

# **Von Giganten, Lügnern und Trantüten**

## **Ein (Unit-)Test-Anti-Pattern-Märchen**

Birgit Kratz, 27.06.2023, DWX 2023

# About me

## Birgit Kratz

- Freelancing IT Consultant
- Java-Backend
- More than 20 years experience
- Co-Organizer of Softwerkskammer in Düsseldorf and Köln (Cologne)
- Co-Organizer of SoCraTes-Conf Germany
- Email: mail@birgitkratz.de
- Twitter: @bikratz
- Github: <https://github.com/bkratz>
- Web: <https://www.birgitkratz.de>

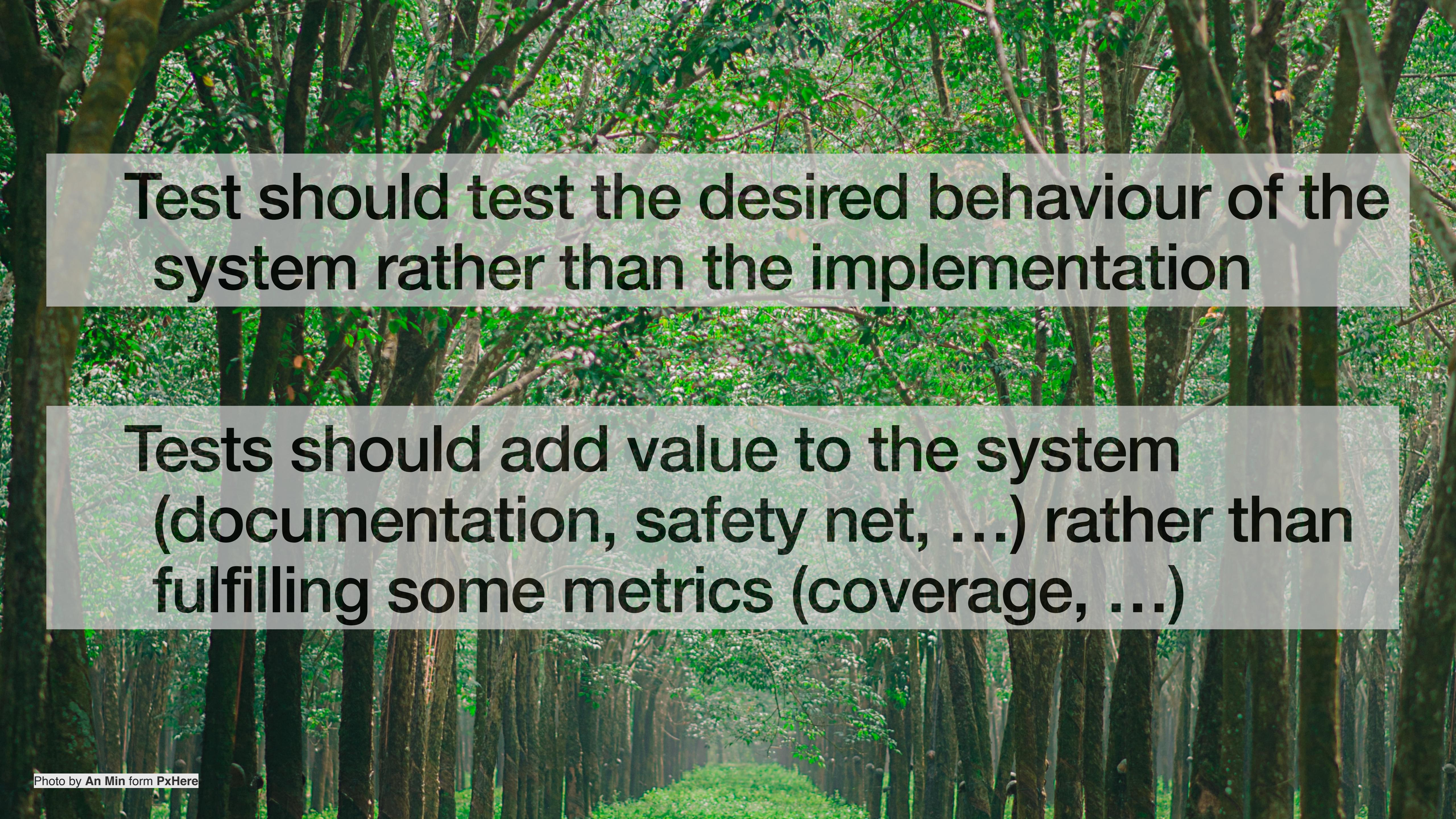




Once upon a time



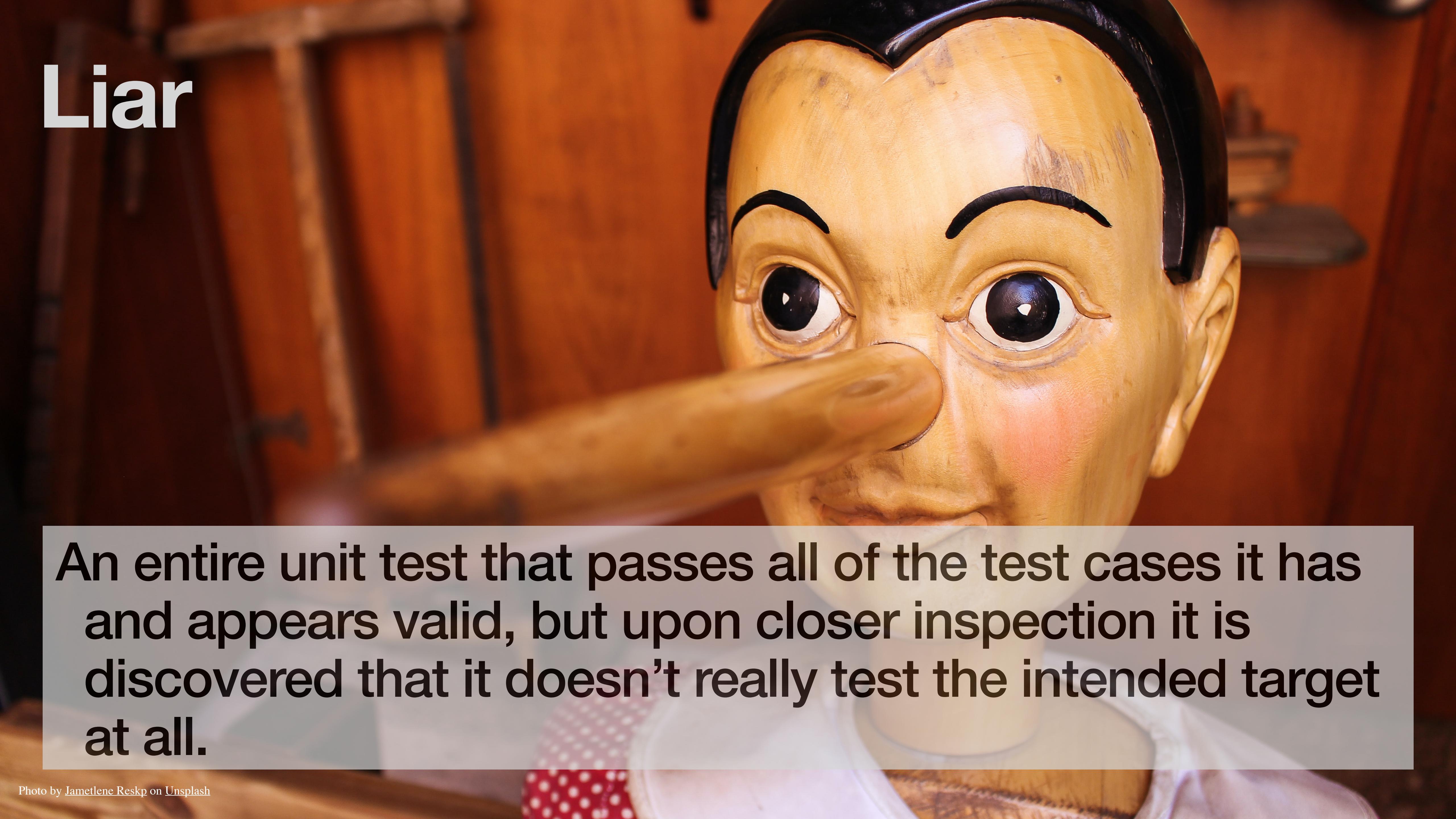




**Test should test the desired behaviour of the system rather than the implementation**

**Tests should add value to the system (documentation, safety net, ...) rather than fulfilling some metrics (coverage, ...)**

# Liar

A close-up photograph of a wooden Pinocchio mask. The mask is light-colored wood with dark hair and eyebrows. It has large, dark, bulging eyes and a prominent, articulated wooden nose that is currently pointing downwards. The background is a warm-toned wooden surface.

An entire unit test that passes all of the test cases it has and appears valid, but upon closer inspection it is discovered that it doesn't really test the intended target at all.

# Liar



<b>Example</b>	<b>Cause</b>	<b>Solution</b>
passes all tests with no useful assertions (aka: Line Hitter)	chasing test coverage not practicing test-first approach	add meaningful assertions or delete the test
test method name and test method content do not match	refactoring, but somehow the tests are still green not practicing test-first approach	keep test method names and test method content in sync

