Testing Without Mocks
Using Nullables

Presented by James Shore
jamesshore.com

v2023-06-02

Hi, I'm James Shore. This is the lecture video for the “Using Nullables” module of the "Testing Without Mocks” training course. In this module, you’ll learn how to use a technique called “Nullables” to test application-level code.

Watching this video is optional—I’ll cover the same material during the course—but it’s available if you want to study the material ahead of time, or if you want to refresh your memory after the course is over. You can also download the slides, which include a transcript.

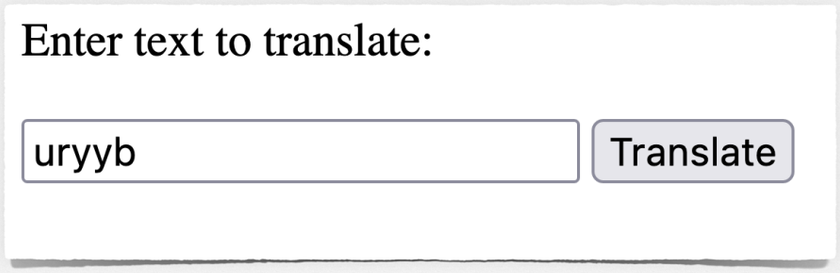
Okay, let’s get started.

1 Why Not Mock?

I first learned about mocks way back in 2000. My team was working on a web site that interacted with a database, and we wanted to test the code without having to set up the database. I came across the original mock objects paper ("Endo-Testing: Unit Testing with Mock Objects" by Tim Mackinnon, Steve Freeman, and Philip Craig) and we tried it out.

It solved our problems! We were able to test the code without setting up the database. Our tests were faster and more deterministic. But the mocks also introduced new problems.

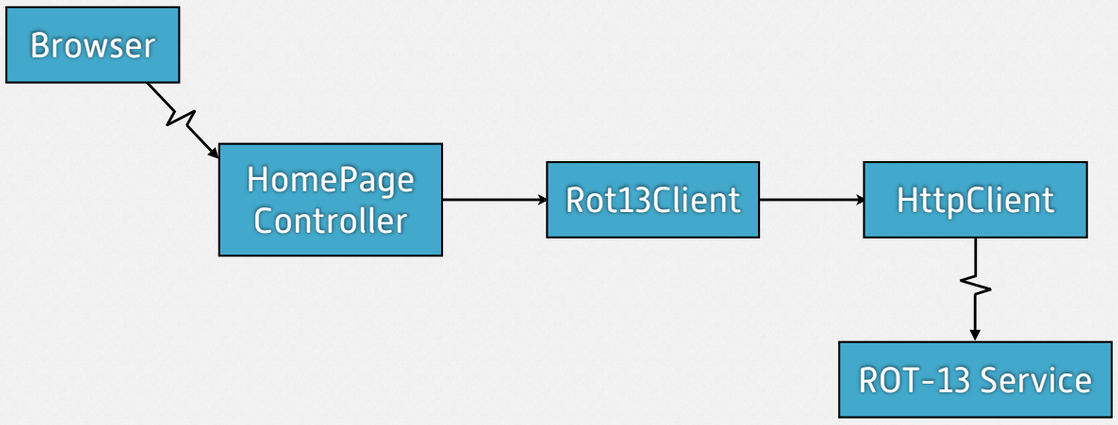
An Example Scenario



Before we talk about the problems, what's a "mock"? You're probably already familiar with them, but just in case:

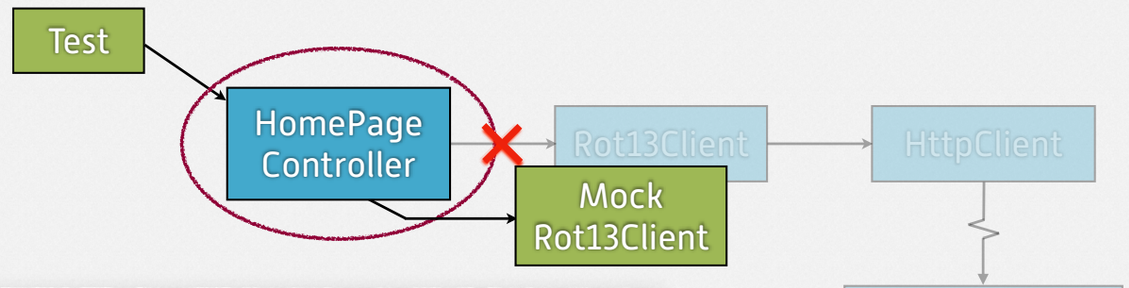
Imagine you have a web page that translates text using ROT-13 encoding. ROT-13 is a spoiler-hiding mechanism you see on the Internet. A becomes M, B becomes N, and so forth. Your web page uses a ROT-13 service to do the encoding.

An Example Scenario



It works like this. The user's browser sends a POST request to the HomePageController. HomePageController asks the Rot13Client to translate the text. The Rot13Client asks the HttpClient to POST to the ROT-13 service.

A Mock-Based Test




```
controller = new Controller(mock)
controller.post();

mock.assertTransformCalled();
```


A mock-based test of this code will inject a MockRot13Client in place of the real Rot13Client. It will run methods on HomePageController and check that MockRot13Client was called correctly.

So, what's the problem?

Problems with Mocks and Spies

- **Comprehension:** the tests are **complicated** and the error messages are **hard to understand.**
- **False negatives:** when behavior changes, the tests should fail, but **don't.**
- **False positives:** when designs change, the tests shouldn't fail, but **do.**

As I said, back in 2000, using mocks solved our database setup problem. But it also introduced new problems of its own.

The first problem was that mocks are hard to understand. My team had trouble visualizing how the mocks overrode real code. They made our tests were more complicated and test failures were harder to diagnose.

They also prone to false negatives, and false positives. They passed when they should have failed, and failed when they should have passed.

Mocks' Fundamental Design Tradeoff

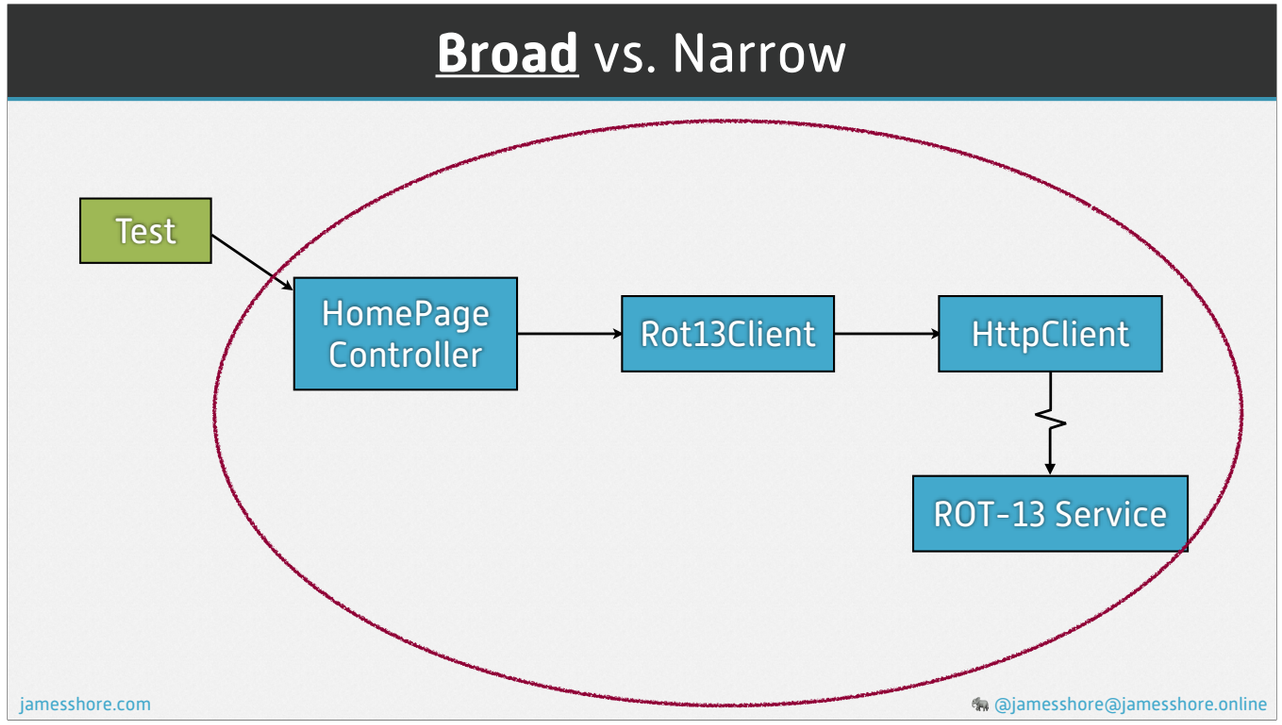
Mocks and spies are used for

- **narrow**
- **solitary** ← false negatives
- **interaction-based** ← false positives

tests.

