Extracted from:

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

Venkat Subramaniam

Edited by Jacquelyn Carter

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-174-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2016

CHAPTER 1

Automation Shall Set You Free

Everyone benefits a great deal when the applications we create actually work. Failure in production is expensive, and we should do our best to minimize that. With today's technology, reach, and visibility, when applications fail the entire world can take notice. With automated tests, we can fail fast and safely, and in the process create resilient applications that work well in production.

Automated testing also has a deep impact on the design of the code. It naturally forces the code to be modular, cohesive, and loosely coupled. That, in turn, makes the code easier to change, and that has a positive impact on the cost of change.

You're probably eager to start coding, but learning a bit about whys and the possible impediments of automated testing will get you ready for the deeply technical things that follow this chapter. Let's quickly discuss the benefits and challenges of automated testing and how you can prepare and make use of the fast feedback loops.

The Challenges of Change

Code gets modified several times in its lifetime. If programmers tells you their code has not changed since the initial writing, they're implying that their project got canceled. For an application to stay relevant, it has to evolve. We make enhancements, add features, and often fix bugs. With each change comes a few challenges:

- The change should be affordable and cost effective.
- The change should have the right effect.

Let's discuss each of these challenges.

The Cost of Change

A good design is flexible, easier to extend, and less expensive to maintain. But how can we tell? We can't wait to see the aftermath of the design to learn about its quality—that may be too late.

Test-driven design can help to address that concern. In this approach, we first create an initial, big picture, strategic design. Then, through small tactical steps, and by applying some fundamental design principles (see *Agile Software Development, Principles, Patterns, and Practices [Mar02]*), we refine the design further. The tests, among other things, provide continuous feedback to ensure that the design being implemented in code meets the requirements. Tests promote good design principles—high cohesion, low coupling, more modular code, a single level of abstraction—traits that make change affordable.

The Effect of Change

"Does it work?" is a dreaded question we often hear when we change code. "I hope" is the usual response we developers give. There's nothing wrong in having hope that our efforts have the right effect, but we can strive for better.

Software is a nonlinear system—a change here can break something over there. For example, one small incorrect change to a data format can adversely affect different parts of a system. If disparate parts of a system begin to fail after a change is deployed, the result is frustration and pain. It's also embarrassing—our customers think of us as amateurs rather than as professionals.

When we make a change, we should quickly know if the code that worked before continues to work now. We *need* rapid, short, *automated* feedback loops.

Testing vs. Verification

Using automated feedback loops doesn't mean no manual testing.

It is not about automated instead of manual—we need the right combination of both. Let's define two different terms that need to stand apart—testing and verification.

Testing is an act of gaining insights. Is the application usable? Is it intuitive, and what's the user experience like? Does the workflow make sense? Are there steps that can be removed? Testing should raise these kinds of questions and provide insight into the key capabilities and limitations of an application.

Verification, on the other hand, is an act of confirmation. Does the code do what it's supposed to do? Are the calculations right? Is the program working as expected after a code or configuration change? Did the update of a third-party library/module break the application? These are largely the concerns of verifying an application's behavior.

Manual testing is quite valuable. On a recent project, after hours of coding-and-automated verification cycle, I manually exercised the application. As soon as the page popped up in the browser I wanted to change quite a few things—that's the *observer effect*. We need to manually exercise and test applications often. However, keep in mind the intent: to gain insights, not to verify.

In the same application, I changed the database within weeks before production. A quick run of the automated tests immediately resulted in verification failures. Within minutes I was able to rectify and reverify, without even bringing up the web server. Automated verification saved my day.

Do Both Testing and Verification

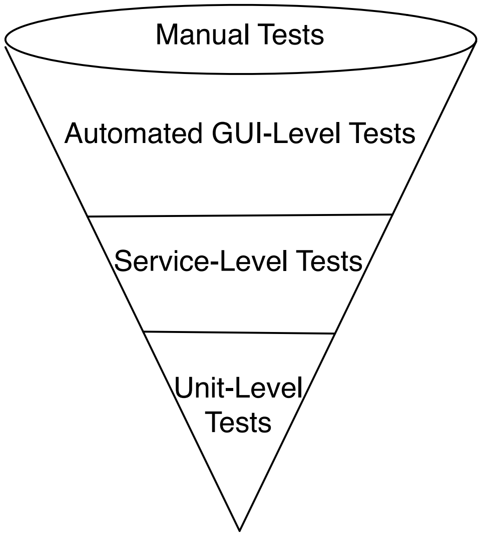
Do manual testing to gain insights and automated verification to influence the design and to confirm the code continues to meet the expectations.