# The Enumerator

A unit test with each test case method name is only an enumeration, i.e. `test1`, `test2`, `test3`. As a result, the intention of the test case is unclear, and the only way to be sure is to read the test case code and pray for clarity.

# The Enumerator

Example	Cause	Solution
test method names are the same except for a number at the end  often in this comes in combination with <b>The Liar</b>	trying to test the same method with different inputs  being not creative enough or just lazy to find good test method names	rename test methods to represent the indicate the input and expected output  possibly use parameterized tests

# The Happy Path

A unit test that only tests the expected behaviour, not testing any boundaries or exceptions. The anti-pattern here is when the developer stops at happy path tests.

# The Happy Path

Example	Cause	Solution
only one test per unit  test only trying to prove the correctness of the business logic/algorithm	not practicing test-first approach  not testing boundaries	start using test-first and start with testing the boundaries, using some out of boundary values  consider using Mutation Testing  consider using Property Based Testing

# Excessive Setup

A photograph of a grand, multi-story castle with light-colored stone walls and dark green, pointed roofs. The castle features several prominent towers, one of which has a tall, thin spire topped with a small flag. The sky above the castle is a clear, pale blue with wispy, white clouds.

A test that requires a lot of work setting up in order to even begin testing. Sometimes several hundred lines of code is used to setup the environment for one test, with several objects involved, which can make it difficult to really ascertain what is tested due to the “noise” of all of the setup going on.