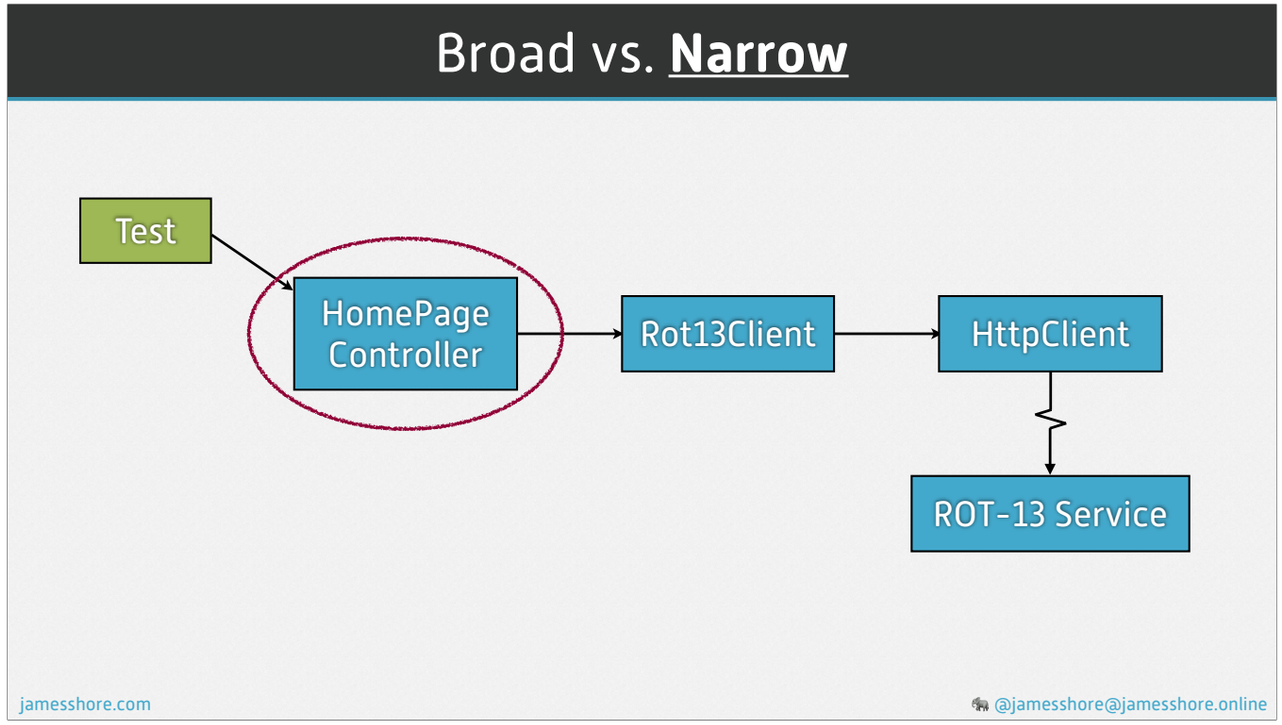
Ultimately, these problems are due to the fundamental design tradeoffs mocks make. Mocks are used to create narrow, solitary, interaction-based tests. Solitary tests will always have false negatives, and interaction-based tests will always have false positives.

Let's look at those tradeoffs in more depth.



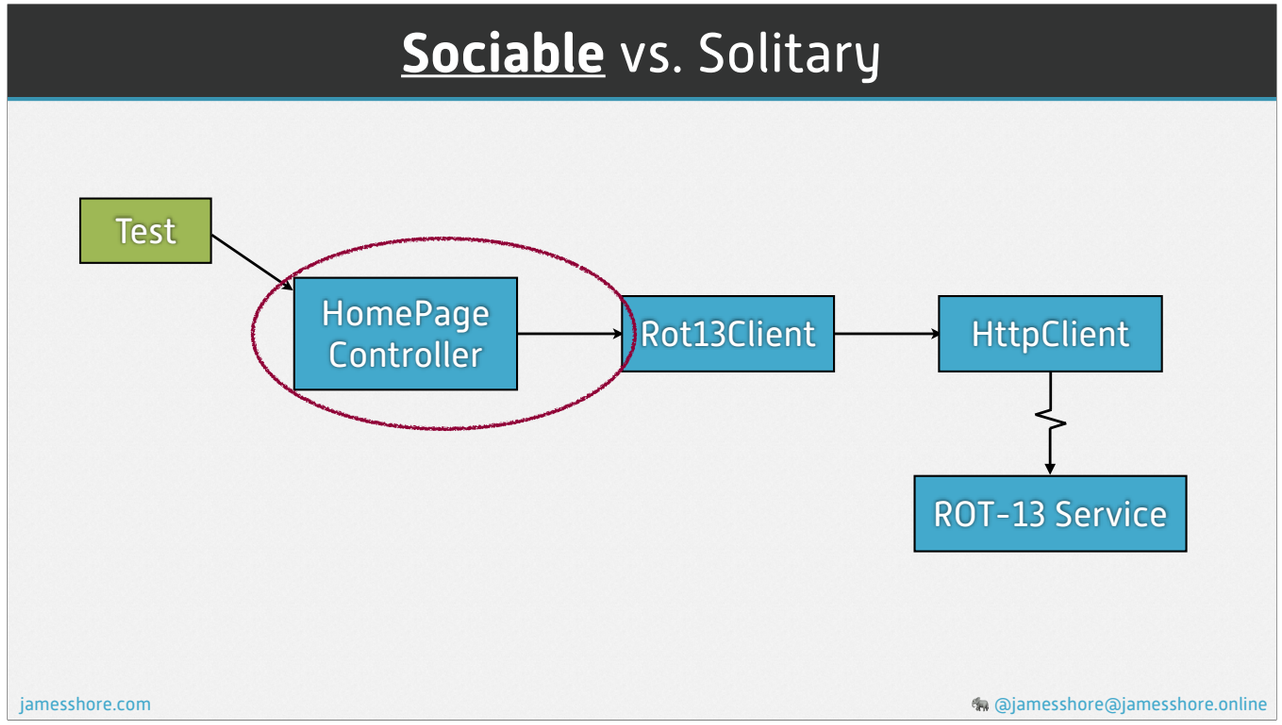
The first tradeoff in test design is broad tests vs. narrow tests.

Broad tests check multiple parts of the system in a single test. End-to-end tests are a classic example. They check a lot of code at once, but they're brittle and slow.



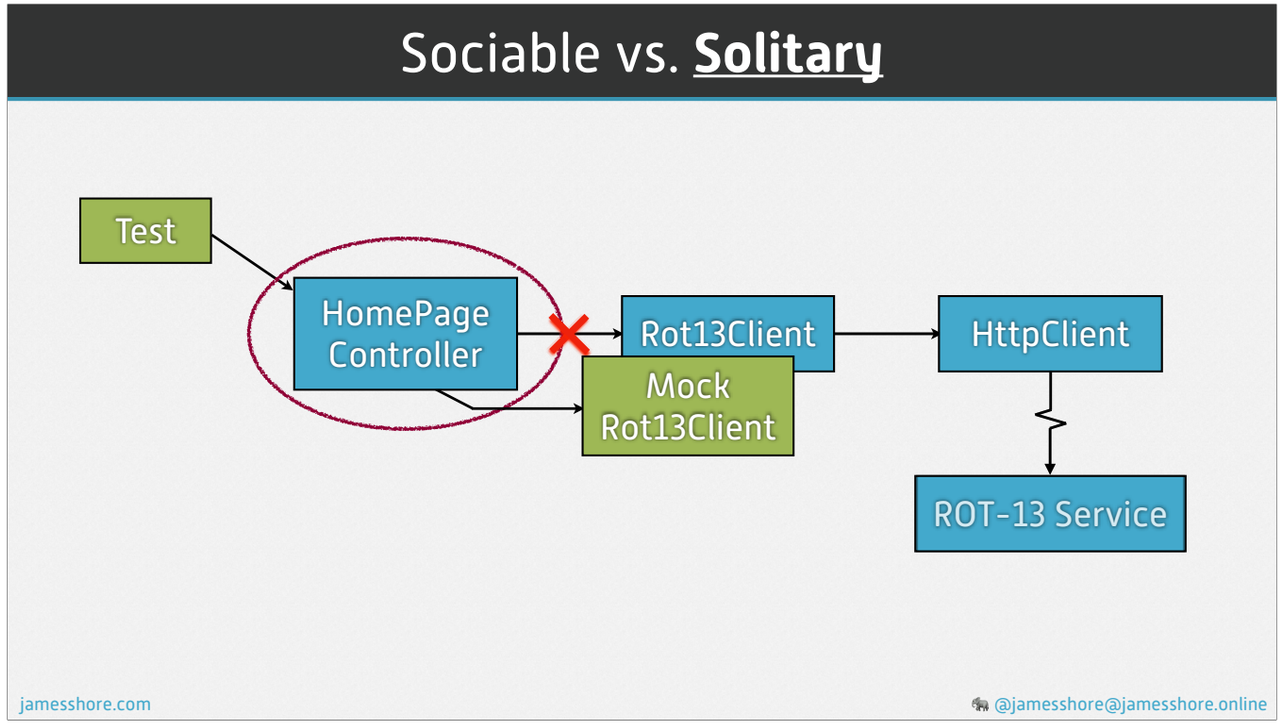
Narrow tests, in contrast, focus on a single, constrained part of the system. Unit tests are a classic example. They're supposed to be fast and deterministic.

