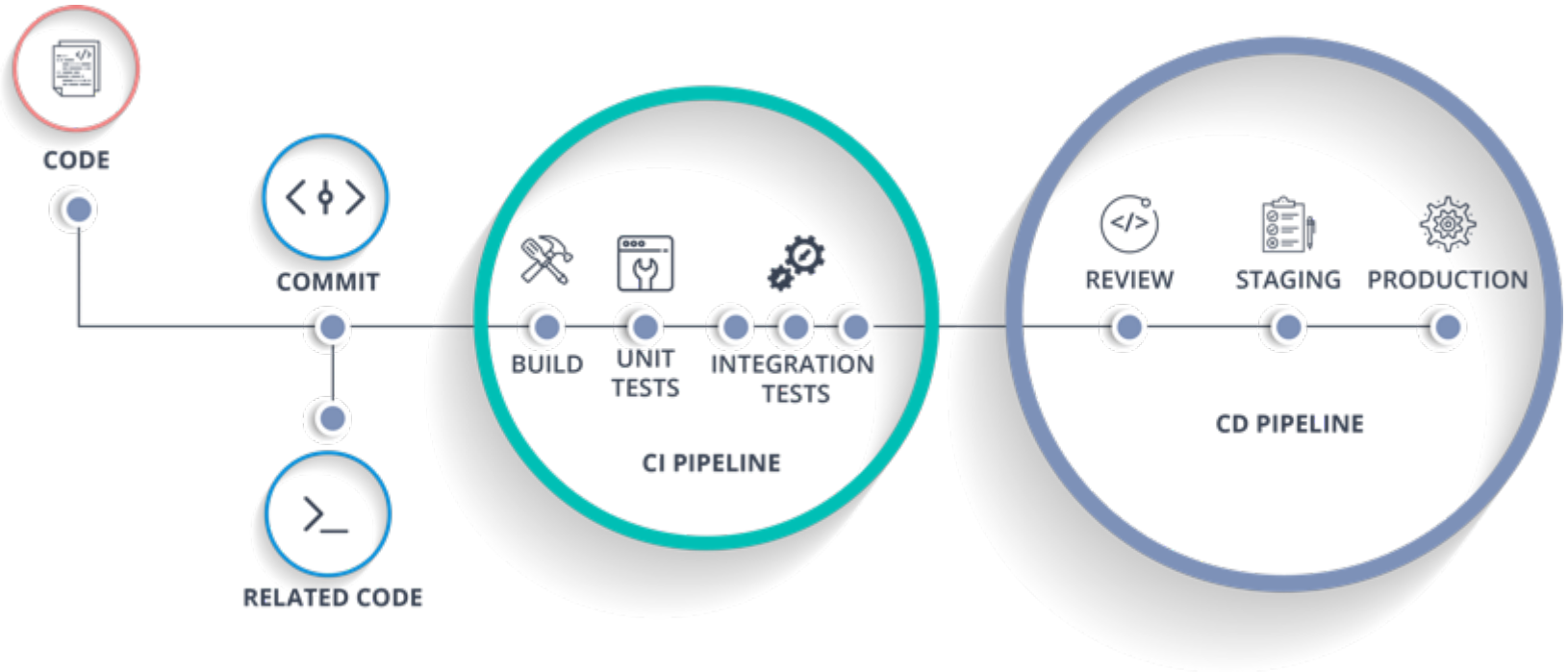
- If it's a bugfix branch, name it XXX-something where XXX is the number of the issue
- If it's a feature branch, create an enhancement issue to announce your intentions, and name it XXX-something where XXX is the number of the issue.

Submit unit tests for your changes. Go has a great test framework built in; use it! Take a look at existing tests for inspiration. Run the full test suite on your branch before submitting a pull request.

Regression testing

- Usual model:
  - Introduce regression tests for bug fixes, etc.
  - Compare results as code evolves
    - **Code1** + **TestSet** → **TestResults1**
    - **Code2** + **TestSet** → **TestResults2**
  - As code evolves, compare **TestResults1** with **TestResults2**, etc.
- Benefits:
  - Ensure bug fixes remain in place and bugs do not reappear.
  - Reduces reliance on specifications, as **<TestSet,TestResults1>** acts as one.

Continuous Integration & Deployment



How good are our tests?
(How can we measure test quality?)

Code Coverage

- Line coverage
- Statement coverage
- Branch coverage
- Instruction coverage
- Basic-block coverage
- Edge coverage
- Path coverage
- …

Code Coverage

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - test			
Test:	coverage.info	Lines:	6092	7293	83.5 %
Date:	2018-02-07 13:06:43	Functions:	481	518	92.9 %

