CS 4530: Fundamentals of Software Engineering

Module 12: Testing Larger Things

Jon Bell, Adeel Bhutta, Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

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

Story so far: Tests Check Return Values

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

    const id1 = db.addStudent('blair’);

    expect(db.nameToIDs('blair')).toEqual([id1])
  });
```

Challenge: How to test the ProducerClock?

clockWithObserverPattern.test.ts

```
export interface IClockWithListeners {
    reset():void  // resets the time to 0
    tick():void  // increment time and notify all listeners
    // add a listener and initialize it with the current time
    addListener(listener:IClockListener):void
}

export interface IClockListener {
    // @param t - the current time, as reported by the clock
    notify(t:number):void
}

export class ProducerClock implements IClockWithListeners {
    // some implementation
}
```

Test the ProducerClock with a Fake ClockListener

clockWithObserverPattern.test.ts

```
export interface IClockWithListeners {
    reset():void  // resets the time to 0
    tick():void  // increment time and notify all listeners
    // add a listener and initialize it with the current time
    addListener(listener:IClockListener):void
}

class ClockListenerForTest implements IClockListener {
    private _time : number = 0
    constructor (private masterClock:IClockWithListeners) {
        masterClock.addListener(this)
    }
    notify (t:number) : void {this._time = t}
    getTime () : number {return this._time}
}
```

Now we can test using the custom observer

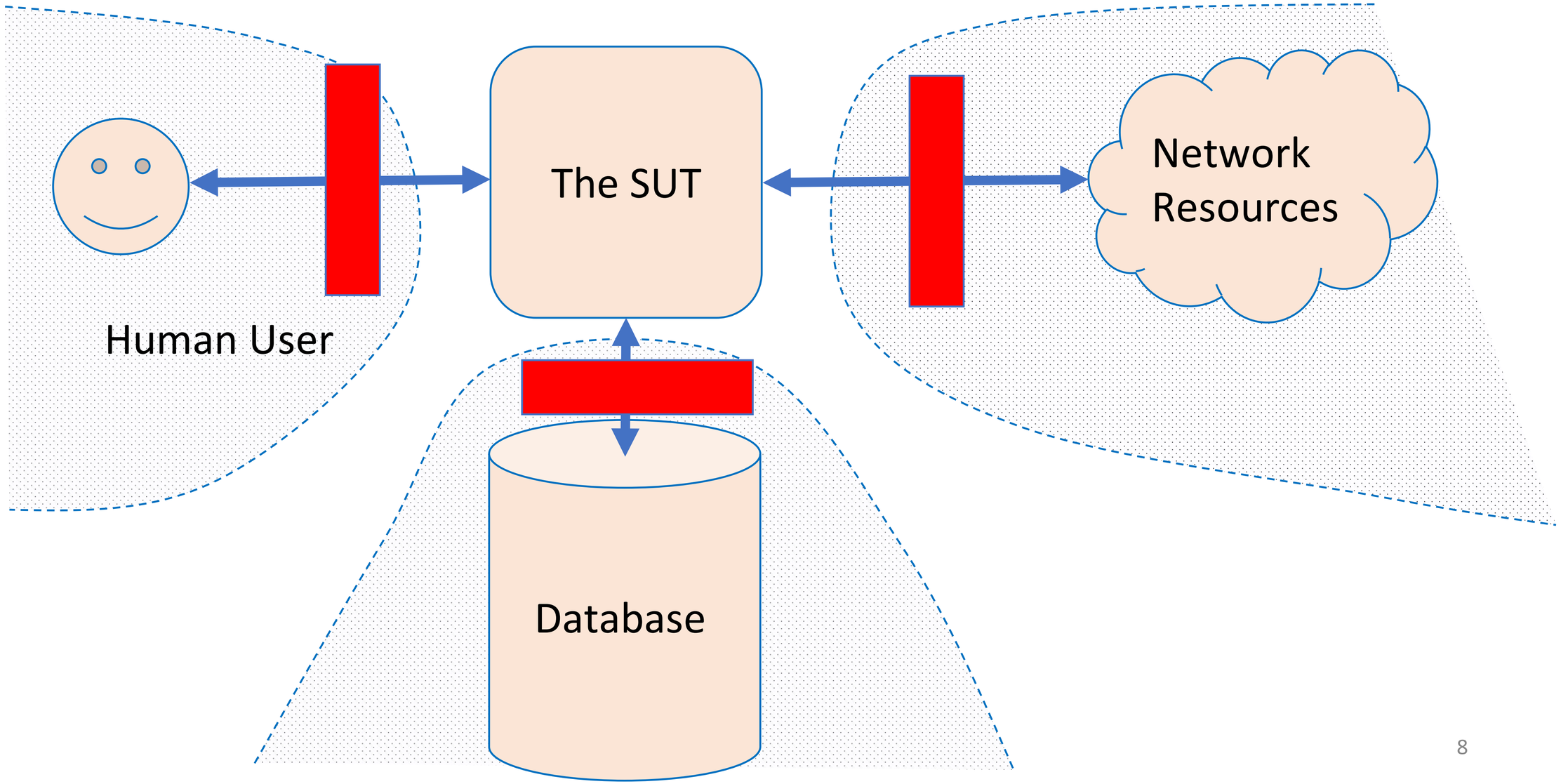
clockWithObserverPattern.test.ts

```
import { ProducerClock } from  "./clockWithObserverPattern";

const clock1 = new ProducerClock
const listener1 = new ClockListenerforTest(clock1)

describe("tests for ProducerClock", () => {
    test("after reset, listener should return 0", () => {
        clock1.reset()
        expect(listener1.getTime()).toBe(0)
      })
    test("after one tick, listener should return 1", () => {
        clock1.reset(); clock1.tick()
        expect(listener1.getTime()).toBe(1)
    })
    test("after two ticks, listener should return 2", () => {
        clock1.reset(); clock1.tick(); clock1.tick()
        expect(listener1.getTime()).toBe(2)
    })
})
```

“Test Doubles” Stand In For Other Components

- Act as a stand-in for components, allowing for testing in isolation
- Fakes: Replace client implementations with dummies for testing
- Mocks: Automatically-generated fake implementations for an interface
- Spies: Automatically-instrument internals of objects, classes or modules

Test doubles replace uncontrollable things with things that you do control



Fake Listener: Discussion