That's the first tradeoff. Broad, or narrow? You're trading off speed and determinism for test coverage.



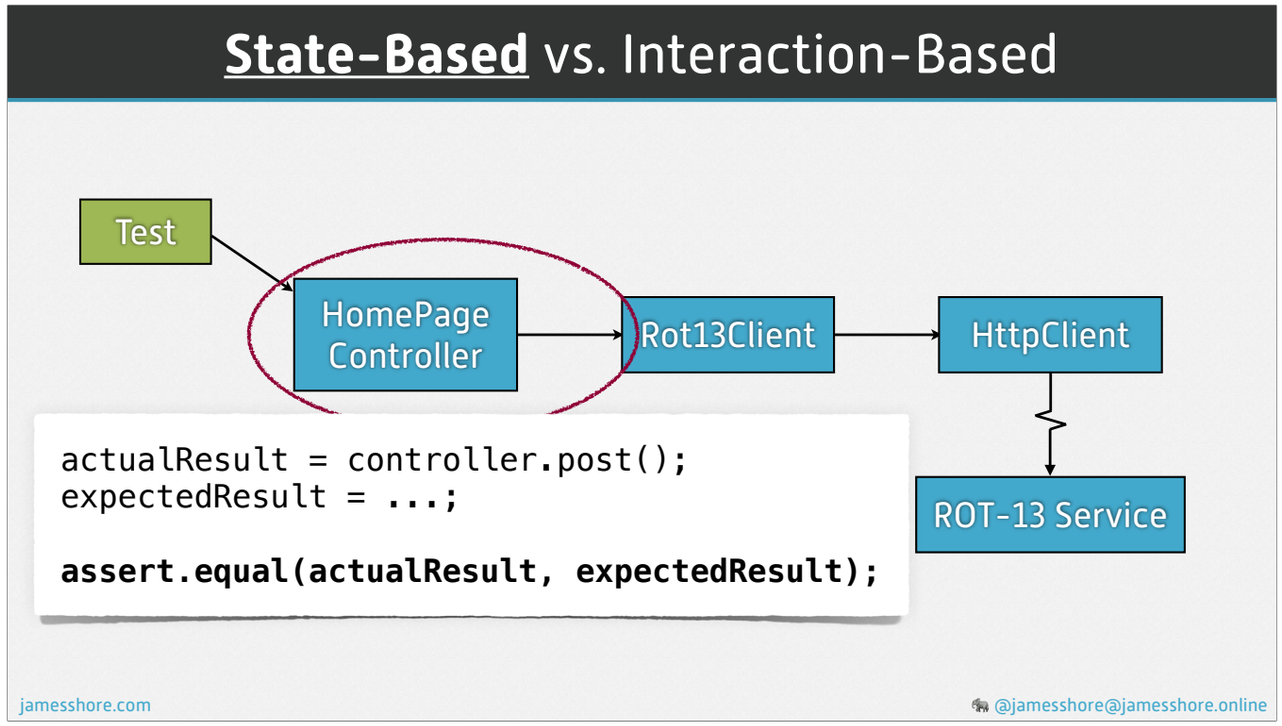
If you choose narrow tests, you have another tradeoff decision: whether your tests are "sociable" or "solitary." Like all narrow tests, a sociable test focuses on a single behavior: in this case, the behavior of HomePageController. But what happens next? HomePageController calls Rot13Client. Rot13Client calls HttpClient. And HttpClient calls the ROT-13 service. A sociable test is focused on the behavior of HomePageController, but it allows it talk to its dependencies.

But that's a problem, because that means our test has to set up the ROT-13 service. It's no longer fast and deterministic. It's not a good narrow test.

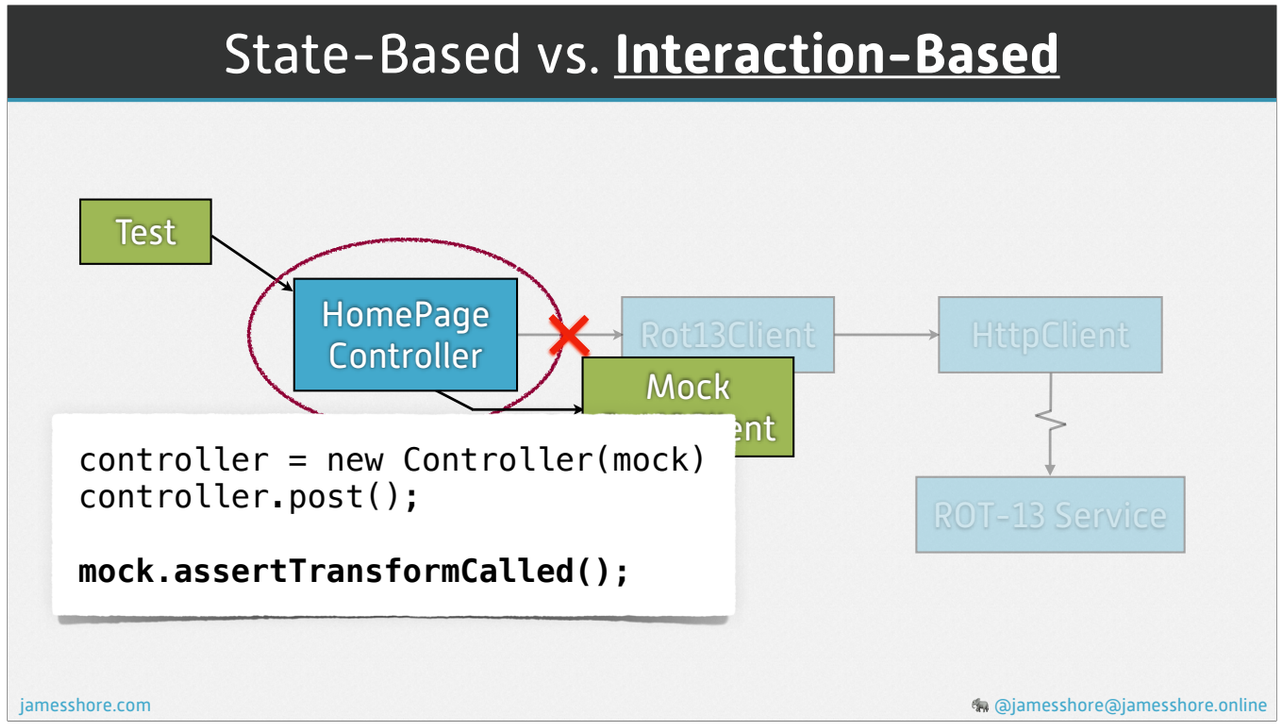


Solitary tests solve that problem. They inject a “test double,” such as MockRot13Client, that takes the place of the real Rot13Client. Now HomePageController doesn't talk to its dependencies, and we don't have to set up the ROT-13 service. But now our test isn't running real code. Not entirely.

That's the second trade-off. Sociable, or solitary? This time, you're trading speed and determinism for realism.

