Intro to QA and Testing

Michael Hilton and **Rohan Padhye**

institute for SOFTWARE RESEARCH | Carnegie Mellon University School of Computer Science

Administrativia?

- HW3 is due Oct 29 (tomorrow)
  - All documents (requirements + reflection)
- HW4 checkpoint is also due Oct 29 (tomorrow)
  - Should be fairly quick
- No recitation next week (Nov 3 and Nov 5)
  - (since no classes on Friday for community engagement)

Learning goals

- Identify the scope and limitations of software testing
- Appreciate software testing as a methodology to use automation in improving software quality
- Estimate the costs of testing and discuss trade-offs of running tests at different times in the software development lifecycle
- 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

Alright, it's time to talk about testing

- 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.
- Other goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

Alright, it's time to talk about testing

- What can we test for? (Software quality attributes)
  - What can we not test for?
- Why should we test? What does testing achieve?
  - What does testing not achieve?
- When should we test?
  - And where should we run the tests?
- What should we test?
  - What CAN we test?
- How should we test?
  - How many ways can you test the sort() function?
- How good are our tests?
  - How to measure test quality?

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

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

Things we might try to test

- Program/system functionality:
  - Execution space (white box).
  - Input or requirements space (black box).
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

Software Errors

Functional errors
Performance errors
Deadlock
Race conditions
Boundary errors
Buffer overflow
Integration errors
Usability errors
Robustness errors
Load errors

Design defects
Versioning and configuration errors
Hardware errors
State management errors
Metadata errors
Error-handling errors
User interface errors
API usage errors
...

Why should we test?
(What does testing help us achieve?)

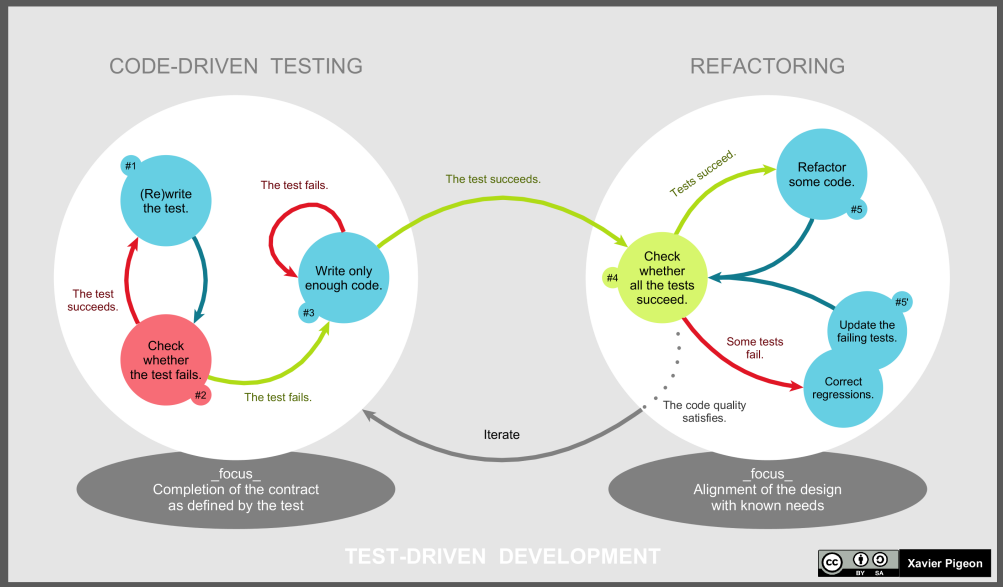
What are the limitations of testing?
(What does testing not achieve?)

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


YOU SCHEDULED THE END OF THE TEST PHASE AFTER THE START OF THE PRODUCTION PHASE.
WE'RE FEELING CONFIDENT.
IT'S TOO BAD THAT BEING SMART DOESN'T COME WITH SOME SORT OF GOOD FEELING LIKE THAT.
Dilbert.com DilbertCartoonist@gmail.com
5-17-11 © 2011 Scott Adams, Inc. Dist. by Universal Uclick

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:

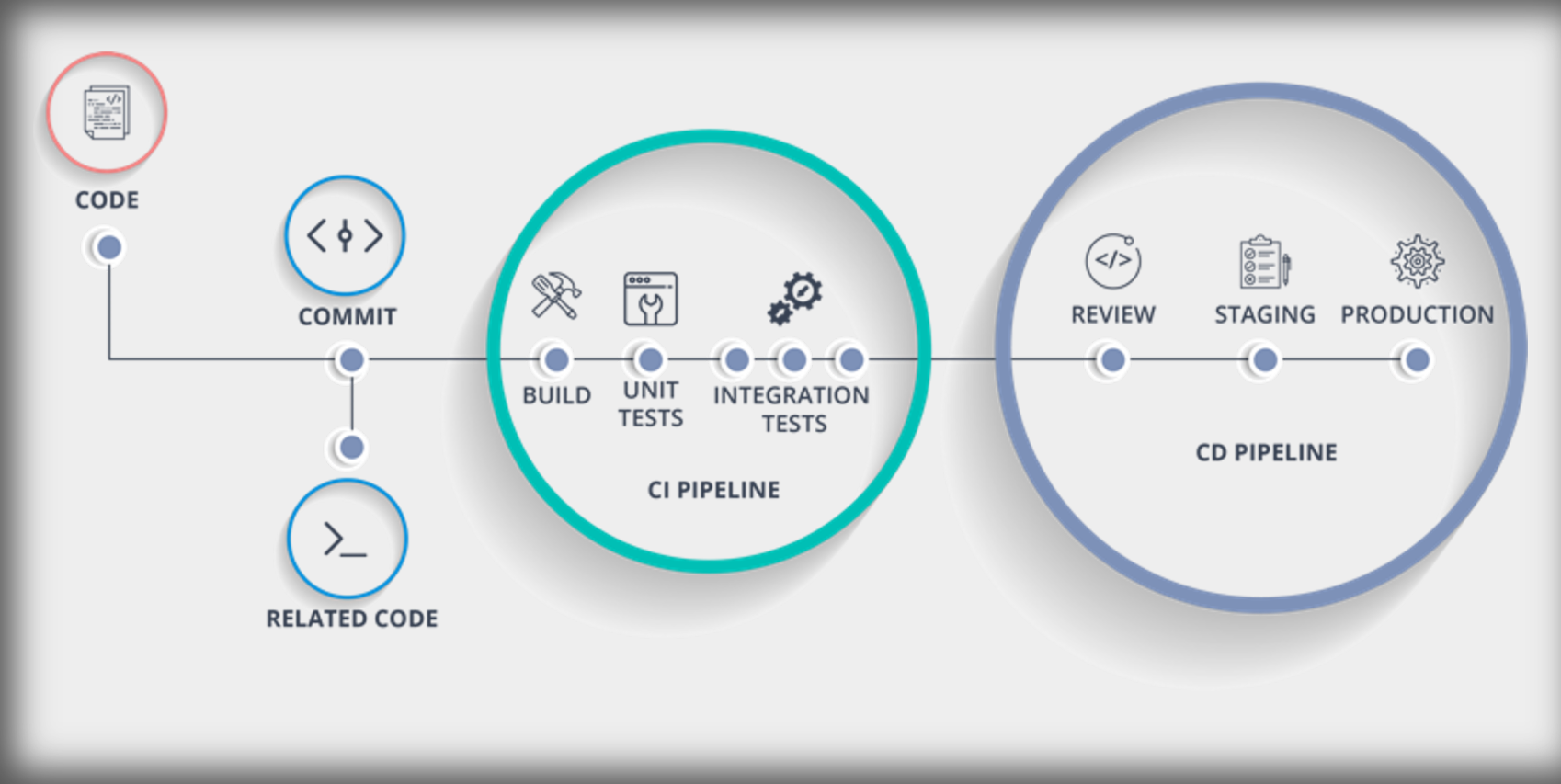
- If it's a bugfix branch, name it XXX-something where XXX is the number of the issue
- If it's a feature branch, create an enhancement issue to announce your intentions, and name it XXX-something where XXX is the number of the issue.

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



What should we test?
(What CAN we test?)

