Extracted from:

The Way of the Web Tester

A Beginner's Guide to Automating Tests

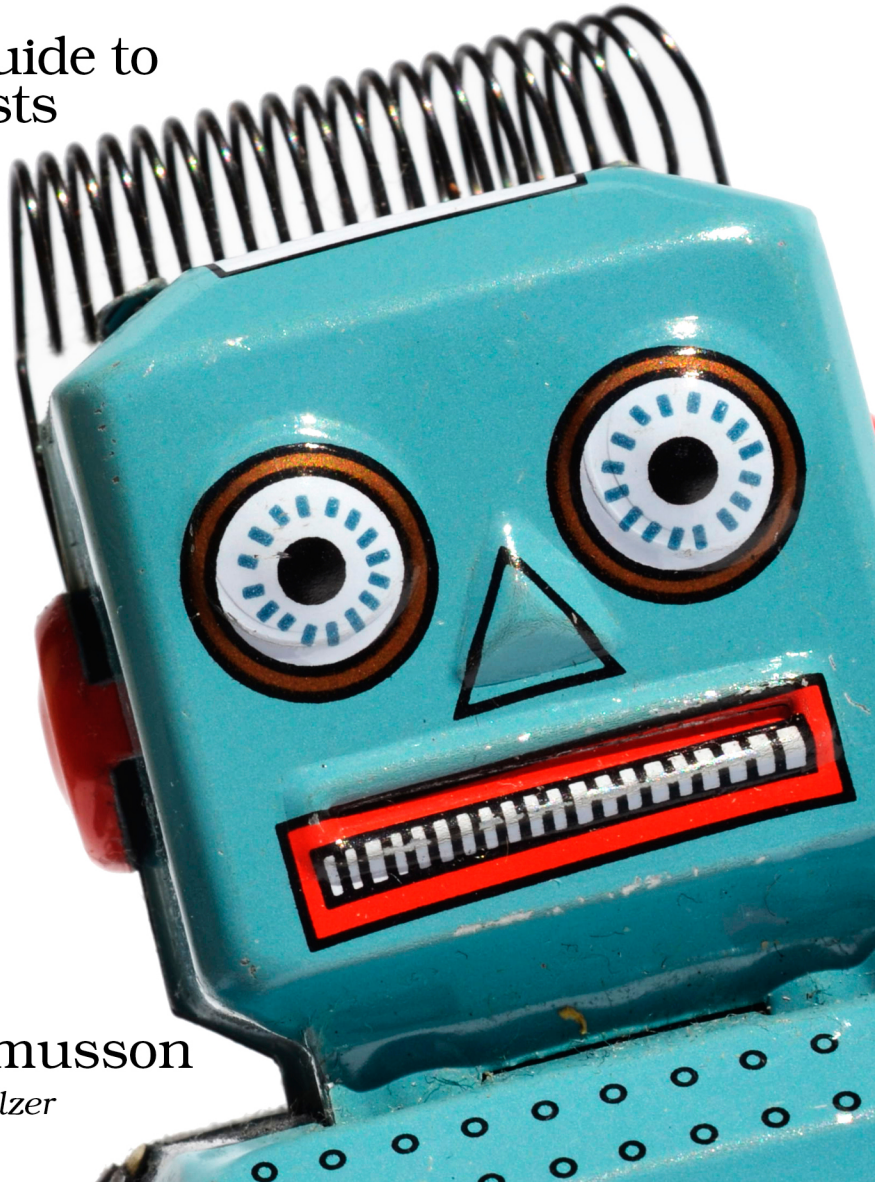
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

The Way of the Web Tester

A Beginner's Guide to Automating Tests



Jonathan Rasmusson

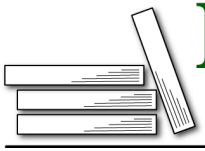
edited by Susannah Pfalzer

The Way of the Web Tester

A Beginner's Guide to Automating Tests

Jonathan Rasmusson

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The yellow adhesive note graphic in Chapter 11 is designed by Layerace from Freepik.com.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (index)
Nicole Abramowitz (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