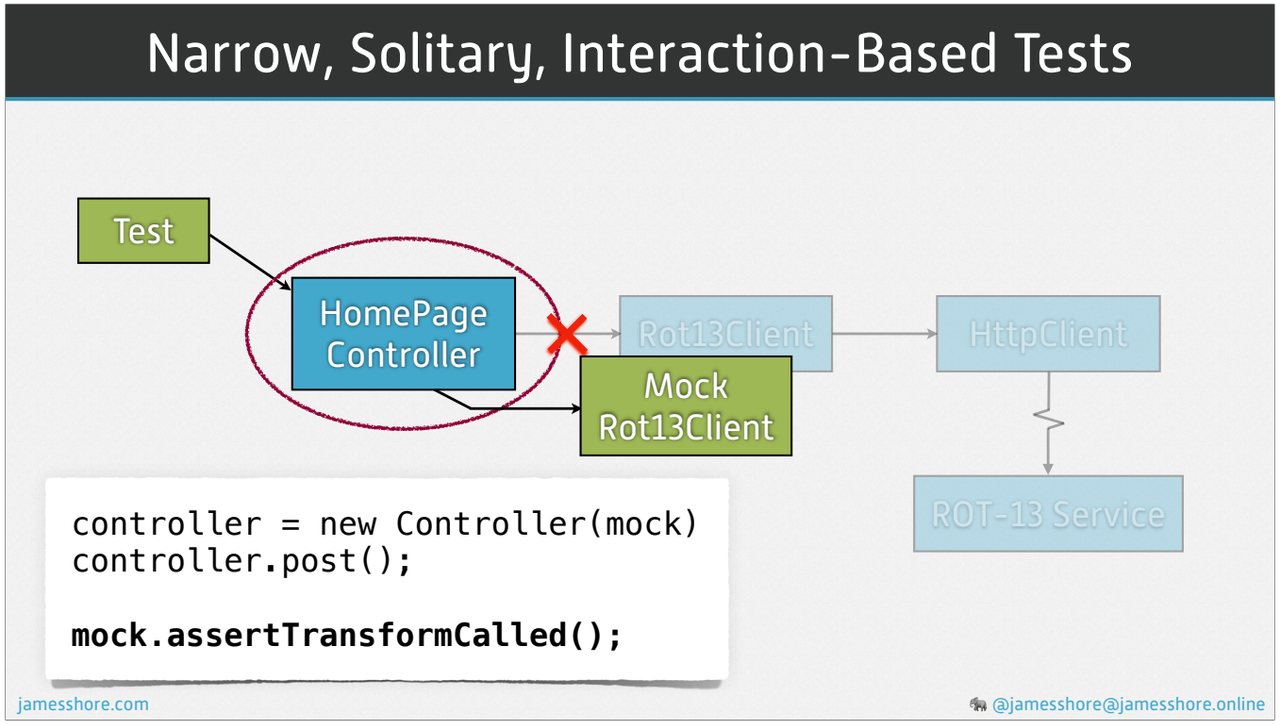


Your third trade-off decision is between state-based and interaction-based tests. State-based tests check the **behavior** and **state** of the system under test. They don't care about its dependencies—they're an implementation detail to be ignored. As long as HomePageController produces the correct results, the test doesn't care how it does it.



Interaction-based tests, in contrast, consider interactions to be a fundamental design question that needs be checked by the test. It comes from the “tell, don’t ask” school of object-oriented design. Tests check whether the system under test calls the correct methods on its dependencies.

And that's your third design trade-off. Are interactions with dependencies an implementation detail to be ignored, or a behavior to be tested?

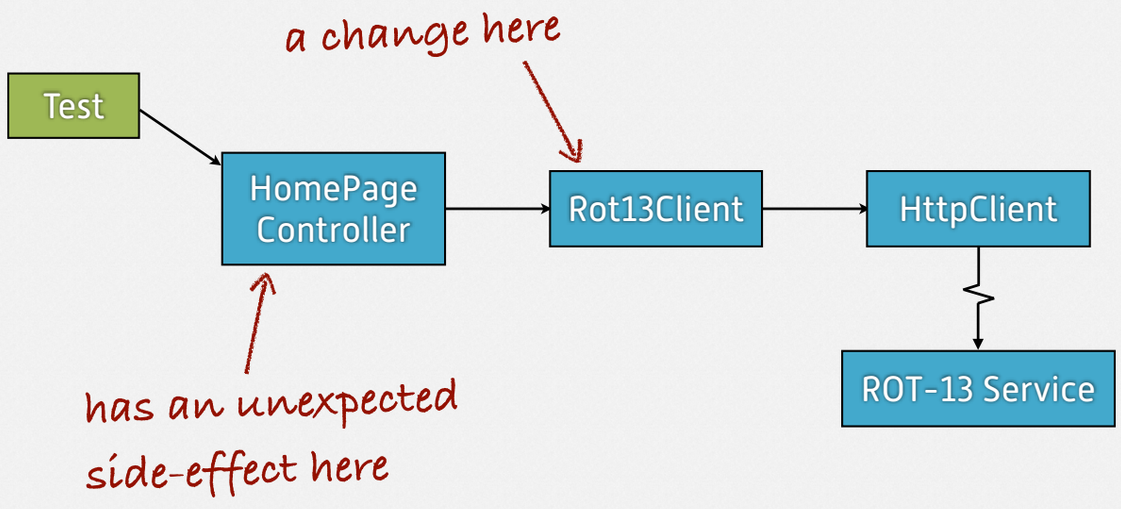


Design is tradeoffs, and there's no one right answer. But it's important to know which tradeoffs you're making. If you use mocks (or spies), you're choosing narrow, solitary, interaction-based tests.

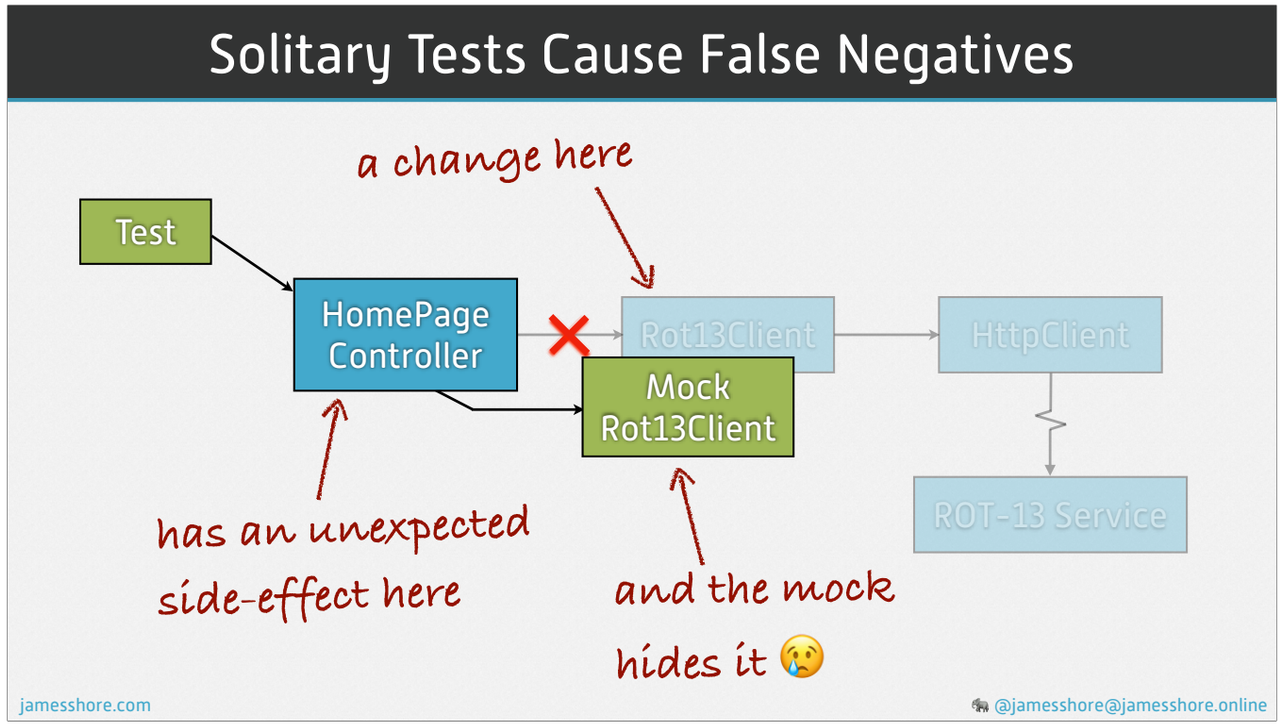
Narrow: they test a specific behavior.
Solitary: they prevent the code under test from calling its real dependencies.
Interaction-based: they check how the code calls its dependencies.

These tradeoffs are why mock-based tests sometimes fail when they should pass, and sometimes pass when they should fail. Let me explain.

