Testing can never demonstrate the _______ of errors in software, only their _________.

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**.
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 are highly automated
• QA/testing group improves testability & tools
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?

- **Speed**
  - E2E
  - Integration
  - Unit
- **Complexity**
  - Static
  - Unit
  - Integration
  - End to End

---

Google testing blog

Kent C Dodds “Write tests. Not too many. Mostly integration.”
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”*
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
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!
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, 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!*
Anatomy of a test (with **Jest**)

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

Individual test

One or more expectations/assertions:

`expect(expression).matcher(assertion)`
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)
const isBirthday = function(birth){
    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.
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()`
- 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
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