```
class ClockListenerForTest implements IClockListener {
    private _time : number = 0
    constructor (private masterClock:IClockWithListeners) {
        masterClock.addListener(this)
    }
    notify (t:number) : void {this._time = t}
    getTime () : number {return this._time}
}
```

- Good news:
  - It works!
  - It doesn’t require learning other libraries
- Bad news:
  - It’s a maintenance burden (what if new methods are added to IClockListener?)
  - It took manual effort to write
  - Richer fakes (e.g. track how many times a method called) are even more effort to write

Mocks are automated fakes

- Jest's mocks return "undefined" by default (can be customized), and track calls to the function

```
test("simplest mock behavior", () => {
    const mockFunction1 = jest.fn();

    const result1 = mockFunction1("17");
    const result2 = mockFunction1("42")

    expect(result1).toBeUndefined();
    expect(result2).toBeUndefined()

    expect(mockFunction1).toHaveBeenCalled();
    expect(mockFunction1).toHaveBeenCalledTimes(2);

    expect(mockFunction1).toHaveBeenCalledWith("17");
    expect(mockFunction1).toHaveBeenCalledWith("42")
});
```

Jest's Mock API: https://jestjs.io/docs/mock-function-api

You can customize your mock in many ways

simpleMocks.test.ts

```
test("customizing mock functions", () => {

    // you can specify the the return value
    const mockFunction3 = jest.fn();
    mockFunction3.mockReturnValue("baz");

    expect(mockFunction3(17)).toBe("baz");
    expect(mockFunction3).toHaveBeenCalledWith(17);

    // or give the mock an implementation
    const mockFunction2 = jest.fn()
    mockFunction2.mockImplementation((n: number) => n + n);

    expect(mockFunction2(3)).toBe(6);
    expect(mockFunction2(14)).toBe(28)
    expect(mockFunction2).toHaveBeenCalledWith(3);
    expect(mockFunction2).toHaveBeenCalledWith(14);

    // you can also reset the mock's history and implementation
    mockFunction2.mockReset()
    expect(mockFunction2).not.toHaveBeenCalledWith(14);
});
```

Mock Classes and Interfaces with Jest-Mock-Extended

clockWithObserverPatternMock.test.ts

```
import { mock, mockClear } from 'jest-mock-extended';
import { IClockListener, ProducerClock } from './clockWithObserverPattern';

const clock1 = new ProducerClock();
//Automatically create an implementation of IClockListener, each method is a mock function
const listener1 = mock<IClockListener>();
clock1.addListener(listener1);

describe('tests for ProducerClock', () => {
    beforeEach(() => {
        mockClear(listener1); //Clear the mock function's history
    });
    test('after one tick, listener should return 1', () => {
        clock1.reset();
        clock1.tick();
        expect(listener1.notify).toHaveBeenLastCalledWith(1);
    });
    test('after two ticks, listener should return 2', () => {
        clock1.reset();
        clock1.tick();
        expect(listener1.notify).toHaveBeenLastCalledWith(1);
        clock1.tick();
        expect(listener1.notify).toHaveBeenLastCalledWith(2);
        expect(listener1.notify).toHaveBeenCalledTimes(2);
    });
});
```

Unlike mocks, spies *instrument* existing implementations

- Consider cases where you *don't* want a complete fake, but *do* want to check side-effects:
  - What was sent on the network?
  - How many times was a problem logged?
  - What was inserted in the database?
- Jest can automatically instrument existing code to make it into a "spy" – a mock but with the original implementation

Spy "remembers"

Use jest.spyOn to create a spy on an object

```
import { ClockListener, ProducerClock } from './clockWithObserverPattern';

const clock1 = new ProducerClock();
const clockClient = new ClockListener(clock1);
const notifySpy = jest.spyOn(clockClient, 'notify’); // Spy on calls to notify on this clock
describe('tests for ProducerClock', () => {
  beforeEach(() => {
    notifySpy.mockClear(); // Clear the mock function's history
  });
  test('after one tick, listener should return 1', () => {
    clock1.reset();
    clock1.tick();
    expect(notifySpy).toHaveBeenLastCalledWith(1);
  });
  test('after two ticks, listener should return 2', () => {
    clock1.reset();
    clock1.tick();
    expect(notifySpy).toHaveBeenLastCalledWith(1);
    clock1.tick();
    expect(notifySpy).toHaveBeenLastCalledWith(2);
    expect(notifySpy).toHaveBeenCalledTimes(2);
  });
});
```

clockWithObserverPatternSpy.test.ts

Spies can be used even when you can’t control the SUT

- Syntax: jest.spyOn(object, methodName)
- Example from last slide:

```
const clock1 = new ProducerClock();

/** Some listener that we happen to know about */
const clockClient = new ClockListener(clock1);

/** Spy on calls to 'notify' on this listener */
const notifySpy = jest.spyOn(clockClient, 'notify’);
```

- 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
  - we'll illustrate this a few slides from now.

Let’s use mocks and spies to test the http client from the async module

EchoClass.ts

```
export class Echo  {

  /** @argument a string
   * @returns a promise to return the same string
   * @requires axios
   * @calls https://httpbin.org/get?answer=${str}
   */

  public static async echo(str: string): Promise<string> {
    const res = await axios.get(`https://httpbin.org/get?answer=${str}`);
    return res.data.args.answer;
  }