Filename	Line Coverage		Functions	
asn1_string_table_test.c	58.8 %	20 / 34	100.0 %	2 / 2
asn1_time_test.c	72.0 %	72 / 100	100.0 %	7 / 7
bad_dtls_test.c	97.6 %	163 / 167	100.0 %	9 / 9
bftest.c	65.3 %	64 / 98	87.5 %	7 / 8
bio_enc_test.c	78.7 %	74 / 94	100.0 %	9 / 9
bntest.c	97.7 %	1038 / 1062	100.0 %	45 / 45
chacha_internal_test.c	83.3 %	10 / 12	100.0 %	2 / 2
ciphername_test.c	60.4 %	32 / 53	100.0 %	2 / 2
crltest.c	100.0 %	90 / 90	100.0 %	12 / 12
ct_test.c	95.5 %	212 / 222	100.0 %	20 / 20
d2i_test.c	72.9 %	35 / 48	100.0 %	2 / 2
danetest.c	75.5 %	123 / 163	100.0 %	10 / 10
dhtest.c	84.6 %	88 / 104	100.0 %	4 / 4
drbgtest.c	69.8 %	157 / 225	92.9 %	13 / 14
dtls_mtu_test.c	86.8 %	59 / 68	100.0 %	5 / 5
dtlstest.c	97.1 %	34 / 35	100.0 %	4 / 4
dtlsv1listentest.c	94.9 %	37 / 39	100.0 %	4 / 4
ecdsatest.c	94.0 %	140 / 149	100.0 %	7 / 7
enginetest.c	92.8 %	141 / 152	100.0 %	7 / 7
evp_extra_test.c	100.0 %	112 / 112	100.0 %	10 / 10
fatalerrtest.c	89.3 %	25 / 28	100.0 %	2 / 2
handshake_helper.c	84.7 %	494 / 583	97.4 %	38 / 39
hmactest.c	100.0 %	71 / 71	100.0 %	7 / 7
ideatest.c	100.0 %	30 / 30	100.0 %	4 / 4
igetest.c	87.9 %	109 / 124	100.0 %	11 / 11
lhash_test.c	78.6 %	66 / 84	100.0 %	8 / 8
mdc2_internal_test.c	81.8 %	9 / 11	100.0 %	2 / 2
mdc2test.c	100.0 %	18 / 18	100.0 %	2 / 2
ocspapitest.c	95.5 %	64 / 67	100.0 %	4 / 4
packettest.c	100.0 %	248 / 248	100.0 %	24 / 24

```
      1  /   1:                if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
      0  /   1:                    goto fail;
                :    }
                :    else {
                :        /* DSA, ECDSA - just use the SHA1 hash */
      0  /   1:        dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
      0  /   1:        dataToSignLen = SSL_SHA1_DIGEST_LEN;
                :    }
                :
      1  /   1:    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
      1  /   1:     hashOut.length = SSL_SHA1_DIGEST_LEN;
      1  /   1:     if ((err = SSLFreeBuffer(&hashCtx)) != 0)
      0  /   1:         goto fail;
                :
      1  /   1:     if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
      0  /   1:         goto fail;
      1  /   1:     if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
      0  /   1:         goto fail;
      1  /   1:     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      0  /   1:         goto fail;
      1  /   1:     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      0  /   1:         goto fail;
      1  /   1:         goto fail;
                :     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                :         goto fail;
                :
                :    err = sslRawVerify(ctx,
                :                       ctx->peerPubKey,
                :                       dataToSign,                     /* plaintext *,
                :                       dataToSignLen,                  /* plaintext l
                :                       signature,
                :                       signatureLen);
                :    if(err) {
                :            sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                :                    "returned %d\n", (int)err);
                :            goto fail;
                :    }
                :
                : fail:
      1  /   1:     SSLFreeBuffer(&signedHashes);
      1  /   1:     SSLFreeBuffer(&hashCtx);
      1  /   1:     return err;
                :
      1  /   1: }
                :
```

Beware of coverage chasing

- Recall: issues with metrics and incentives
- Also: Numbers can be deceptive
  - 100% coverage != exhaustively tested
- “Coverage is not strongly correlated with suite effectiveness”
  - Based on empirical study on GitHub projects [Inozemtseva and Holmes, ICSE’14]
- Still, it’s a good low bar
  - Code that is not executed has definitely not been tested

Coverage != Outcome

- What's better, tests that always pass or tests that always fail?
- Tests should ideally be *falsifiable.* Boundary determines specification
- Ideally:
  - Correct implementations should pass all tests
  - Buggy code should fail at least one test
  - Intuition behind *mutation testing* (we'll revisit this next week)
- What if tests have bugs?
  - Pass on buggy code or fail on correct code
- Even worse: flaky tests
  - Pass or fail on the same test case nondeterministically
- What's the worst type of test?

Test Design principles

- Use public APIs only
- Clearly distinguish inputs, configuration, execution, and oracle
- Be simple; avoid complex control flow such as conditionals and loops
- Tests shouldn’t need to be frequently changed or refactored
  - Definitely not as frequently as the code being tested changes

Anti-patterns

- Snoopy oracles
  - Relying on implementation state instead of observable behavior
  - E.g. Checking variables or fields instead of return values
- Brittle tests
  - Overfitting to special-case behavior instead of general principle
  - E.g. hard-coding message strings instead of behavior
- Slow tests
  - Self-explanatory (beware of heavy environments, I/O, and `sleep()`)
- Flaky tests
  - Tests that pass or fail nondeterministically
  - Often because of reliance on random inputs, timing (e.g. `sleep(1000)`), availability of external services (e.g. fetching data over the network in a unit test), or dependency on order of test execution (e.g. previous test sets up global variables in certain way)

Takeaways

- Most tests that you will write will be muuuuuuch more complex than testing a sort function.
- Need to set up environment, create objects whose methods to test, create objects for test data, get all these into an interesting state, test multiple APIs with varying arguments, etc.
- Many tests will require mocks (i.e., faking a resource-intensive component).
- General principles of many of these strategies still apply:
  - Writing tests can be time-consuming
  - Determining test adequacy can be hard (if not impossible)
  - Test oracles are not easy
  - Advanced test strategies have trade-offs (high costs with high returns)