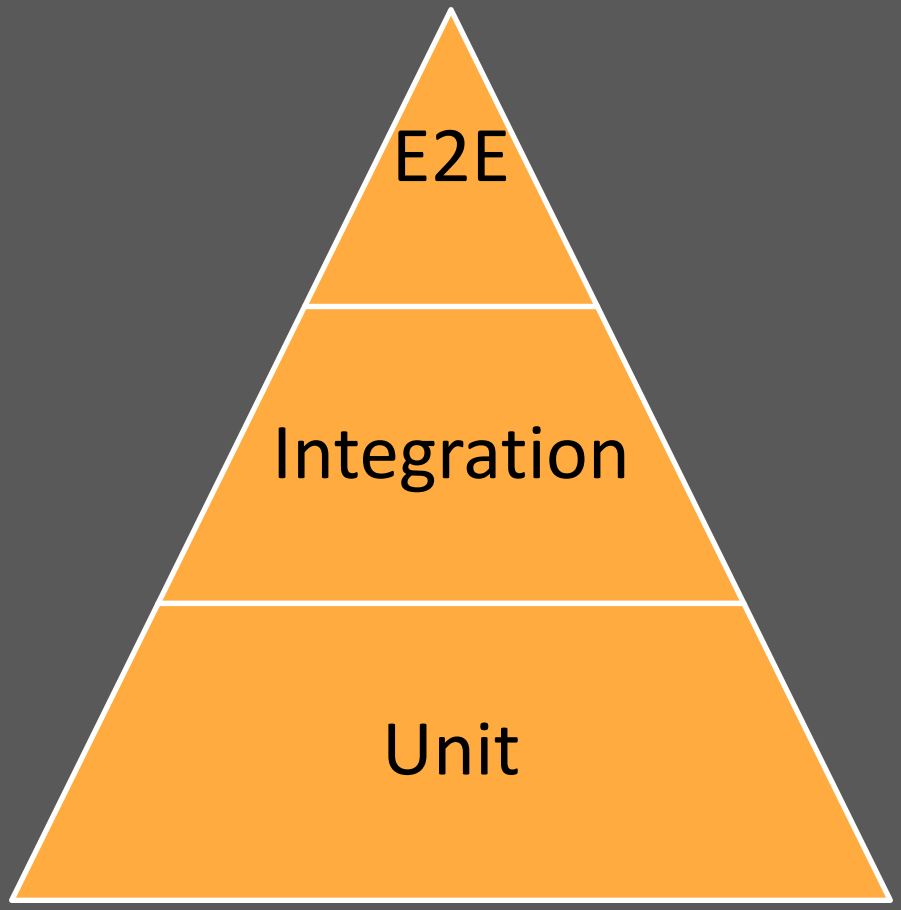
Testing Levels

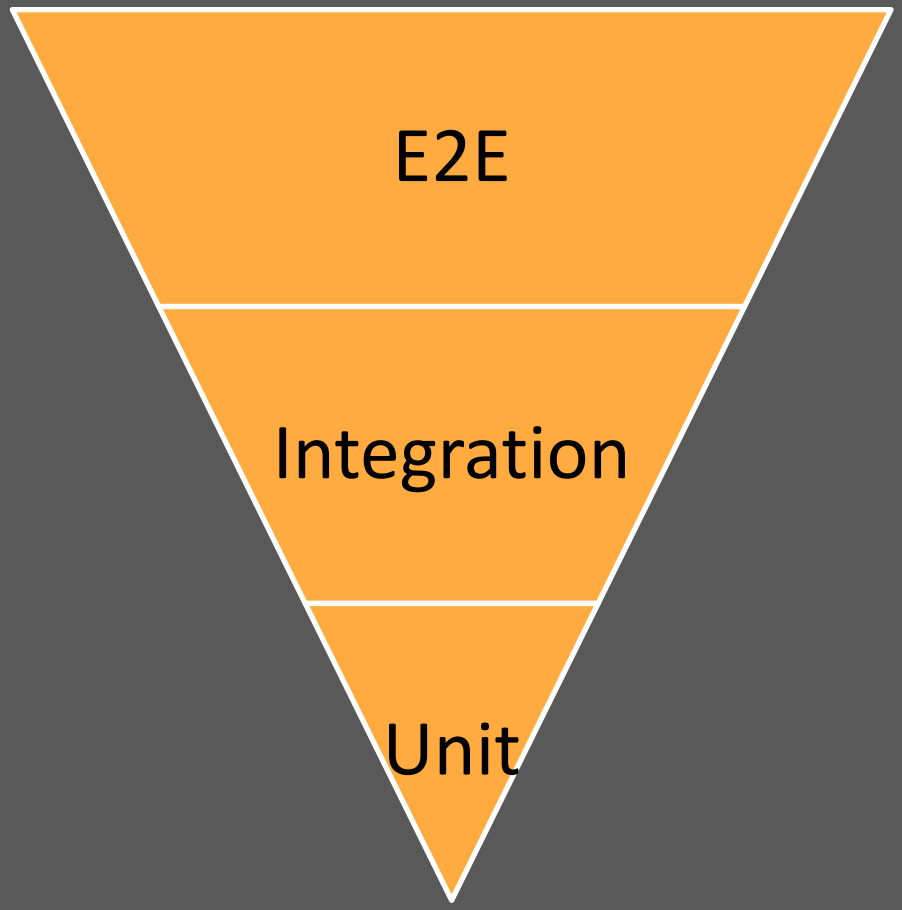
- Unit testing
- Integration testing
- System testing

