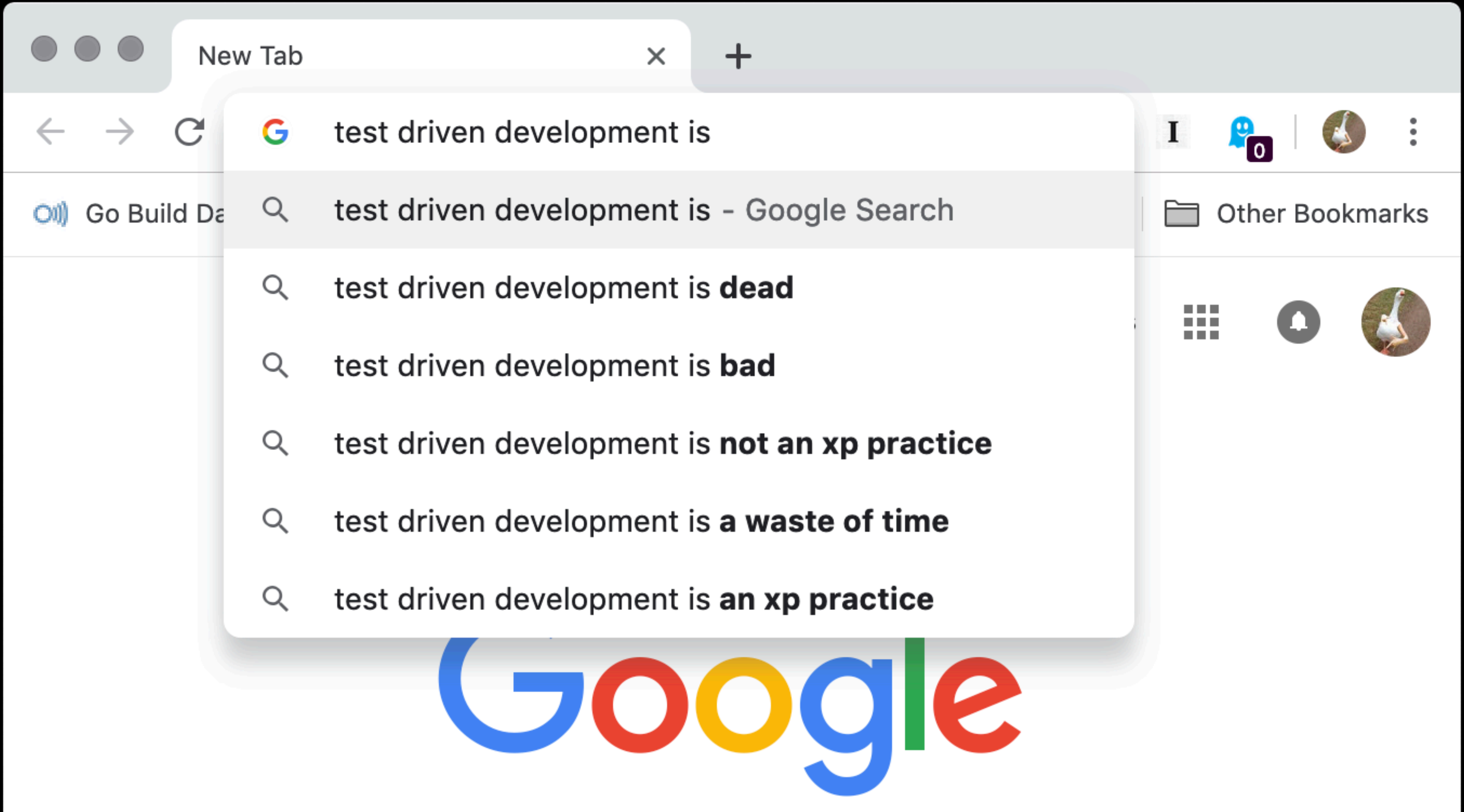
Absolute Unit (test)

Go Sydney Users' Group

This idea starts with testing

Test Driven Development

00
00:



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**

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

ArticleS.UncleBob.TheThreeRu
Not Secure
butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
but UncleBob .com
Edit
Properties
Refactor
Where Used
Search
Files
Versions
Recent Changes
User Guide

ArticleS. UncleBob.

