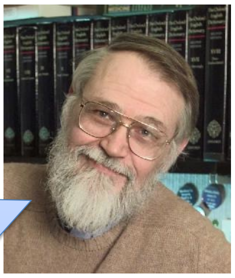
Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

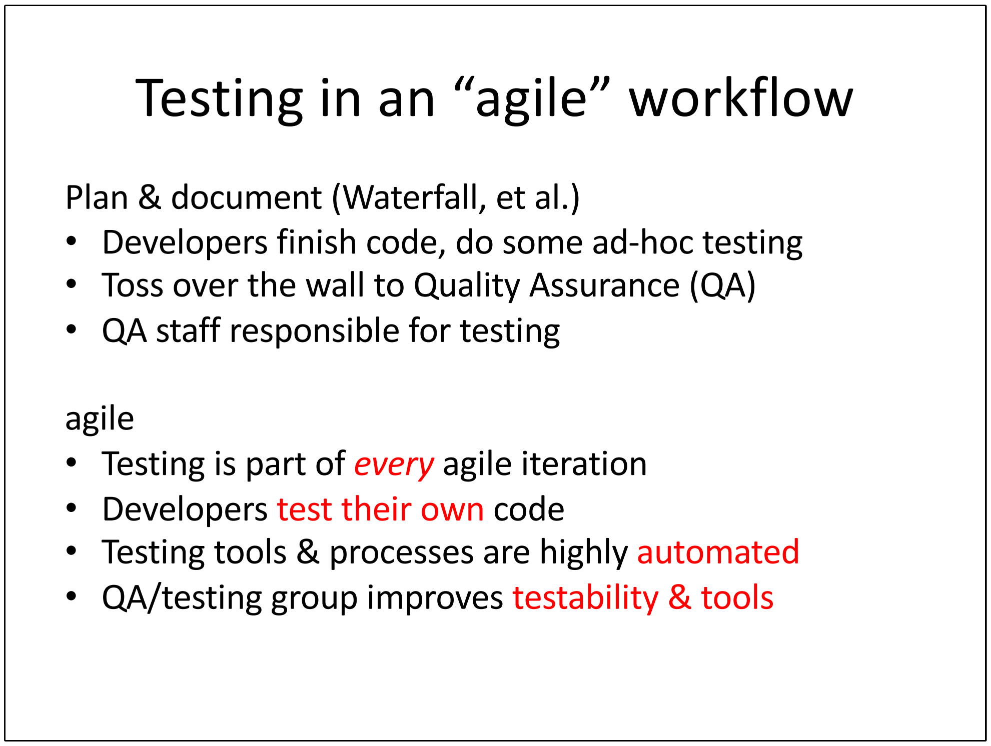


Testing can never demonstrate the absence of errors in software, only their presence

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

The heart of Dijkstra's message 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

Plan & document (Waterfall, et al.)

- Developers finish code, do some ad-hoc testing
- Toss over the wall to Quality Assurance (QA)
- QA staff responsible for testing

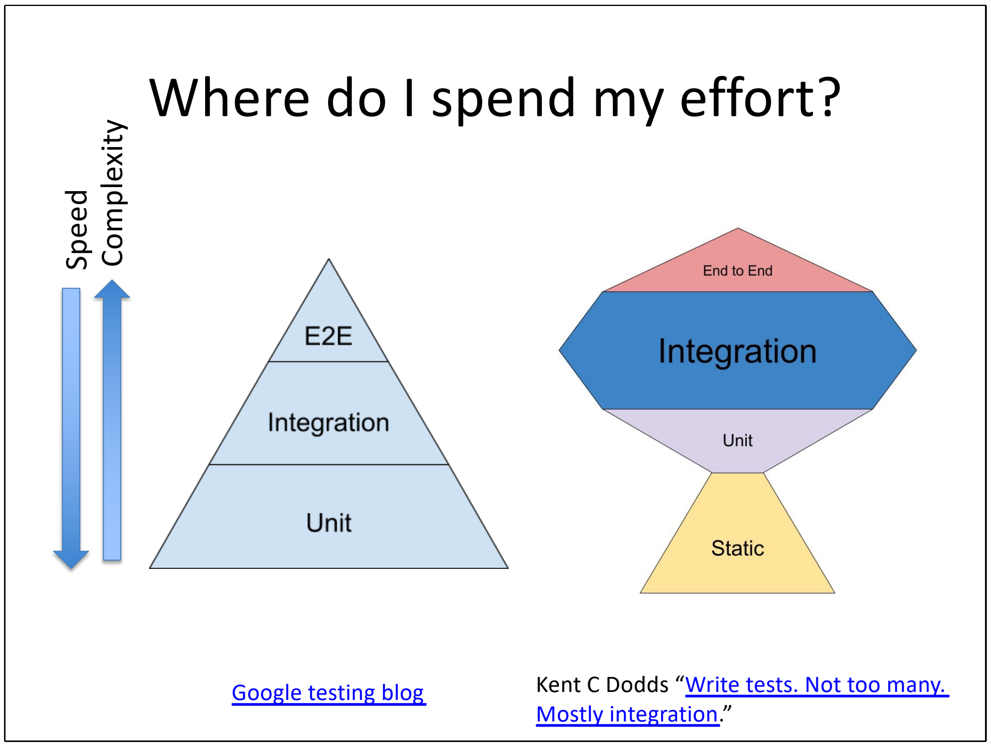
agile

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

What do I mean by "ad hoc" testing — it is the kind of testing we all do (myself included!)... You try a couple of examples and when it doesn't blow up, you declare it working!