Testing Levels

- 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

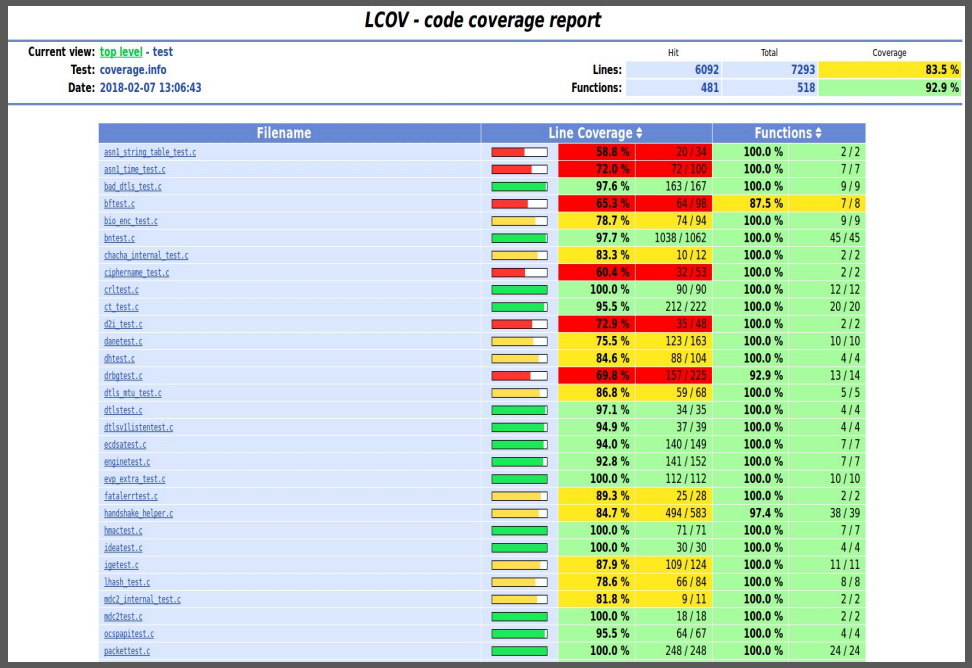
What's a good distribution of test levels?