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 productio code, or refactoring, we keep the system executing at all times. The time between running tests is on the order o seconds, or minutes. Even 10 minutes is too long.

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

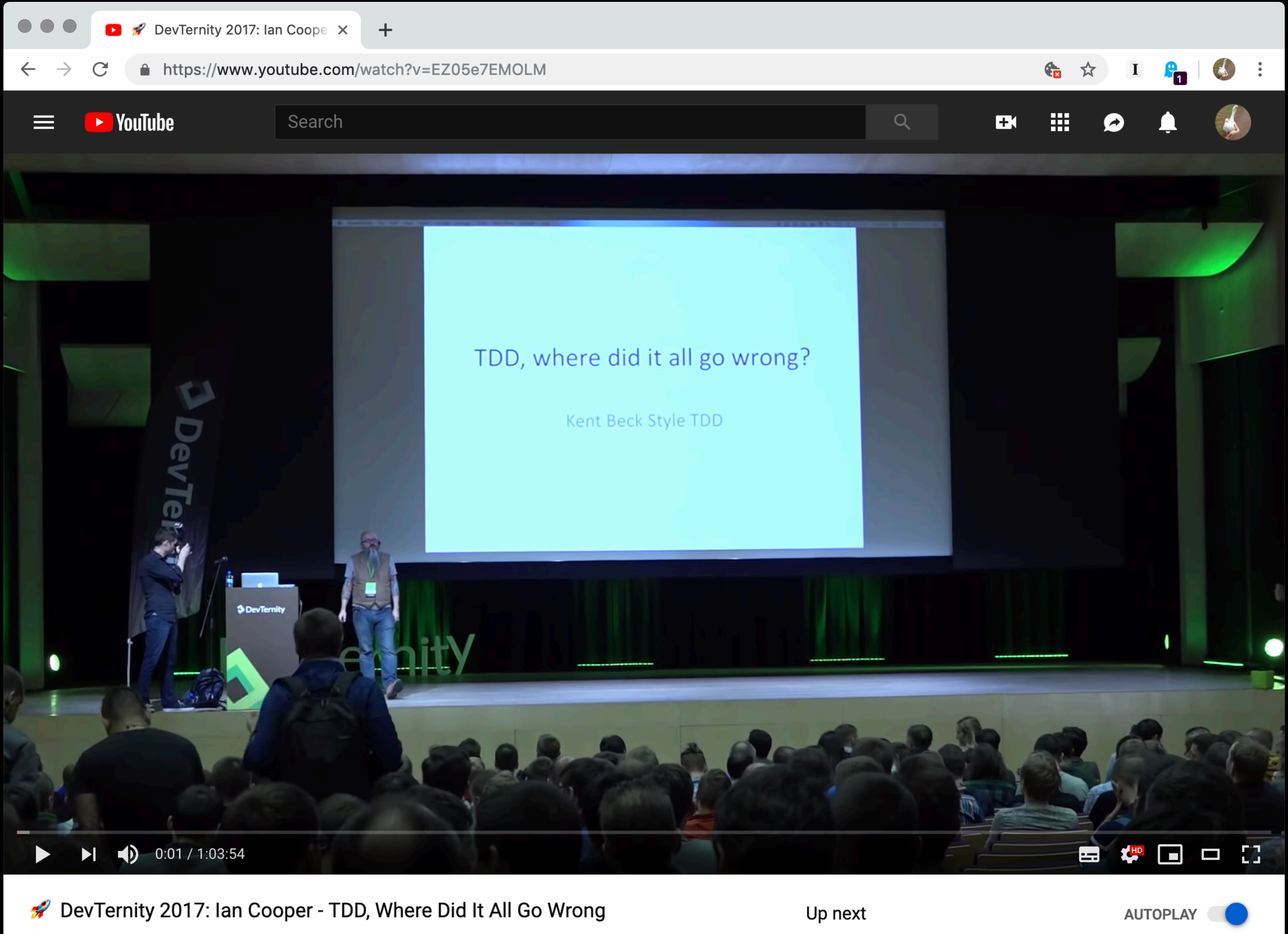
My life with TDD

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 feels safe, changing the tests feels like cheating



DevTernity 2017: Ian Coope

https://www.youtube.com/watch?v=EZ05e7EMOLM

TDD, where did it all go wrong?