}
```

Create a spy on (axios, 'get')

echo.test.ts

```
import { Echo } from './EchoClass';

// etc...

test('just spying on a function runs the original', async () => {
    test('echo should return its argument', async () => {
      const spy1 = jest.spyOn(axios, 'get');
      const str = '34';
      const res = await Echo.echo(str);
      expect(spy1).toHaveBeenCalled();
      expect(res).toEqual(str);
    });
  });
```

Next step: define a mock for the axios call

echo.test.ts

```
async function mockAxiosCall(url: string) {
  return { data: { args: { answer: url.split('=')[1] } } };
 }

// Hmm, we better test mockAxiosCall!

describe('tests for mockAxiosCall', () => {
  test('mockhttpbin should return its argument', async () => {
    const url = 'https://httpbin.org/get?answer=33'
    const res = await mockAxiosCall(url);
    expect(res).toEqual({ data: { args: { answer: "33" } } });
  });
})
```

Now install the mock, so the 'get' doesn't get called.

echo.test.ts

```
test('mock axios.get so httpbin is not called', async () => {
    jest.resetAllMocks();
    const spy1 = jest.spyOn(axios, 'get').mockImplementation(mockAxiosCall);
    const str = '34';
    const res = await Echo.echo(str);
    expect(spy1).toHaveBeenCalled();
    expect(res).toEqual(str);
  })
```

What if we wanted to test a client of echo?

- But we didn't want to issue any http requests

echoClient.ts

```
import { Echo } from './EchoClass';

/** calls echo twice and concatenates the results */
export async function echoClient(str: string) {
  const res1 = await Echo.echo(str);
  const res2 = await Echo.echo(str);
  return res1 + res2;
}
```

Solution: create a mock for Echo

echoClient.test.ts

```
import { echoClient } from './echoClient';

