Extracted from:

Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

Adam Tornhill

edited by Adaobi Obi Tulton

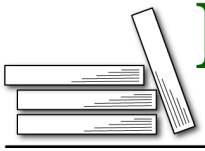
Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

Adam Tornhill

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 team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Adaobi Obi Tulton
Copy Editor: Candace Cunningham
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

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

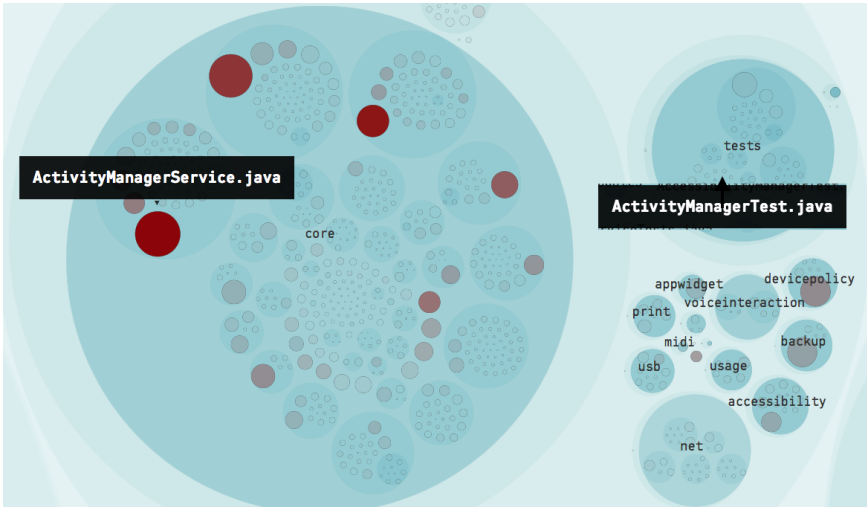
For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-272-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2018

Build Temporary Tests as a Safety Net

Before you apply a splinter refactoring you have to ensure that you won't break the behavior of the code. Unfortunately, most hotspots lack adequate test coverage and writing unit tests for a hotspot is often impossible until we've refactored the code. Let's look at an example from the Android codebase that we discussed earlier.

As you see in the figure on page 6, there's a big difference in the amount of application code in Android's `core` package versus the amount of test code in the `test` package.



That figure should put fear into any programmer planning a refactoring, because the unit test for the main hotspot, ActivityManagerService.java, with 20,000 lines of code, is a meager 33 (!) lines of test code. It's clear that this test won't help us refactor the code.

In situations like this you need to build a safety net based on *end-to-end tests*. End-to-end tests focus on capturing user scenarios and are performed on the system level. That is, you run with a real database, network connections, UI, and all other components of your system. End-to-end tests give you a fairly high test coverage that serves as a regression suite, and that test suite is the enabler that lets you perform the initial refactoring without breaking any fundamental behavior.

The type of end-to-end tests you need depends upon the API of your hotspot. If your hotspot exposes a REST API—or any other network-based interface—it's straightforward to cover it with tests because such APIs decouple your test code from the application. A UI, like a web page or a native desktop GUI, presents more challenges as it makes end-to-end tests much harder to automate. Our cure in that situation comes with inconvenient side effects but, just like any medicine, if you need it you really need it. So let's look at a way to get inherently untestable code under test.

Introduce Provisional End-to-End Tests

The trick is to treat the code as a black box and just focus on its visible behavior. For web applications, tools like Selenium let you record existing

interactions and play them back to ensure the end-user behavior is unaffected.[8] This gives you a way to record the main scenarios that involve your hotspot from a user's point of view. Tools like Sikuli let you use the same strategy to cover desktop UI applications with tests.[9]

The test strategy is based on tools that capture screen shots and use image recognition to interact with UI components. The resulting tests are brittle—a minor change to the style or layout of the UI breaks the regression suite—and expensive to maintain. That's why it's important to remember the context: your goal is to build a safety net that lets you refactor a central part of the system. Refactoring, by its very nature, preserves existing behavior since it makes for a safer and more controlled process.

