CS 4530: Fundamentals of Software Engineering

Module 10: Testing Larger Things

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be prepared to:
  - Explain why you might need a test double in your testing
  - Design test cases for code using fakes, mocks and spies
  - Explain why you might need tests that are larger than unit tests
  - Explain how large, deployed systems lead to additional testing challenges

Why do we test?

- Unit Testing
  - Does the SUT satisfy its specification?
- Integration Testing
  - Do the SUT and its context work correctly *together*?
- Acceptance Testing
  - Does the SUT satisfy the customer
  - “Good” test suite answers: Are we building the right system ?

The Story So Far: Testing Things That Look Like Functions

- Unit tests are tests of functions with no dependencies

```
test('addStudent should add a student to the database', () => {
  expect(db.nameToIDs('blair')).toEqual([])
  const id1 = db.addStudent('blair');
  expect(db.nameToIDs('blair')).toEqual([id1])
});
```

- REST API tests are integration tests, but have the same basic shape

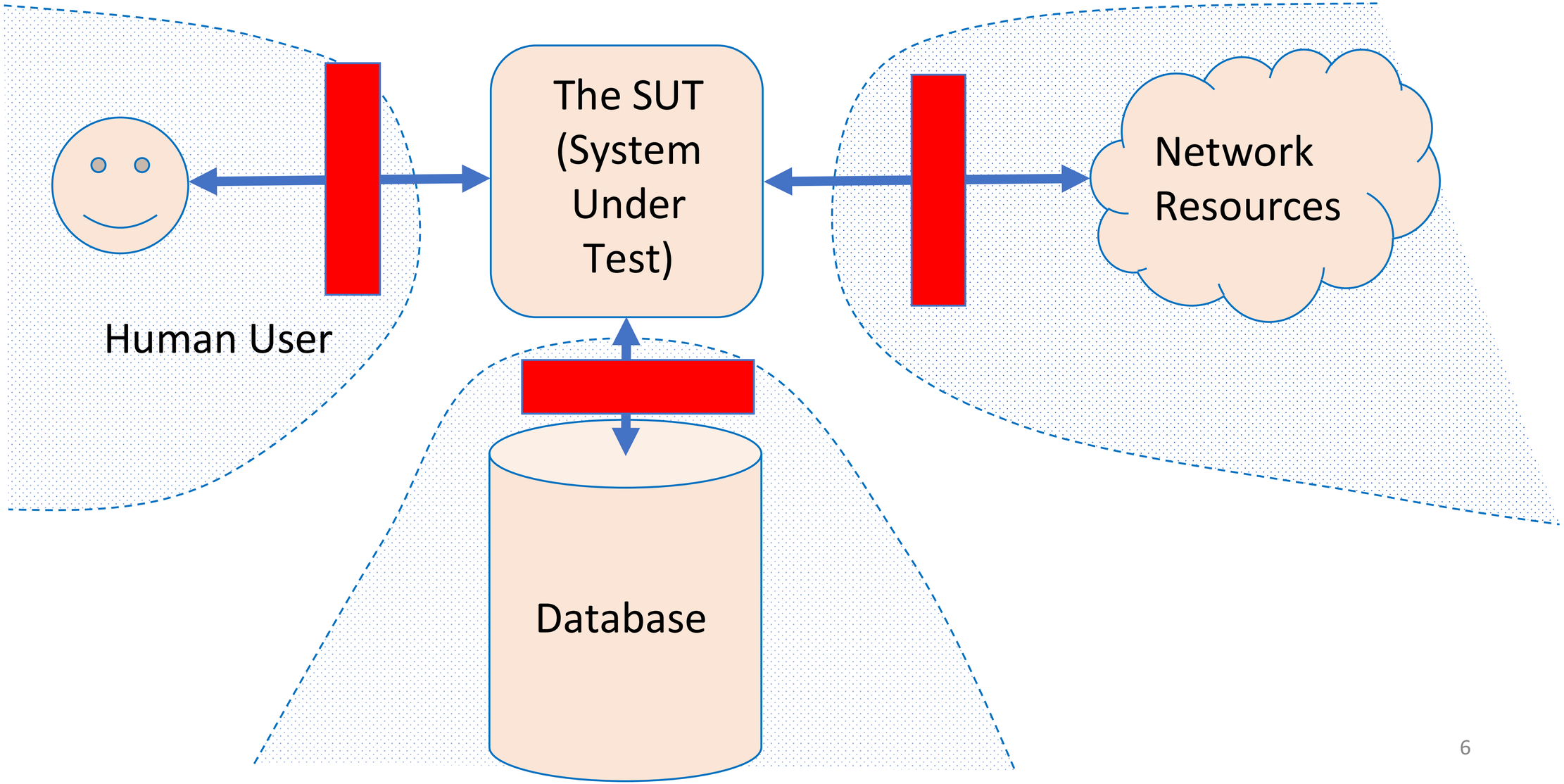
```
test('GET /api/user/:id should 404 for if no user', async () => {
  response = await supertest(app).get(`/api/user/nobody`);
  expect(response.status).toBe(404);
  expect(response.body).toStrictEqual({ error: 'No user' });
});
```

Why Does The “Story So Far” Stop Working?