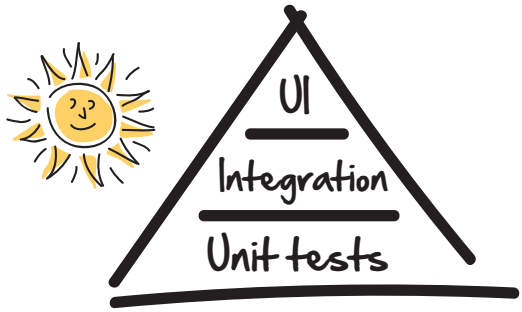
For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-183-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2016

Enter the Testing Pyramid

The testing pyramid, first coined by Mike Cohn in *Succeeding with Agile [Coh09]*, is a model that teams use to show how three different kinds of tests complement each other.



At the top of the pyramid, we've got these things called *user interface* or *UI tests*. These tests go end-to-end through the entire system and act just like a user would if they were using the system. We'll cover UI tests in Chapter 2, *Smoking User Interface Tests*, on page ?.

Then we've got *integration tests*. These are like UI tests, except they don't go through the user interface. They instead go one layer beneath and directly test the underlying services that make our user interfaces go. We cover these in Chapter 4, *Connecting the Dots with Integration Tests*, on page ?.

Then at the base we've got these things called *unit tests*: small, fast, precise code-level tests developers write to tell instantly when things are broken. These come later, in [xxx](#unit).

Chapter Ordering

Now when it comes to exploring the pyramid, we're going to start at the top with the UI tests and then work our way down to the bottom. We're going to do this for three reasons:

1. Quick wins.

   UI tests are the easiest of the three types of tests to get going with, and scoring some quick wins will put some wind in our sails and make tackling the subsequent chapters easier.

2. We need some basics.

The chapter on JavaScript won't make sense until you understand a few mechanics about how HTML and CSS work. So we are going to cover those first in Chapter 2, *Smoking User Interface Tests*, on page ?.

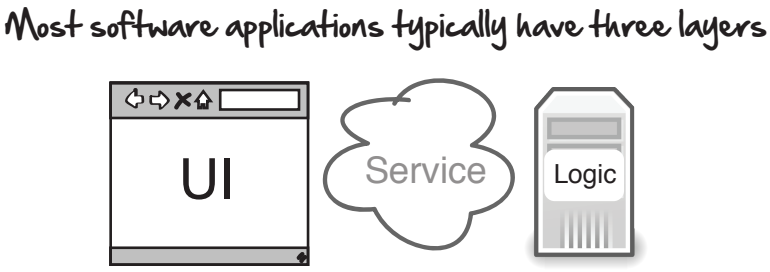
3. Sticky learning.

   Over the course of the book, I am going to occasionally lead you down some garden paths and show you how some things seem great, only to then show you where they fail. This will give you a better feel for what each type of test can do, along with where their limits lie.

So don't think of the chapter ordering as showing the levels of importance—most teams start with unit tests first. But we are starting at the top to aid with learning, which will hopefully make the material more sticky and fun along the way.

Three Levels

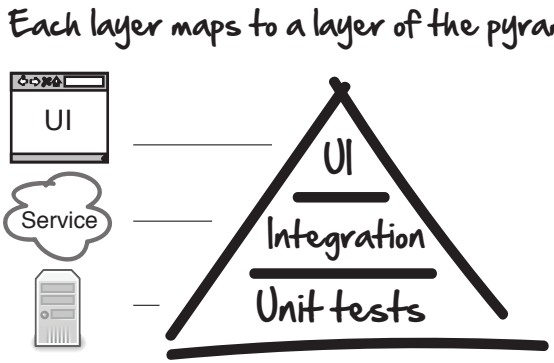
The testing pyramid makes more sense once you understand that most web software architectures are made up of three distinct layers.



There's a UI layer, which contains the buttons and controls your customers use when using your application. There's the *service layer*, which feeds your UI layer the data it needs to update its displays. And then there is the *logic layer*, which contains the math, calculations, and brains of the operation.

Now of course not every application is built this way. Some have business logic built into the service layer. Some applications don't have any UI. These differences don't usually matter. The fundamentals of the pyramid still tend to hold.

