Defensive Design Strategies to Prevent Flaky Tests

By Steve Berczuk - April 29, 2020

Agile code is at the heart of agile software development. It’s not enough to embrace change in the requirements process; you need to be able to deliver incremental changes along the way, since agile methods deal with uncertainty in the product space by enabling stakeholders to inspect and adapt. Without working code that can be delivered into production quickly, it’s hard to inspect things in a meaningful way.

There are many elements to agile code. Designing with patterns to allow for flexibility with minimal overhead is essential, but not enough. You also need testing in order to get feedback.

More testing isn’t always better testing, though. If your tests are unreliable, they can cause more harm than good. “Flaky tests” aren’t as rare as we’d like to think, and there are a few things you can keep in mind to avoid them and help your development lifecycle move more smoothly.

Flaky tests are tests that fail intermittently, and whose failures don’t really provide useful indications of significant errors in the product code. Flaky tests could be the result of issues in the code, but more often they are the result of assumptions in the test code that lead to non-relatable results. Common examples are UI tests that rely on identifying elements based on positions, or data-driven tests that make assumptions about the initial state of the data without verifying it.

While integration tests are often the most likely to exhibit flakiness, unit tests are not immune. Unit tests based on time computations can fail on time zone transitions, or if a legacy test is written with the assumption of a future date that is fixed.

There are many reasons that tests can fail intermittently, and some can be easily avoided by applying good defensive design strategies. Sometimes the problem is that the test is too simple—for example, comparing a response to an API call as a string, when you have no control over order or even whitespace. Instead, compare fields in a structured object, checking only for the fields you expect.

Sometimes the issue is testing something irrelevant. Comparing all the fields in a response rather than just the ones that matter to the client that you are testing can lead to test failures that don’t imply code failures.

Stateful integration tests can fail when you make assumptions about the data that you can’t verify. Ideally, you would start with a known external system state, but if that’s not possible, consider structuring tests so that the query can examine the current state, make a change, and then examine the new state.

While one can argue that tests that provide more noise than signal should just be skipped, that is not always a good practice. Even very simple tests that provide valuable insight can be flaky if not written well.

Test code *is* code, and it should be treated with the same level of good design that you treat production code. It should be adaptable and only make assumptions that it verifies. By making this a practice, you can keep all of your code