Thus, we need to consider our UI-based safety net as a temporary creation that we dispose of once we've reached our intermediate goal. You emphasize that by giving the temporary test suite a provocative name, as we discussed in *Signal Incompleteness with Names, on page ?*.

Finally, measure the *code coverage* of your test suite and look for uncovered execution paths with high complexity.[10] You use that coverage information as feedback on the completeness of your tests and record additional tests to cover missing execution paths. You could also make a mental note to extract that behavior into its own splinter module.

Maintainable Tests Don't Depend on Details

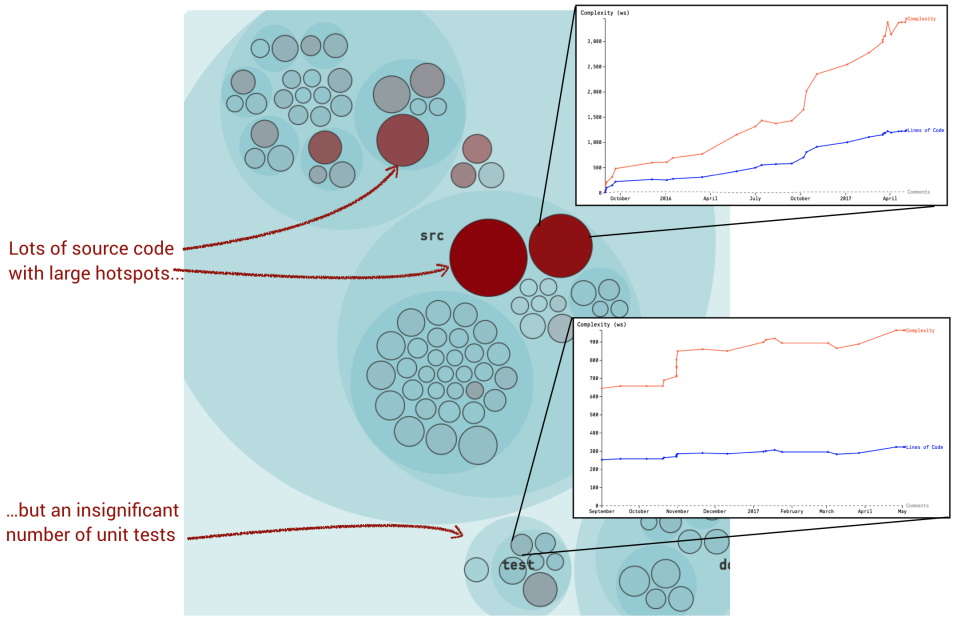
Maintainable end-to-end tests don't depend on the details of the rendered UI. Instead they query the DOM based on known element identities or, in the case of desktop applications, the identity of a specific component.

Reduce Debt by Deleting Cost Sinks

It's a depressingly common case to find hotspots with inadequate test coverage. That doesn't mean there aren't any tests at all, just that there aren't any tests where we would need them to be. Surprisingly often, organizations have unit-test suites that don't grow together with the application code, yet add to the maintenance costs. Let's look at the warning signs in the figure on page 8.

As you see in the figure, the ratio between the amount of source code versus test code is unbalanced. The second warning sign is that the complexity trends

8. http://www.seleniumhq.org/
9. http://www.sikuli.org/
10. https://en.wikipedia.org/wiki/Code_coverage



show different patterns for the hotspot and its corresponding unit test. This is a sign that the test code isn't doing its job by growing together with the application code, and a quick code inspection is likely to confirm those suspicions.

This situation happens when a dedicated developer attempts to introduce unit tests but fails to get the rest of the organization to embrace the technique. Soon you have a test suite that isn't updated beyond the initial tests, yet needs to be tweaked in order to compile so that the automated build passes.

You won't get any value out of such unit tests, but you still have to spend time just to make them build. A simple cost-saving measure is to delete such unit tests, as they do more harm than good.