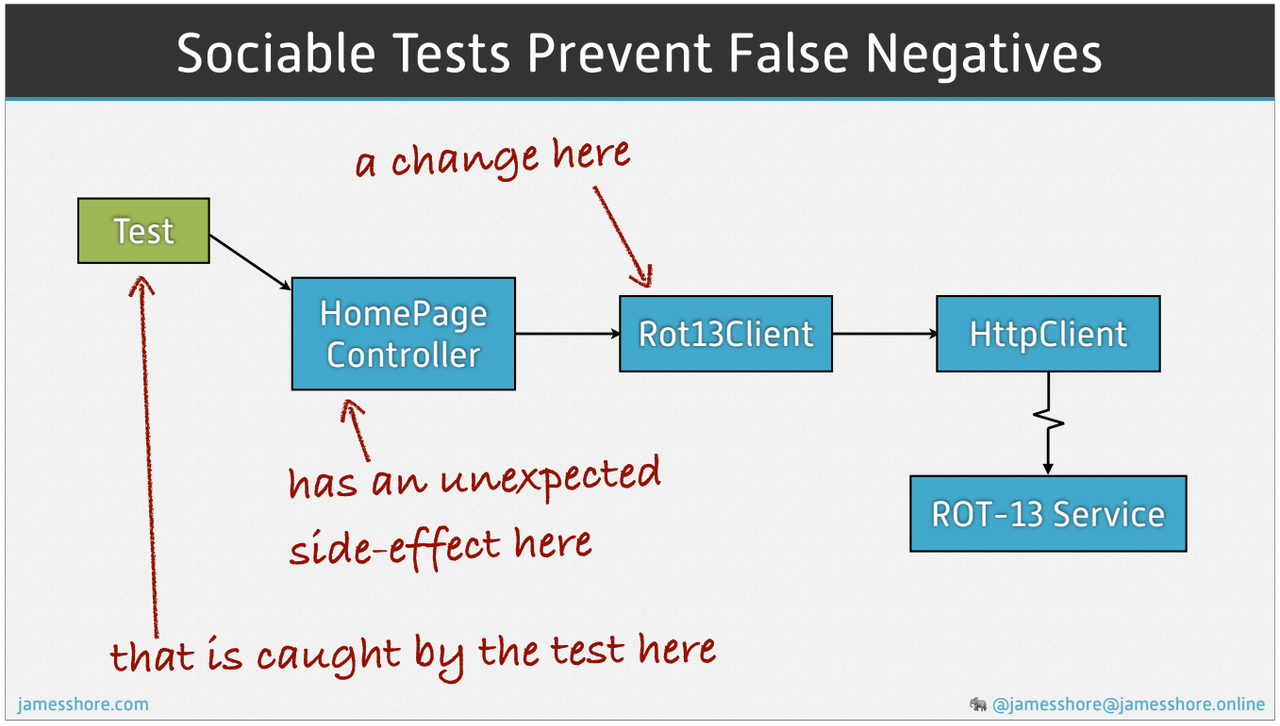
Bugs Live in the Boundaries



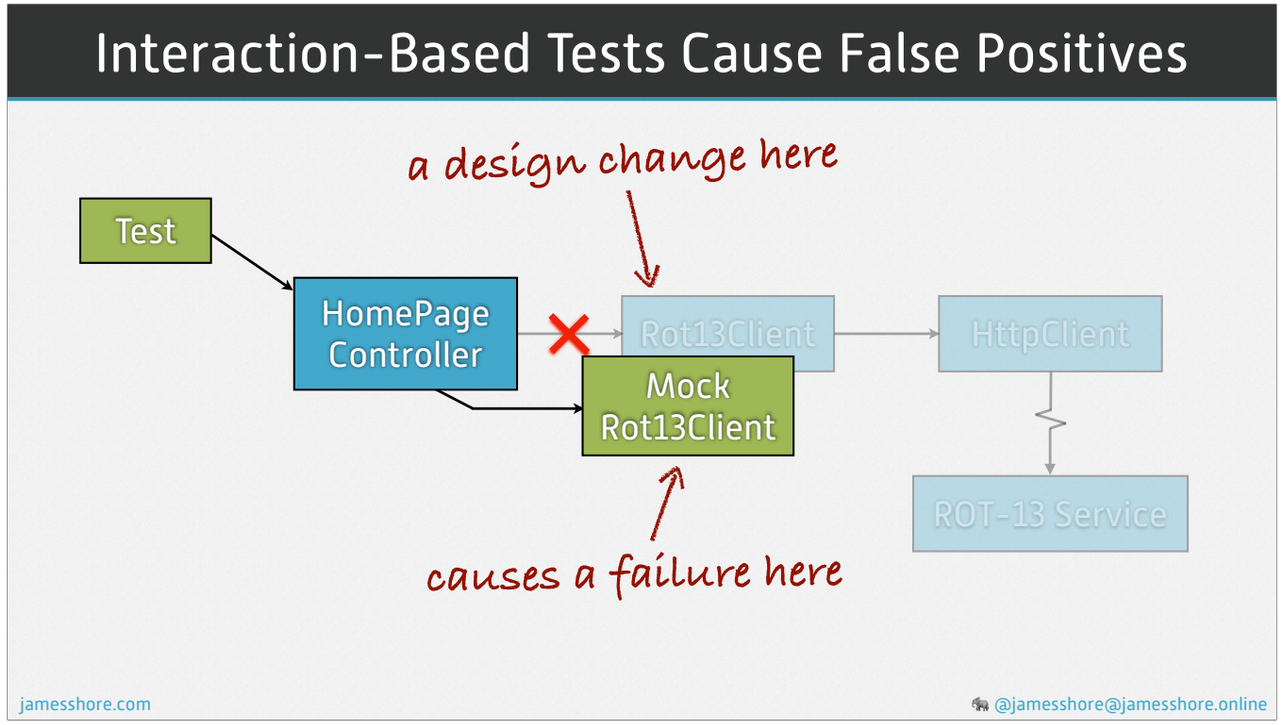
Let's say someone makes a well-intentioned change to Rot13Client. They even update its tests. But HomePageController assumes Rot13Client will behave in a particular way, and it no longer does. The change introduces a bug.



A solitary test hides that bug. The mock is coded with particular assumptions. Even though those assumptions are no longer true, there's nothing to check that the mock is correct. The HomePageController is broken, but the HomePageController's tests still work.

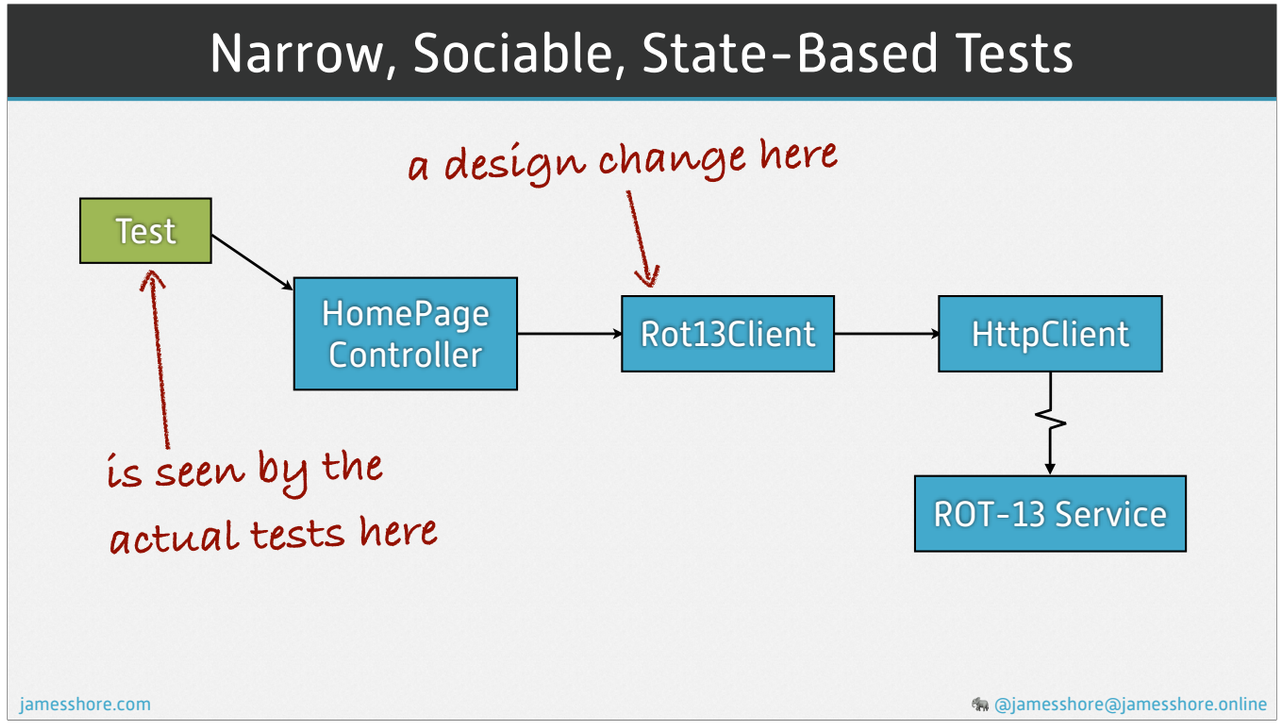


Sociable tests don't have that problem. If you make a change to Rot13Client that changes the behavior of HomePageController, the tests catch it, because the tests run real code.