describe('tests for echoClient', () => {
  beforeEach(jest.resetAllMocks);
  beforeEach(() => {
    // mock echo with a correct return value
    jest.spyOn(Echo, 'echo').mockImplementation((str: string) => Promise.resolve(str));
    // mock axios.get to always throw an error;
    // if our test calls the real axios.get, it will fail
    jest.spyOn(axios, 'get').mockRejectedValue('axios.get should not be called');
  })
  test('echoClient should return its argument twice', async () => {
    const str = '345';
    const res = await echoClient(str);
    expect(res).toEqual(str + str);
  });
})
```

Test Doubles Have Weaknesses

- Some failures may occur purely at the integration between components:
  - The test may assume wrong behavior (wrongly encoded by mock)
  - Higher fidelity mocks can help, but still just a snapshot of the real world
- Test doubles can be brittle:
  - Spies expect a particular usage of the test double;
  - The test is "brittle" because it depends on internal behavior of SUT;
- Potential maintenance burden: as SUT evolves, mocks must evolve.

Did we correctly model the behavior of httpbin?

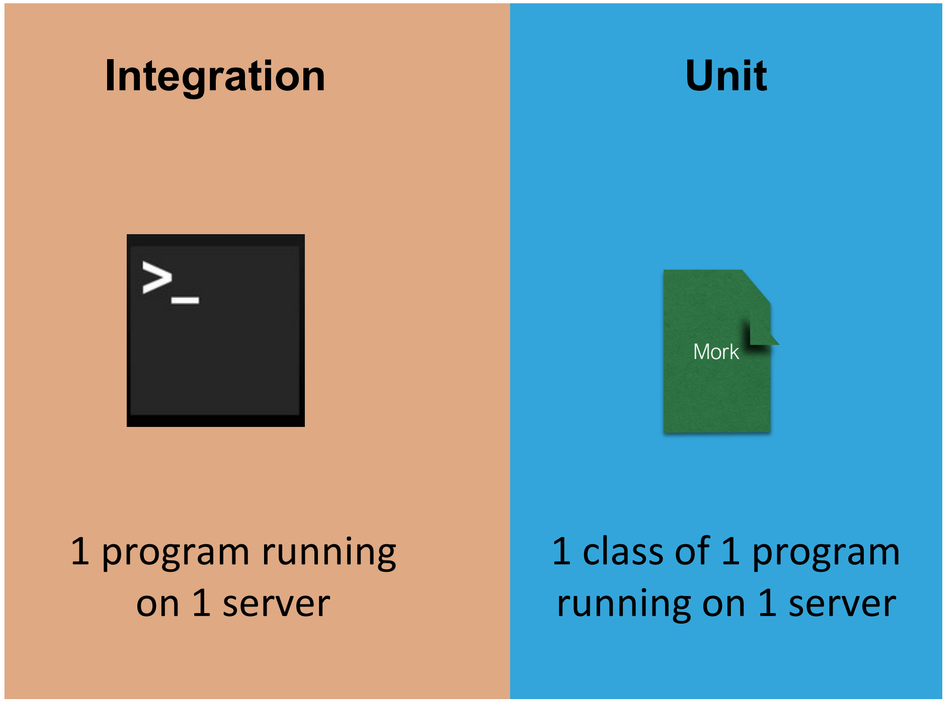
Not just its IO behavior, but also its dependencies

What if we didn't want to make assumptions about how httpbin behaves?

- We'd need to actually call httpbin.
- This is no longer a unit test; it's an integration test
- Which brings us to our next topic.

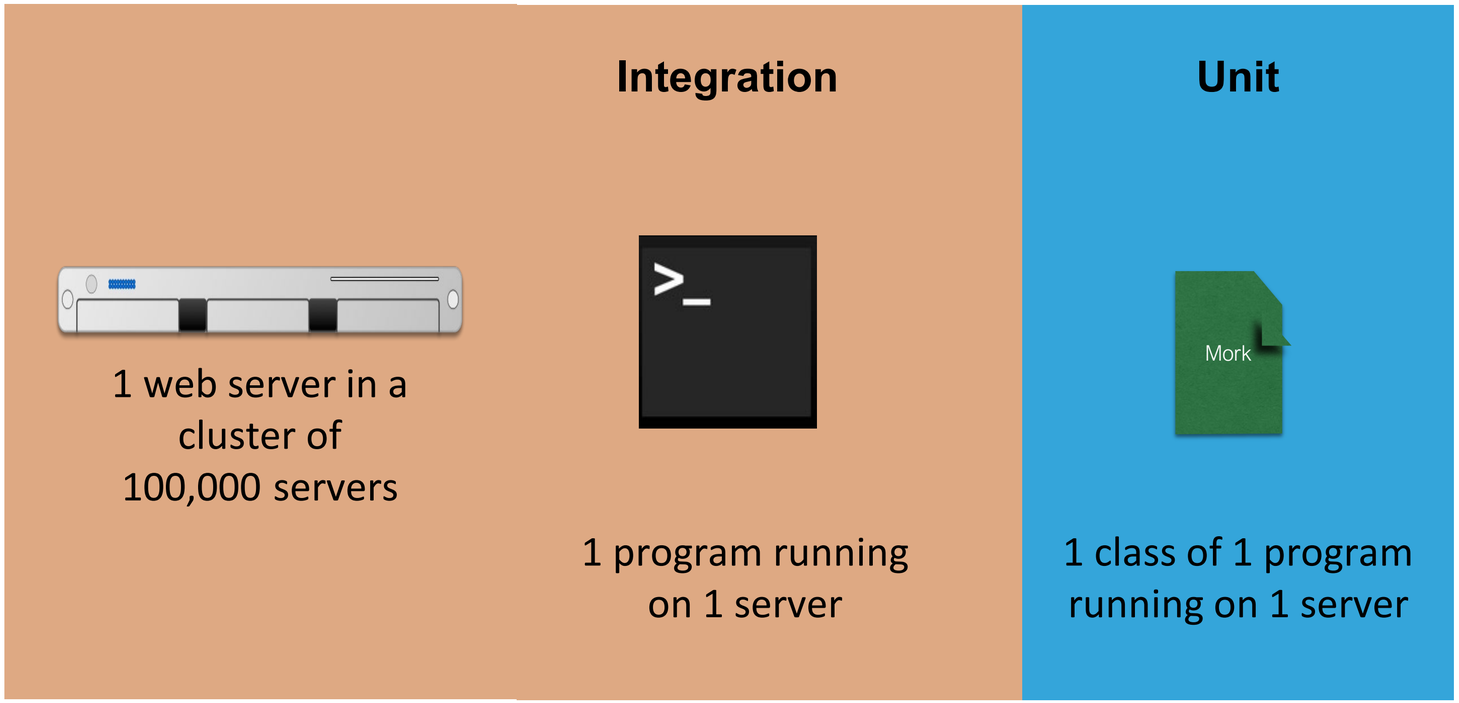