Kent Beck Style TDD

0:01 / 1:03:54

DevTernity 2017: Ian Cooper - TDD, Where Did It All Go Wrong

System

Under

Test

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 Go software is the package

Test behaviour, not implementation

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.

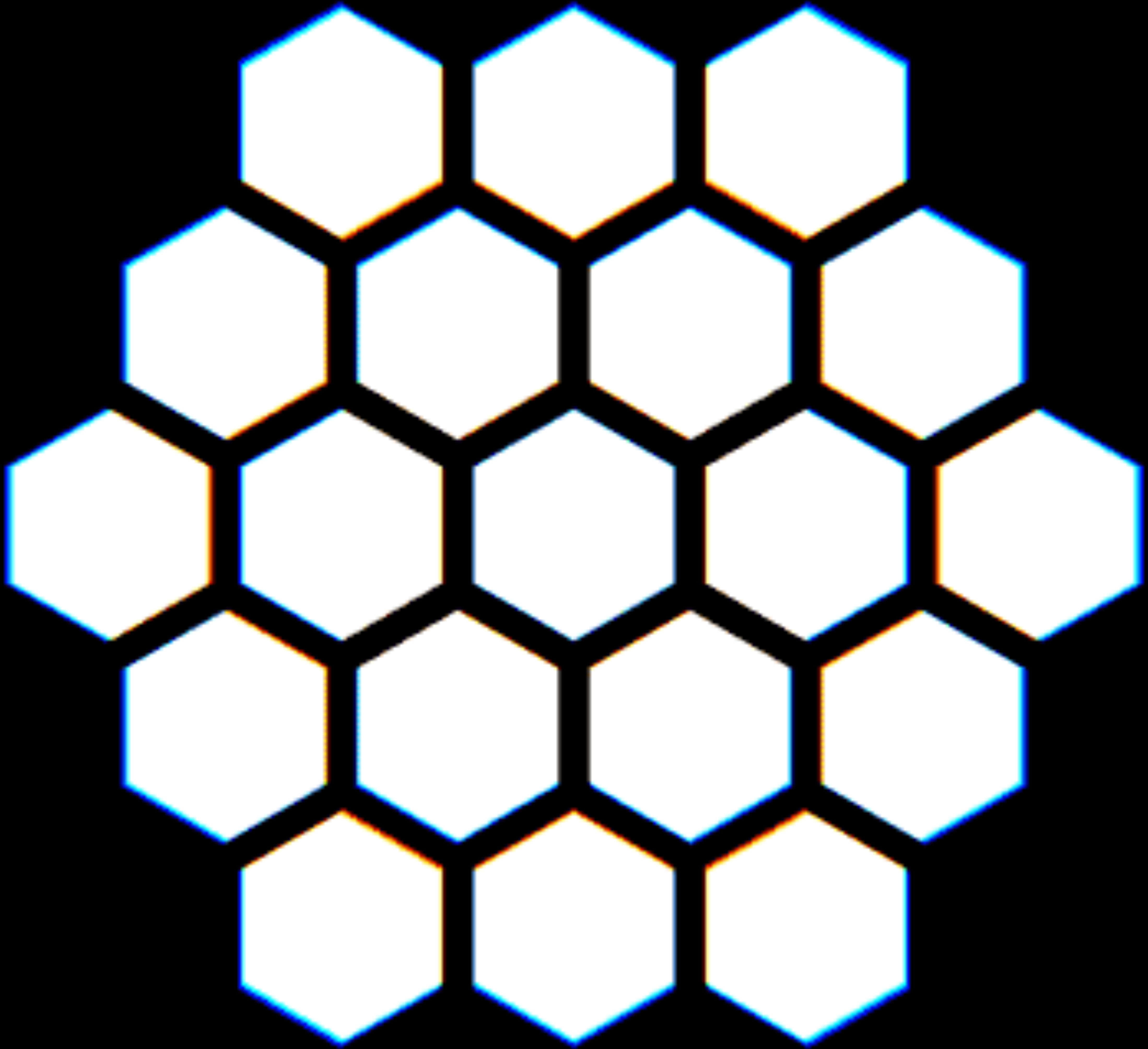
Refactor with confidence

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.



In a nutshell

Each Go package is a self contained unit.