# Excessive Setup

Example	Cause	Solution
lots of mocked dependencies lots of code to form a scenario always set up the whole application context, instead of using only what is needed	tested class or method do too much, poor separation of concerns  tests and code are highly coupled  not practicing test-first approach  not practicing object calisthenics	start improving abstraction and separation of concerns  practice test-first  practice object calisthenics

# Giant



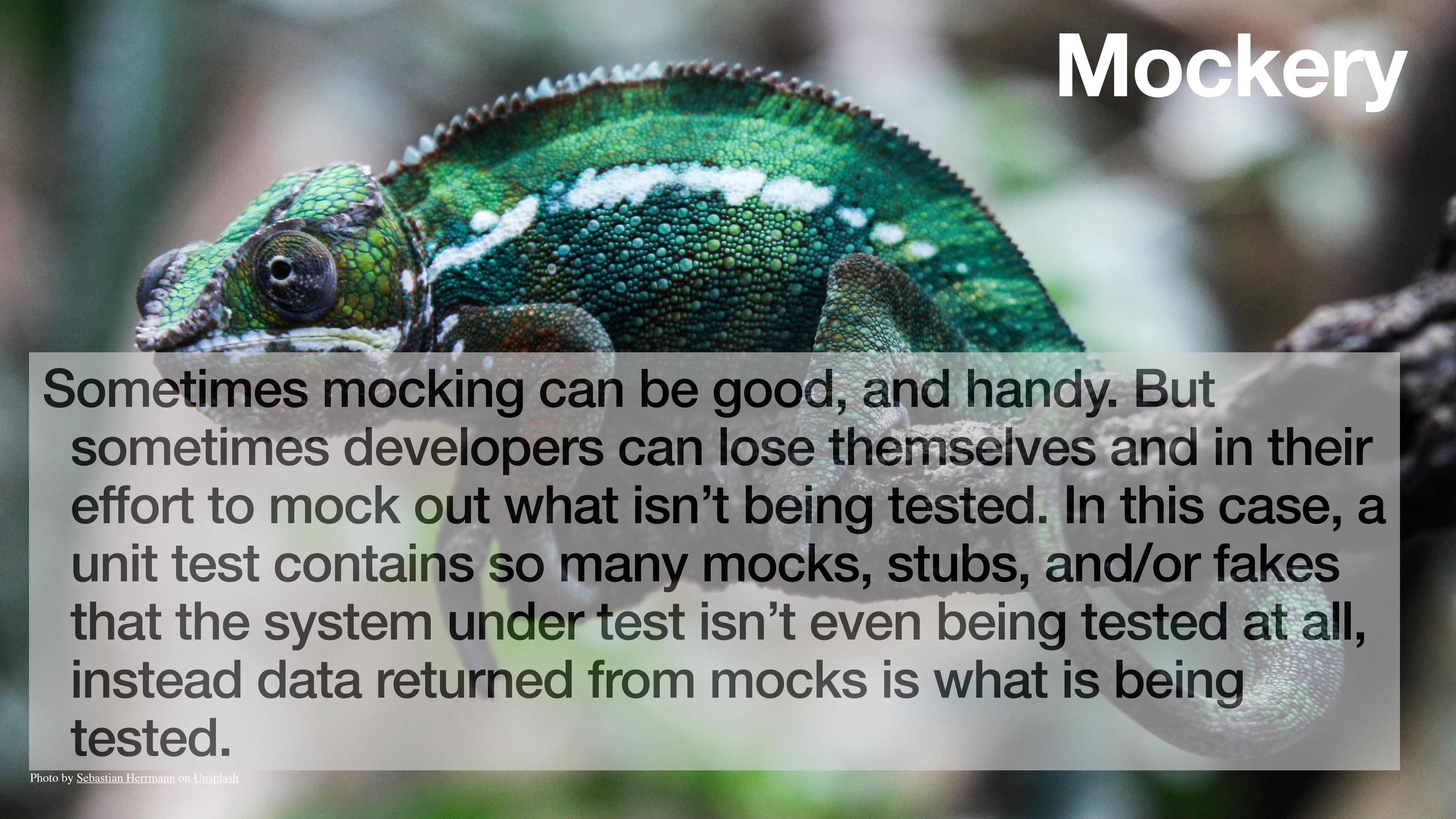
A unit test that, although it is validly testing the object under test, can span thousands of lines and contain many many test cases. This can be an indicator that the system under tests is a God Object

# Giant



Example	Cause	Solution
<p>test with many lines of code, it takes ages scrolling the test and nothing can be found</p> <p>tests with comment lines separating different sections within the test class</p>	<p>its easy to put everything in one class to keep dependencies low</p> <p>a util class to collect all util methods used within the program, no matter where they are used</p>	<p>refactoring the tested class to several classes with separate concerns</p> <p>practice object calisthenics</p>

