@TheMERL

Follow

look at this absolute unit.

6:52 AM - 9 Apr 2018

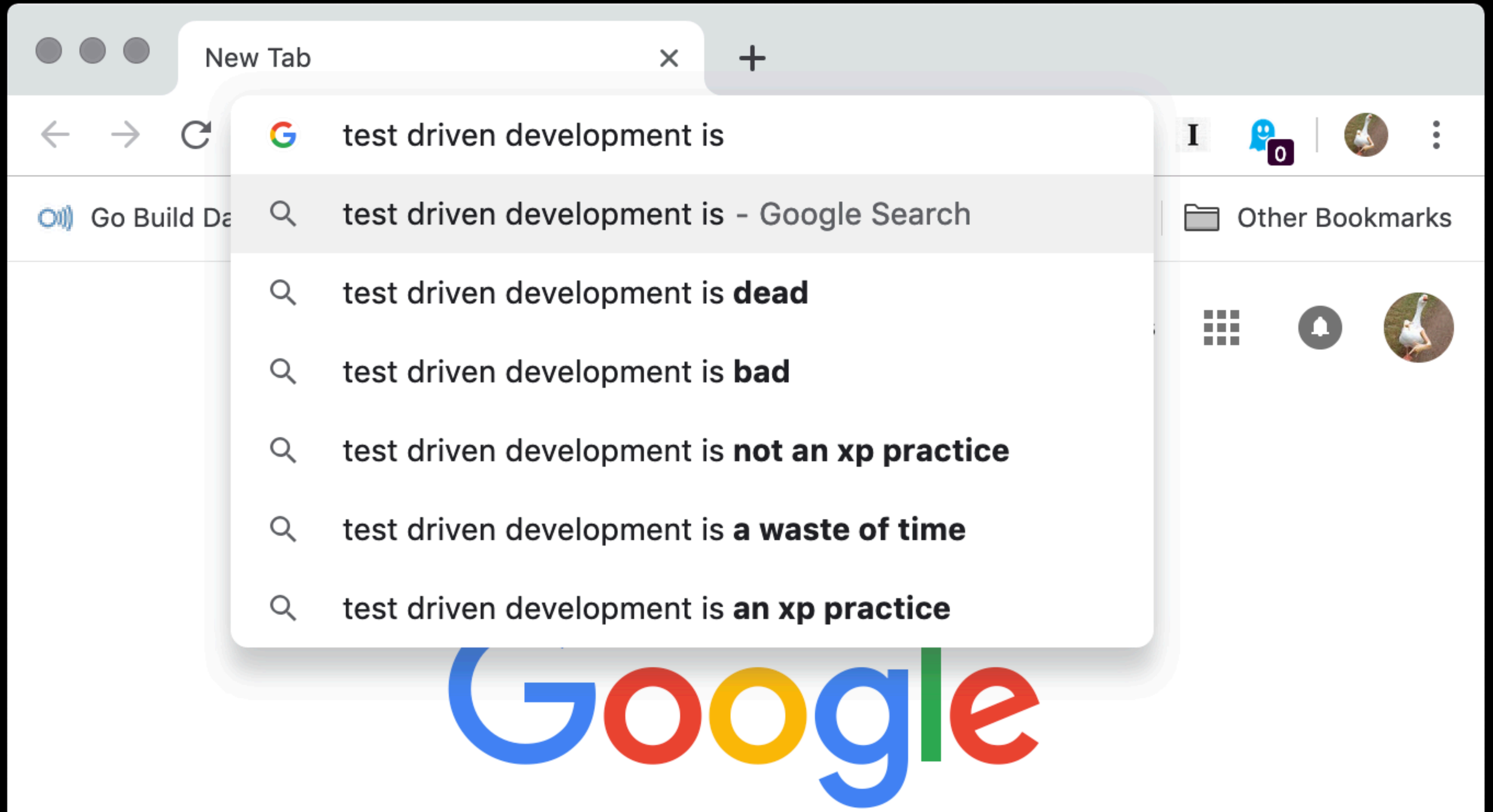
Absolute Unit (test)

Go London User Group

This idea starts with **testing**

Test Driven Development

Test Driven Development


New Tab
test driven development is
test driven development is - Google Search
test driven development is dead
test driven development is bad
test driven development is not an xp practice
test driven development is a waste of time
test driven development is an xp practice
Go Build Da
Other Bookmarks
Google

Whose idea was TDD anyway?

