# CS 312 Software Development

**Testing**

---

---
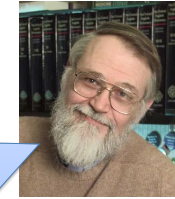
# Testing in an "agile" workflow

Previously (Waterfall, et al.)
  Developers finish code, some ad-hoc testing
  Toss over the wall to Quality Assurance (QA)
  QA staff manually poke at software

Agile
  Testing is part of *every* Agile iteration
  Developers test their own code
  Testing tools & processes highly automated
  QA/testing group improves testability & tools

---

# Test-driven development (TDD)

- Think about one thing the code *should* do

- Capture that thought in a test, which fails

- Write the simplest possible code that lets the test pass

- Refactor: DRY out commonality w/other tests

- Continue with next thing code should do

<div align="center">

Red – Green – Refactor

*Aim for "always have working code"*

</div>

# Hierarchy of testing

- **System (or end-to-end) testing**: Testing the entire application (typically to ensure compliance with the specifications, i.e. "acceptance" testing)

- **Integration testing**: Tests of combinations of units (i.e. integration of multiple units)

- **Unit testing**: Tests for isolated "units", e.g. a single function or object

- **Static testing**: Compile or build time testing

---

# Where do I spend my effort?



Google testing blog      Kent C Dodds "Write tests. Not too many. Mostly integration."

---
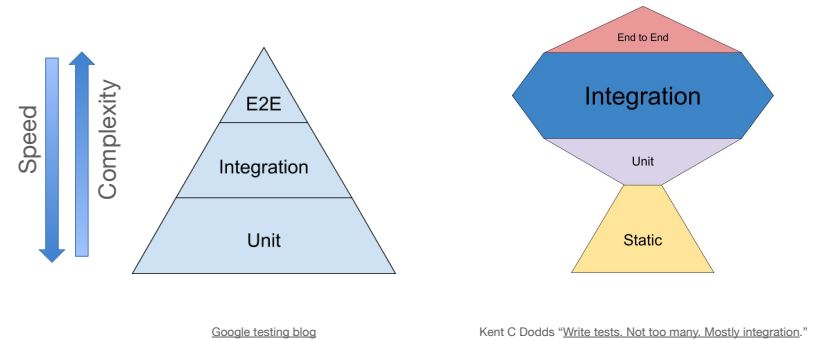
# How much testing is enough?

- Bad: "Until it's time to ship"

- A bit better: *X%* of coverage, i.e. 95% of code is exercised by tests

- Even better?

"You rarely get bugs that escape into production, [and] you are rarely hesitant to change some code for fear it will cause production bugs." –Martin Fowler

---

# Moderation in all things

✗ "I kicked the tires, it works"

✗ "Don't ship until 100% covered & green"

☑ Use coverage to identify untested or undertested parts of code


✗ "Focus on unit tests, they're more thorough"

✗ "Focus on integration tests, they're more realistic"

☑ Each finds bugs the other misses

## In spite of good testing, debugging happens

To minimize the time to solution take a "scientific" approach to debugging:

1. What did you expect to happen (be as specific as possible)?
2. What actually happened (again as specific as possible)?
3. Develop a hypothesis that could explain the discrepancy
4. Test your specific hypothesis (with console.log, the debugger, etc.)

*1 & 2 aren't that different than writing tests!*

---

# CS 312 Software Development
**Writing tests**

---

## Anatomy of a test (with <u>Jest</u>)

```
// Import fib function from module
const fib = require('./fibonacci');
```

Set of tests with common purpose, shared setup/teardown

```
describe('Computes Fibonacci numbers', () => {
  test('Computes first two numbers correctly', () => {
    expect(fib(0)).toBe(0);
    expect(fib(1)).toBe(1);
  });
});
```

Individual test

One or more:
expect(*expression*).*matcher*(*assertion*)

---

## Tests should be F.I.R.S.T.

- **F**ast: Tests need to be fast since you will run them frequently
- **I**ndependent: No test should depend on another so any subset can run in any order
- **R**epeatable: Test should produce the same results every time, i.e. be deterministic
- **S**elf-checking: Test can automatically detect if passed, i.e. no manual inspection
- **T**imely: Test and code developed concurrently (or in TDD, test developed first)

## How would you test this function?

```
const moment = require('moment');
const isBirthDay = function (birthday) {
  // moment() initializes with current date
  return moment().isSame(birthday, 'day');
};

describe('Checks if today is birthdate', () => {
  let _Date;
  beforeAll(() => { _Date = Date; });

  afterAll(() => { Date = _Date; // Reset Date });

  beforeEach(() => {
    Date.now = // Set a fixed date
      jest.fn(() => new Date('01 Jan 2018').valueOf());
  });
  …
});
```

moment() calls Date.now(). Replace with "mock" function to control current date

## An example of *seams*

•**Seam**: A place where you can change app's *behavior* without changing its *source code*. -Michael Feathers, *Working Effectively With Legacy Code*

- Useful for testing: *isolate* behavior of code from that of other code it depends on
- Here we use JS's flexible objects to create a seam for `Date.now()`
- Make sure to reset all mocks, etc. to ensure tests are Independent

## Seams, not just for Independence

Development is an iterative process

- Work from the "outside in" to identify code "collaborators"
- Implement "the code you wish you had" at seam
- Efficiently test out the desired interface