# Mockery

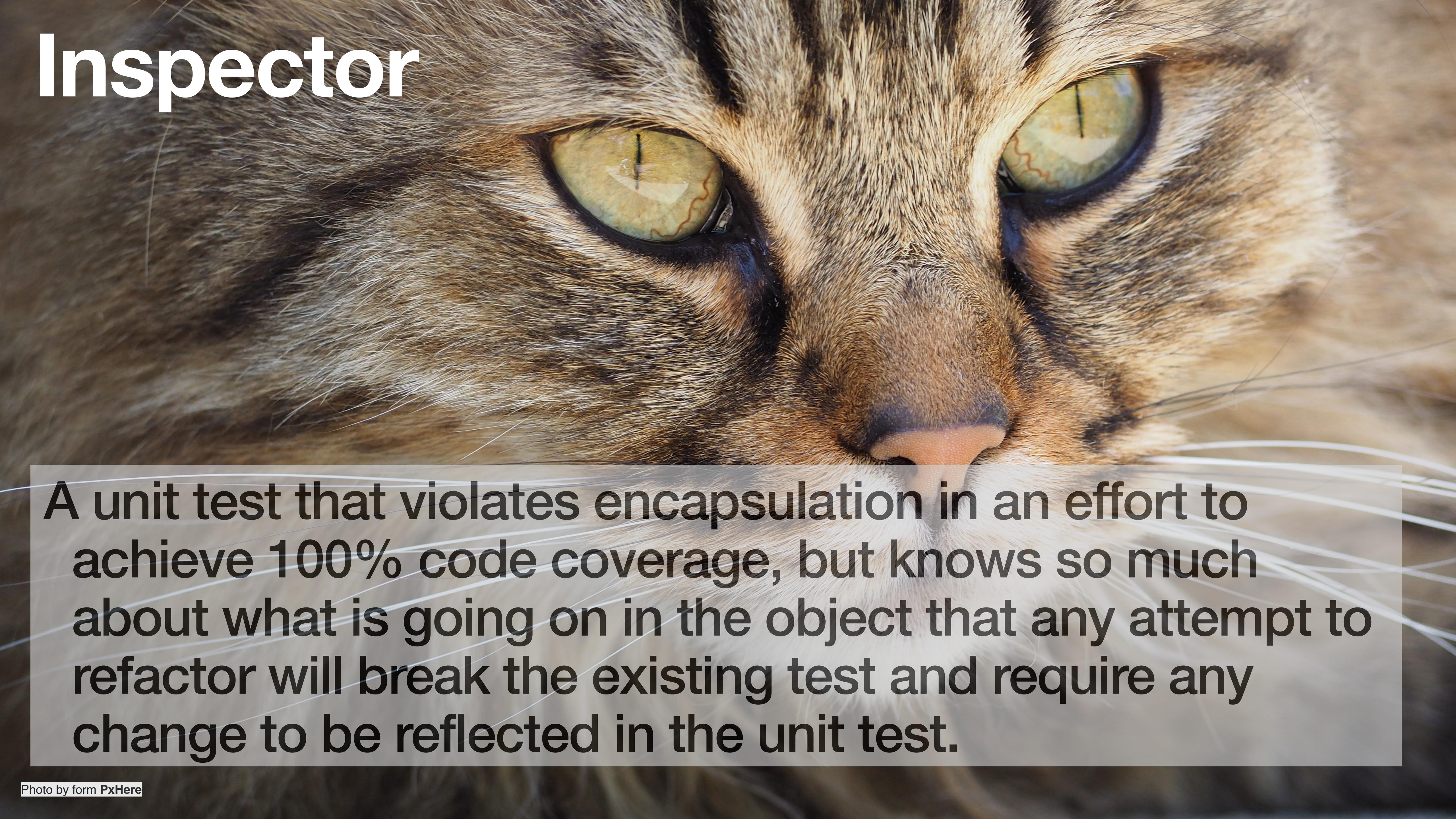


Sometimes mocking can be good, and handy. But sometimes developers can lose themselves and in their effort to mock out what isn't being tested. In this case, a unit test contains so many mocks, stubs, and/or fakes that the system under test isn't even being tested at all, instead data returned from mocks is what is being tested.

# Mockery

Example	Cause	Solution
lots of dependencies that need mocking to isolate the code to test even partially mocking the class under test	class under test contains methods that do not really belong there and therefore have to be mocked  tests and code are highly coupled  see: Excessive Setup	possibly refrain from using mocking frameworks and write your own Mocks, Stubs, Fakes, Test-Doubles (which will make you think about mocking)  refactoring to less dependencies using abstraction and separation of concerns