Whose idea was TDD anyway?

Uncle Bob Martin

Whose idea was TDD anyway?

Uncle Bob Martin

Martin Fowler

Whose idea was TDD anyway?

Uncle Bob Martin

Martin Fowler

Kent Beck

Whose idea was TDD anyway?

Uncle Bob Martin

Martin Fowler

Kent Beck

Alan Perlis

“A software system can best be designed if the testing is interlaced with the designing instead of being used after the design.”

–Alan Perlis, NATO Software Engineering Conference, 1968
(source https://www.infoq.com/presentations/1-9-6-8)

I hold it as an article of faith that writing tests **at the same time** as the code is a **good thing.**

TheThreeRulesOfTdd [add child]

The Three Laws of TDD.

Over the years I have come to describe Test Driven Development in terms of three simple rules. They are:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

You must begin by writing a unit test for the functionality that you intend to write. But by rule 2, you can't write very much of that unit test. As soon as the unit test code fails to compile, or fails an assertion, you must stop and write production code. But by rule 3 you can only write the production code that makes the test compile or pass, and no more.

If you think about this you will realize that you simply cannot write very much code at all without compiling and executing something. Indeed, this is really the point. In everything we do, whether writing tests, writing production code, or refactoring, we keep the system executing at all times. The time between running tests is on the order of seconds, or minutes. Even 10 minutes is too long.

Too see this in operation, take a look at The Bowling Game Kata.

My life with TDD

If I knew what I wanted to write, then writing the failing test first was easy.

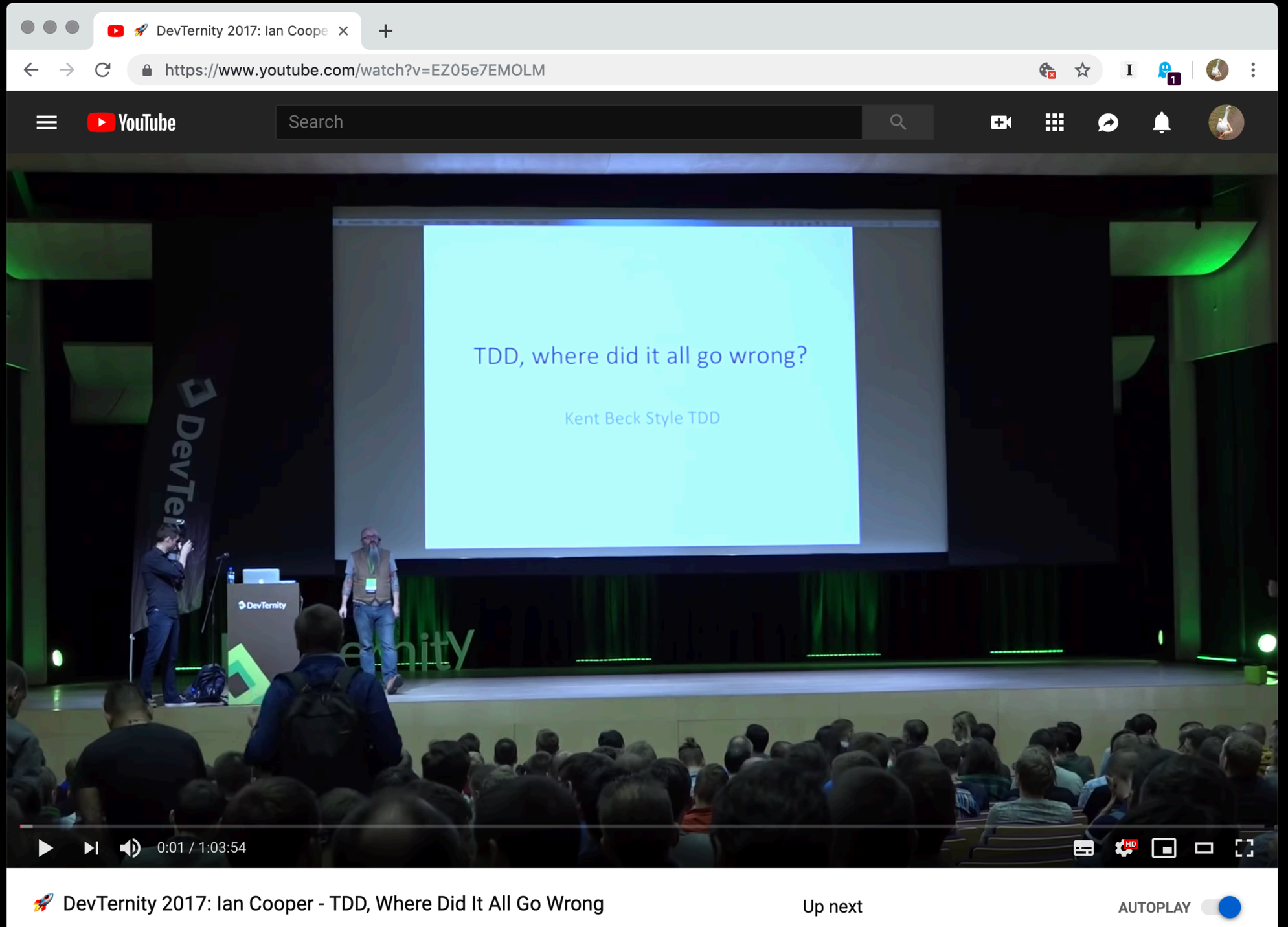
If I *didn't* know what I was writing, the cost of experimentation *started* at 200%; change the code, fix the test.