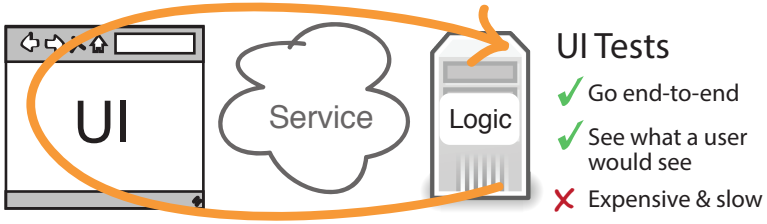
What matters is understanding that each layer of these applications maps to a specific level in our pyramid, and that each level has a certain kind of test.



Let's take a quick look at each of these layers now.

UI Tests

The user interface tests test the application from the UI layer down.



This is what makes UI tests so desirable. They cut through all the layers of the architecture and ensure everything is hooked up. That's what we mean when we say UI tests go *end-to-end*.

The downside to this end-to-end awesomeness is speed and fragility. UI tests tend to be slow and fragile. UI tests don't have to be brittle (we'll look at some ways to make them more robust in Chapter 2, *Smoking User Interface Tests*, on page ?). But there's no getting around the fact that they are slow—orders of magnitude slower than unit tests. So they are not the greatest for giving rapid feedback. This is why UI tests sit at the top of the pyramid and tend to be used more sparingly on projects.

Integration Tests

Integration tests, on the other hand, don't go through the UI. They start one layer down and test the underlying services. This gives them the advantage of not having to deal with the fragility of the UI, while still retaining some of the ability to check that things are properly hooked up and connected.

The only downside to integration tests is that they aren't very precise. By precise, I mean that while they are great at telling you something is broken, they can't always tell you exactly where.

So we like integration tests, because they are great at testing connectivity, but we still don't use them for everything because they can't always tell us exactly where our problems lie.

Unit Tests

For precision, speed, and coverage, we rely on unit tests. Unit tests are the granddaddy of all automated tests. Developers started writing these things years ago with the rise of agile methods like extreme programming,[1] and they have become a staple in modern programming languages and platforms.

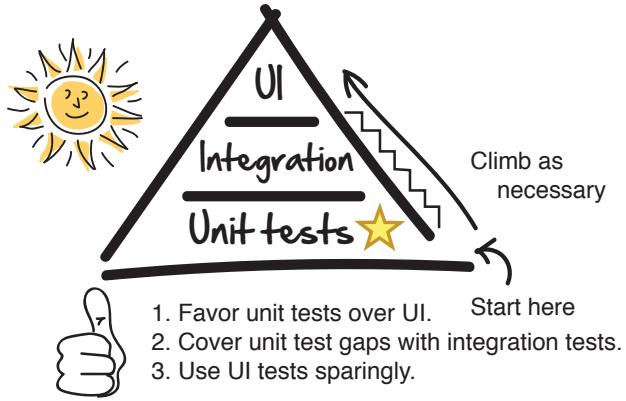
They are extremely quick and very precise. And when things break, they tell us exactly where things went wrong. They are essential for rapid iterative development, and without these, we would be flying blind.

The only downside to all that speed and precision is integration. Sometimes unit tests miss things. Certain bugs only appear when we hook things up. This is why integration tests are still so valuable. And why developers will typically write both when testing their systems.

When we bring all these tests together, some rules of thumb start to form.

1. http://www.agilenutshell.com/xp

Rules of Thumb



The pyramid takes its shape from experience, which has taught us that it is better to do the bulk of our automated testing down near the bottom, where the tests are fast and cheap, than at the top where they are slow and expensive.

Not all projects have or need end-to-end UI-style tests. Some get by with just unit and integration.

That's why whenever we go and add new tests to the system, we always start at the bottom first, and work our way up from there.

When adding a new test, always see if you can cover it with a unit test first.