# Inspector

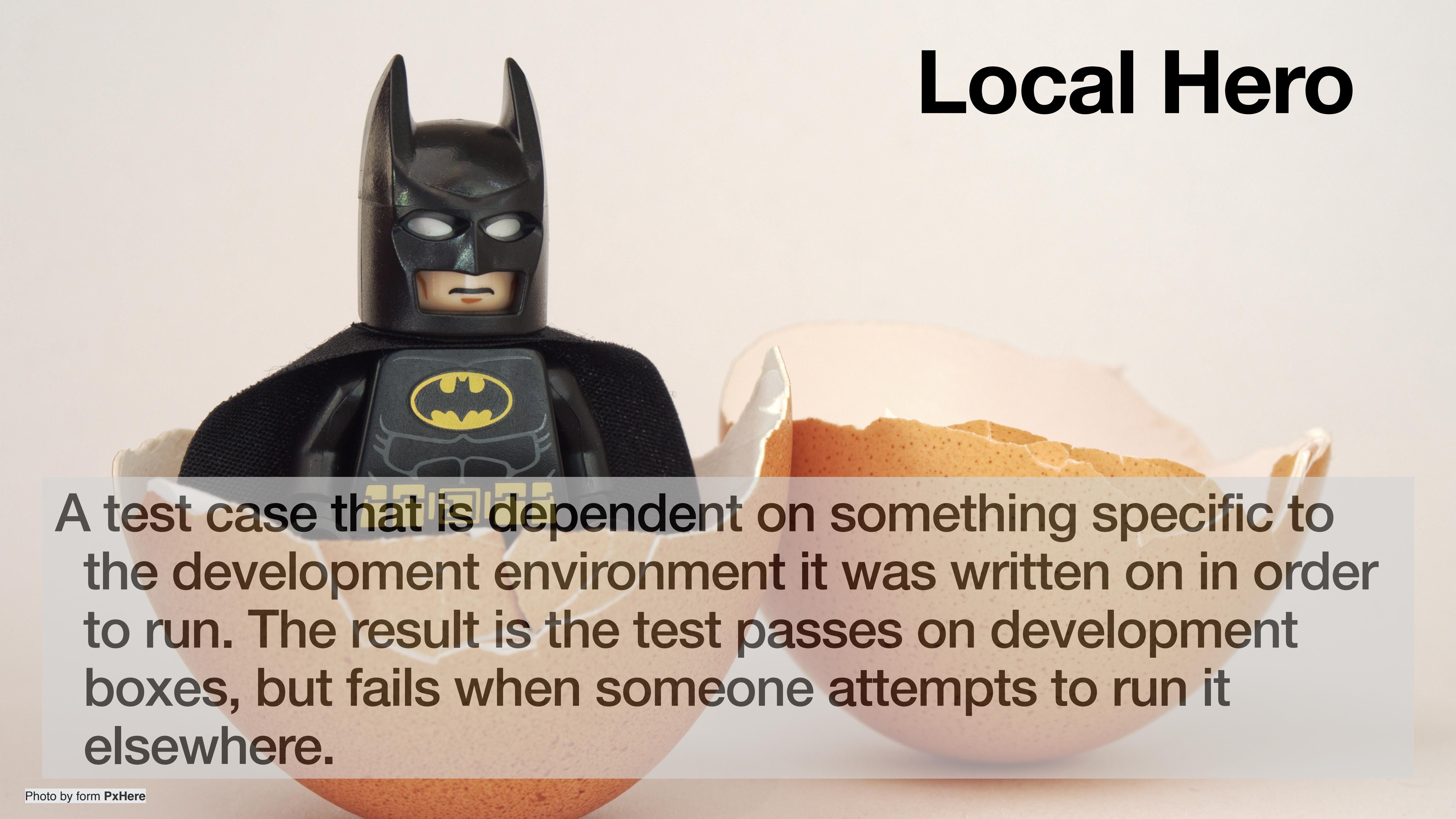
A close-up photograph of a cat's face, focusing on its intense green eyes and the texture of its brown and black fur. The cat has a slightly weary or focused expression. The background is blurred, making the cat's features stand out.

A unit test that violates encapsulation in an effort to achieve 100% code coverage, but knows so much about what is going on in the object that any attempt to refactor will break the existing test and require any change to be reflected in the unit test.

# Inspector

Example	Cause	Solution
making methods public, just to be able to test them writing getter-method that is only ever used by the test use reflection to get access to private fields	not practicing test-first approach chasing test coverage poor use of dependency injection	start improving abstraction and separation of concerns by refactoring methods out to another object never compromise encapsulation for tests instead design for testability

# Local Hero

A LEGO Batman minifigure stands next to a cracked egg. The minifigure is wearing a black suit with a yellow bat emblem on the chest. The egg is brown with a large, jagged crack running down its side. A small white piece of debris is visible near the crack.

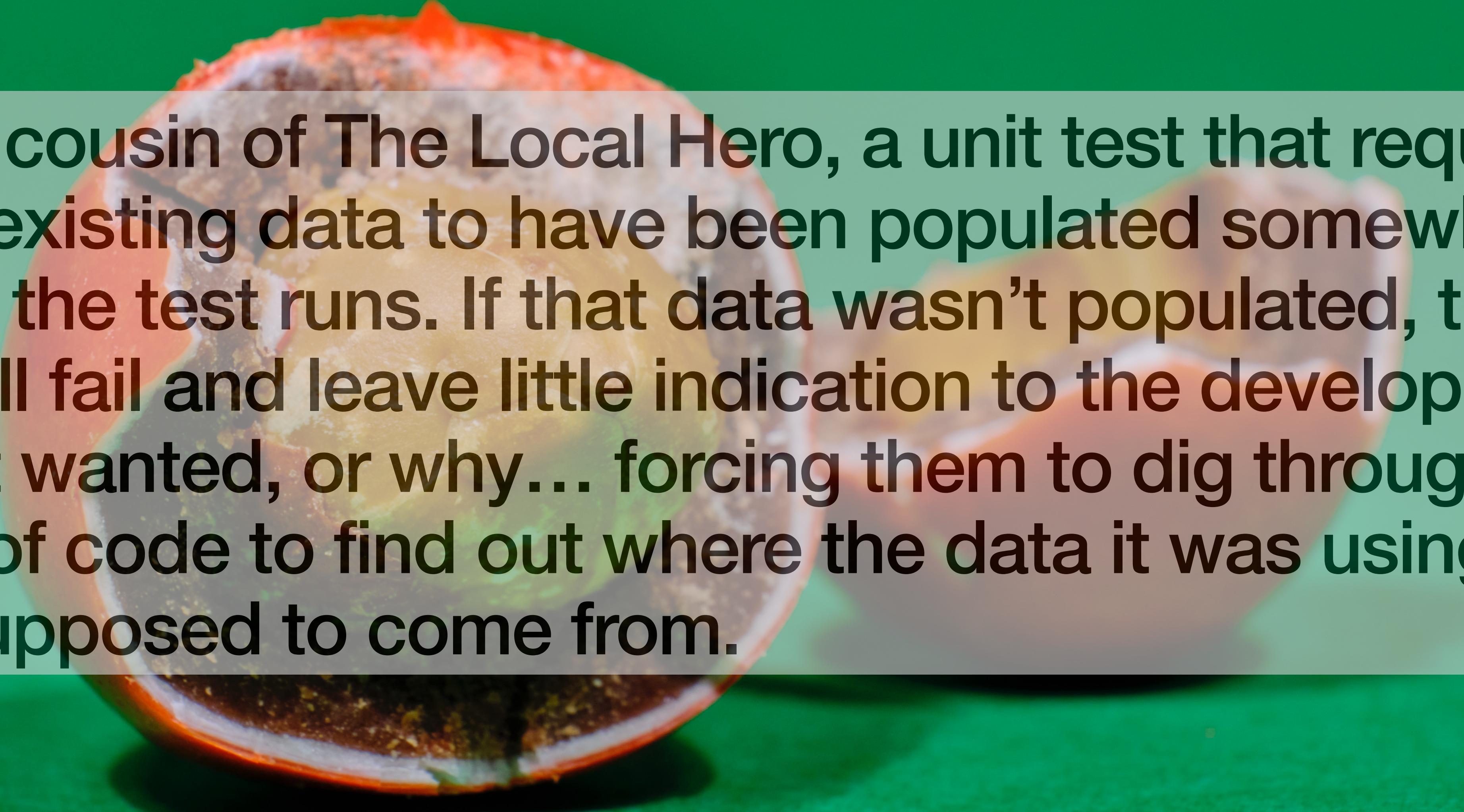
A test case  that is dependent on something specific to the development environment it was written on in order to run. The result is the test passes on development boxes, but fails when someone attempts to run it elsewhere.

