The Pragmatic Programmers

# Your Code as a Crime Scene

Second Edition

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

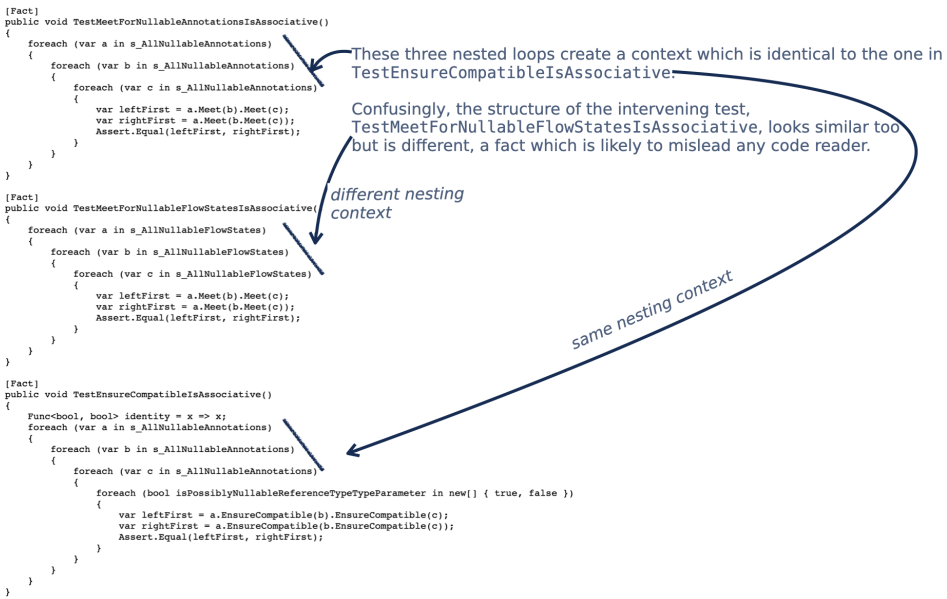
## Understand Why Test Code Isn't Just Test Code

Even if test automation—and its distant relative, Test-Driven Development (TDD)—are in widespread use today, both are fairly recent additions to mainstream software development. Consequently, we as a community might not yet have learned what good tests are, what works, and what doesn't. Let's illustrate the point with a prominent example, which you can also view interactively.[3]

Roslyn is the implementation of the C# and Visual Basic compilers, together with an API for writing tools. It's a large-scale codebase with six million lines of code. The .Net team has also invested heavily in test automation.

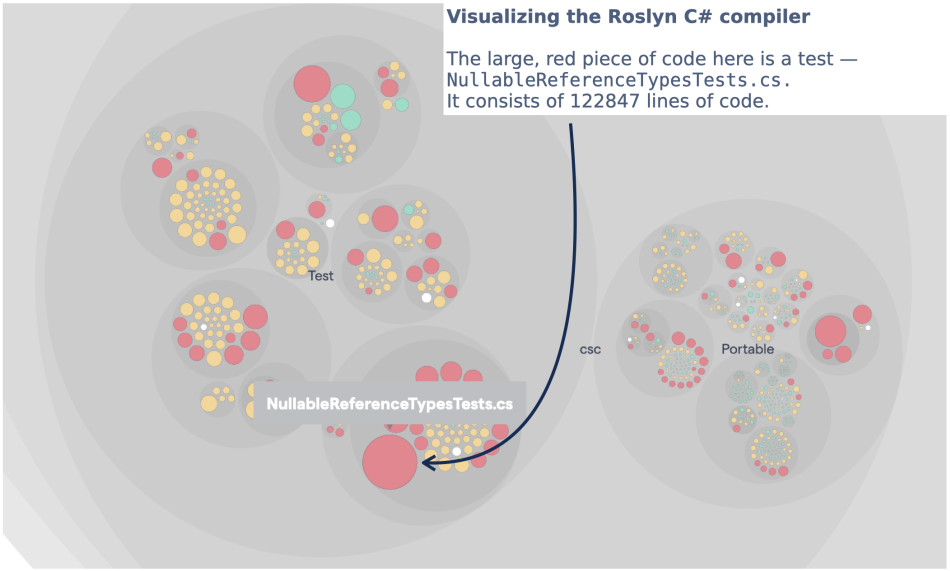
Let's peek under the hood in the visualization on page 4.