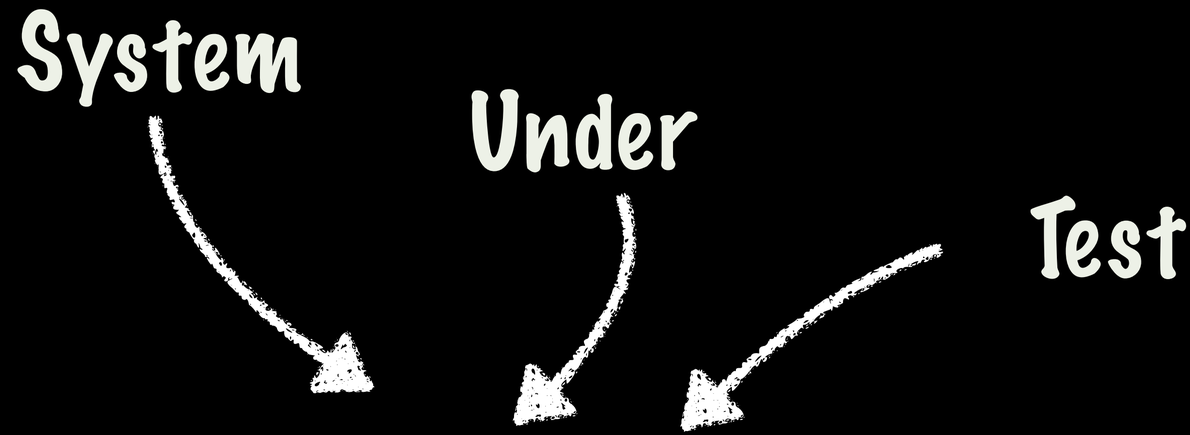
This cost went up and to the right when one function called another.

Changing well covered code felt safe, changing the tests felt like cheating


THE DEVELOPERS LOOK UNHAPPY.
ADD LESS TESTS.


THE DEVELOPERS LOOK UNHAPPY.
ADD LESS TESTS.


DevTernity 2017: Ian Coope
https://www.youtube.com/watch?v=EZ05e7EMOLM
YouTube
Search
TDD, where did it all go wrong?
Kent Beck Style TDD
DevTernity
0:01 / 1:03:54
DevTernity 2017: Ian Cooper - TDD, Where Did It All Go Wrong
Up next
AUTOPLAY



The SUT is not the class

What is the unit of code in a C program?

What is the unit of code in a Java program?

What is the unit of code in a Go program?

“Go packages embody the spirit of the UNIX philosophy.
Go packages interact with one another via interfaces.
Programs are composed, just like the UNIX shell, by
combining packages together.”

