SUBJECT7
QUALITY IS A CULTURE

Rethinking the Testing Pyramid

By Matthew Heuser for Subject7

Early in my career I read an incredible book that filled me with hope about software. The book is called *Peopleware*, it is thirty-three years old and still in print. A highlight of my career has been working in a small way with Tim Lister, one of the co-authors. The other co-author is Tom Demarco, who even earlier, back in 1982, wrote *Controlling Software Projects: Management, Measurement, and Estimation*, where he penned the infamous line "You can't control what you can't measure." That quote spawned a generation of metrics consultants who ran around, measuring what was easy to measure, often to the detriment of the productivity or quality people actually wanted.

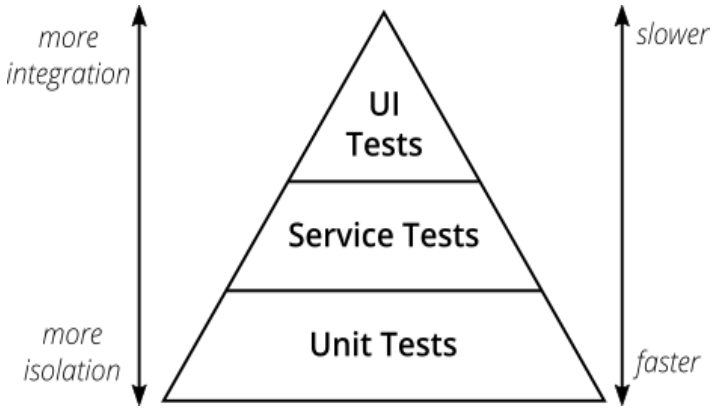
Half a career later, DeMarco published a follow-up where he asked "I'm wondering, was its advice correct at the time, is it still relevant, and do I still believe that metrics are a must for any successful software development effort?" His answers were no, no, and no. In one sentence, Demarco argued that the most promising projects were harder to measure, and the ones that could be controlled offered less return on investment. If this world only had more Tom DeMarcos, more people willing to admit their mistakes, or when their context has changed, we would all be so much better off.

Today, I want to take a look at the Test Automation Pyramid in much the same way. Was it correct at the time, is it still relevant today, and do I believe that all projects should be structured with mostly unit tests, some integration or system tests, and a few end-to-end GUI tests at the top?

Let's talk about it.

The Automation Pyramid in Context

Mike Cohn popularized the pyramid in his 2009 book, *Succeeding with Agile*. Here's a part of what he wrote: "Even before the ascendancy of Agile Methodologies like Scrum, we knew we should automate our tests. But we didn't. Automated tests were considered expensive to write and were often written in months, or in some cases years, after a feature had been programmed. One reason teams found it difficult to write tests sooner was because they were automating *at the wrong level.*"



The assumption here is that automated unit tests are cheap, easy, fast to run and isolated, compared to those slow, brittle, hard to write end-to-end tests that require a full working system and a web

browser or mobile device. Cohn proposed the foundation of a test effort should be unit tests, with fewer service tests and very few end-to-end tests, creating a bit of a pyramid.

Source: The "The Practical Test Pyramid" from Martin Fowler's Blog

The pyramid had other advantages. The high quality of individual pieces developed through unit testing should result in a system that is of higher quality overall. A high-quality overall system would have fewer defects out of the gate, fewer regressions, and need less testing. Personally, I've found Cohn's logic, originally developed with peers like Lisa Crispin in 2003, to be both beautiful and compelling. They helped push an entire industry away from test automation snake oil and toward a more balanced approach.

They also came before micro services, before viable test tooling existed, before the iPhone or modern smartphone, before the cloud, containers, or virtualization as we know it. When the pyramid was created, Microsoft Visual Source Safe was probably the most popular version control system; the *extreme programming* people were talking about having a physical "build machine" to copy files to. End-to-end automation ran on a desktop computer over a lunch break, not on eight virtual browsers in ten minutes.

I dare suggest that the system forces may have changed a little.