Adopting Automated Verification

The efforts toward automated verification, or what we'll loosely call *automated testing*, varies widely across the industry; broadly there are three different adoption efforts:

- No automation, only manual verification. These are largely done by teams in denial. They struggle to validate their applications after each change and suffer from the consequences.
- Automation mostly at the user interface (UI) level. Quite a few teams have realized the need for automation verification, but have focused most of their efforts at the UI level. This leads to pathway-to-hell automation—we'll shortly discuss why.
- Automation at the right levels. This is done by mature teams that have gone beyond the first step of realizing the need for automation. They've invested in short feedback loops at various levels of their application, with more tests at lower levels.

Extreme focus on UI-level test automation results in the ice-cream cone antipattern.[1]



One reason why teams end up with this antipattern is the lack of alignment between different members of the teams. Eager to automate, the teams hire automation engineers tasked with creating automated test suites. Unfortunately, the programmers are often not on board and do not provide test hooks at different layers in the application—that's not something they had to do before. As a result, the automation engineers are limited to writing tests at the level they can get their arms around. This often is the GUI and any external-facing APIs.

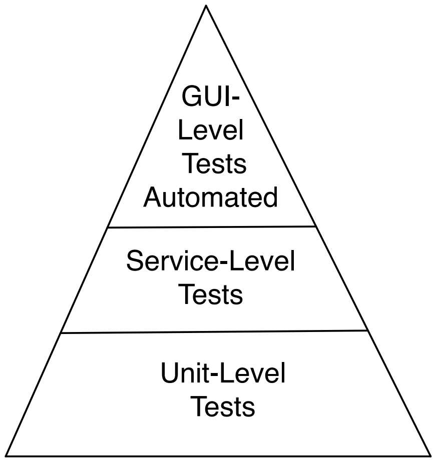
Testing mostly at the UI level has many drawbacks:[2]

- It's brittle. UI-level tests break often. UI is one of the most fluid parts of an application. It changes when the code it depends on changes. It also changes when customers or testers walk by, giving their feedback on how the UI should look and feel. Keeping tests in sync with changes in the UI is quite arduous, much more than at lower levels.
- It has too many moving parts. UI-level tests often need tools like Selenium and require different browsers to be up and running. Keeping these dependencies up and running takes effort.

1. http://watirmelon.com/2012/01/31/
2. http://martinfowler.com/bliki/TestPyramid.html

- It's slow. These tests need the entire application to be up and running: the client side, the server side, the database, and connections to any external services that may be needed. Running thousands of tests through full integration is often much slower than running isolated tests.
- It's hard to write. One of my clients struggled for well over six months to write a UI-level test for a simple interaction. We eventually discovered the problem was due to the timing of the tests compared to the running of the client-side JavaScripts that produced the results.
- It does not isolate the problem area. When a UI-level test fails, we only know something is amiss—can't tell readily where or at what level.
- It does not prevent logic in the UI. We all know putting logic in UI is bad, yet we've all seen it permeate and duplicate at this level. UI-level tests do nothing to mitigate this.
- It does not influence a better design. UI-level tests don't discourage the so-called "big ball of mud"—the opposite of modular code.

Mike Cohn suggests in *Succeeding with Agile: Software Development Using Scrum [Coh09]* the idea of a test pyramid—more tests at the lower levels and fewer end-to-end tests at the higher levels.



We should follow the test pyramid rather than falling into the traps of the ice-cream cone antipattern. Writing more tests at the lower levels has many benefits. The tests at the lower level run faster, they're easier to write, and they provide shorter feedback loops. They also lead to better modular design, and as a result, it gets easier to isolate and identify problems when they happen.

We'll follow that spirit in this book, to write tests at the right levels—more unit tests, then functional tests, and a few end-to-end UI-level tests.

Automated Verification Is Not Optional

Developing any nontrivial application without automated tests at the right level is an economic disaster.

If automation at the right level is so critical, then why don't more developers do it? Let's discuss some reasons next.