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
- **Static testing (analysis):** Compile or build time testing/analysis



Where do I spend my effort?

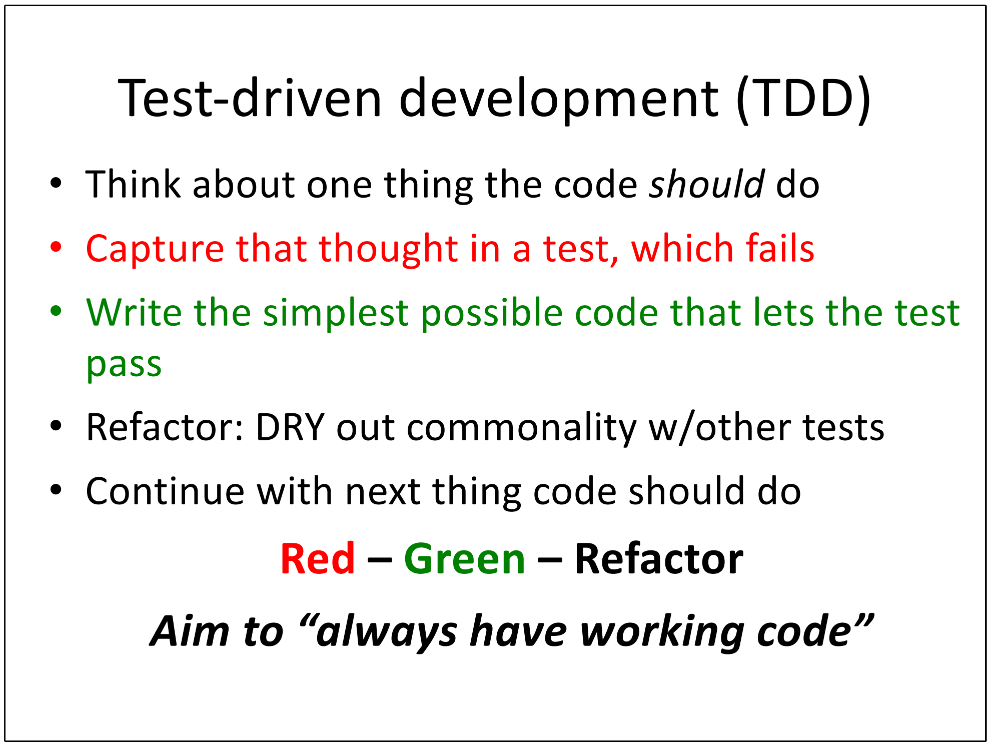
Google testing blog

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

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 the Google blog post 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



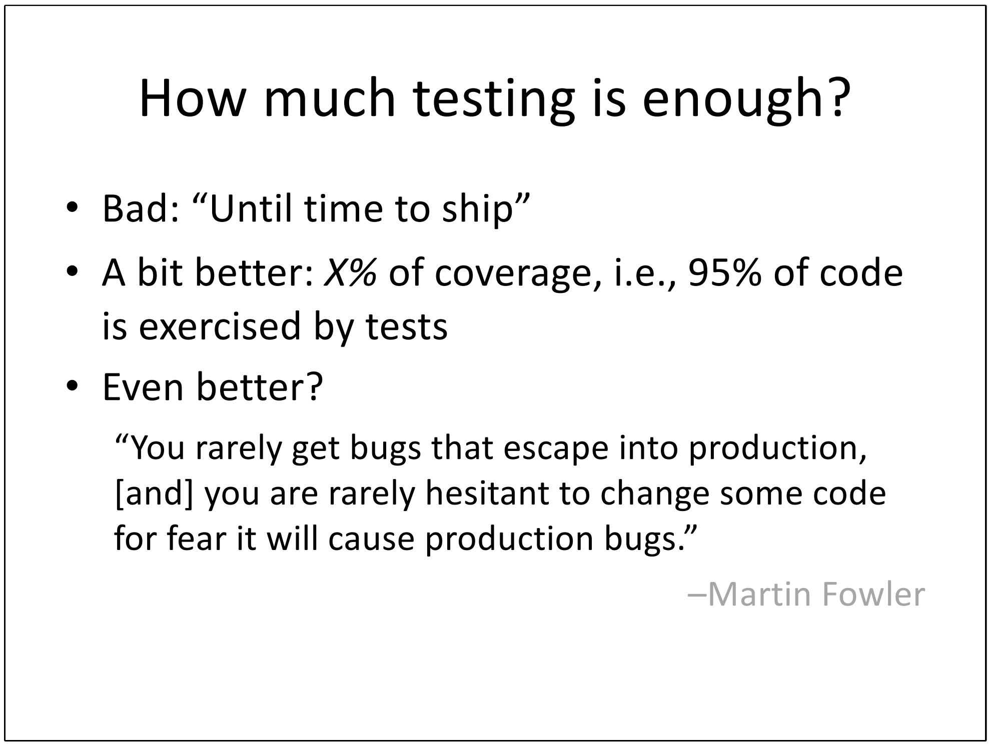
Test-driven development (TDD)

- 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

Red – Green – Refactor

Aim to "always have working code"

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...)



