



[Refactoring](#) [Agile](#) [Architecture](#) [About](#) [Thoughtworks](#)  

On the Diverse And Fantastical Shapes of Testing

Pyramids, honeycombs, trophies, and the meaning of unit testing



Martin Fowler

 [TEST CATEGORIES](#)


02 June 2021


There's been a recent resurgence on twitter and the like about how teams should divide up their testing efforts. In particular, Tim Bray [argues compellingly](#) in favor of taking automated testing seriously. Anyone familiar with my writing will know that I'm very much in agreement with him.

One of the points he raises in his post refers to this couple of images:

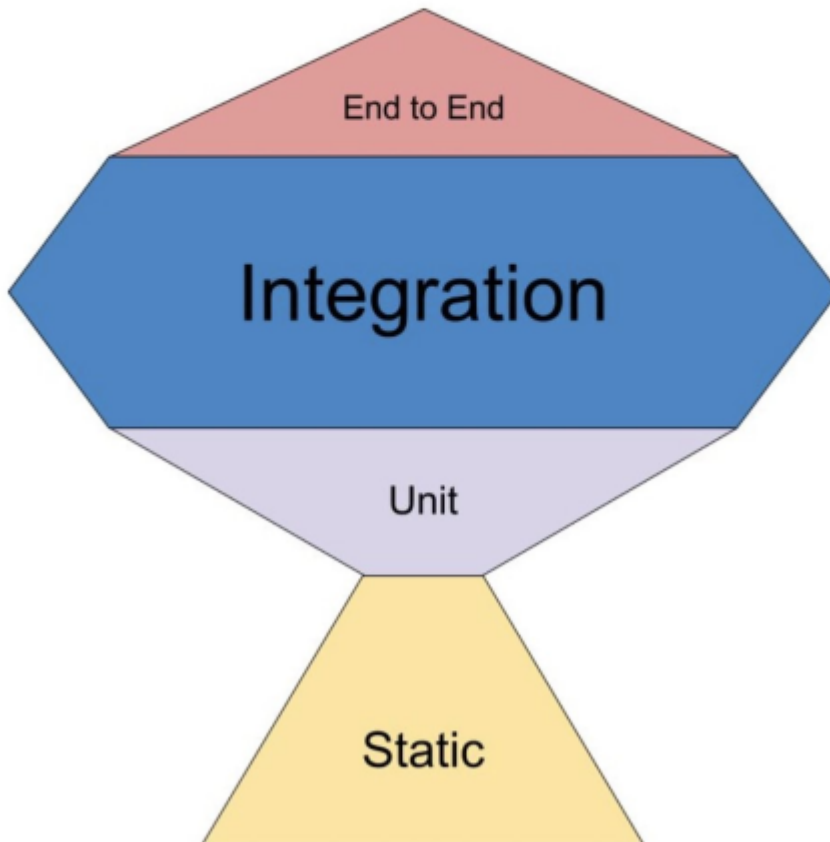
 **Kent C. Dodds** 
@kentcdodds



"The Testing Trophy" 

A general guide for the ****return on investment****  of the different forms of testing with regards to testing JavaScript applications.

- End to end w/ @Cypress_io 
- Integration & Unit w/ @fbjest 
- Static w/ @flowtype F and @geteslint 