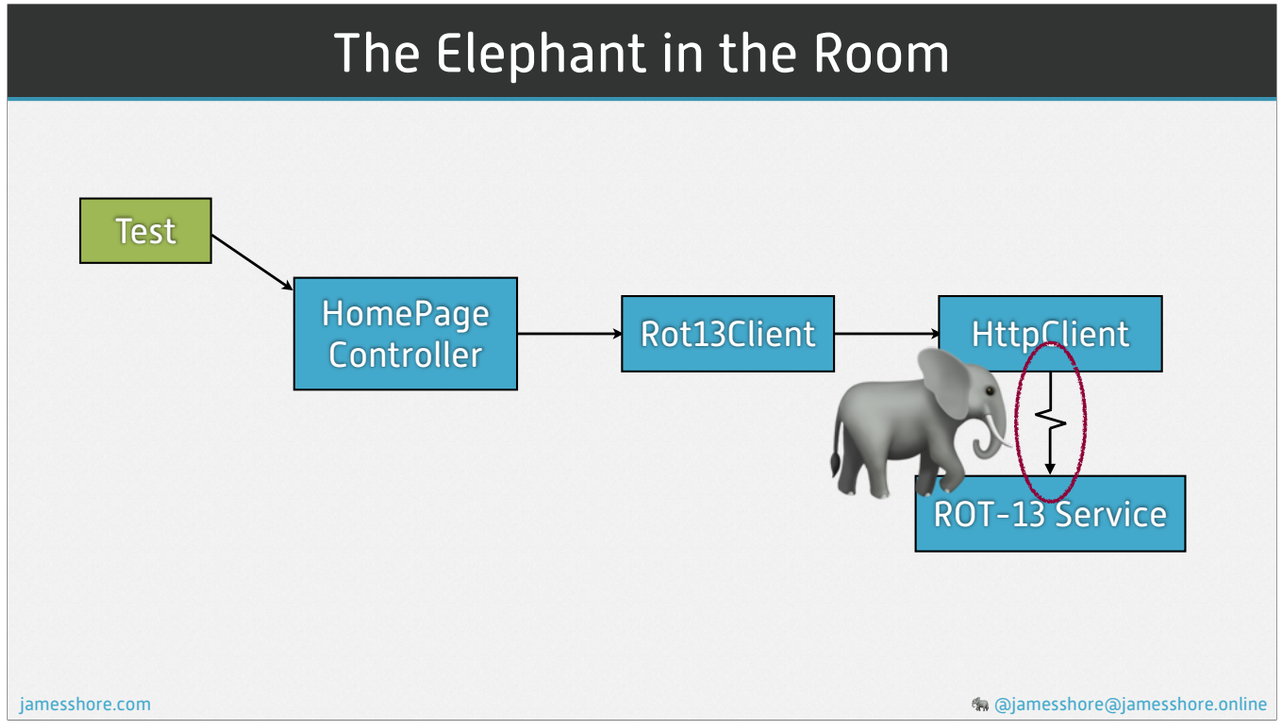


Mocks also result in tests that fail when they should pass. That's a result of choosing interaction-based tests.

Let's say you're refactoring Rot13Client. You're getting ready to add some new capabilities to HomePageController, or maybe you're just cleaning things up. You're not changing any behavior yet. You refactor Rot13Client and update HomePageController accordingly. Now HomePageController's tests are failing, even though you didn't break anything. That's because HomePageController's interactions with Rot13Client have changed. You have to change your tests to match the new interactions to get things working again.



State-based tests don't have that problem. Interactions with dependencies aren't tested—just the behavior of your code. If you refactor Rot13Client and it doesn't change the way HomePageController works, your tests still pass.



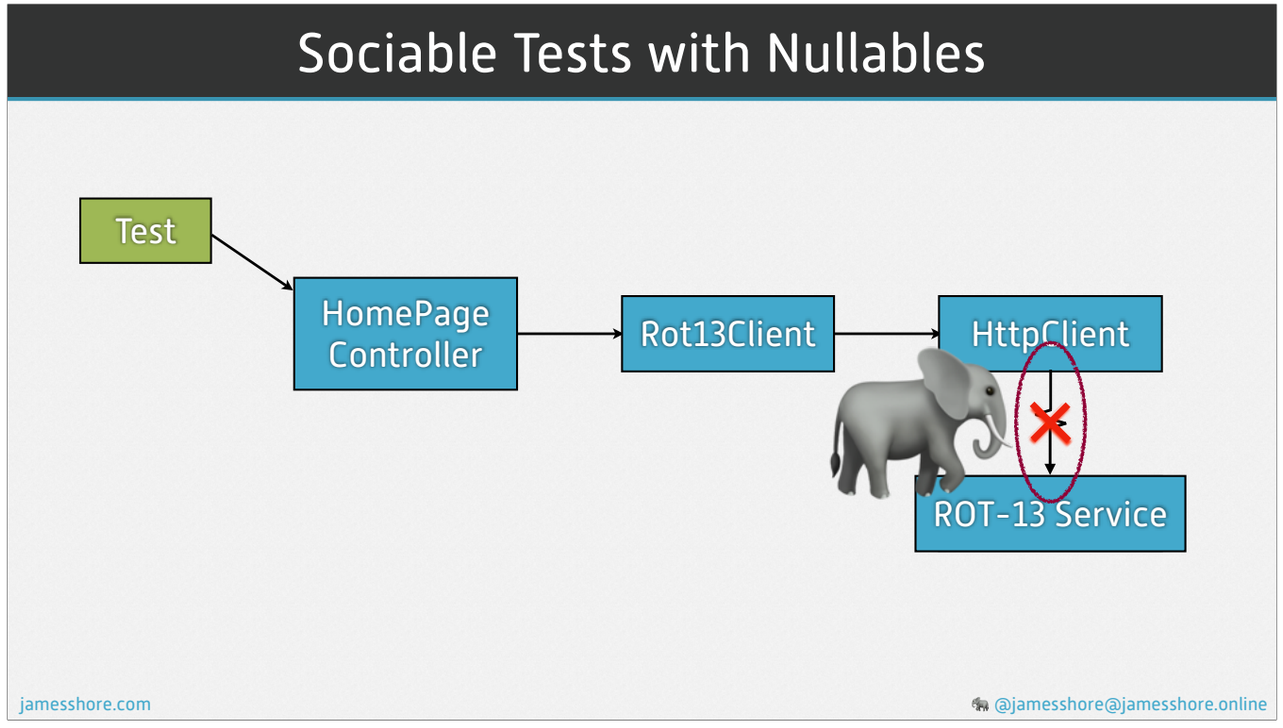
Those are some pretty big tradeoffs. Solitary tests make tests pass when they should fail, and interaction-based tests make tests fail when they should pass. So why don't we just use sociable, state-based tests?

That's what I want, too. But sociable tests have a big problem: they run real dependencies. HomePageController talks to Rot13Client. Rot13Client talks to HttpClient. And HttpClient talks to the ROT-13 service.

Now our test has to set up the ROT-13 service. It's no longer fast and reliable. And that's a problem.

I wrestled with this problem for years. I wasn't willing to put up with the downsides of mocks, but I didn't want slow tests, either. I tried a bunch of different solutions. Honestly, they were giant hacks, and I was embarrassed to share them.