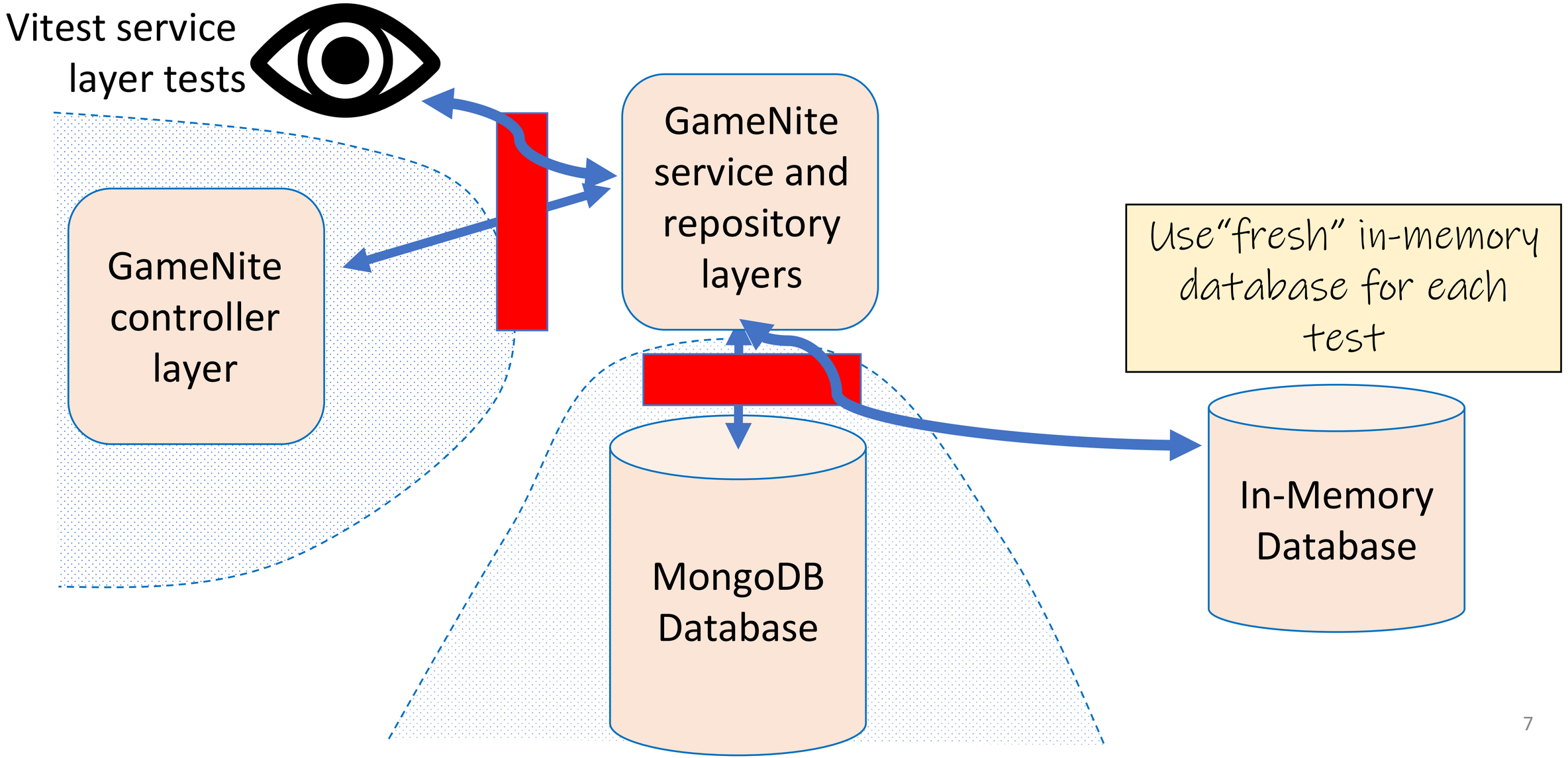
- Integration tests become too big and complicated, with many dependencies
  - Test doubles can replace dependencies to isolate the function

- Tests stop looking like functions (or we need to test non-function-like properties of the functions, like their effects)
  - Test doubles can track the changes to the environment in a way that we can double check from within our code.
  - UI tests don’t look like functions; UI testing frameworks like Playwright are a bit different.

Test doubles replace uncontrollable things with things that you do control



Test doubles replace uncontrollable things with things that you do control



“Test Doubles” Stand In For Other Components

- Act as a stand-in for components, allowing for testing in isolation
- **Fakes: Replace “real” implementation with some simplified, but at-least-sort-of-working, version for testing**
- Mocks: Automatically-generated fake implementations for an interface
- Spies: Automatically-instrument internals of objects, classes or modules

“Test Doubles” Stand In For Other Components

- Act as a stand-in for components, allowing for testing in isolation
- Fakes: Replace “real” implementation with some simplified, but at-least-sort-of-working, version for testing
- **Mocks: Automatically-generated fake implementations for an interface**
- **Spies: Automatically-instrument internals of objects, classes or modules**

How do we test this function? What do we even want to test?

```
/** Returns true half the time, o/w false */
function coinFlip() {
  return Math.random() >= 0.5;
}
```

"Test Doubles" Stand In For Other

Aside: **Test Oracles**

To assess a function, you need to know what it ought to do!

- Function returns the exact "right" answer
- Function returns an acceptable answer
- Returns the same value as last time

Snapshot testing is a version of this

- Function returns without crashing
- Function crashes (as expected)
- Function has the right effects on its environment

Mocks/spies are very good here

How do we test this function? What do we even want to test?

```
/** Returns true half the time, o/w false */
function coinFlip() {
  return Math.random() >= 0.5;
}
```

Spies *Instrument* Dependencies

```
/** Returns true half the time, otherwise false */
function coinFlip() {
  return Math.random() >= 0.5;
}

test('coinFlip should call Math.random', () => {
  const spyRandom = vi.spyOn(Math, 'random');
  coinFlip();
  expect(spyRandom).toHaveBeenCalled();
  expect(spyRandom).toHaveBeenCalledTimes(1);
});
```

Spied-On Functions Can Be Mocked, Changing Their Implementation

```
/** Returns true half the time, otherwise false */
function coinFlip() {
  return Math.random() >= 0.5;
}

test('coinFlip notices Math.random’s output', () => {
  const spyRandom = vi.spyOn(Math, 'random');
  spyRandom.mockImplementationOnce(() => 0.9);
  expect(coinFlip()).toBe(true);
  spyRandom.mockImplementationOnce(() => 0.1);
  expect(coinFlip()).toBe(false);
});
```

Spies Can be Used Even When You Can't (Fully) Control the SUT

- You can specify *any* object, and *any* method name (even private methods)
- Spy on objects *or* entire modules
- The spy logs *all* calls to that method of that object or module
- The call to the **original still gets made**, unless the spy explicitly supplies a substitute mock

Overspecification is Very Easy With Spies

```
/** Returns true half the time, otherwise false */
function coinFlip() {
  return Math.random() >= 0.5;
}
```

- Do we actually care that coinflip is getting its randomness from Math.random?
- Do we actually care which Math.random values lead to true and which to false?