But some bugs are observable only when multiple components interact.

- 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

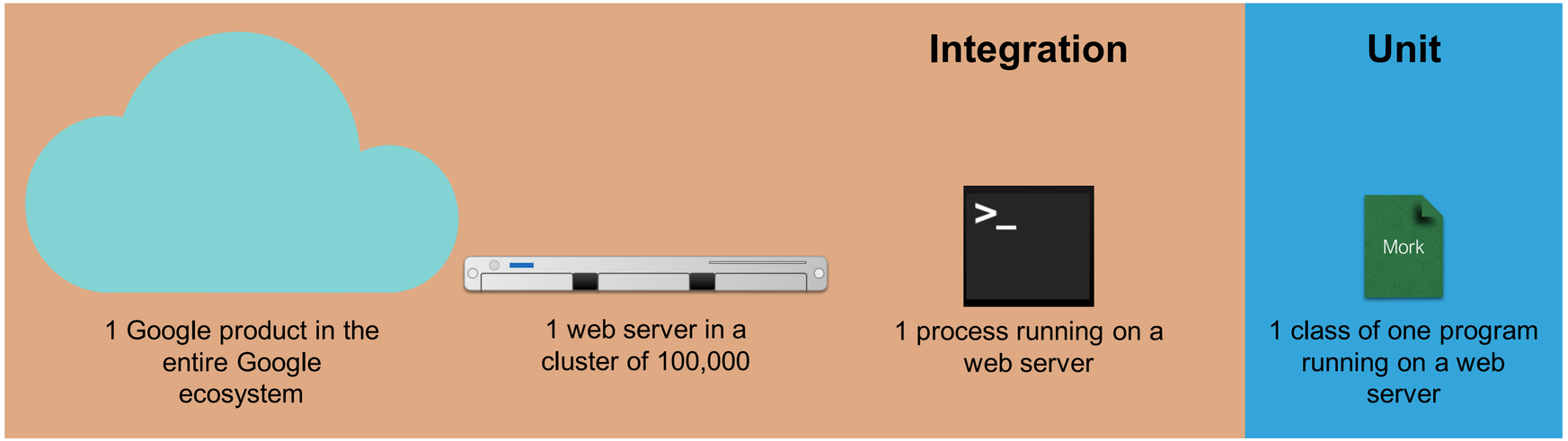


Integration tests may be larger

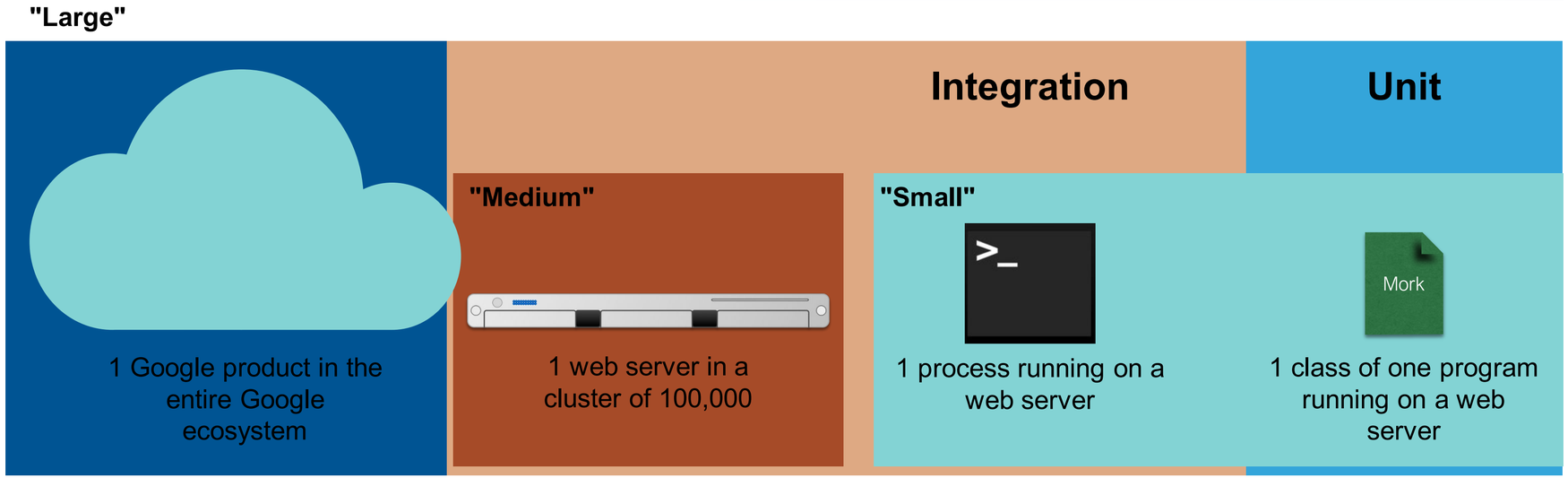
- Does the presence of other jobs on our server change the behavior of our program?
- Does the presence of the other servers change the behavior of our program?



Some Tests are Enormous



Google classifies tests by “size”



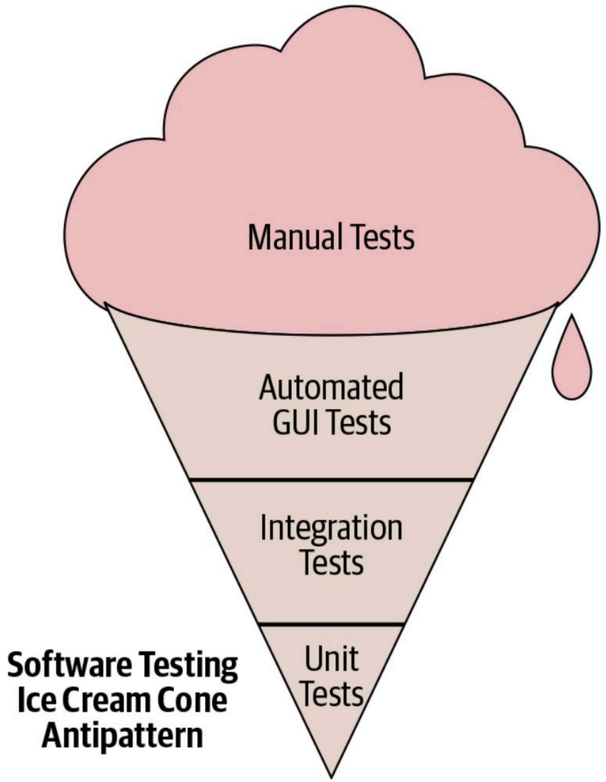
- “small” = single process
- “medium” = single machine
- “large” = bigger than that.

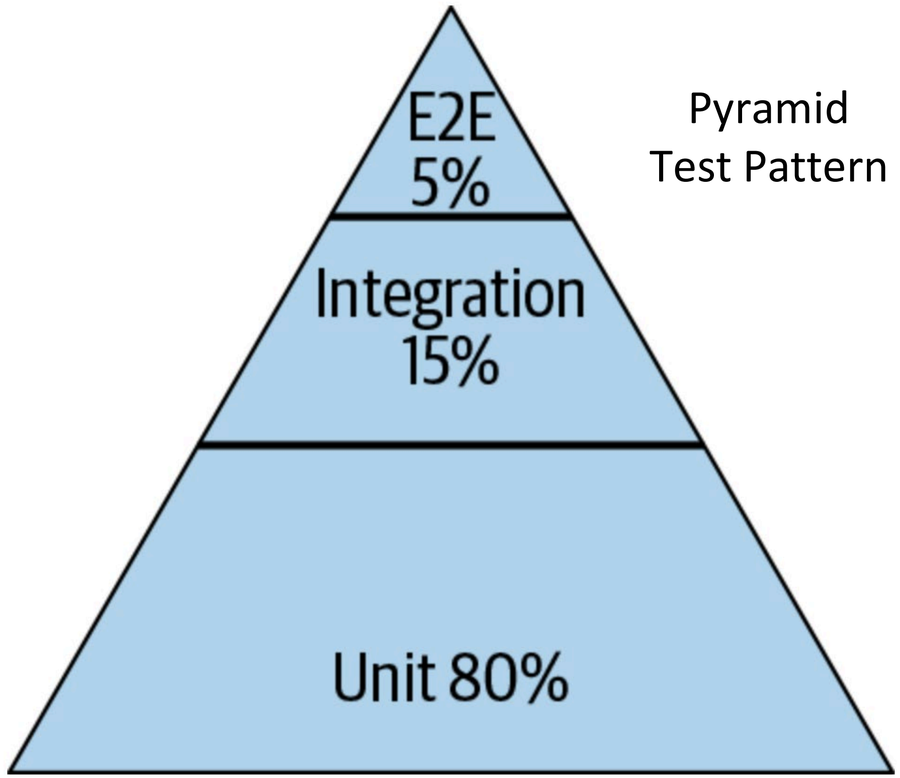
"Software Engineering at Google: Lessons Learned from Programming Over Time," Wright, Winters and Manshreck, 2020 (O’Reilly)