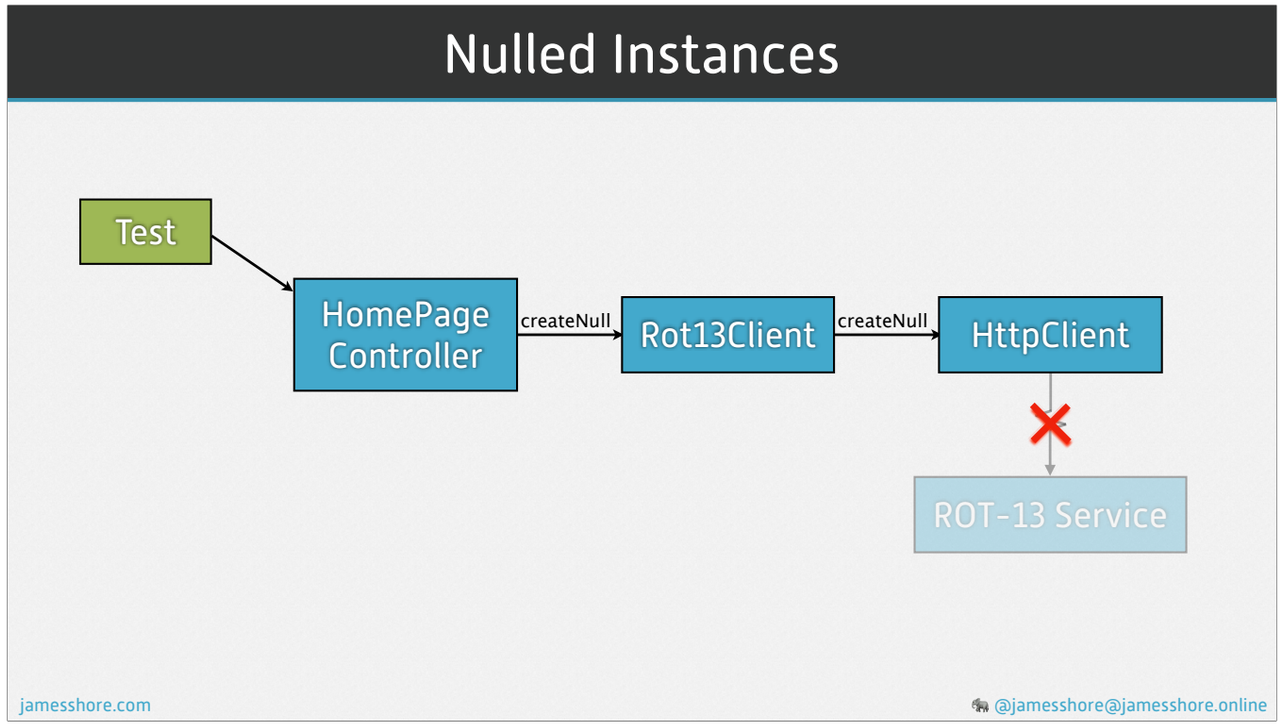
Until I figured out Nullables.



Nullables are production code with an "off" switch. They allow you to run real production code that doesn't talk to the outside world. This gives you the best of both worlds: now you can write sociable tests that are fast and deterministic like mock-based tests, and simple and reliable like sociable tests.

Nullables still have tradeoffs. Design is tradeoffs. I'll share those tradeoffs later. Today, we'll focus on how Nullables are used.

2 Core Techniques



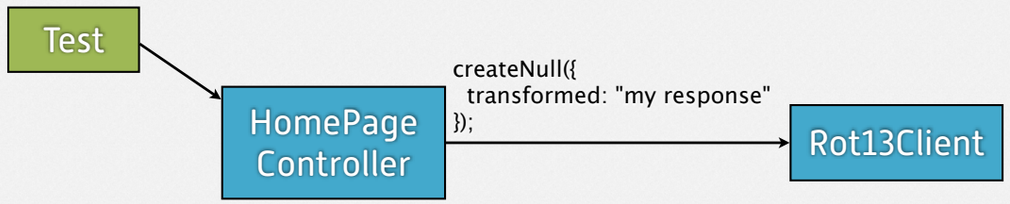
Let's say you're implementing HomePageController and you already have a Nullable Rot13Client. Your test will create a “Nulled” instance of Rot13Client and pass it into your system under test. It does that by calling the “createNull()” factory method on Rot13Client.

From the perspective of your test, that's it! Remember, dependencies are an **implementation detail**. Your code doesn't know how Rot13Client works, and doesn't want to. All you know is that Rot13Client talks to the ROT-13 service—somehow—and the “Nulled” version turns that behavior off. Now the test can do whatever it wants with HomePageController without worrying about making network calls.

In the “Testing Without Mocks” pattern language, this is called **Parameterless Instantiation.** Any class can be instantiated without parameters. It will take responsibility for instantiating its dependencies and making sure it's in a working state.

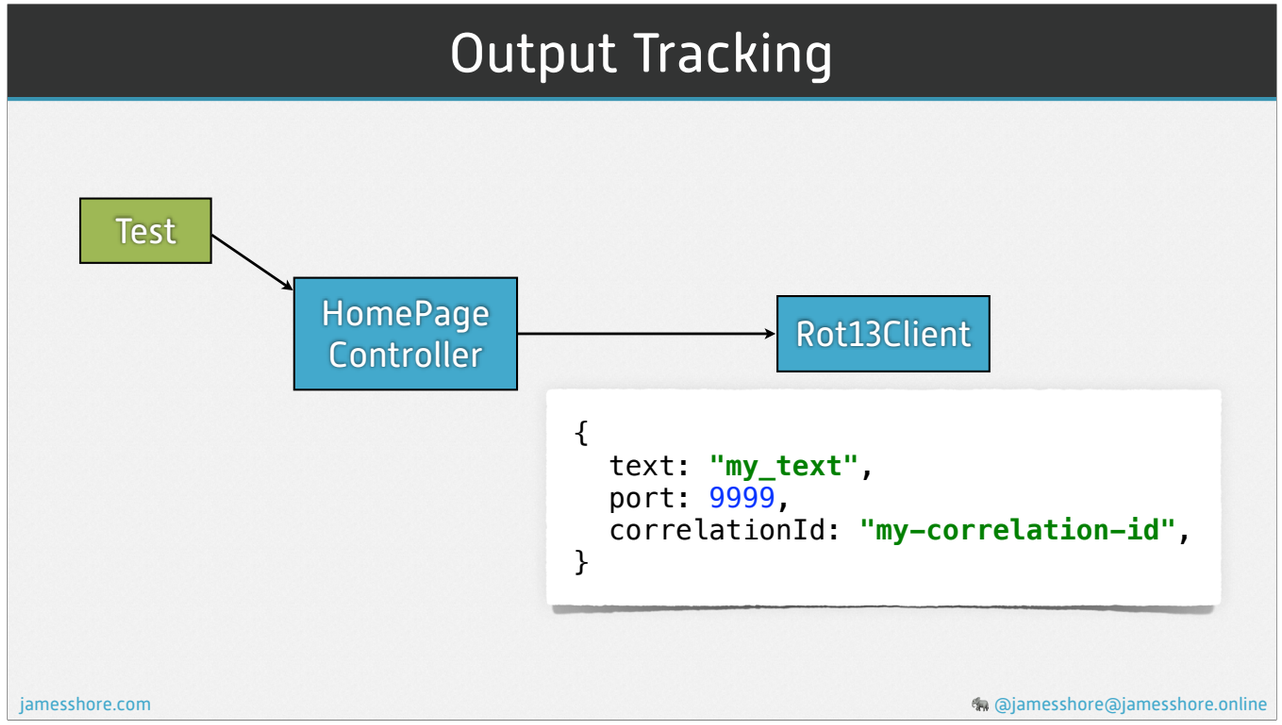
Under the covers, of course, Rot13Client is instantiating HttpClient. Because Rot13Client is Nulled, HttpClient is also Nulled. And HttpClient takes responsibility for making sure the communication with the ROT-13 service is turned off. We'll look at the details of how that works in later modules.