Dependency Injection is Passing The Dependencies as Function Arguments

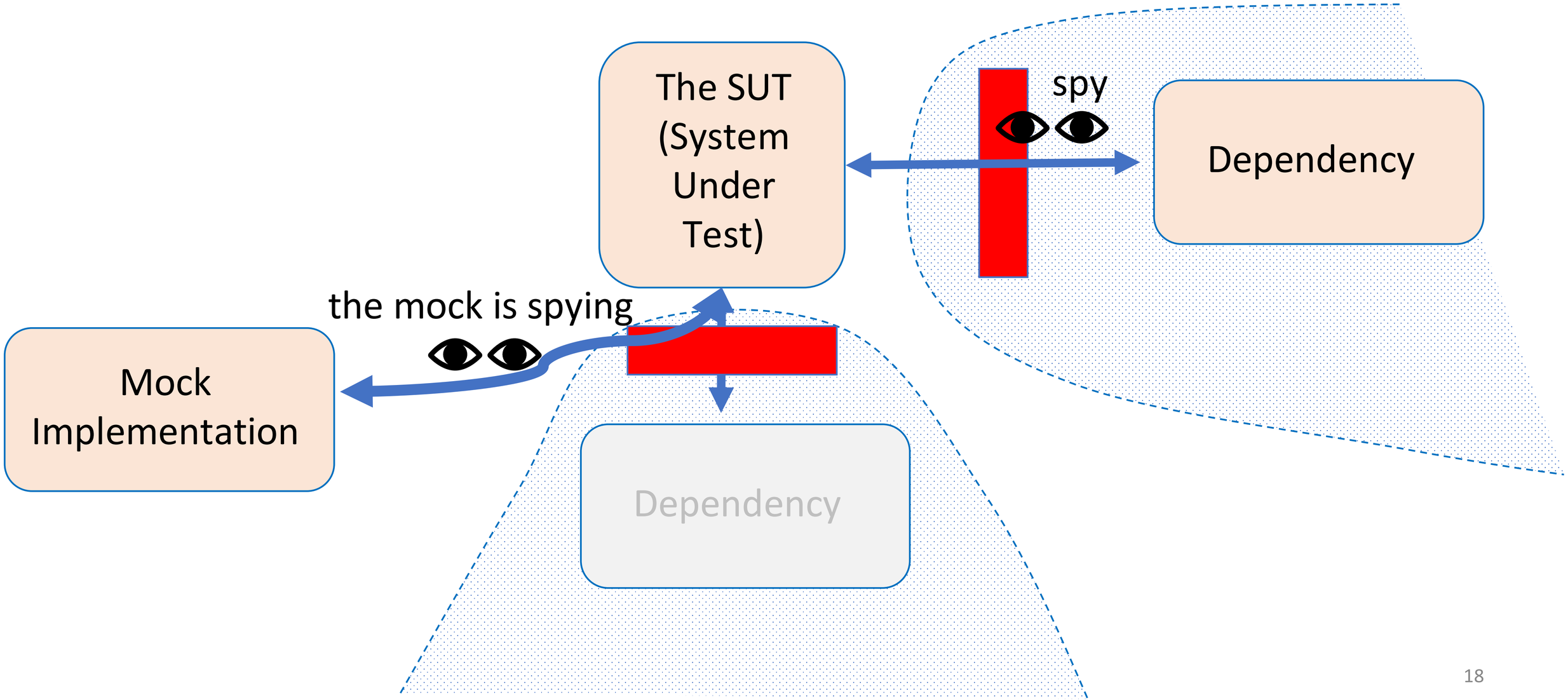
```
/**
 * @param randomSource - randomness source
 * @returns true if randomSource returns >= 0.5
 */
function coinFlip2(randomSource: () => number) {
  return randomSource() >= 0.5;
}

test('coinFlip2 uses the randomness source', () => {
  expect(coinFlip2(() => 0.9)).toBe(true);
  expect(coinFlip2(() => 0.1)).toBe(false);
});
```

Simple Mocks

- Vitest mocks return “undefined” by default (can be customized), and also spies on all the calls to the function

```
test('a simple mock', () => {
  const myMock = vi.fn();
  expect(myMock(104)).toBeUndefined();
  expect(myMock('Hello', 'you')).toBeUndefined();
  expect(myMock.mock.calls).toEqual([
    [104],
    ['Hello', 'you'],
  ]);
});
```