# Local Hero



Example	Cause	Solution
using OS specific settings (i.e. line breaks) in tests  relying on some tool installed locally (databases, ...)	being unaware of build on different machines or OS  being unaware of usage of a local tool	for instance: consistently use UTF-8  possibly use tool libraries instead of the tool itself  use In-Memory databases or Testcontainers

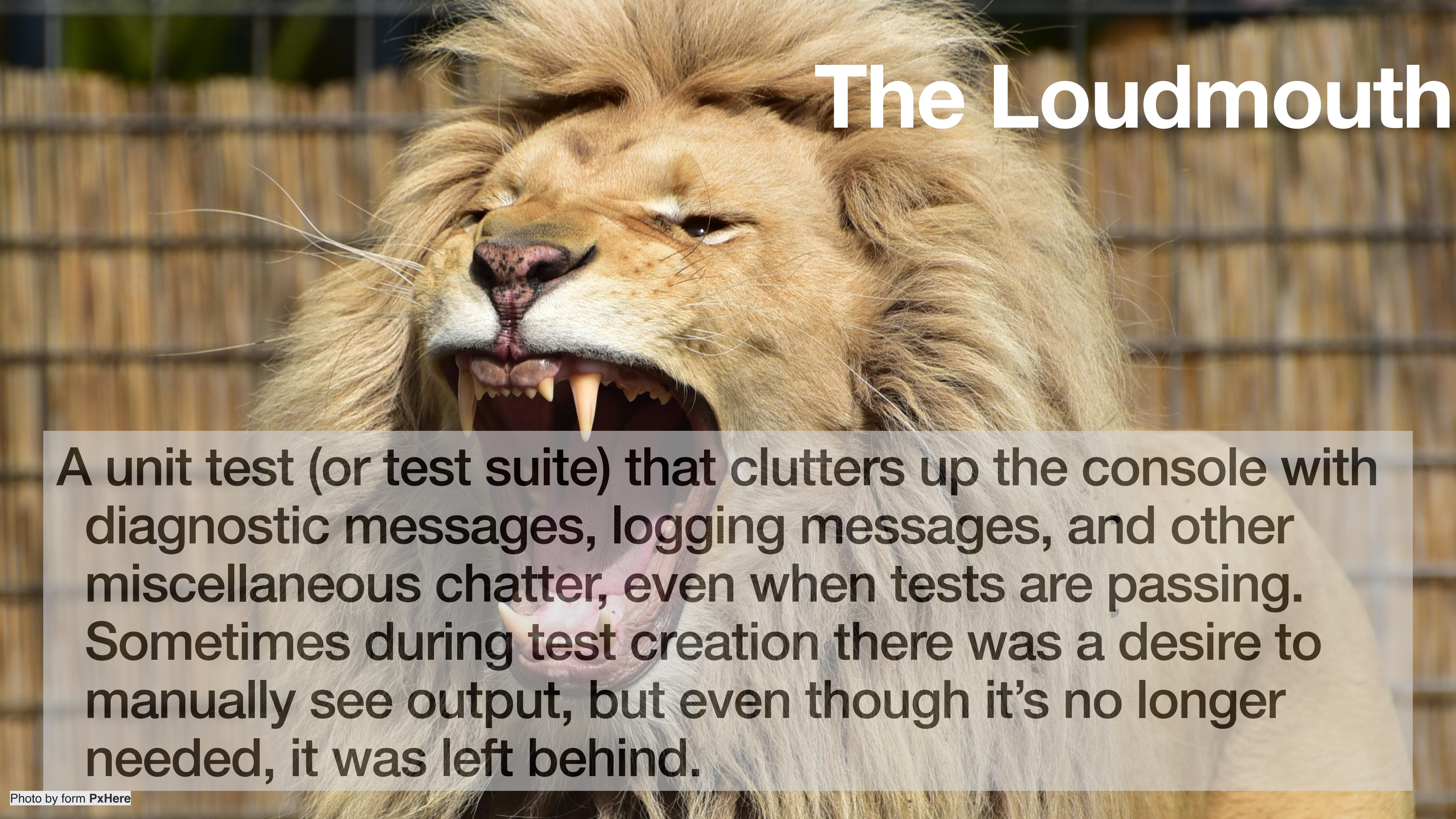
# The Hidden Dependency



A close cousin of The Local Hero, a unit test that requires some existing data to have been populated somewhere before the test runs. If that data wasn't populated, the test will fail and leave little indication to the developer what it wanted, or why... forcing them to dig through acres of code to find out where the data it was using was supposed to come from.