How big is my test?

- 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

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



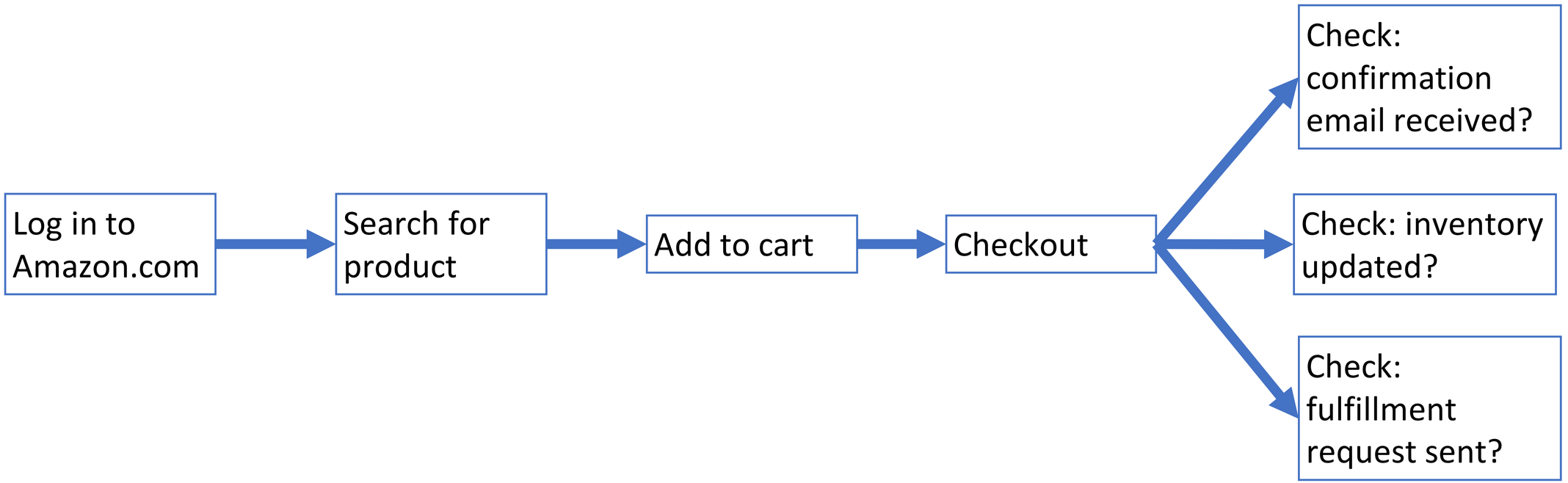


Pyramid Test Pattern

From SoftEng @ Google Chapter 11

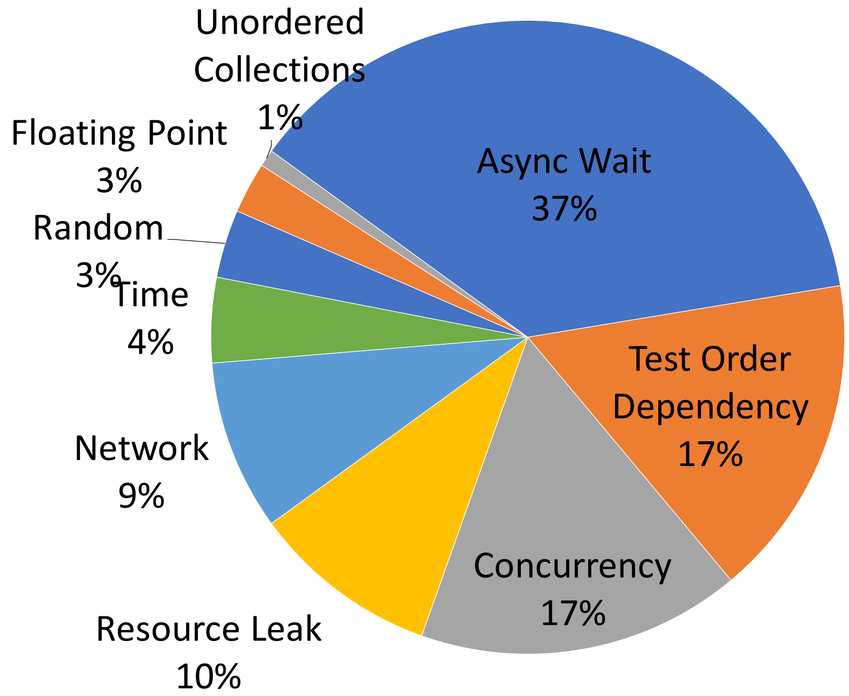
- https://learning.oreilly.com/library/view/software-engineering-at/9781492082781/ch11.html#testing_overview

“End-to-End” Tests are Enormous



Medium and Large Tests can be Flaky

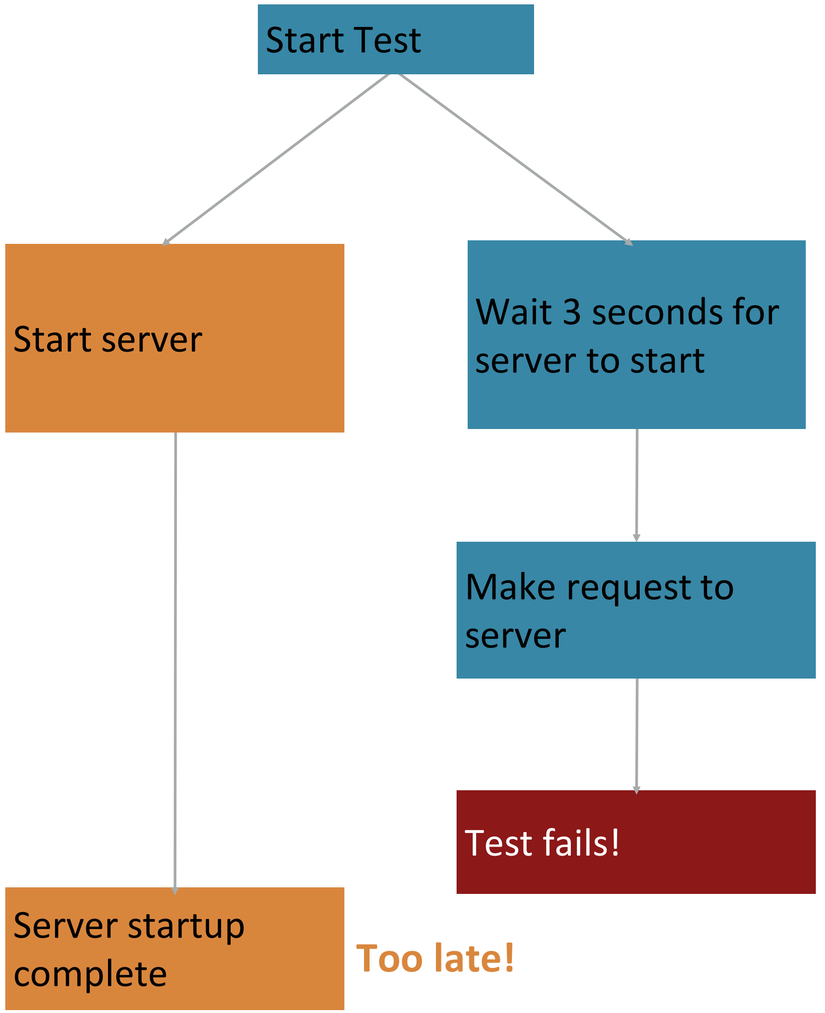
- Flaky test failures are false alarms
- Tests that are hermetic defend against “test order dependency” - failures due to tests running in other orders
- Most common cause of flaky test failures: “async wait” - tests that expect some asynchronous action to occur within a timeout
