1. Automated testing

What and why

What is automated testing?

- Testing a program with a program
- Many approaches, as we'll see
- Usually involves more code, written in parallel to actual application code

Why test? (1)

You get the perspective of the caller

- tests of a module will (typically) use the same interface that callers do
- helps keep things taut and minimal; reduces incidental complexity
- i.e. keep things <u>simple</u>

Why test? (2)

Continuous Deployment

- provides a safety net that you're not deploying a bug

Why test? (3)

tests are developer-oriented documentation

- reveals intentions of the code
- example

Why test? (4)

it helps future devs know whether they break something

- This enables agility
- What do we do when we don't feel confident that we understand? We become hesitant and slow.
- Remember that courage is a core value of Extreme Programming

Why test? (5)

it enables refactoring

- another source of hesitation: striving for excellence in code
- refactoring essentially means to improve the clarity of code
- key insight: decouple getting it *working* from getting it *simple*
- write tests, get it working, then refactor with *courage*

Why test? (6)

it enables tight feedback loops

- minimize the time gap
- if there's a bug, it's likely to be in what you just wrote, so no context switching needed for debugging
- improved "time locality"
- we'll see this in action later
- goal: traction and steady progress on your code

Why not test?

- It doesn't catch everything
    - But still catches many things
- It takes longer; you must write "extra" code
    - True, but often worth it, in my estimation
- It adds inertia to the interface
    - Interface changes are less agile

Analysis

So when should I test?

- Main factor: what's the likely longevity of the project?
  - Quick spike? Don't test.
  - Longer project? Higher likelihood of things changing with a long time gap. Test.

2. Testing basics

What to test

- Test the code of your project
- Don't test functionality of dependencies
- Typical setup: one test "suite" per code module/class/namespace
- And 1 test block (a "describe" block) per function/method
- And >= 1 test case per "behavior"

Test isolation spectrum

- "end-to-end" or "system" tests test everything working together
- for example, there are ways to remote control a browser to put a web app through its paces
- "unit" tests only test one unit in isolation, e.g. a particular function or method
- "integration" tests combine at least two units, testing units as well as interfaces between units
    - fuzzy term; some say "integration test" and imply some amount of integration with a system, especially a database

Testing tutorial

Testing exercise

1. git clone `https://gitlab.com/jeff.terrell/testing-tutorial.git`
2. cd testing-tutorial
3. npm install
4. npm test
5. edit src/sum.test.js and implement the first test
6. get to green: fix bug in src/sum.js
7. optionally, implement other two tests