Configurable Responses



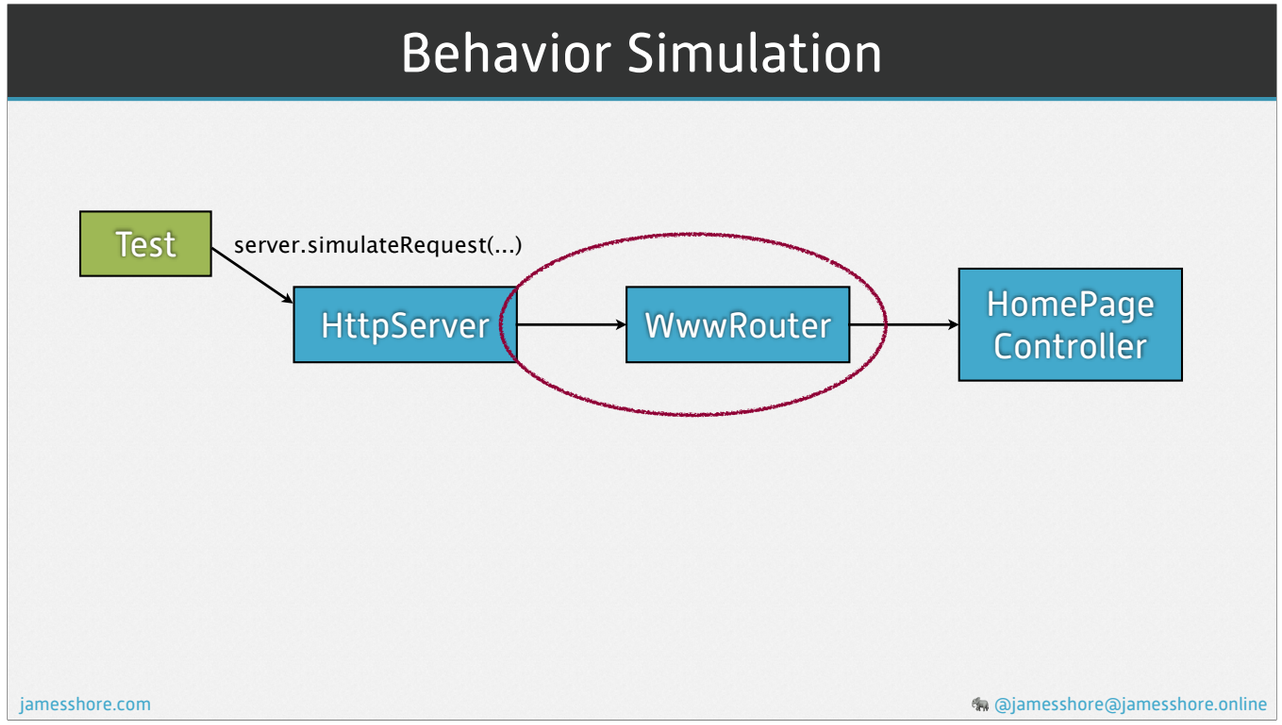
Some of your HomePageController tests will check how HomePageController handles Rot13Client's return values. Does HomePageController parse return values correctly? How does it handle malformed return values? And exceptions? And hangs?

You can control the behavior of a Nulled instance by providing configuration parameters to the createNull() factory. This is called **Configurable Responses**.



Similarly, some of your tests will want to make sure the correct data is sent to the ROT-13 service. In a state-based test, you don't check interactions, so how do you ensure the right data was sent? It doesn't result in any state changes or behavior you can observe.

The answer: make it stateful. This is called **Output Tracking**, and it's done in a way that doesn't use any memory or CPU when it's not in use. You tell Rot13Client to start tracking requests, and then later you examine what it tracked. You'll see how it works when you do the exercises.



There's one more aspect of Nullables that won't come up in the exercises. Sometimes your code receives events from external systems. For example, when the a web browser makes a request, that results in an "GET" event in the server code. Nullables can simulate those events using a pattern called **Behavior Simulation**. It works be adding a "simulate()" method to a Nullable that runs the same code as a real event.

For example, the HomePageController is called by WwwRouter, and the WwwRouter is called by HttpServer. Although our tests could call the same WwwRouter method that HttpServer does, that leaves us vulnerable to changes in HttpServer's behavior. So instead, we add a "simulateRequest()" method to HttpServer that runs the same code a real request does. Our test calls that method, which turns around and calls WwwRouter.

Signature Shielding

```
it("GET renders home page", async () => {
   const rot13Client = Rot13Client.createNull();
   const clock = Clock.createNull();
   const controller = new HomePageController(rot13Client, clock);

   const request = HttpServerRequest.createNull();
   const config = WwwConfig.createTestInstance();

   const response = await controller.getAsync(request, config);

   const expected = homePageView.homePage();
   assert.deepEqual(response, expected);
});
```

```
it("GET renders home page", async () => {
   const { response } = await getAsync();
   assert.deepEqual(response, homePageView.homePage());
});
```


There's another pattern you'll see used in the examples: **Signature Shielding.** It's basically test helpers. When production code changes, you often end up having to change a bunch of tests. DRY up your tests with a helper method that encapsulates the setup your tests need. It's better than using your testing framework's "setup" code because you can parameterize it.

Summary of Techniques

- When a dependency involves external systems:
  → **Make the dependency Nullable**
- When you want to simulate a response:
  → **Use a Nullable with Configurable Responses**
- When you want to check invisible actions:
  → **Use Output Tracking**
- When you want to simulate an event:
  → **Use Behavior Simulation**
- When you want to protect your tests against code changes:
  → **Use Signature Shielding**

Here's a summary of all the techniques I've discussed. If it seems like a lot, don't worry. The exercises will make things more clear.

Further Reading

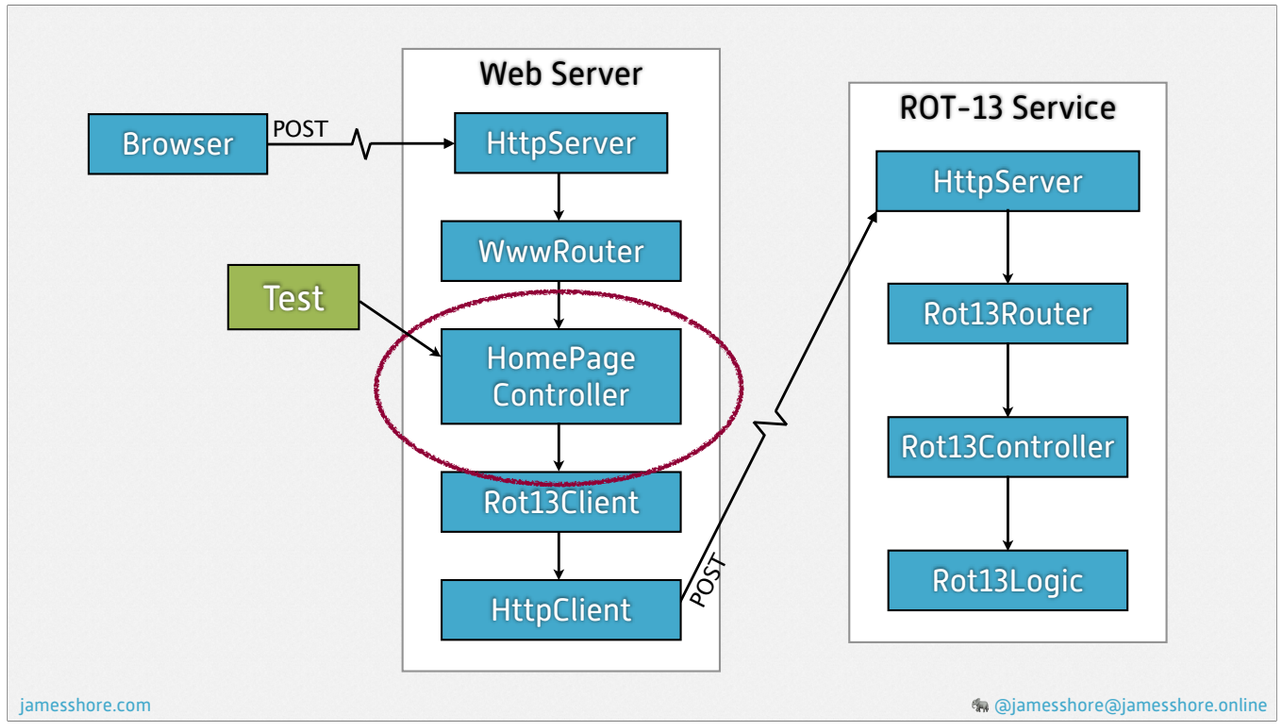
Patterns in **jamesshore.com/s/nomocks**:

- Narrow Tests
- State-Based Tests
- Overlapping Sociable Tests
- Signature Shielding
- Parameterless Instantiation
- Nullables
- Configurable Responses
- Output Tracking
- Behavior Simulation

This course is focused on learning by doing, so I've ignored some details in favor of spending more time on practical examples. For all the nitty-gritty details, read the “Testing Without Mocks” article. It's a good reference. These are the specific patterns I've discussed today. You can find them at this URL.

3 Exercises

Speaking of learning by doing...



In this module's exercises, you'll be implementing the HomePageController. Here's how it fits into the overall application architecture. The browser talks to the web server, and the web server talks to the ROT-13 service.

You don't need to worry about every detail. Your focus will be on the HomePageController. You'll inject a Nulled Rot13Client and test that HomePageController handles requests properly.

As you work on the exercises, be sure to take advantage of all the resources I've provided. There's an extensive series of API documentation, JavaScript primers, and hints available.

(Walk through exercise setup.)

(Walk through API docs, primers, and hints.)

Testing Without Mocks
Using Nullables

Presented by James Shore
jamesshore.com

v2023-06-02

That's a brief introduction to using Nullables. You'll use the exercises to solidify this foundation. Good luck!