- Good tests avoid relying on timing



[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 consider:
  - 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

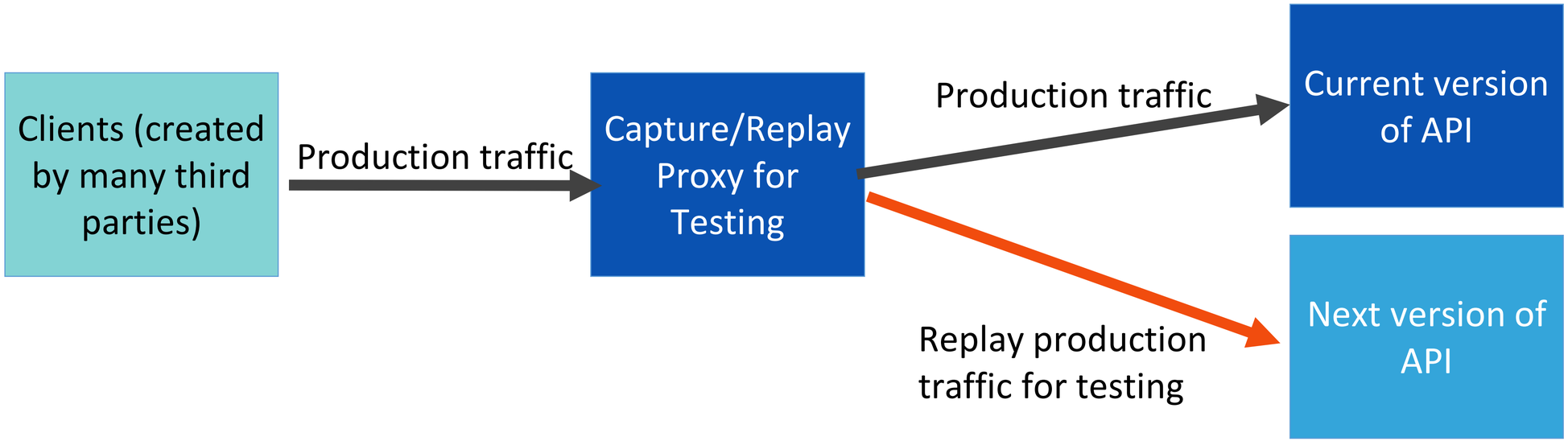


Deployed systems create even more testing challenges

- 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")

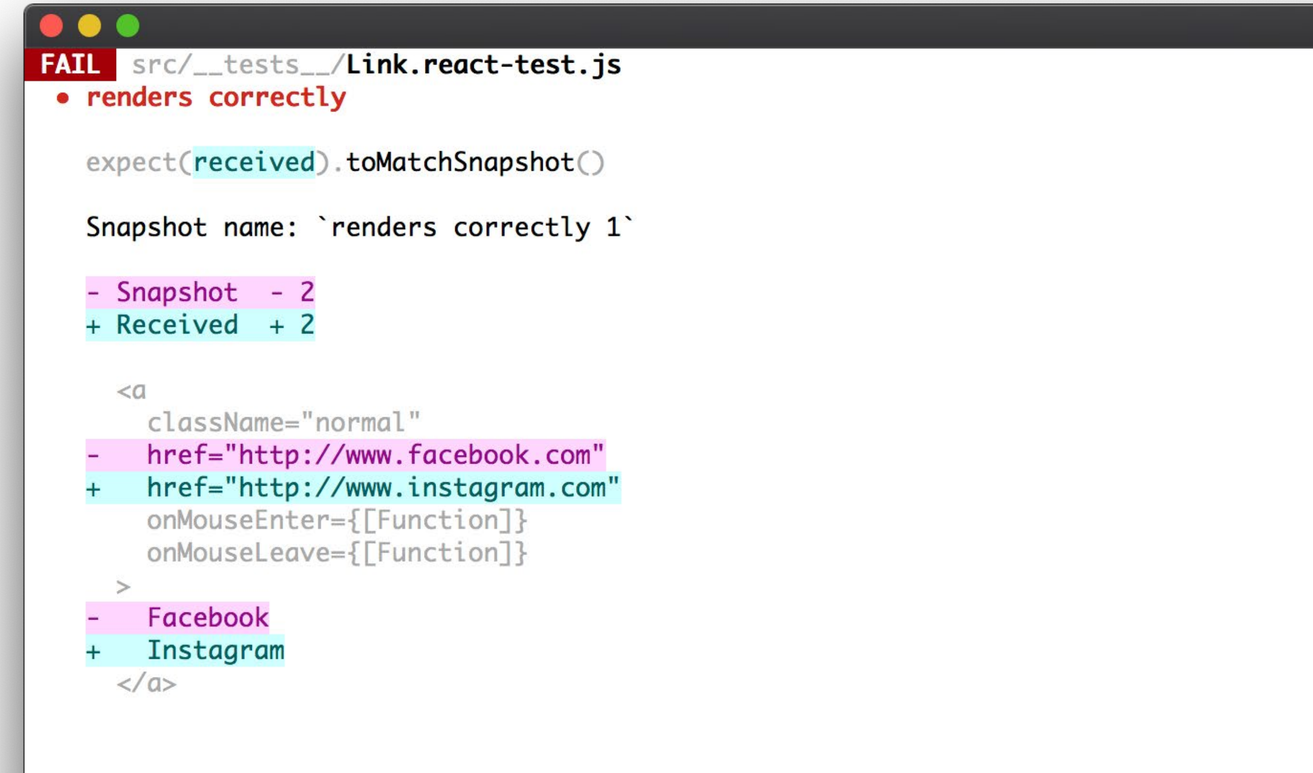


https://www.tradeweb.com/our-markets/data--reporting/replay-service/

Snapshot Tests Can Detect GUI Changes

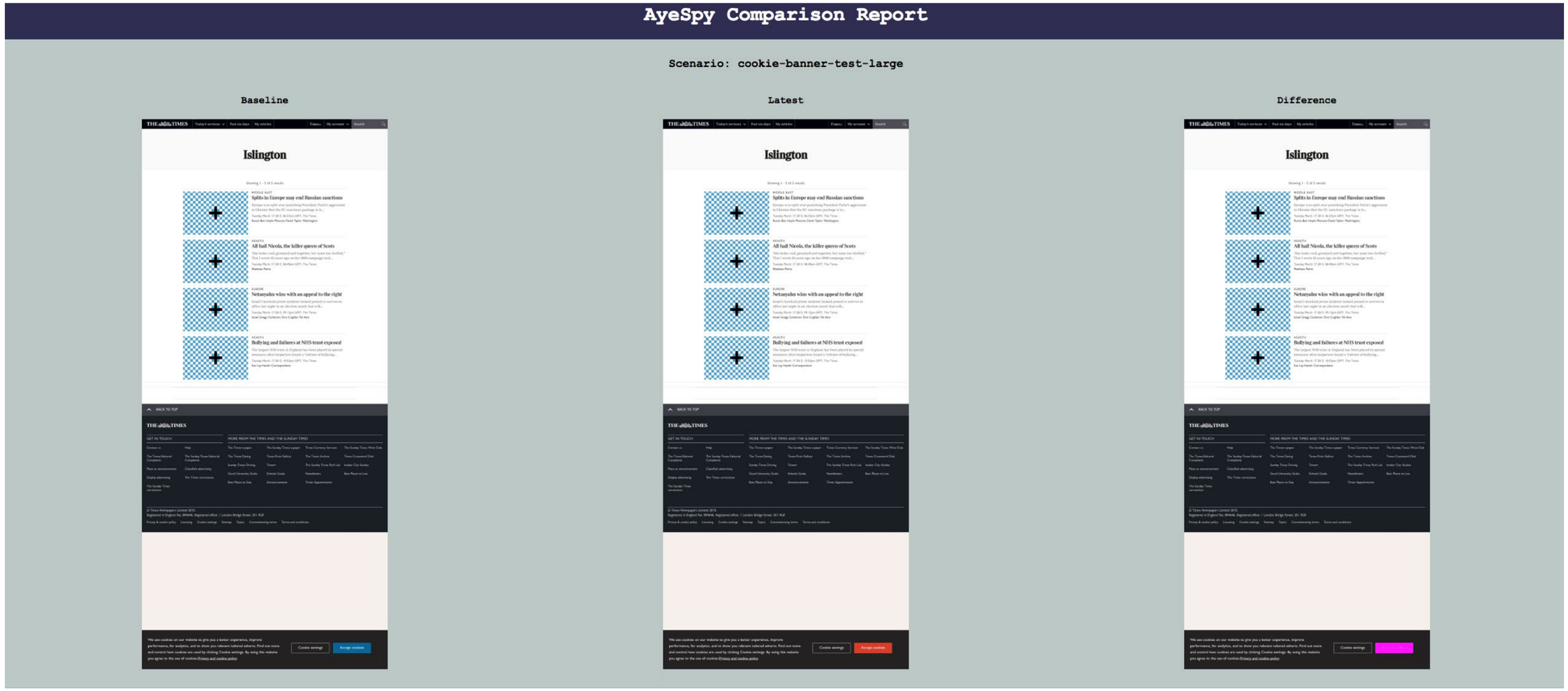
- The first time the test runs, it saves a "snapshot" of the rendered GUI
- Subsequent runs will fail if the snapshot changes

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