–Me, a few years ago

The unit of software in Go is the package

Test the **behaviour** of your unit, not its **implementation**

The public API of a package declares *this is **what** I do*, not *this is **how** I do it*

Behaviour

The public API of a package declares *this is **what** I do*, not *this is **how** I do it*

Behaviour

The public API of a package declares *this is what I do*, not *this is how I do it*

Implementation

The unit is *not* the function

TDD doctrine states that you shall not write a line of production code until you have a failing unit test.

The result is every helper or private function inside your package has a test, and those tests break all the time when you refactor.

This wasted work discourages refactoring. There is a risk that the tests will end up fitting the observed behaviour of the code under test, not asserting the expected behaviour.

The unit of code is the package

In Go the unit of code is the package. You only need to test the behaviour of your package that can be observed.

Code coverage is your guide. If there are branches that cannot be covered via the public API, delete that code.

When you refactor, use coverage to tell you where you need to add tests. Beware Hyrum's Law.

If you have high coverage, consider adding fuzz testing to check that you've covered all the edge cases.

Let's make this concrete

The product I've been working on for the last 14 months is effectively a translator from k8s objects to gRPC objects.

Before I applied these ideas, I would have to write out the translations twice; once in code, once for the test fixtures.

Whenever the logic would change, I'd have to change the test. Whenever the gRPC definitions would change, I'd have to change both the business logic *and* the test.