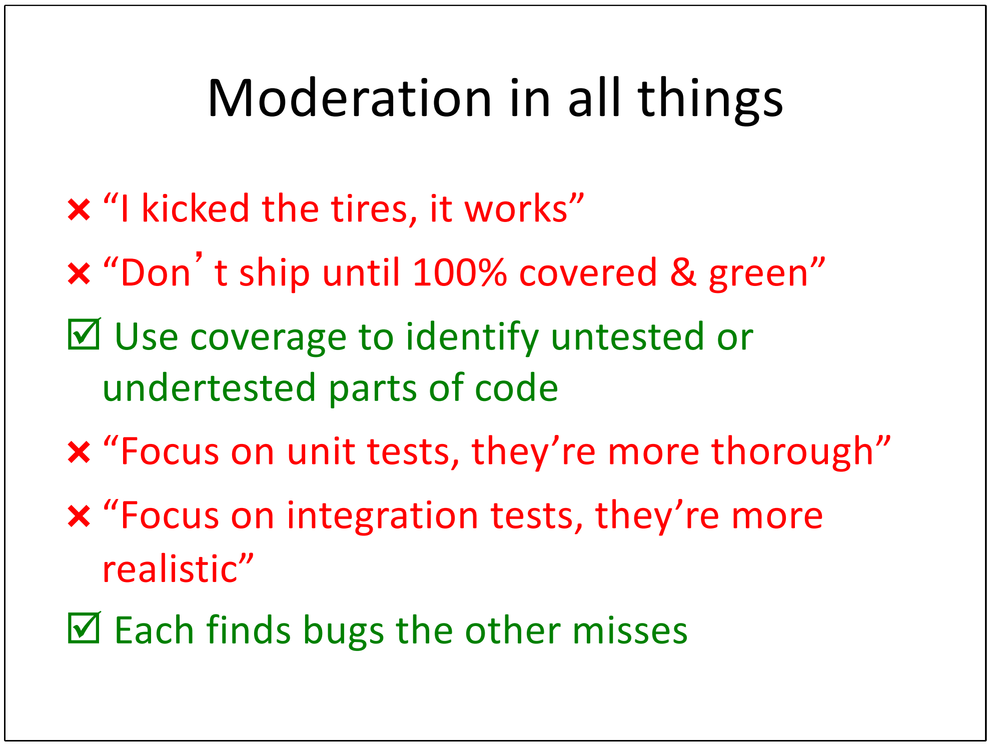
How much testing is enough?

- Bad: “Until 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.”

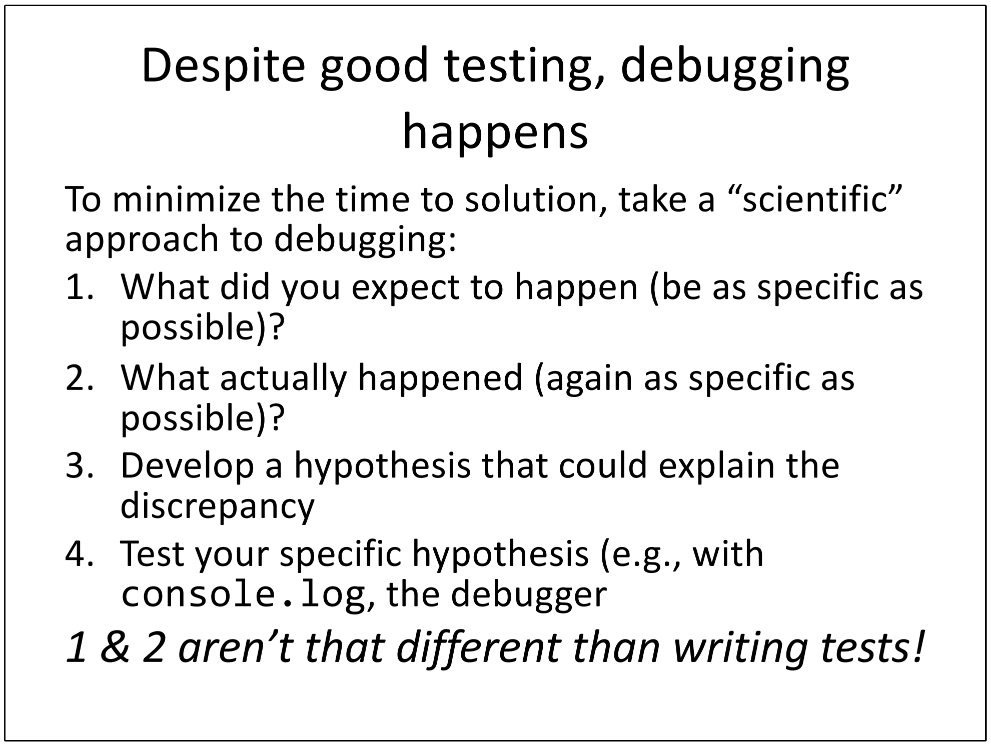
–Martin Fowler

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.