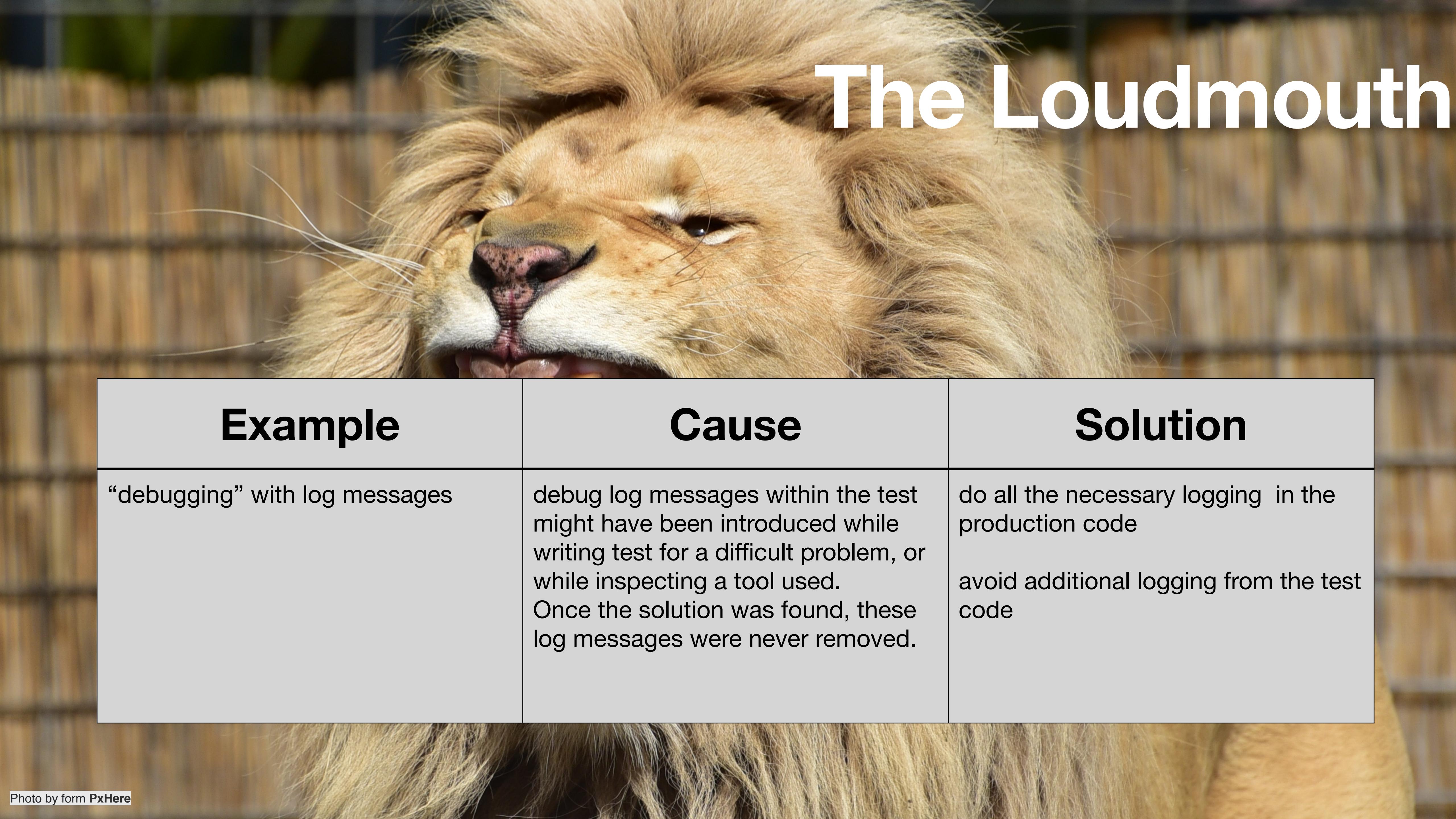
# The Hidden Dependency

Example	Cause	Solution
tests reads from a database that is expected to be filled with data test reads a file that is expected to be present	tests that do not prepare their needed data themselves but rather assume, that certain data are prepared for them	tests should take care of the needed data setup itself



# The Loudmouth

A unit test (or test suite) that clutters up the console with diagnostic messages, logging messages, and other miscellaneous chatter, even when tests are passing. Sometimes during test creation there was a desire to manually see output, but even though it's no longer needed, it was left behind.



# The Loudmouth

Example	Cause	Solution
“debugging” with log messages	debug log messages within the test might have been introduced while writing test for a difficult problem, or while inspecting a tool used. Once the solution was found, these log messages were never removed.	do all the necessary logging in the production code  avoid additional logging from the test code

# The Slow Poke

A unit test that runs incredibly slow. When developers kick it off, they have time to go to the bathroom, grab a smoke, or worse, kick the test off before they go home at the end of the day.



# The Slow Poke

<b>Example</b>	<b>Cause</b>	<b>Solution</b>
testing a time-consuming algorithm with all possible inputs  asynchronous test that waits for an answer	algorithm need lots of CPU-power  in case of asynchronous setup, timeout are too long if another system does not answer	consider using less input data covering the boundaries and one or two happy paths  if making these tests faster is not possible, then run them less often (after careful consideration)

# The Sequencer

A unit test that depends on items in an unordered list appearing in the same order during assertions.