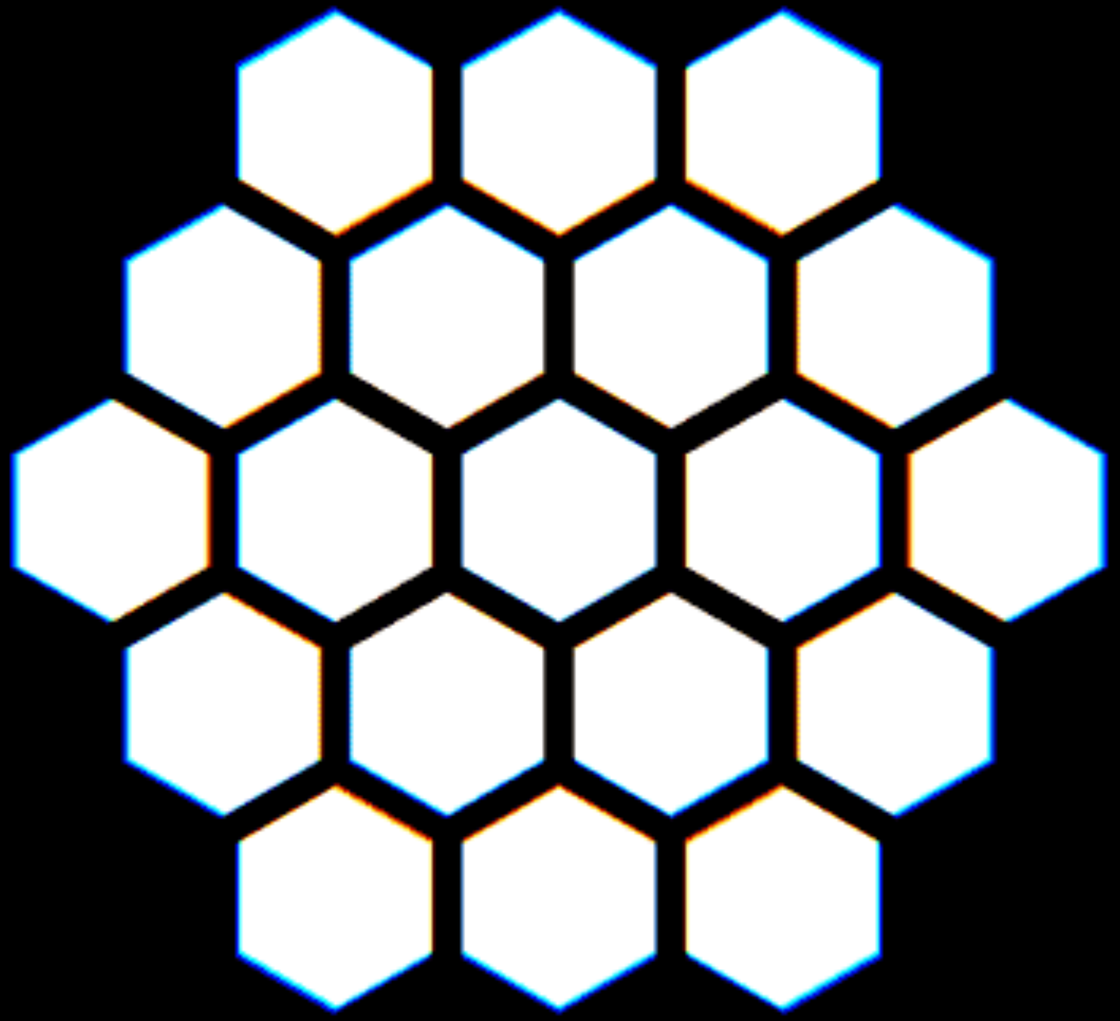
Refactoring with **impunity**

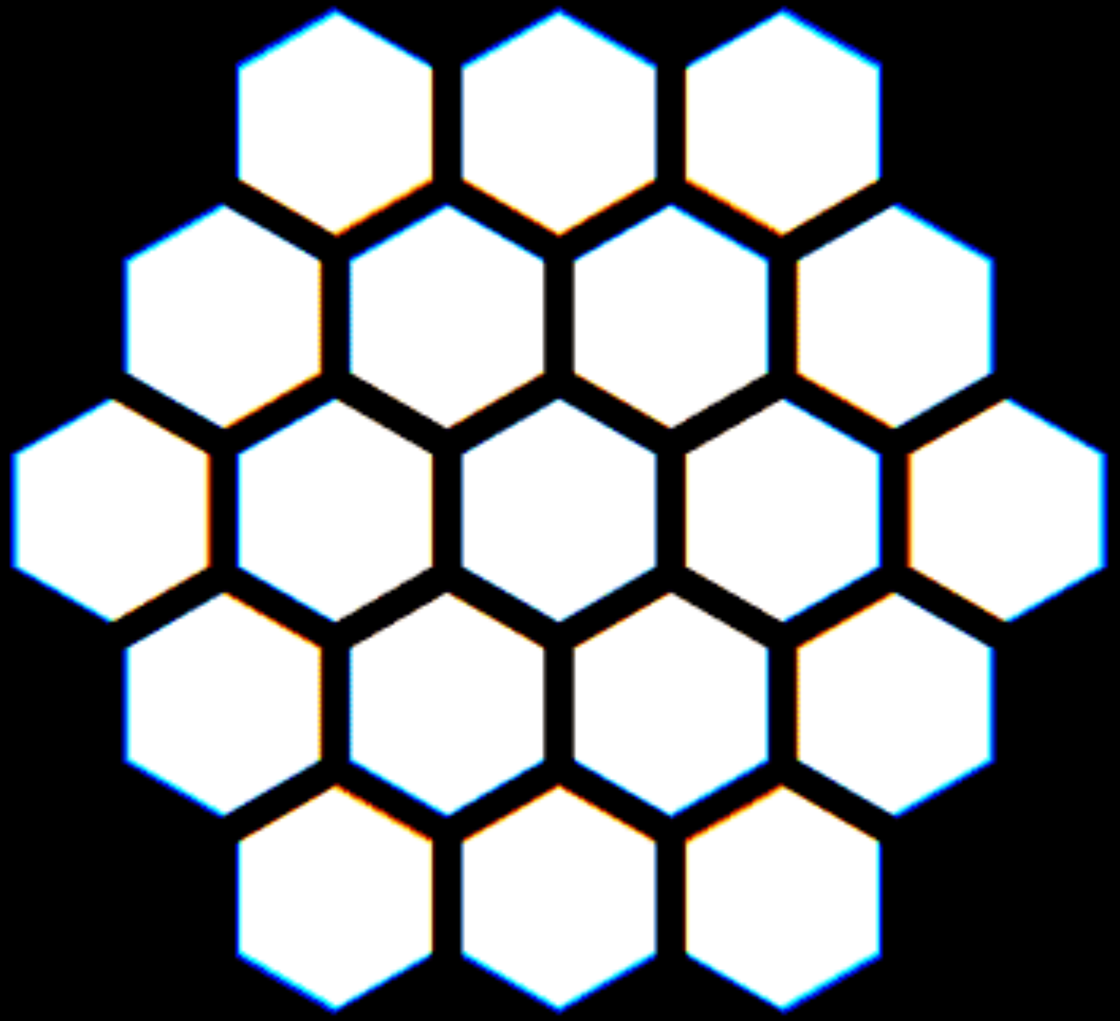
I moved all the gRPC translation logic to its own package, tested in isolation from the k8s mechanics.