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

Code Coverage

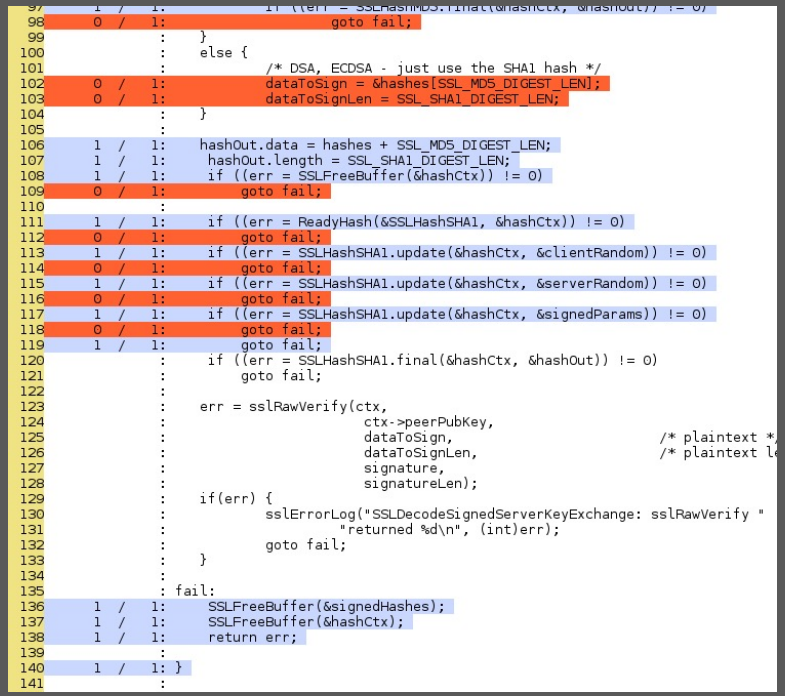


LCOV - code coverage report

Current view: top level - test
Test: coverage.info
Date: 2018-02-07 13:06:43

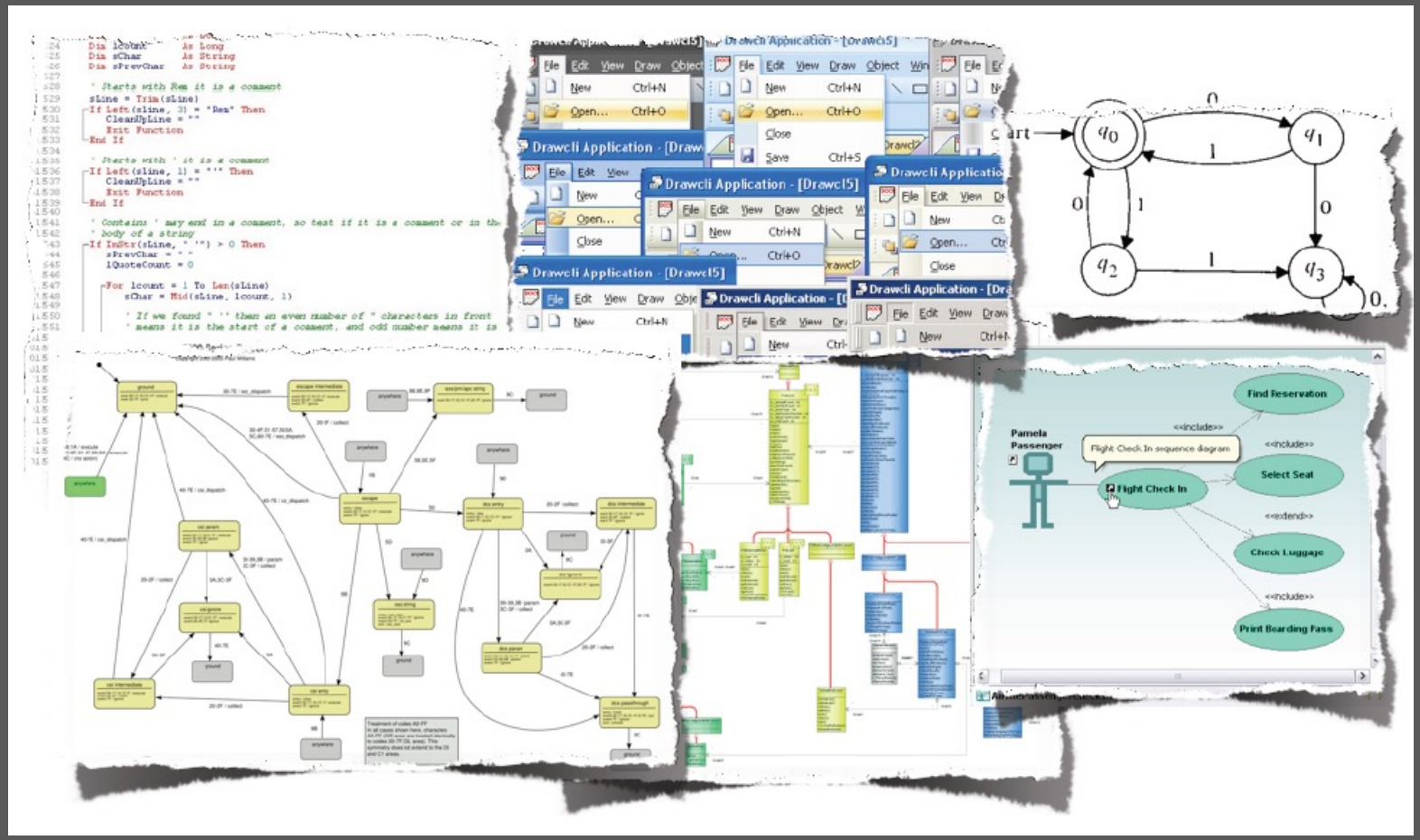
	Hit	Total	Coverage
Lines:	6092	7293	83.5 %
Functions:	481	518	92.9 %

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
dtlsvllistentest.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: }
        :
```

We can measure coverage on almost anything



A. Zeller, Testing and Debugging Advanced course, 2010

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 of what?

- Distinguish code being tested and code being executed
- Library code >>>> Application code
  - Can selectively measure coverage
- All application code >>> code being tested
  - Not always easy to do this within an application

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?

How should we test?

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available (Maven, Gradle, etc.)
- Can be used as design mechanism

```
@Test
public void testSort() {
    int[] input = {8, 16, 15, 4, 42, 23};
    int[] output = {4, 8, 15, 16, 23, 42};
    assertEquals(sort(input), output);
}
```

Basic Elements of a Test

```
@Test
public void testSort() {
    int[] input = {8, 16, 15, 4, 42, 23};
    int[] output = {4, 8, 15, 16, 23, 42};
    assertEquals(sort(input), output);
}
```

- Tests usually need an *input* and *expected output.*

- More generally, a *test environment,* a *test harness*, and a *test oracle*
  - Environment: Resources needed to execute a family of tests
  - Harness: Triggers execution of a test case (aka *entry point*)
  - Oracle: A mechanism for determining whether a test was successful

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)

Next Week

How many ways can you test the sort() function?
And other matters on designing tests…