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



Despite good testing, debugging happens

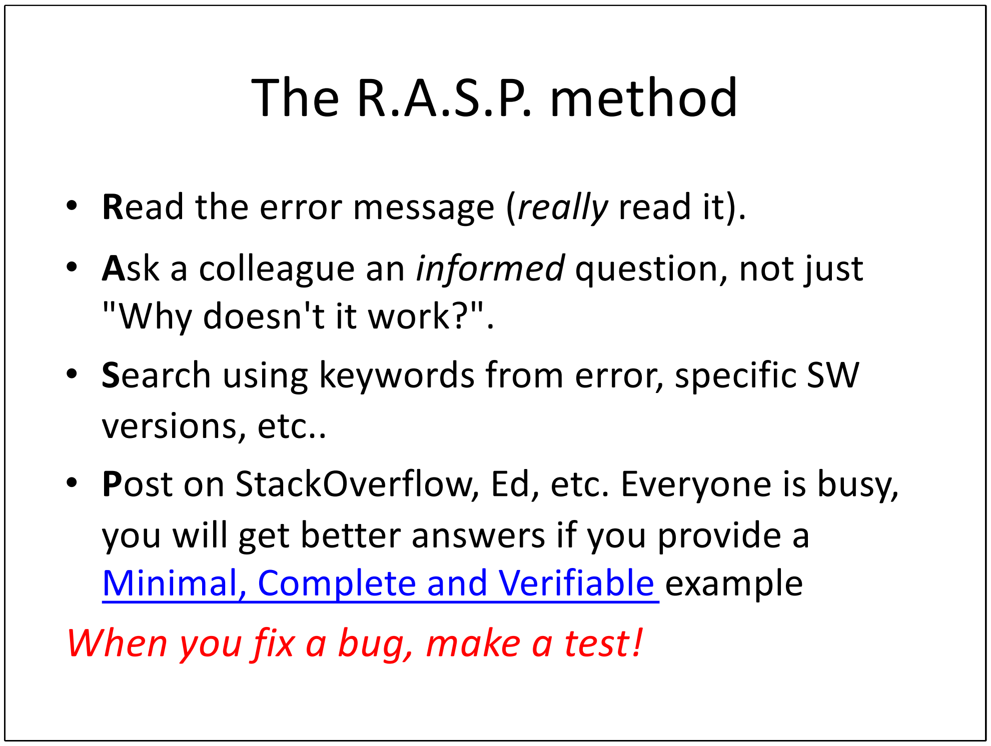
To minimize the time to solution, take a "scientific" approach to debugging:

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 (e.g., with `console.log`, the debugger

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

One of the things I find most challenging about software development, generally, is the indeterminate time required for debugging. No amount of experience suddenly makes debugging predictable. You will have to spend the time (and often an unknown amount of time). But I want to distinguish between good time and bad time. This type of scientific debugging represents good time. You have a specific hypothesis of the problem and tests that you can use to confirm/falsify that hypothesis. When you no longer have a specific idea, when you are just making changes randomly, that is "bad time" where you have stopped being productive. That is a good moment to pause, do something else to clear your head and seek out help (in our case, Ed discussion board, drop-in hours, office hours, your classmates). Our assignments are not intended to be trivial, but they are also not intended to be slogs. Before you start, set an expected amount of time for yourself. When that time has elapsed evaluate if you are spending good time or bad time.

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



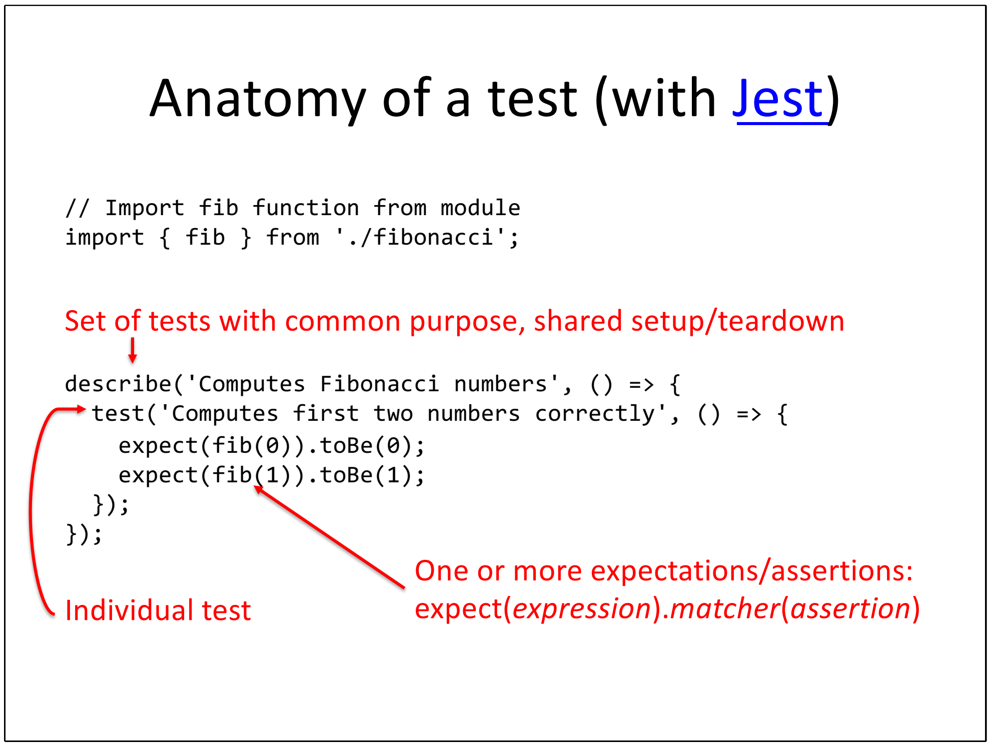
The R.A.S.P. method

- **R**ead the error message (*really* read it).
- **A**sk a colleague an *informed* question, not just "Why doesn't it work?".
- **S**earch using keywords from error, specific SW versions, etc..
- **P**ost on StackOverflow, Ed, etc. Everyone is busy, you will get better answers if you provide a 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.