The Testing Picture Today

The first thing I'd like to point out is the slope of that pyramid. You'll note it is not a pie chart, with a percentage of time suggested for each activity. For that matter, even if it were a pie chart, it is unclear what people would measure. Would you measure the effort in time, number of tests, or something else? Toward the customer acceptance side, it is unclear how much testing would be "enough," or if some scenarios require more end-to-end tests. For example, most domains have *core paths*, such as path-to-purchase in eCommerce. Those *core paths* might benefit much more from end-to-end tests running on every build than "customer reviews," "your recent views," or "buy again." For the latter examples, a single test per major flow might be sufficient. The idea of "less" end to end tests doesn't provide much guidance on these tough decisions.

Speaking of simple quick tests, modern infrastructure gives us a new, powerful choice: test in production. This could involve creating complex test accounts with fake credit card numbers that go to nowhere, but it can start as easily as doing everything other than clicking the final checkout button. Tests that do this can run on repeat in production all the time. If an outage or bad deploy breaks a major feature, your team will be the first to know. These sorts of tests can also provide real-time performance information: how long it is really taking a browser to load a real web page in production. When customers call and complain the site is "slow," you'll have an objective measure of how long it was taking when they called, at say 4:00PM Eastern, instead of someone "unable to reproduce" the problem because it's now three hours later.

Starting with existing end-to-end tools, then pivoting to run in production is not quite free, but probably close. That kind of "two for one deal" was not really in mind when Cohn created the pyramid.

Then there are the tools themselves.

In 2003, if you wanted to test a website with a tool, you were probably testing on a Windows application ("Win32") using record/playback. Every feature you added would slow things down by another five minutes. Eventually the tests would take hours to run, and you would give up. The arguments above for a "narrow point" for a few powerful end-to-end tests make sense.

Today, the tools are better. You can create them in a browser using *Software as a Service*. For the most part, modern Adaptive tools visualize the workflow, making it understandable by both the programmer and the analyst. The scenario flows are also easier to change when the underlying code changes the expectations.

Meanwhile, on the programmer side, the Is Test Driven Development (TDD) Dead? debate has been striving to find a middle ground for unit tests between "test drive every line of code" and nothing. Experts like Kent Back suggest writing fewer, more powerful unit tests that demonstrate the important outcomes of a function as you write the function, instead of a dogmatic test/code/refactor loop. That means less unit tests and more end-to-end tests. Meanwhile integration tests, especially for Microservices, are largely context-dependent. A company like Amazon.com that runs services as the architecture is going to need a huge number of service-level tests. Organizations that are not Amazon and have a different architecture may have less or no need for such tests. As it turns out, like the pirates code in the *Pirates of the Caribbean*, the test automation pyramid is more of a guideline to be filled in by context -- and the context is changing.

The Bottom Line

In a different era of computing, the guidance was to focus less on end-to-end tests than on unit tests. When you consider that in 2003, the typical team was doing *zero* automated unit tests, the shift in emphasis feels more like a mandate. I would argue that Brett Pettichord's "Hey Vendors, Give Us Real Scripting Languages" which came out around that time captured the spirit of the age. The tools of that time were slow, designed to be used on a single device, and had their own custom programming languages, often with bugs in the scripting languages themselves. The proponents of the test pyramid did not see them as viable, and instead suggested getting the business logic out of the user interface. If all the user interface did was pass messages and display the result of the message, then true end-to-end testing was a bit redundant.

Almost twenty years later, I have to say, that vision was never quite realized. Most web and mobile applications I work with today still have validation and transformation logic in the front-end, which is code, and that code should be tested.

Today, adaptive tools can create tests in the cloud which are multi-user-simultaneous-edit. Continuous Integration can do setup and kick off the test run. Thanks to the power of running them in the cloud in parallel, a "thirty minute" test suite might take something like five minutes to run.

These ideas aren't new, exactly. When Sean McMillan and I presented at the Google Test Automation Conference, (About 24 minutes in) we suggested testing larger pieces of the application and doing more end-to-end testing.

The pyramid was wonderful for its time. The top tools were not good, and the systems were not stable without a solid base. Today, some of the emerging tools are radically better.

It may be time to shift the pendulum back.