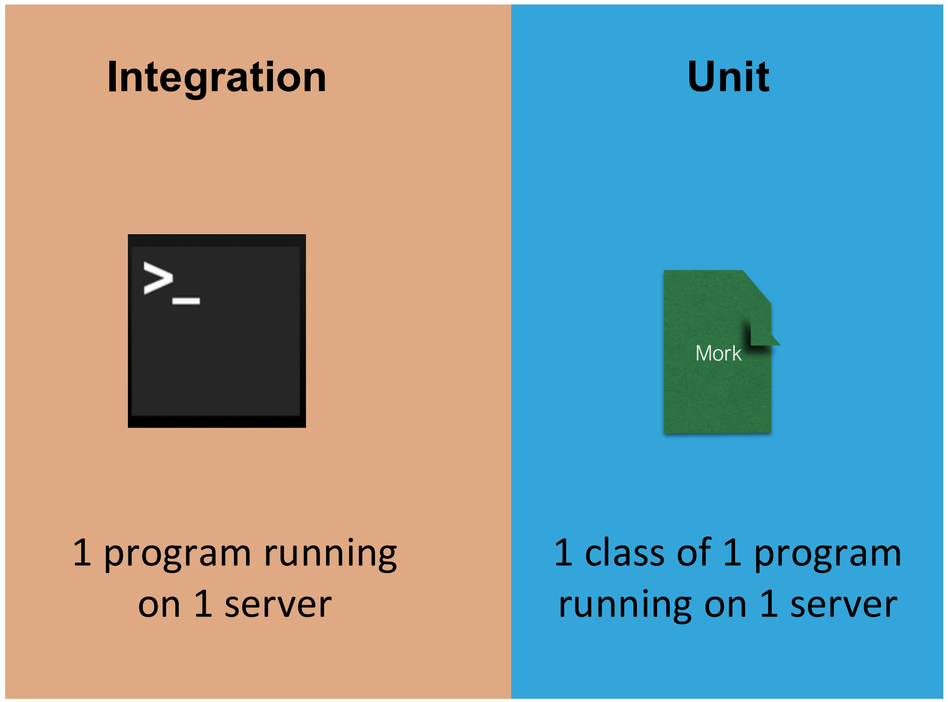
Mocks vs Spies



Integration Testing

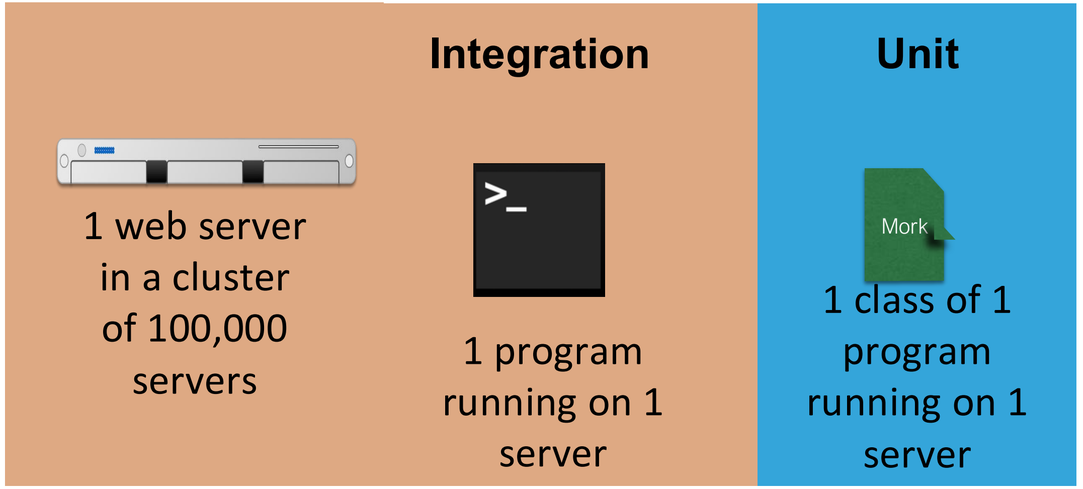
But some bugs are observable only when multiple components interact.

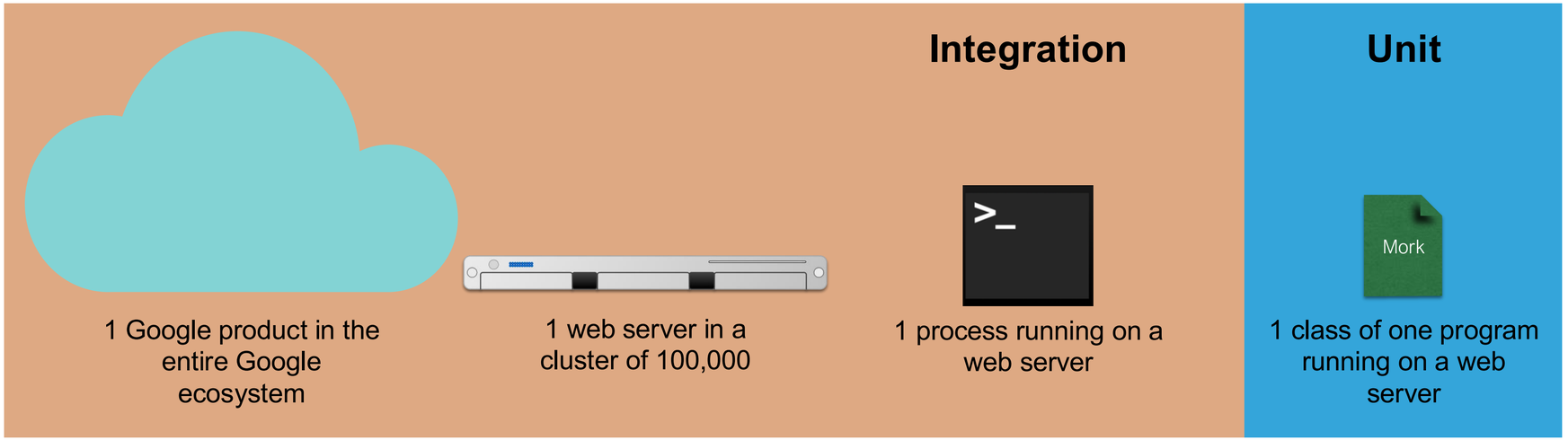
- These are usually because one module has made incorrect assumptions about some other module
- Unit tests won't reveal such bugs
- Mocks won't help, either (since they may incorporate our incorrect assumptions)
- So you really need integration tests