Anatomy of a test (with Jest)

```
// Import fib function from module
import { fib } from './fibonacci';
```

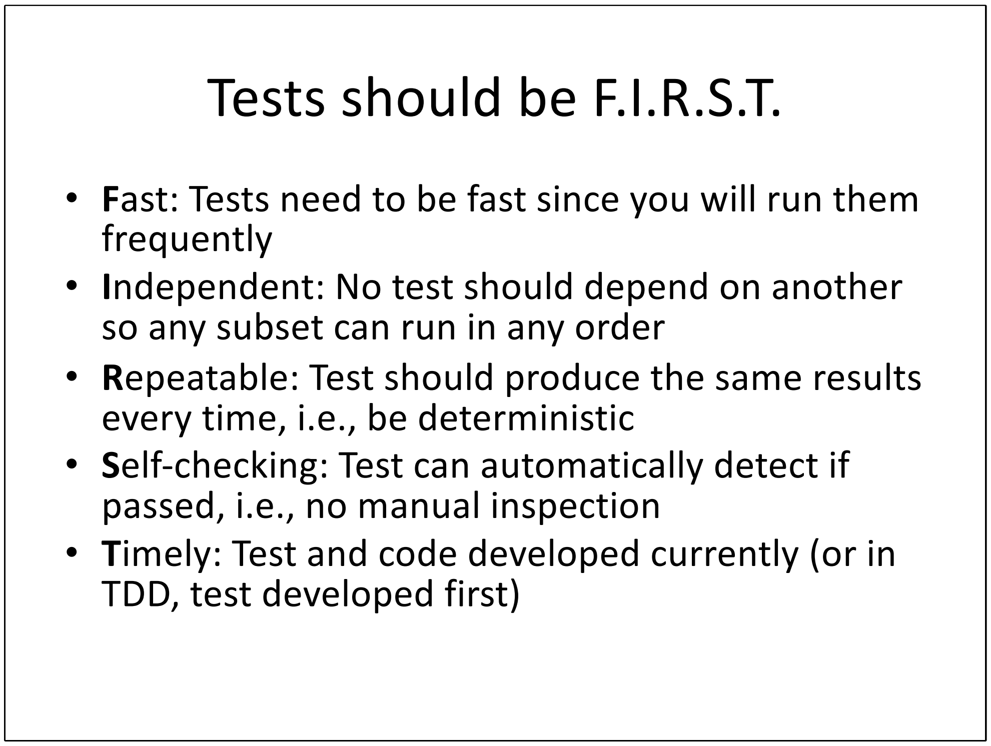
Set of tests with common purpose, shared setup/teardown

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

Individual test

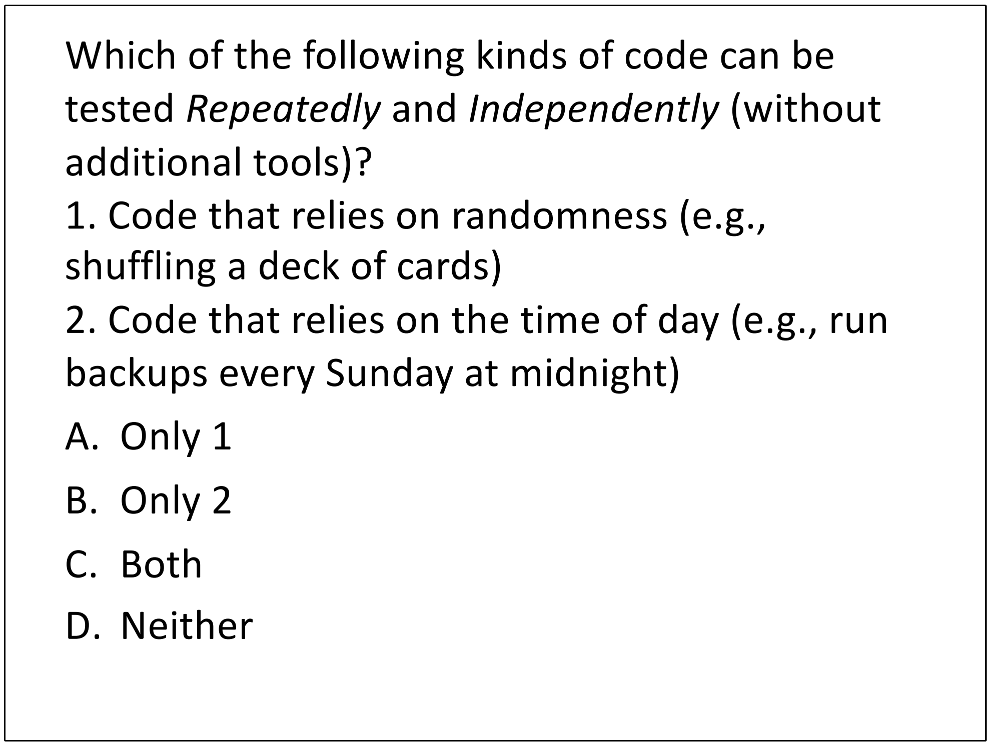
One or more expectations/assertions: expect(*expression*).*matcher*(*assertion*)

Note that this just the start of the tools available within Jest. Check out its documentation, with a particular eye to the many different matchers it provides. In general, we want to use the most specific matcher possible, i.e., expect(fib(0)).toBe(0) is preferred over expect((fib(0) === 0).toBeTruthy(). Why? The test is clearer (and thus easier to maintain) and we will get more informative error messages, i.e., that we didn’t get the return value expected as opposed we didn't get a the Boolean value we expected.