Only test public functions of that package (almost everything is now a public function)

Business logic and the gRPC generation now separate, we can use the gRPC in both business logic *and* tests.

Changes to the gRPC definitions were contained to one package.



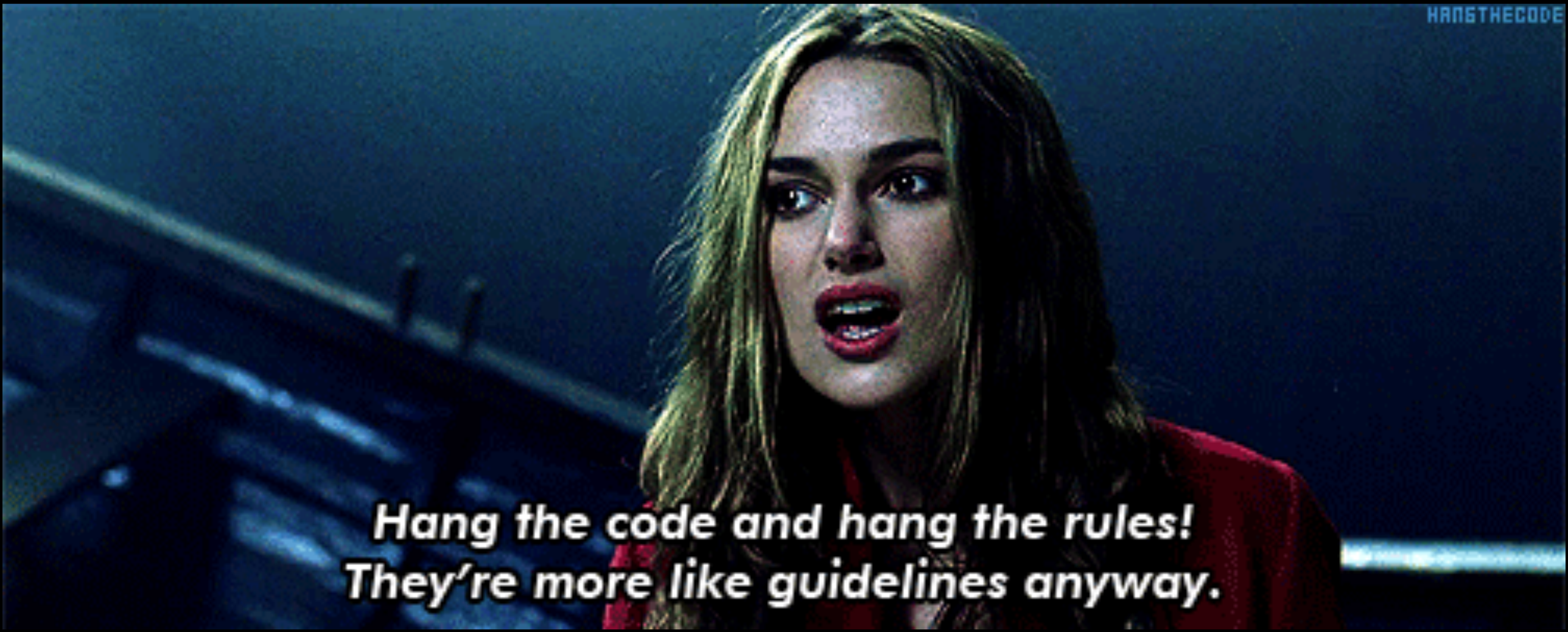


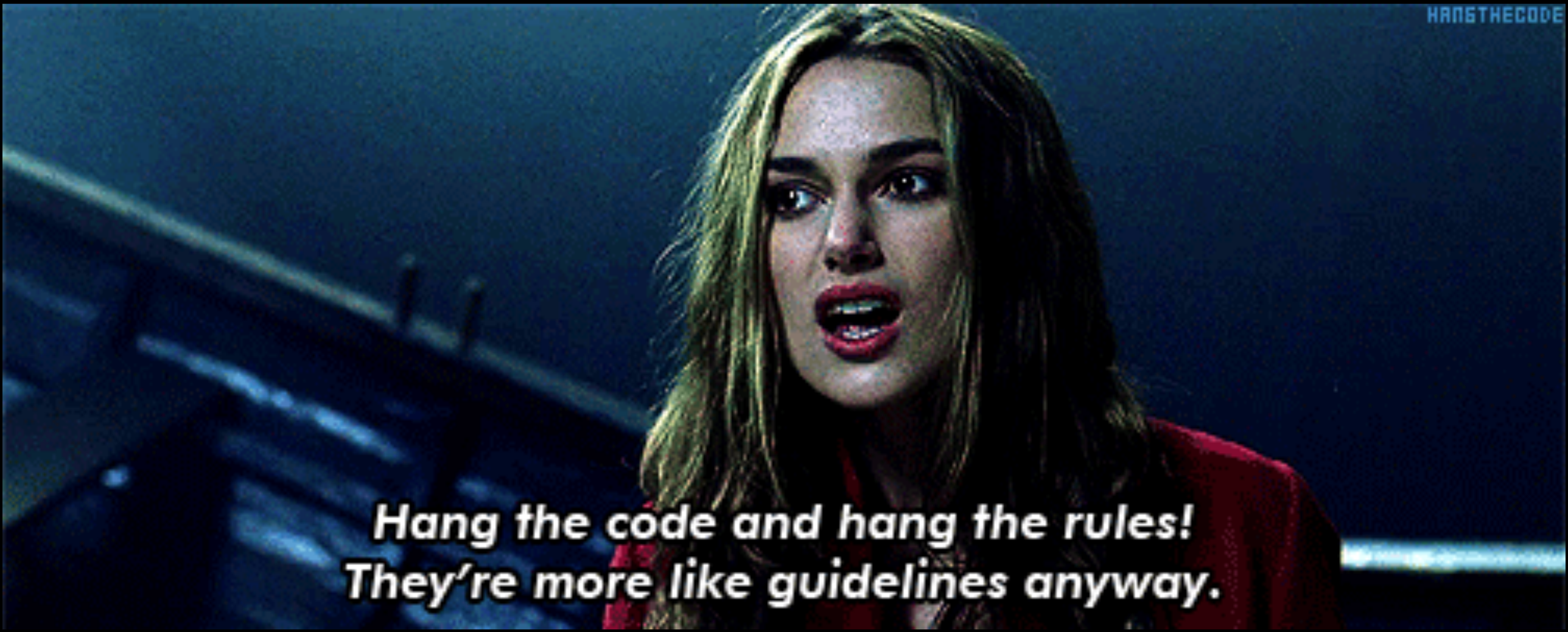
♥ David Crawshaw liked

Matt Klein
@mattklein123

I find myself increasingly doing "assert driven development." I write code and add assert(false) in interesting logic spots, and then write tests that hit them (followed by more code/asserts) until they are all gone. I find this process is fast and yields excellent coverage.

7:03 AM · Feb 23, 2019 · Twitter Web App


HANGTHECODE
Hang the code and hang the rules!
They're more like guidelines anyway.


HANGTHECODE
Hang the code and hang the rules!
They're more like guidelines anyway.

Guidelines, not rules

Guidelines, not rules

You should write tests.

Guidelines, not rules

You should write tests.

You should write tests at the same time as you write your code.

Guidelines, not rules

You should write tests.

You should write tests at the same time as you write your code.

Each Go package is a self contained unit.

Guidelines, not rules

You should write tests.

You should write tests at the same time as you write your code.

Each Go package is a self contained unit.

Your tests should assert the observable behaviour of your package, not its implementation.

Guidelines, not rules

You should write tests.

