Turing Award winner Edsger Dijkstra
Turing Award winner Brian Kernighan, co-inventor of the C language

The heart of Kernighan’s quote is that only formal methods that prove correctness can demonstrate the absence of bugs. But those tools only work in very limited scenarios (that aren’t relevant to what we will be working on). Thus for us, testing can’t prove the negative, the absence of bugs, only show the presence of bugs. Thus we want to try to write tests that are likely to expose possible bugs, i.e. the test both “happy” paths (working code) and “sad” paths (error scenarios), with particular attention to corner cases that are likely to reveal bugs.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
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

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Hierarchy of testing (from “high” to “low” level)

- **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
What are the advantage of unit tests (in the view of the Google authors)? Fast, Reliable, and Isolates failure (each unit test is focused on a small part of the code). The limitation is that the tests don’t simulate a “real user”.

Why the difference between Google and Dodds? Unit testing doesn’t verify components work together. In a UI setting, for example, unit tests often verify that framework works as documented (click invokes handler) not that your application does the right thing in that handler.

“It doesn’t matter if your button component calls the onClick handler if that handler doesn't make the right request with the right data! So while having some unit tests to verify these pieces work in isolation isn’t a bad thing, it doesn’t do you any good if you don’t also verify that they work together.” -Kent Dodds
Recall our focus is on agile development methods, which are all about short development cycles that improve working (but not yet complete) code. To that end we will practice test-driven development in which we write the tests first, then implement the code that passes those tests (I suspect this is very different from the way you typically work...)

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Anatomy of a test (with **Jest**)

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

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

Set of tests with common purpose, shared setup/teardown

One or more expectations/assertions:

- `expect(expression).matcher(assertion)`

See notes with additional tests
Tests should be F.I.R.S.T.

- **Fast**: Tests need to be fast since you will run them frequently
- **Independent**: No test should depend on another so any subset can run in any order
- **Repeatable**: Test should produce the same results every time, i.e. be deterministic
- **Self-checking**: Test can automatically detect if passed, i.e. no manual inspection
- **Timely**: Test and code developed currently (or in TDD, test developed first)

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Which of the following kinds of code can be tested *Repeatedly* and *Independently* (without additional tools)?

1. Code that relies on randomness (e.g. shuffling a deck of cards)
2. Code that relies on the time of day (e.g. run backups every Sunday at midnight)

A. Only 1  
B. Only 2  
C. Both   
D. Neither

Answer: D

Both kinds of code are not repeatable, since the random shuffle could be different each time and the backup test would depend on the current day of the week when testing. Both would require some form of mocking to make the random number generator deterministic and control the "current" day of the week.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
How would you test this function?

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

describe('Checks if today is birthdate', () => {
    let _Date;
    beforeEach(() => { _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
```

Mock functions allow us to control the return value (and assert it was called with specific arguments, etc.)

Does this make our tests FIRST? Mocking makes them repeatable, and by saving the original Date module so we can restore it, we make our tests independent.

Note that we could tested this by creating inputs offset from the current date, e.g. today, yesterday, etc.
An example of *seams*

**Seam**: A place where you can change app's *behavior* without changing its *source 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

Definition of seam from Michael Feathers in *Working Effectively With Legacy Code*

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
What is the primary benefit of mocking an external service (such as an API accessed via the Internet)?

A. Keeps each test Independent (from other tests)
B. Keeps tests Fast and Repeatable
C. Tests can be written before the code
D. Tests can be self checking

Answer: B (although A is arguable)

The primary benefit is to keep tests Fast and Repeatable. Fast because they are not actually accessing the Internet and Repeatable because the tests are not dependent on the current state of an external service (that may not be under your control). Simply creating mocks does not automatically make tests independent. You need to make sure the mocks are created in a way (e.g. initialized for each test) that guarantees tests are independent.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
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

Example, what if we wanted search functionality...

• Who would collaborate to provide that search (mock a search function on our database)
• Test the “happy path” with search results, and the “sad path” without

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Coverage alone is limited measure of test quality. A high-quality test suite will likely have high coverage but a high coverage test suite does not guarantee high quality. A key use for code coverage can be to help you find the portions of the code base that are not being tested.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
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

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
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!

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
The R.A.S.P. method

- **Read** the error message *(really read it).*
- **Ask** a colleague an *informed* question, not just "Why doesn't it work?".
- **Search** using keywords from error, specific SW versions, etc..
- **Post** on StackOverflow, Piazza, etc. Everyone is busy, you will get better answers if you provide a [Minimal, Complete and Verifiable](https://en.wikipedia.org/wiki/Minimal_complete_and_verifiable) example

*When you fix a bug, make a test!*

Or in keeping with TDD, once you identify the bug, make a test that exercises the bug (and thus fails), fix the bug and then verify that your test now passes.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Student advice: TDD & Testing

- “TDD is hard at first, but it gets easier”
- “Was great for quickly noticing bugs [regression] and made them easy to fix”
- “Helped me organize my thoughts as well as my code” [the code you wish you had]
- “Wish we had committed to it earlier & more aggressively”
- “We didn’t always test before pushing, and it caused a lot of pain”

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Student advice: Test coverage, code quality

- “Good coverage gave us confidence we weren’t breaking stuff when we deployed new code”
- “Felt great to get high grade from [coverage estimator]”
- “Pull-request model for constant code reviews made our code quality high”
- “Wish we had committed to TDD + coverage measurement earlier”

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.