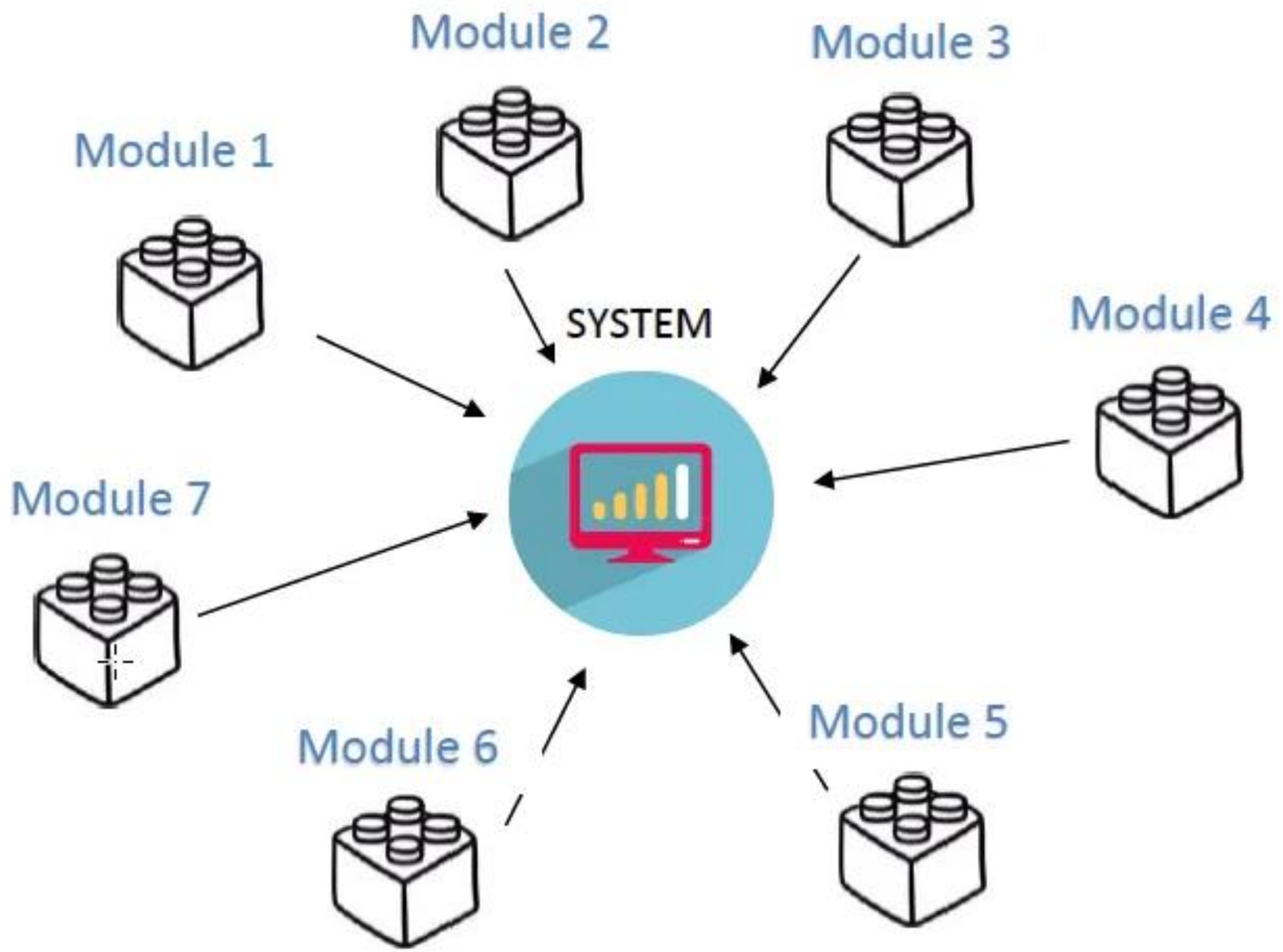
Integration tests may be larger, even enormous

- Does the presence of other jobs on our server change the behavior of our program?
- Does the presence of the other servers change the behavior of our program?



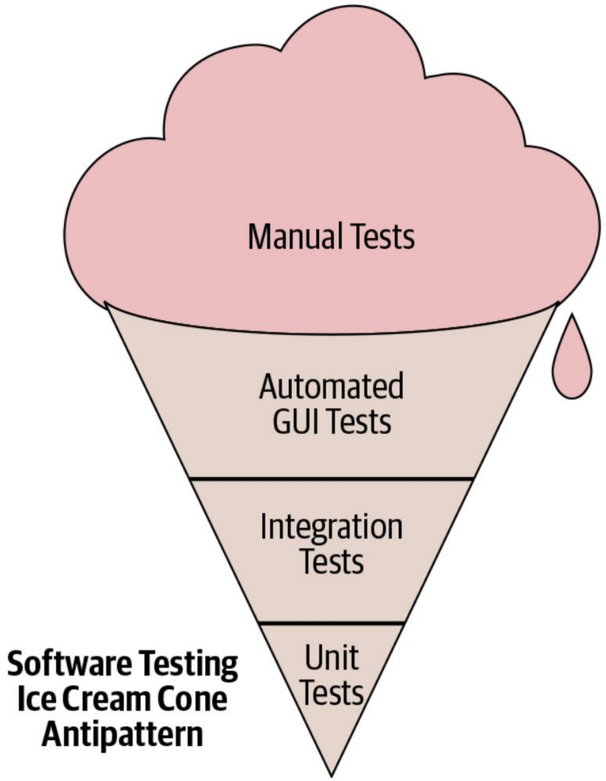


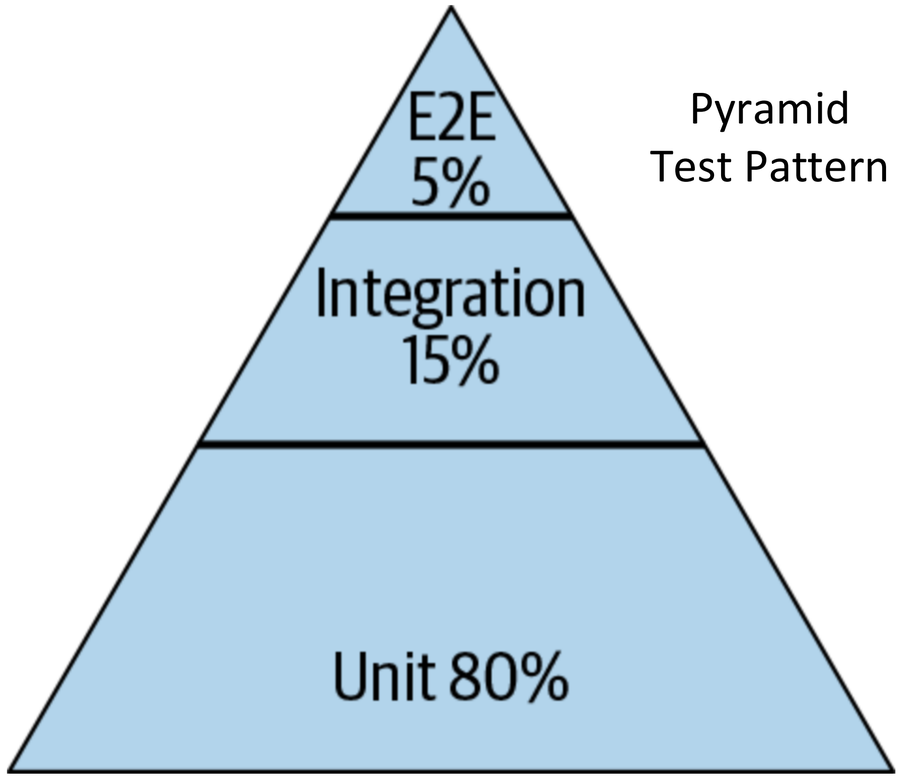
Integration tests can be done in many ways