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 currently (or in TDD, test developed first)



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

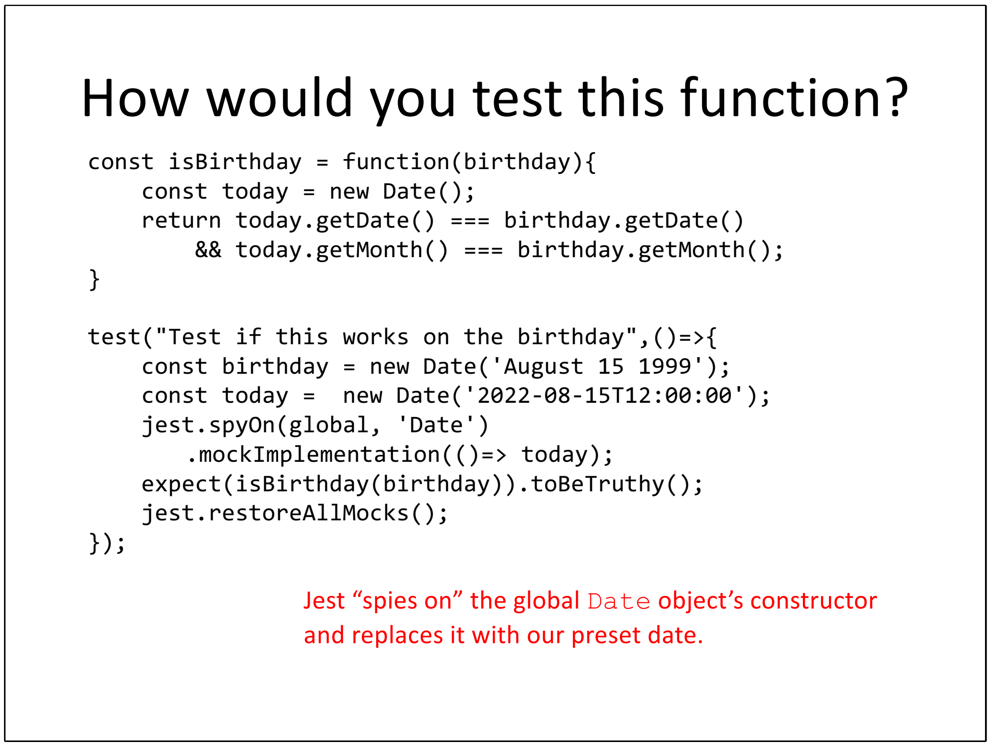
B. Only 2

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 (replacing code during testing) to make the random number generator deterministic and control the "current" day of the week.



How would you test this function?

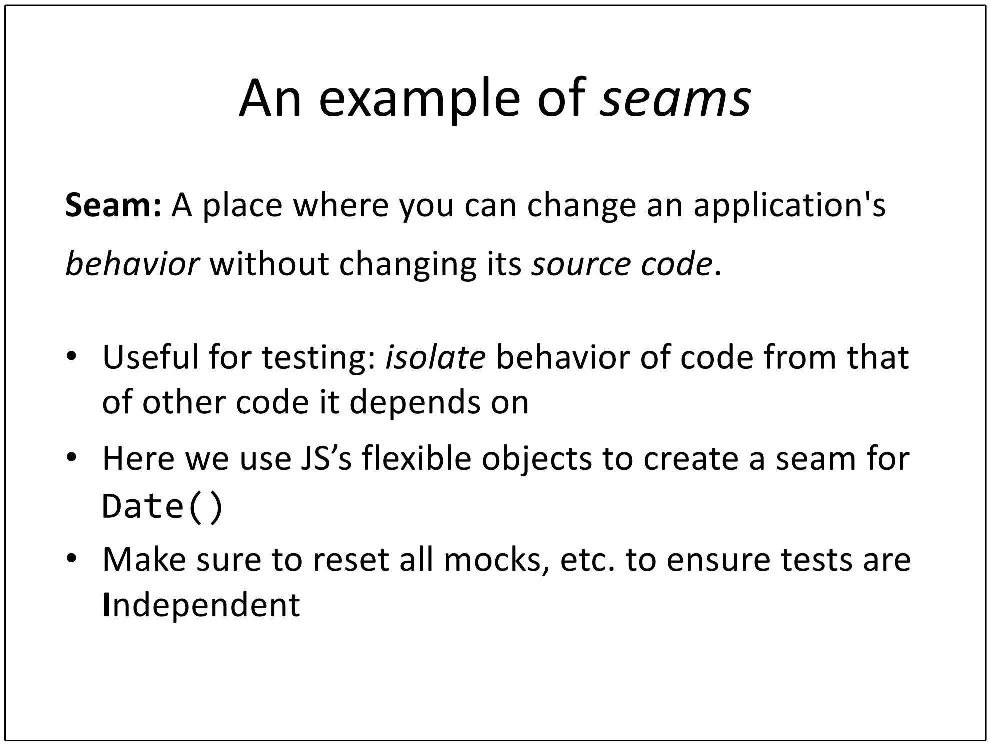
```
const isBirthday = function(birthday){
    const today = new Date();
    return today.getDate() === birthday.getDate()
        && today.getMonth() === birthday.getMonth();
}

test("Test if this works on the birthday",()=>{
    const birthday = new Date('August 15 1999');
    const today =  new Date('2022-08-15T12:00:00');
    jest.spyOn(global, 'Date')
       .mockImplementation(()=> today);
    expect(isBirthday(birthday)).toBeTruthy();
    jest.restoreAllMocks();
});
```

Jest "spies on" the global `Date` object's constructor and replaces it with our preset date.