Test your package's behaviour, not their implementation.

More importantly, design your packages around their behaviour, not their 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

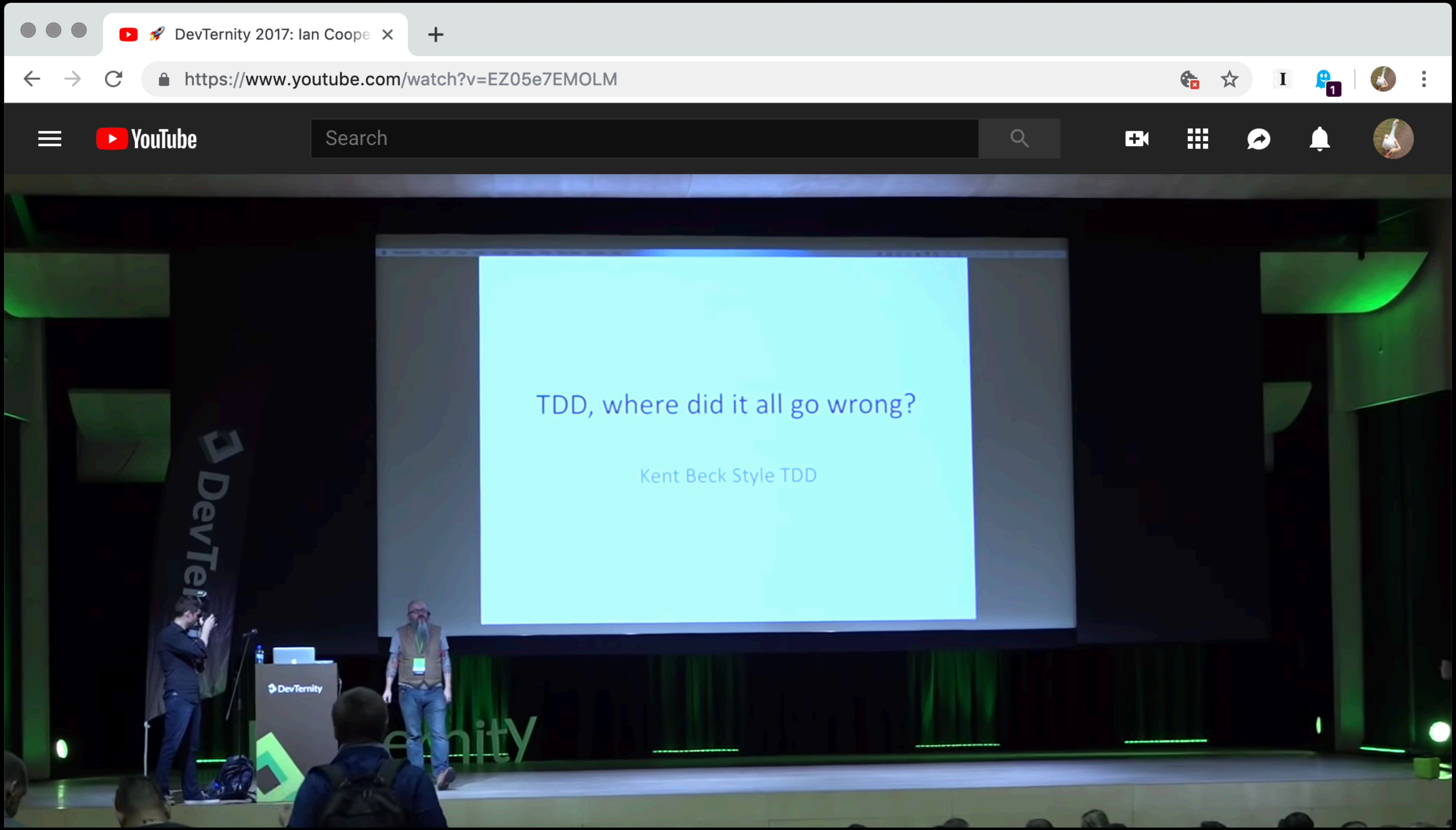
In Go the unit of code is the package. You only need to test the behaviour of your package that can be observed.

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.

Don 'T BE A STOOGE, WATCH THIS TALK.


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

https://youtu.be/EZ05e7EMOLM