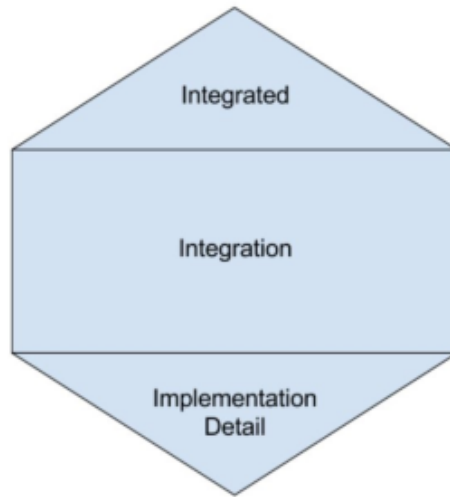
556 11:53 AM - Feb 6, 2018



164 people are talking about this

Microservices test strategy

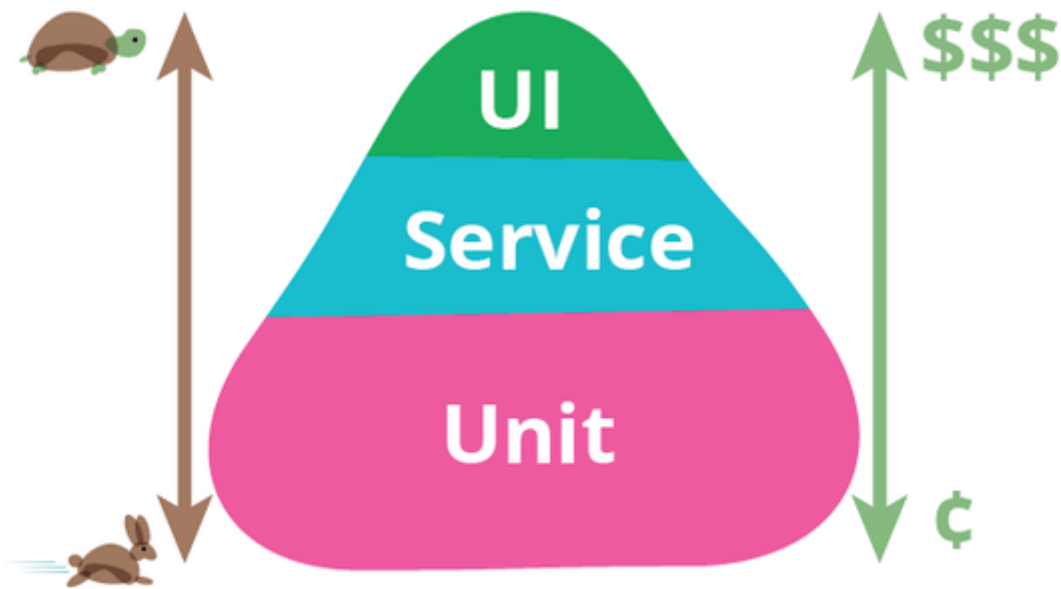
A more fitting way of structuring our tests for Microservices would be the Testing Honeycomb.



Microservices Testing Honeycomb

That means we should focus on Integration Tests, have a few Implementation Detail Tests and even fewer Integrated Tests (ideally none).

Both of these "misshapen blobs" are a reaction to the older image of the Test Pyramid:



The point of these images is to indicate the amount of effort we should expend on various types of tests, in particular the balance between unit

and broader tests. The pyramid argues that you should have most testing done as unit tests, the honeycomb and trophy instead say you should have a relatively small amount of unit tests and focus mostly on integration tests.

The second biggest issue I have with this discussion is that it's rendered opaque by the fact that it's not clear what people see as the difference between unit and integration tests.

The terms “unit test” and “integration test” have always been rather murky, even by the slippery standards of most software terminology. As I originally understood it, they were primarily an organizational issue. Let's go back to the days of large waterfall software projects. I'm working on a hunk of code for several months. I may be working on it alone, or in a small team. Either way I think of this hunk as a conceptual unit which we can work on in relative separation from its neighbors. Once we've finished coding it we can hand it off to the unit testing team, who then test that unit on its own. After a month or two to make those tests work, we can then integrate it with its neighbors and carry out integration tests against a larger part of the system, or indeed the entire system. The key distinction is that the unit tests test my/our code in isolation while integration tests how our code works with code developed separately.

Many people today ran into unit tests as part of the Xunit family of testing tools, pioneered by Kent Beck as part of Extreme Programming. Kent used “unit test” to indicate tests written by developers as part of their day-to-day work.

Programmers write unit tests so that their confidence in the operation of the program can become part of the program itself. Customers write functional tests so that their confidence in the operation of the program can become part of the program too.