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

Does just creating the mock make our tests FIRST? No, it gets up part of the way there. The mock makes the test repeatable, while invoking restoreAllMocks, which resets properties, like the Date constructor, back to their original value, make his test independent (or more precisely other tests independent of this one).

Note that a full test suite would also test the "sad path" for days that aren't the birthday.



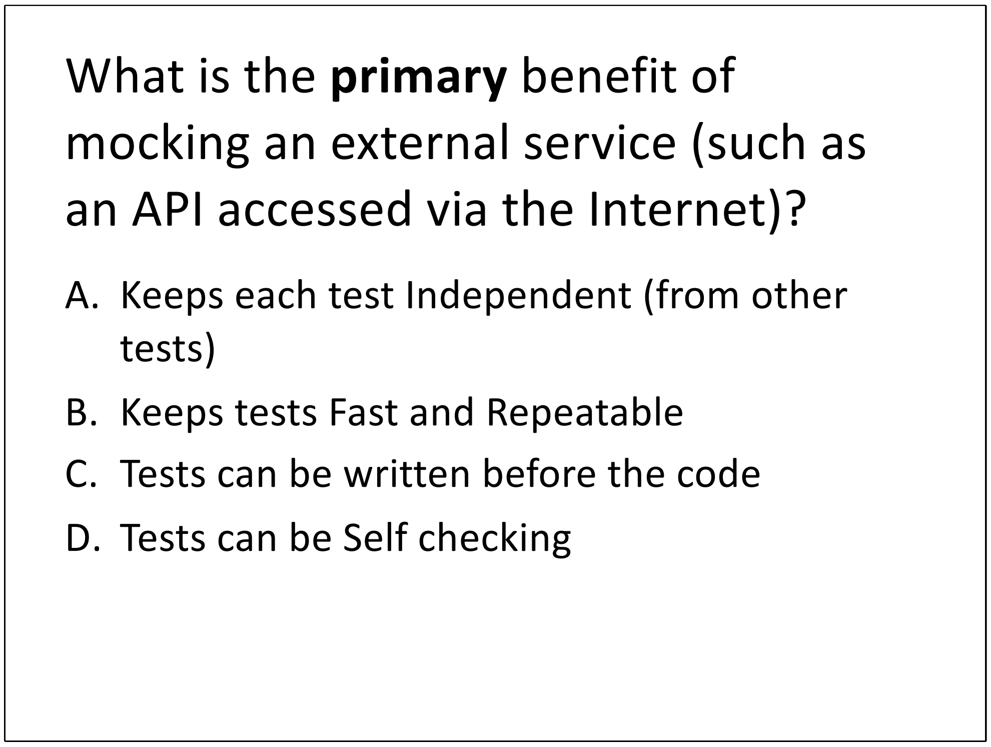
An example of *seams*

Seam: A place where you can change an application'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()`
- Make sure to reset all mocks, etc. to ensure tests are **I**ndependent

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.

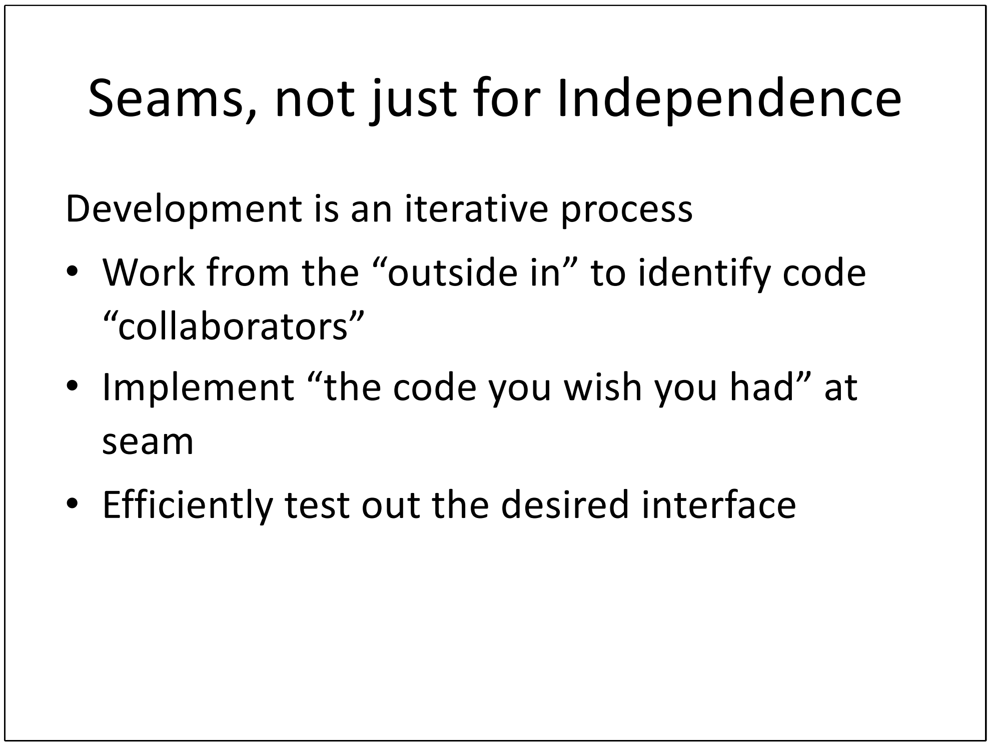


What is the **primary** benefit of mocking an external service (such as an API accessed via the Internet)?