- All at once ("Big Bang")
- Top-down
- Bottom-up
- Middle-out
- Top-Bottom-Middle
- etc., etc., etc.

Testing Distribution (How much of each kind of testing we should do?)





Pyramid Test Pattern

From SoftEng @ Google Chapter 11

- https://learning.oreilly.com/library/view/software-engineering-at/9781492082781/ch11.html#testing_overview

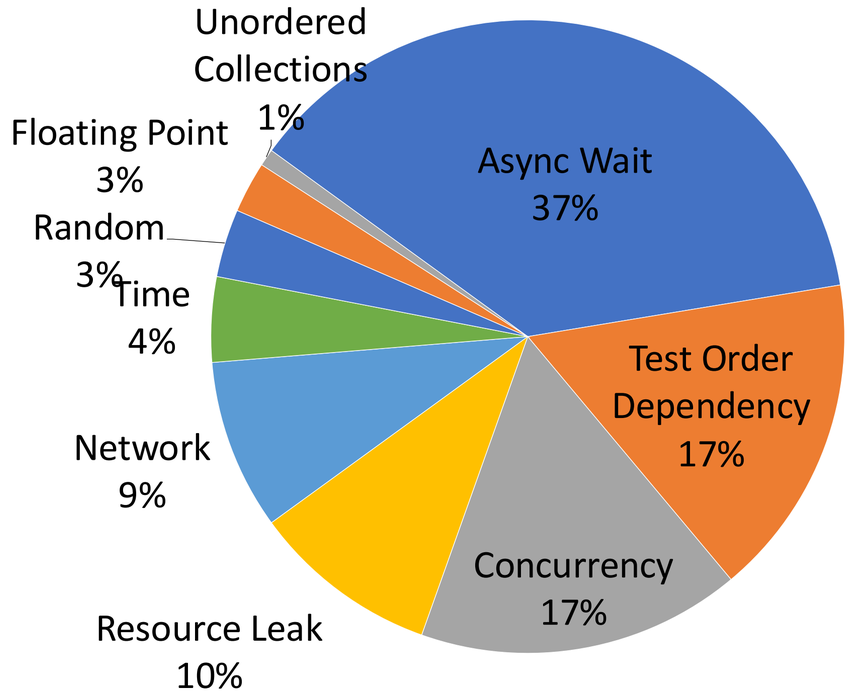
How big is my test? Google's Classification

- Small: run in a single thread, can't sleep, perform I/O or make blocking calls
- Medium: run on single computer, can use processes/threads, perform I/O, but only contact localhost
- Large: Everything else

"Software Engineering at Google: Lessons Learned from Programming Over Time," Wright, Winters and Manshreck, 2020 (O'Reilly)

Integration Tests can be Flaky

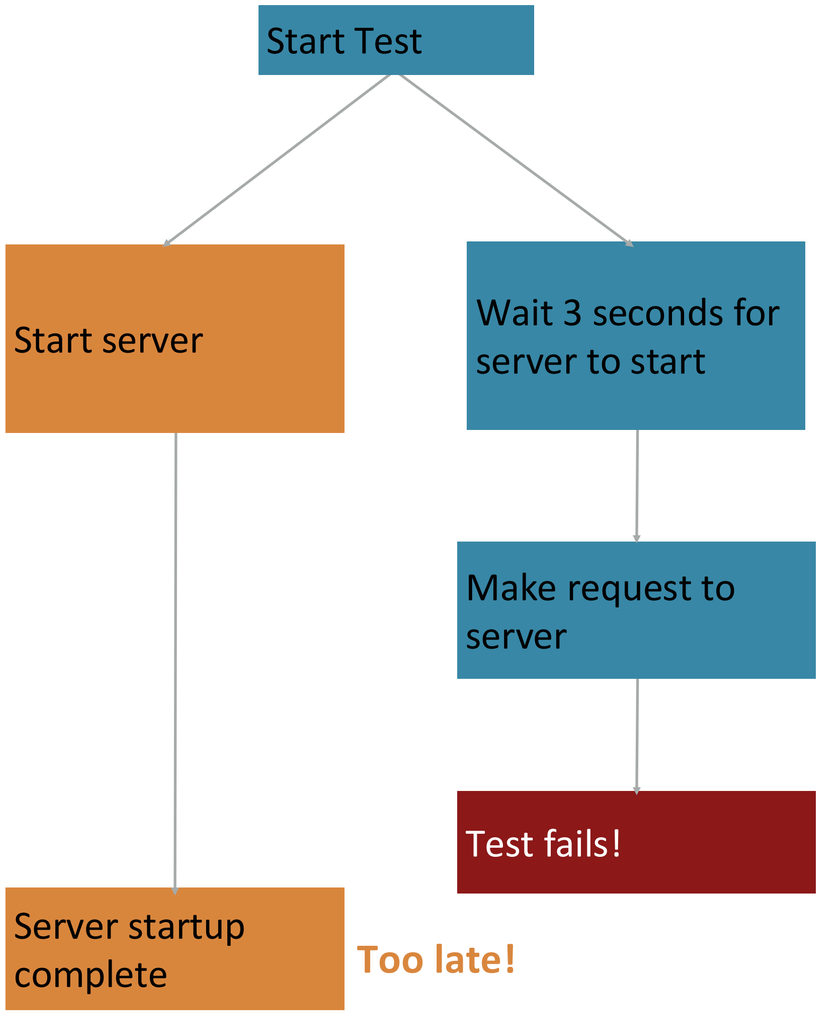
- Flaky test failures are false alarms
- Most common cause of flaky test failures: “async wait” - tests that expect some asynchronous action to occur within a timeout
- UI Testing is often flaky and slower
- Good tests avoid relying on timing
- Good tests avoid relying on the order in which the tests are run