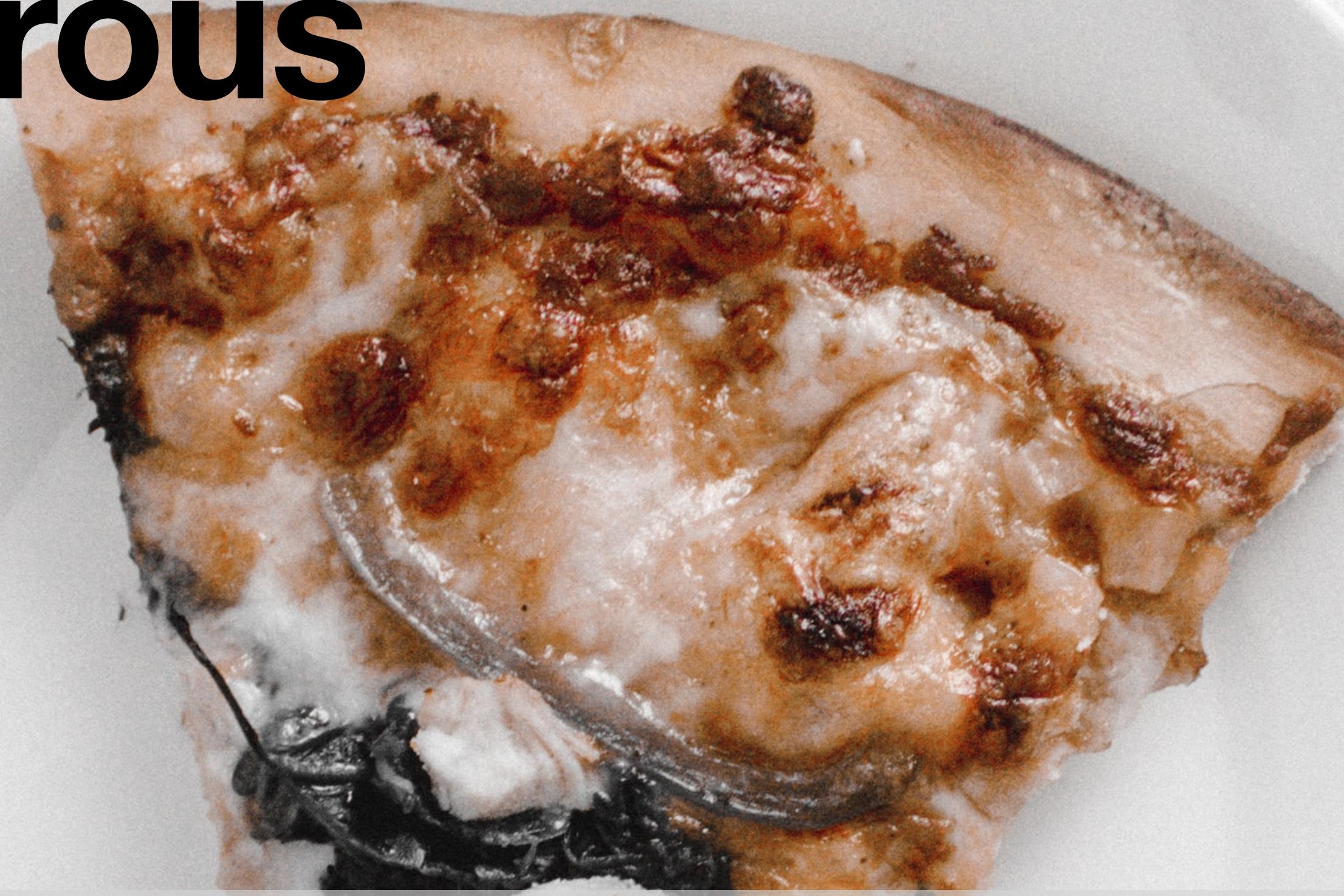


# The Sequencer

Example	Cause	Solution
reading data from a database or from a list (that is not guaranteed to be sequential)	order of items may differ on different machines	make test not depending on the order of inputs or results



# The Generous Leftovers



An instance where one unit test creates data that is persisted somewhere, and another test reuses the data for its own devious purposes. If the “generator” is ran afterward, or not at all, the test using that data will outright fail.

# The Generous Leftovers



Example	Cause	Solution
	unit test framework usually runs tests in random order  flaky tests	design tests so that they never depend on one another or on a certain order to be run

# Other...?





Photo by Johannes Plenio on [Unsplash](#)



Happily Ever After

Starts Here



# Some ressources

- James Carr:  
<https://web.archive.org/web/20100105084725/http://blog.james-carr.org/2006/11/03/tdd-anti-patterns/>
- Dave Farley:  
<https://www.youtube.com/watch?v=UWtEVKVPBQ0>
- Yegor Bugayenko:  
<https://www.yegor256.com/2018/12/11/unit-testing-anti-patterns.html>  
<https://www.youtube.com/watch?v=KiUb6eCGHEY>

# Questions?

# Thank you

Slides:

[https://www.birgitkratz.de/uploads/DWX June 2023 TestAntipattern.pdf](https://www.birgitkratz.de/uploads/DWX_June_2023_TestAntipattern.pdf)

- Email: mail@birgitkratz.de
- Twitter: @bikratz
- Mastodon: [@birgitkratz@jvm.social](https://birgitkratz@jvm.social)
- Github: <https://github.com/bkratz>
- Web: <https://www.birgitkratz.de>