Now if you're a tester, this is hard advice to follow because you won't be automating things near the bottom. You will instead be working with the higher-level tests closer to the top. So the flip side of this for you is to:

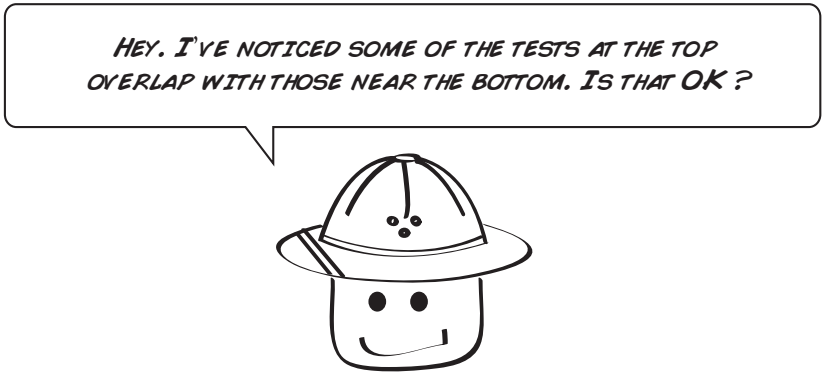
Always push tests as far down the pyramid as you can.

That means if you can handle a given test case with an integration test, that's favorable to trying to automate everything up in the UI.

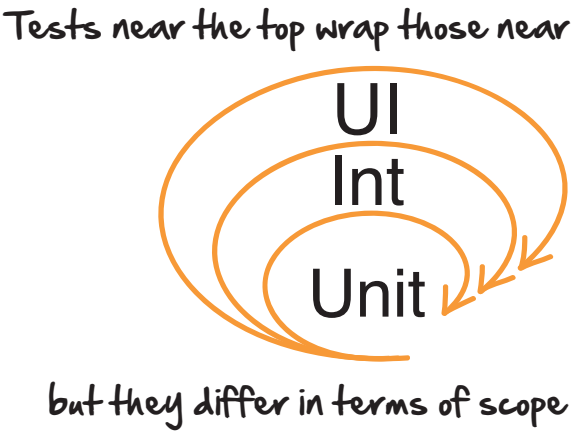
And this final rule of thumb takes a moment to say but a lifetime to master:

Don't try to automate everything. Instead automate just enough.

As wonderful as automated tests are, every test has a price in terms of cost and maintenance. So we don't want to automate everything. Instead we want to automate just enough. Easy to say—hard to do. We will explore this Zen-like principle more as we get further into the book.



Some overlap of tests in terms of functionality is inevitable, because tests near the top are always going to be supersets of those near the bottom.



For example, we might have a unit test that verifies that passwords need to be at least eight characters in length, while any UI test that logs in will inadvertently end up testing the same thing too. So there's no avoiding that.

What we can avoid, however, is blatant duplication. We never want to write the exact same tests between different layers of the pyramid because that

would be wasteful. If we know we've got some scenario covered at the unit test level, there's no sense in duplicating it directly up top in the UI.

If it helps, think of the difference between UI and unit tests like this.

Unit tests	vs	UI tests
are about development		are about verification
rapid feedback		slow feedback
very low level		very high level
very local		go end-to-end
cheap		expensive
fast		slow
solid		fragile
reliable / deterministic		flaky / non-deterministic
used to develop		used to test
test from developer's POV		test from customer's POV

UI and integration tests are about connectivity. It's OK for those tests to be slower because they go through more layers of the architecture. That's why we love them! They are making sure things work end-to-end.

Unit tests, on the other hand, are about speed and feedback. We write unit tests when we are looking for feedback about things that are important to us during development. Things like:

- Did we get our design right?
- Did we break anything with the last set of changes?
- Do all our assumptions and edge cases check out?
- Is it safe to add new functionality?

Unit tests are what enable us to iterate quickly. UI and integration tests are about making sure things work end-to-end. Both serve an important purpose. They're just two different sides of the same coin.

So yes, some duplication in functionality is perfectly fine, just so long as we are not duplicating intent.

And that's basically it! That's the pyramid. The rest of the book is just going to focus on the details of where and when to write each of these tests, and show you how they work in the real world for the web.

But that does leave us with one interesting question. If you are on a mixed team made up of developers and testers, who exactly should be writing these things?