[Luo et al, FSE 2014 “An empirical analysis of flaky tests”]

Flaky Test Example: Async/Wait

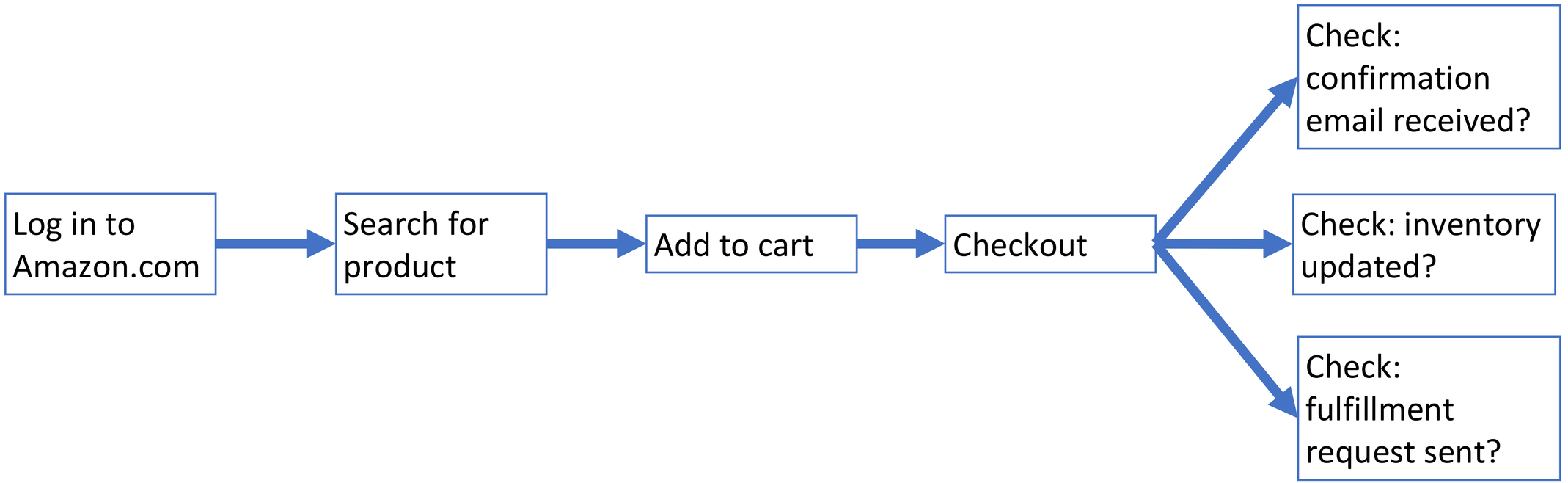
- Most common root cause of flakiness
- Difficult to avoid, but there are mitigations:
  - Have more “small” tests that don’t require concurrency
  - Ensure sufficient resources available for running tests
  - Embed reasonable error detection to classify test failures as likely to be “flaky” vs true failures



We make flaky tests anyway

```
- name: Test that the backend server starts
  run: |
    npm start -w=server & sleep 5
    echo "Checking if home page is served"
    curl --fail 'http://localhost:8000/' > /dev/null 2>&1

    echo "Checking if login page is served"
    curl --fail 'http://localhost:8000/login' > /dev/null 2>&1

    echo "Checking if api endpoint returns several threads"
    curl --fail 'http://localhost:8000/api/thread/list' 2> /dev/null | jq 'if length < 4 then error("Too few posts returned from api
```

“End-to-End” Tests can be Enormous



- Most effective end-to-end tests focus on high value user interactions (UI Testing)

Acceptance Testing

Acceptance Tests can be formulated as scenarios

- Acceptance tests are written to verify behavior from a user's perspective.
- The focus is on treating the application as a black-box
- Tests may be specified as given-when-then scenarios:

given there's a logged in user
and there's an article "bicycle"
when the user navigates to the "bicycle" article's detail page
and clicks the "add to basket" button
then the article "bicycle" should be in their shopping basket

https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test

But how to make these human-readable scenarios into executable tests?

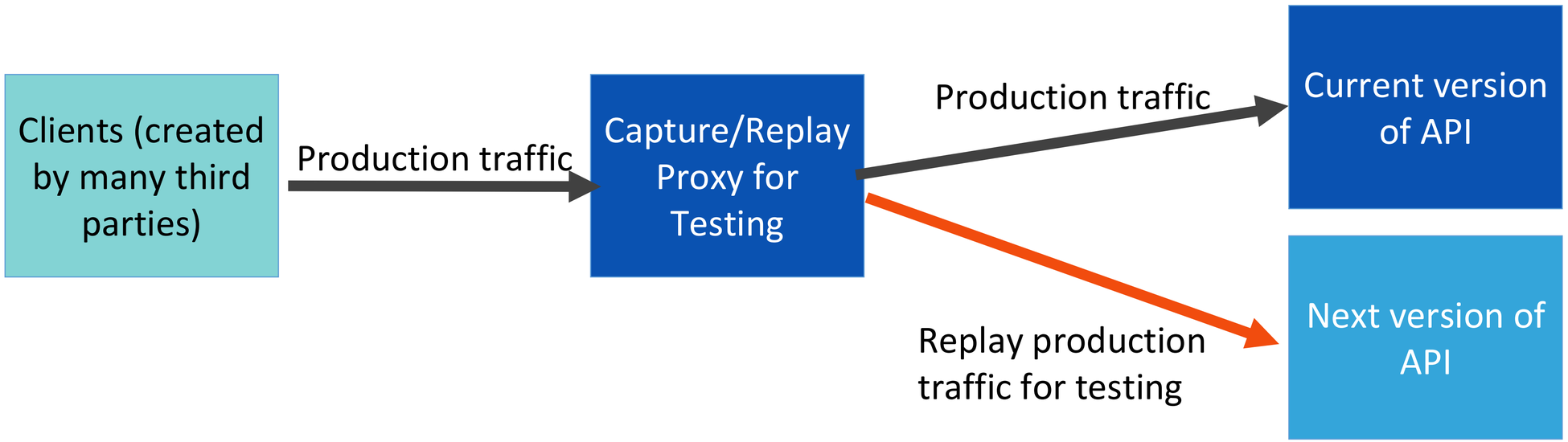
- Scenarios like the one above are readable by humans (e.g. customers)
- But they are not directly executable
- Tools like Playwright and Cypress help fill this gap
  - link on module page