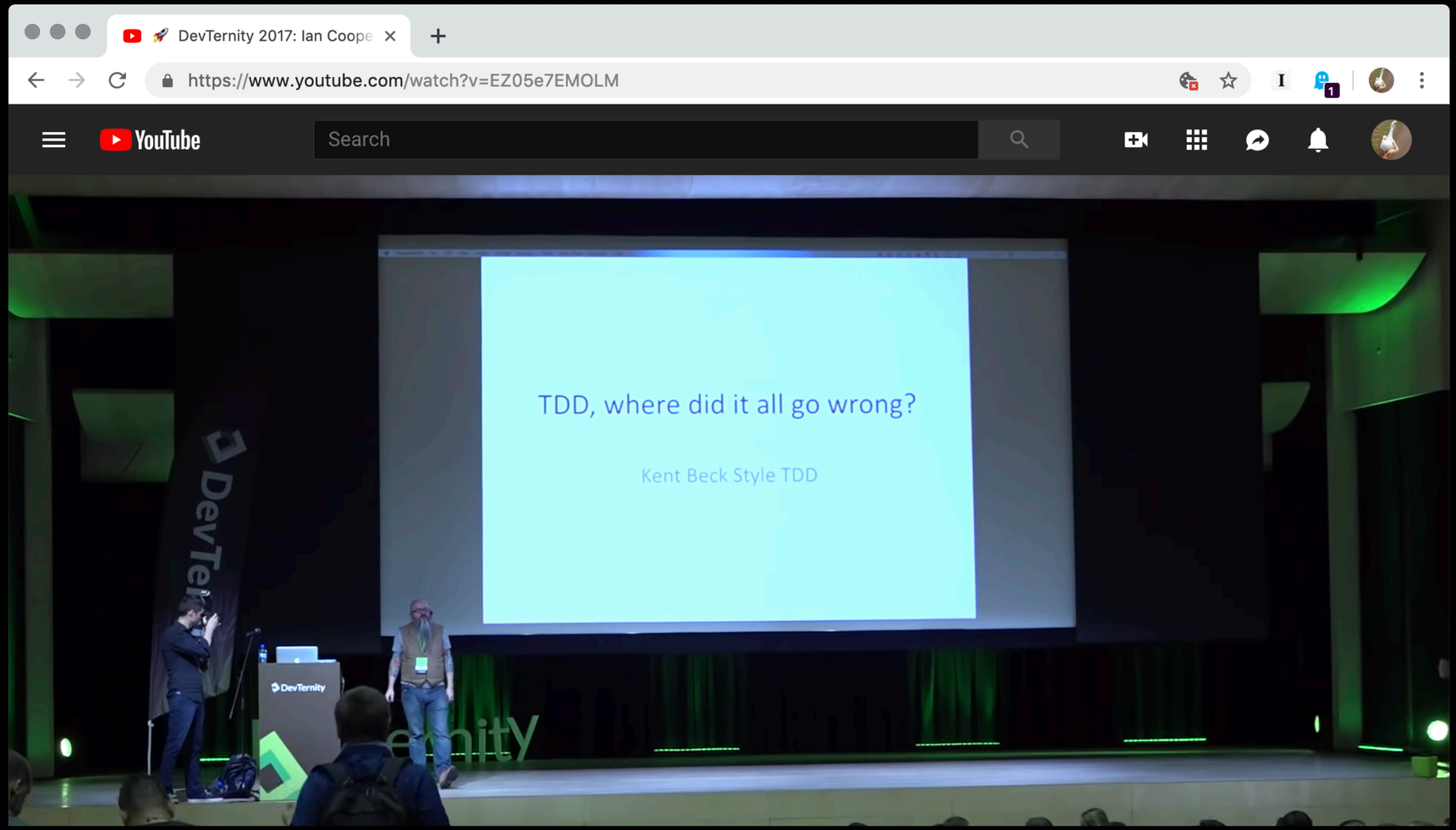
You should write tests at the same time as you write your code.

Each Go package is a self contained unit.

Your tests should assert the observable behaviour of your package, not its implementation.

You should design your packages around their behaviour, not their implementation.

Don’t be a stooge, watch this talk.



https://youtu.be/EZ05e7EMOLM