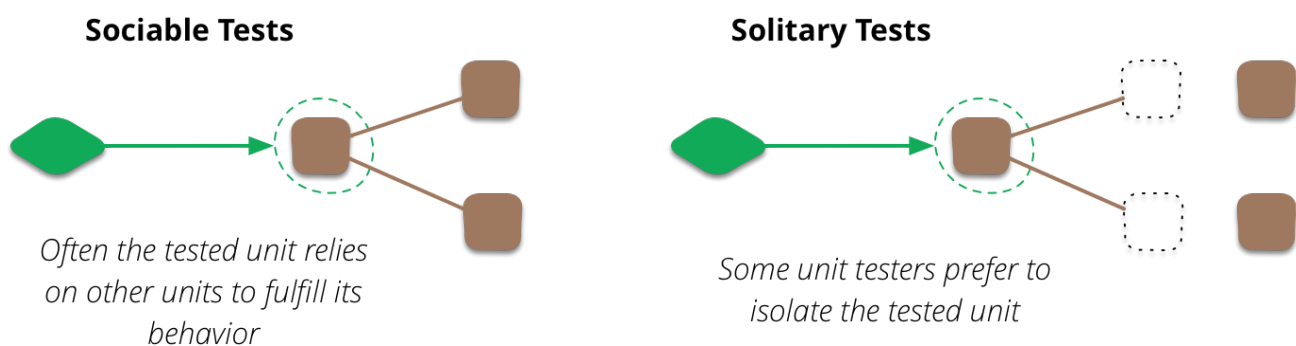
-- Kent Beck (Extreme Programming Explained, 1st Edition)

Notice that in Kent's original formulation, “unit test” means anything written by the programmers as opposed to a separate testing team. At C3 when we wrote a unit test, we'd usually focus on a single class's behavior. But we'd set up a test fixture that created that object with all the necessary dependencies so it could execute its methods. Those other objects would execute too, but we'd assume for this test that all the other code was working correctly (and usually the other code had its own tests).

I remember considerable discussion over this use of “unit test”. One test expert vigorously lambasted Kent for this usage. We asked him how he would define unit test, and his reply was something like “in the first morning of my training course I cover 24 different definitions of unit test”.

From the early days of XP-inspired unit testing there were those who disliked the term “unit test” and proposed using names like “microtest” or “programmer test” instead.

For many, the biggest issue in this was these dependent objects. If I'm testing an order object, and it collaborates with a customer object, then my test of the order object could fail due to a bug in the customer object. This led to a different style of writing unit tests, where any collaborating object would be replaced with a mock, stub or other kind of Test Double. I've since found it useful to describe these styles of Unit Test as sociable and solitary. [1]



Intertwined with this distinction are the growth of two schools of XP unit testing practice, which I call classic and mockist. Classic XP unit testing

follows the sociable approach we originally used, while mockist style favors solitary tests.

So, going back to pyramids versus honeycombs, when I read advocates of honeycomb and similar shapes, I usually hear them criticize the excessive use of mocks and talk about the various problems that leads to. From this I infer that their definition of “unit test” is specifically what I would call a solitary unit test. Similarly their notion of integration test sounds very much like what I would call a sociable unit test. This makes the pyramid versus honeycomb discussion moot, since any descriptions I've heard of the test pyramid consider unit tests to be sociable and/or solitary.

This semantic picture is made even muddier by the definition of Integration Test, which makes “unit test” look tightly defined. The take-away here is when anyone starts talking about various testing categories, dig deeper on what they mean by their words, as they probably don't use them the same way as the last person you read did.

If you're paying my careful prose its properly due attention, you'll notice that I said earlier that this lack of clarity about unit and integration tests is the second biggest issue I have with the honeycomb/pyramid discussion. My biggest issue is well-summed-up by this tweet.

Justin Searls 

@searls



People love debating what percentage of which type of tests to write, but it's a distraction. Nearly zero teams write expressive tests that establish clear boundaries, run quickly & reliably, and only fail for useful reasons. Focus on that instead.

swyx @swyx



Footnotes

1: Jay Fields came up with the terms "solitary" and "sociable"



© Martin Fowler | [Privacy Policy](#) | [Disclosures](#)