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. As we saw before, you need to make sure the mocks are created in a way (e.g., initialized for each test) that guarantees tests are independent.



Seams, not just for Independence

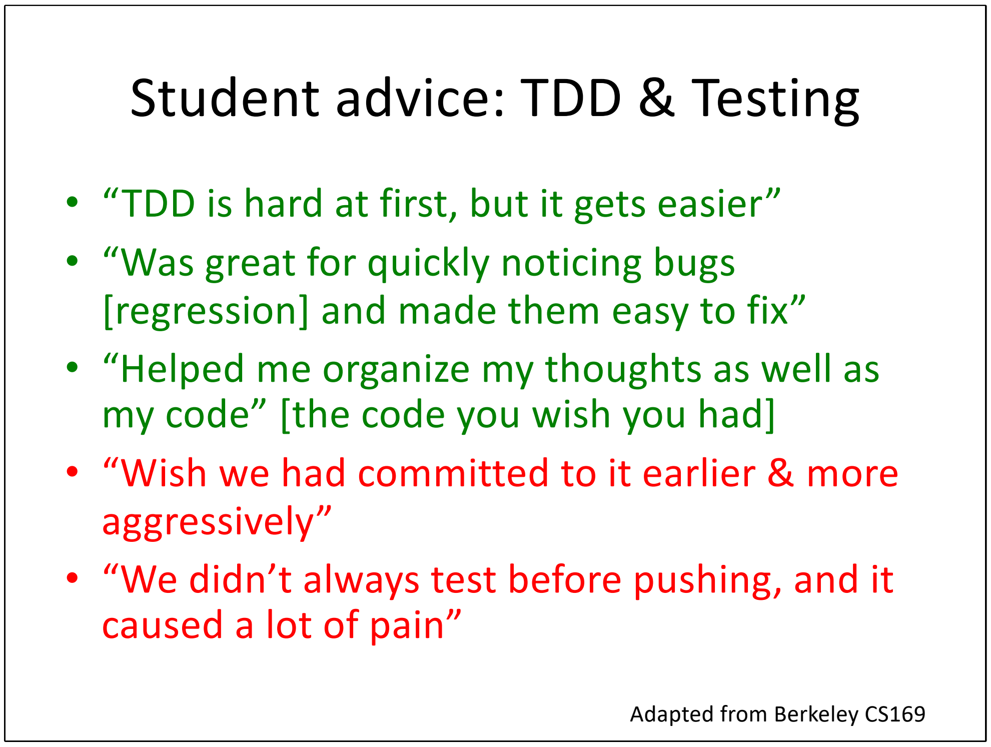
Development is an iterative process

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

I.e., one of the roles for seams is to decouple development of different portions of the application.

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

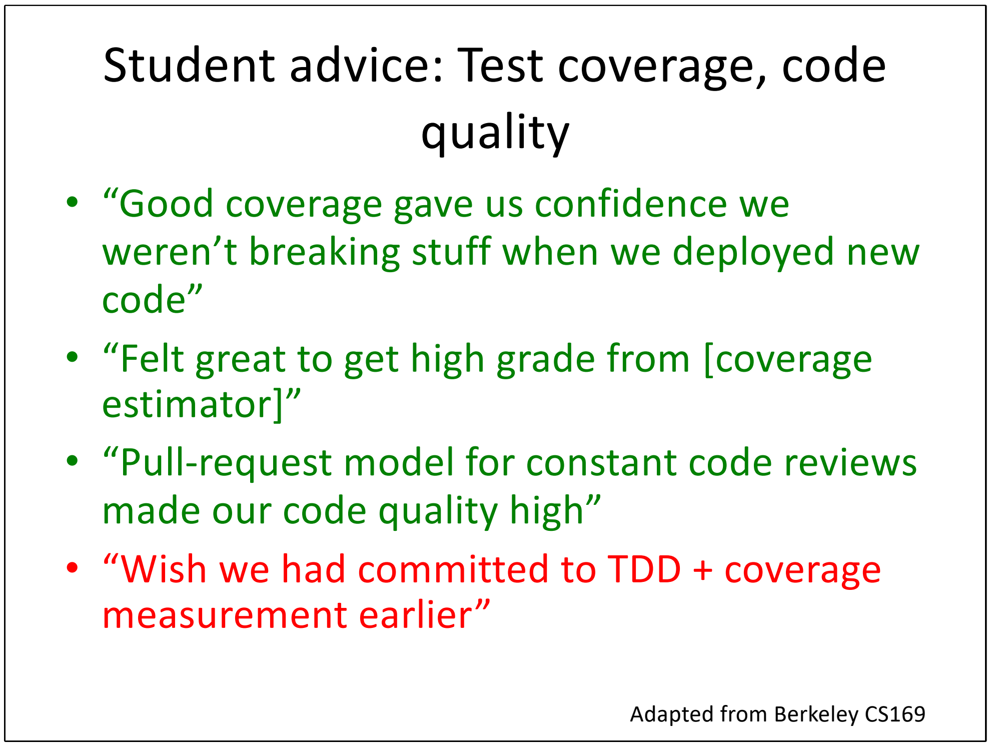
Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



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 Berkeley CS169



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 Berkeley CS169