Deployed systems create even more testing challenges

- Clients believe “how it is now is right”,
  - Not “how the API intended it to be is right”
  - Writing thorough test suite is even harder, less useful
  - What is a “breaking change”?
- Still: vital to detect breaking changes
- Examples:
  - Detailed layout of GUIs
  - Side-effects of APIs, particularly under corner-cases

Mock System-Level Components with Capture/Replay

- Record the API requests and responses that clients make
- Test new versions of the API by identifying requests that result in different responses ("breaking changes")



https://www.tradeweb.com/our-markets/data--reporting/replay-service/

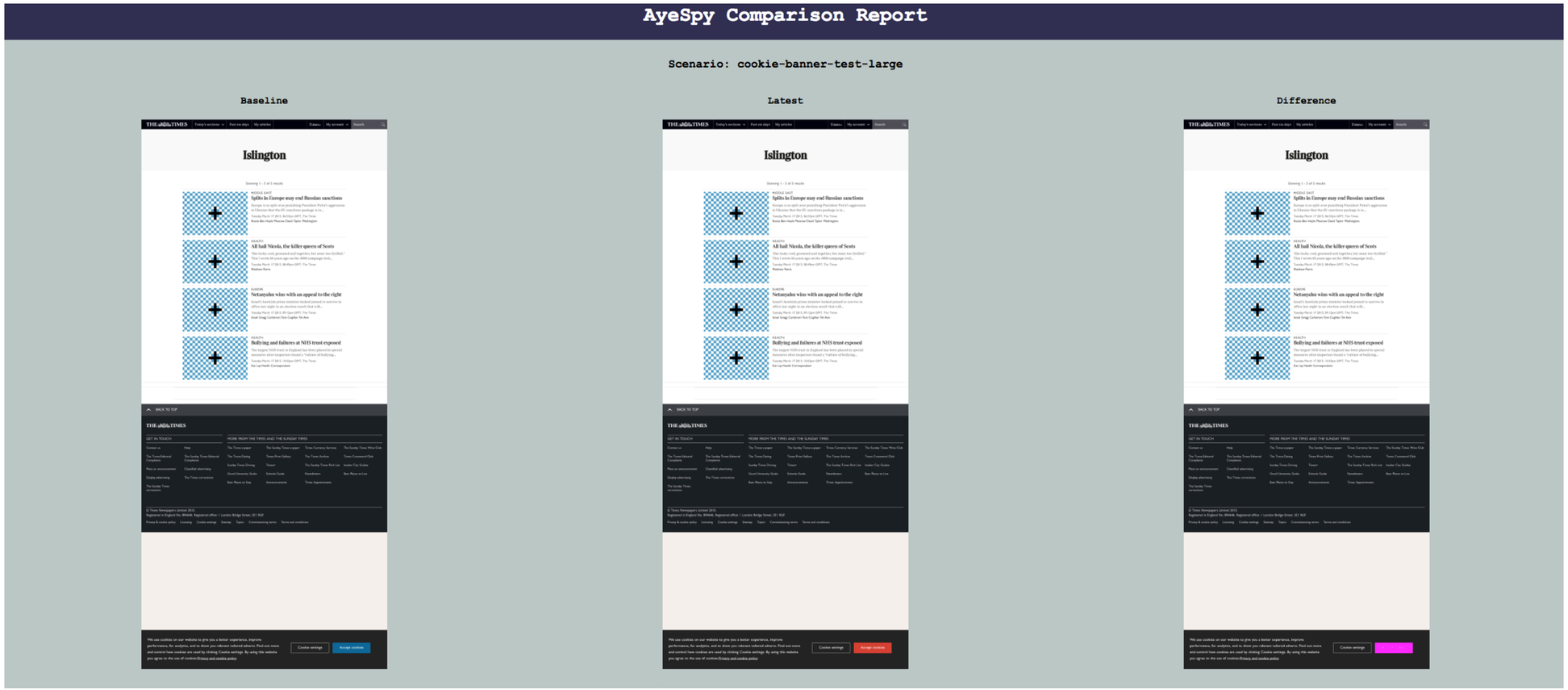
Snapshot Tests Can Detect GUI Changes

- The first time the test runs, it saves a "snapshot" of the rendered GUI
- Subsequent runs will fail if the snapshot changes

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link
page="http://www.facebook.com">Facebook</Li
nk>)
      .toJSON();
  expect(tree).toMatchSnapshot();
});
```

Product Owners can Assess Visual Snapshot Tests

- Capture a visual snapshot of an application under a state
- If that snapshot changes, produce a visual report for manual sign-off



https://github.com/newsuk/AyeSpy

Learning Objectives for this Lesson

- You should now be prepared to:
  - Explain why you might need a test double in your testing
  - Design test cases for code using fakes, mocks and spies
  - Explain why you might need tests that are larger than unit tests
  - Explain how large, deployed systems lead to additional testing challenges