As you see, the tests have a massive piece of red code: NullableReferenceTypesTests.cs. That test suite alone consists of more than 120k lines of code! A quick scan of the source code reveals several instances of duplication, which makes the code harder to understand, not easier.




Now, you might think that I pulled out an extreme case with the Roslyn hotspot just to make my point. And you'd be correct. I did. But I did it for a reason. Over the past decade, I've probably analyzed 300-plus codebases. During all those analyses, I observed that we developers are fairly conscious

3. https://tinyurl.com/roslyn-code-health




of the DRY principle…in application code. When it comes to test code, well, not so much. Consequently, some of the worst technical debt I find tends to be in tests, similar to what we found in the Roslyn platform.

We already discussed the fallacy of treating test code as "just test code." From a productivity perspective, the test scripts you create are every bit as important as the application code you write, and technical debt in automated tests spells just as much trouble. We'd never accept a 120k-line monstrosity in our application code, would we?

**Joe asks:**

## What Else Could You Achieve with 120,000 Lines of Code?

Finding a single test suite with 120,000 lines of code is astonishing. To help you put it into perspective, the complete source code for the Apollo 11 Guidance Computer measures a mere 115,000 lines of code.[a] Implementing nullable reference types in C# seems to be a harder problem than landing on the moon. Respect.

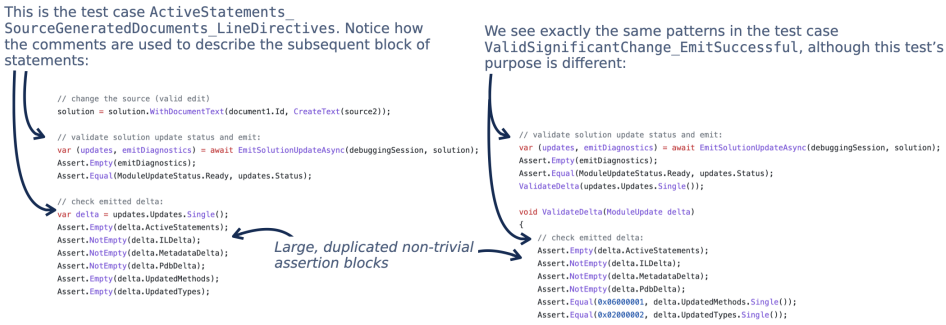
a. https://github.com/code-as-a-crime-scene/Apollo-11

Of course, there's always the counterargument that if we abstract our tests too much, they become harder to understand. That's absolutely true. But it's

also true that there's a whole gulf of abstractions between "not at all" and "too much." Perhaps there's a mid-point where we can pay attention to the abstraction level in our tests without going completely overboard with abstraction acrobatics? Again, parameterized tests, which we met in Reduce Duplication via Parameterized Tests, on page ?, are a much-underutilized tool for striking this balance.

## Encapsulate the Test Criteria

Virtually all automated tests contain assertions used to verify the test outcome. (If your tests don't, then it's likely they are there merely to game the code coverage metrics; see Reverse the Perspective via Code Coverage Measures, on page ?.) In many codebases, these assertions tend to be repetitive and leaky abstractions. Let's see this in action by inspecting another Roslyn test suite, EditAndContinueWorkspaceServiceTests.cs (see the figure on page 5).




The preceding figure reveals chunks of large and non-trivial assertion blocks that are duplicated across the test suite. Again, this is an exceedingly common test smell. The problems with this testing style are a) the intent of the test becomes harder to understand and b) the duplication makes it very easy to miss updating the test criteria in all places when it changes.

The duplicated-assertion-blocks smell is a classic example of a lack of encapsulation. The solution is straightforward: encapsulate the test criteria in a custom assert with a descriptive name that can communicate without the need for the code comment, and re-use the custom assert when your tests call for it. Test data has to be encapsulated just like any other implementation detail.