Chapter 11. Testing Overview

Written by Adam Bender

Edited by Tom Manshreck

Testing has always been a part of programming. In fact, the first time you wrote a computer program you almost certainly threw some sample data at it to see whether it performed as you expected. For a long time, the state of the art in software testing resembled a very similar process, largely manual and error prone. However, since the early 2000s, the software industry's approach to testing has evolved dramatically to cope with the size and complexity of modern software systems. Central to that evolution has been the practice of developer-driven, automated testing.

Automated testing can prevent bugs from escaping into the wild and affecting your users. The later in the development cycle a bug is caught, the more expensive it is; exponentially so in many cases.[1] However, "catching bugs" is only part of the motivation. An equally important reason why you want to test your software is to support the ability to change. Whether you're adding new features, doing a refactoring focused on code health, or undertaking a larger redesign, automated testing can quickly catch mistakes and this makes it possible to change software with confidence.

Companies that can iterate faster can adapt more rapidly to changing technologies, market conditions, and customer tastes. If you have a robust testing practice, you needn't fear change—you can embrace it as an essential quality of developing software. The more and faster you want to change your systems, the more you need a fast way to test them.

The act of writing tests also improves the design of your systems. As the first clients of your code, a test can tell you much about your design choices. Is your system too tightly coupled to a database? Does the API support the required use cases? Does your system handle all of the edge cases? Writing automated tests forces you to confront these issues early on in the development cycle. Doing so generally leads to more modular software that enables greater flexibility later on.

Much ink has been spilled about the subject of testing software, and for good reason: for such an important practice, doing it well still seems to be a mysterious craft to many. At Google while we have come a long way, we still

face difficult problems getting our processes to scale reliably across the company. In this chapter, we'll share what we have learned to help further the conversation.

Why Do We Write Tests?

To better understand how to get the most out of testing let's start from the beginning. When we talk about automated testing, what are we really talking about?

The simplest test is defined by:

- A single behavior you are testing, usually a method or API that you are calling
- A specific input, some value that you pass to the API
- An observable output or behavior
- A controlled environment such as a single isolated process

When you execute a test like this, passing the input to the system and verifying the output, you will learn whether the system behaves as you expect. Taken in aggregate, hundreds or thousands of simple tests (usually called a *test suite*) can tell you how well your entire product conforms to its intended design and, more important, when it doesn't.

Creating and maintaining a healthy test suite takes real effort. As a codebase grows, so too will the test suite. It will begin to face challenges like instability and slowness. A failure to address these problems will cripple a test suite. Keep in mind that tests derive their value from the trust engineers place in them. If testing becomes a productivity sink, constantly inducing toil and uncertainty, engineers will lose trust and begin to find workarounds. A bad test suite can be worse than no test suite at all.

In addition to empowering companies to build great products quickly, testing is becoming critical to ensuring the safety of important products and services in our lives. Software is more involved in our lives than ever before and defects can cause more than a little annoyance: they can cost massive amounts of money, loss of property, or, worst of all, loss of life.[2]

At Google, we have determined that testing cannot be an afterthought. Focusing on quality and testing is part of how we do our jobs. We have learned, sometimes painfully, that failing to build quality into our products

and services inevitably leads to bad outcomes, as a result, we have built testing into the heart of our engineering culture.

The Story of Google Web Server

In Google's early days, engineer-driven testing was often assumed to be of little importance. Teams regularly relied on smart people to get the software right. A few systems ran large integration tests, but mostly it was the Wild West. One product in particular seemed to suffer the worst, it was called the Google Web Server, also known as "GWS."

GWS is the web server responsible for serving Google Search queries and is as important to Google Search as Air Traffic Control is to an airport. Back in 2005, as the project swelled in size and complexity, productivity had slowed dramatically. Releases were becoming buggier, and it was taking longer and longer to push them out. Team members had little confidence when making changes to the service, and often found out something was wrong only when features stopped working in production. (At one point more than 80% of production pushes contained user-affecting bugs that had to be rolled back).

To address these problems the Tech Lead (TL) of GWS decided to institute a policy of engineer-driven, automated testing. As part of this policy, all new code changes were required to include tests and those tests would be run continuously. Within a year of instituting this policy, the number of emergency pushes *dropped by half*. This drop occurred despite the fact that the project was seeing a record number of new changes every quarter. Even in the face of unprecedented growth and change, testing brought renewed productivity and confidence to one of the most critical projects at Google. Today GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures.

The changes in GWS marked a watershed for testing culture at Google as teams in other parts of the company saw the benefits of testing and moved to adopt similar tactics.

One of the key insights the GWS experience taught us was that you can't rely on programmer ability alone to avoid product defects. Even if each engineer writes only the occasional bug, after you have enough people working on the same project you will be swamped by the ever-growing list of defects. Imagine a hypothetical 100-person team whose engineers are so good that they each write only a single bug a month. Collectively this group of amazing engineers still produces five new bugs every workday. Worse yet, in a

complex system, fixing one bug can often cause another, as engineers adapt to known bugs and code around them.

The best teams find ways to turn the collective wisdom of its members into a benefit for the entire team. That is exactly what automated testing does. After an engineer on the team writes a test it is added to the pool of common resources available to others. Everyone else on the team can now run the test and will benefit when it detects an issue. Contrast this to an approach based on debugging, wherein each time a bug occurs, an engineer must pay the cost of digging into it with a debugger. The cost in engineering resources is night and day, and was the fundamental reason GWS was able to turn its fortunes around.

Testing at the Speed of Modern Development

Software systems are growing larger and evermore complex. A typical application or service at Google is made up of thousands or millions of lines of code. It uses hundreds of libraries or frameworks, and must be delivered via unreliable networks to an increasing number of platforms running with an uncountable number of configurations. To make matters worse, new versions are pushed to users frequently, sometimes multiple times each day. This is a far cry from the world of shrink-wrapped software that saw updates only once or twice a year.

The ability for humans to manually validate every behavior in a system has been unable to keep pace with the explosion of features and platforms in most software. Imagine what it would take to manually test all of the functionality of Google Search, like finding flights, movie times, relevant images, and of course web search results (see Figure 11-1). Even if you can determine how to solve that problem, you then need to multiply that workload by every language, country, and device Google Search must support, and don't forget to check for things like accessibility and security. Attempting to assess product quality by asking humans to manually interact with every feature just doesn't scale. When it comes to testing there is one clear answer: automation.

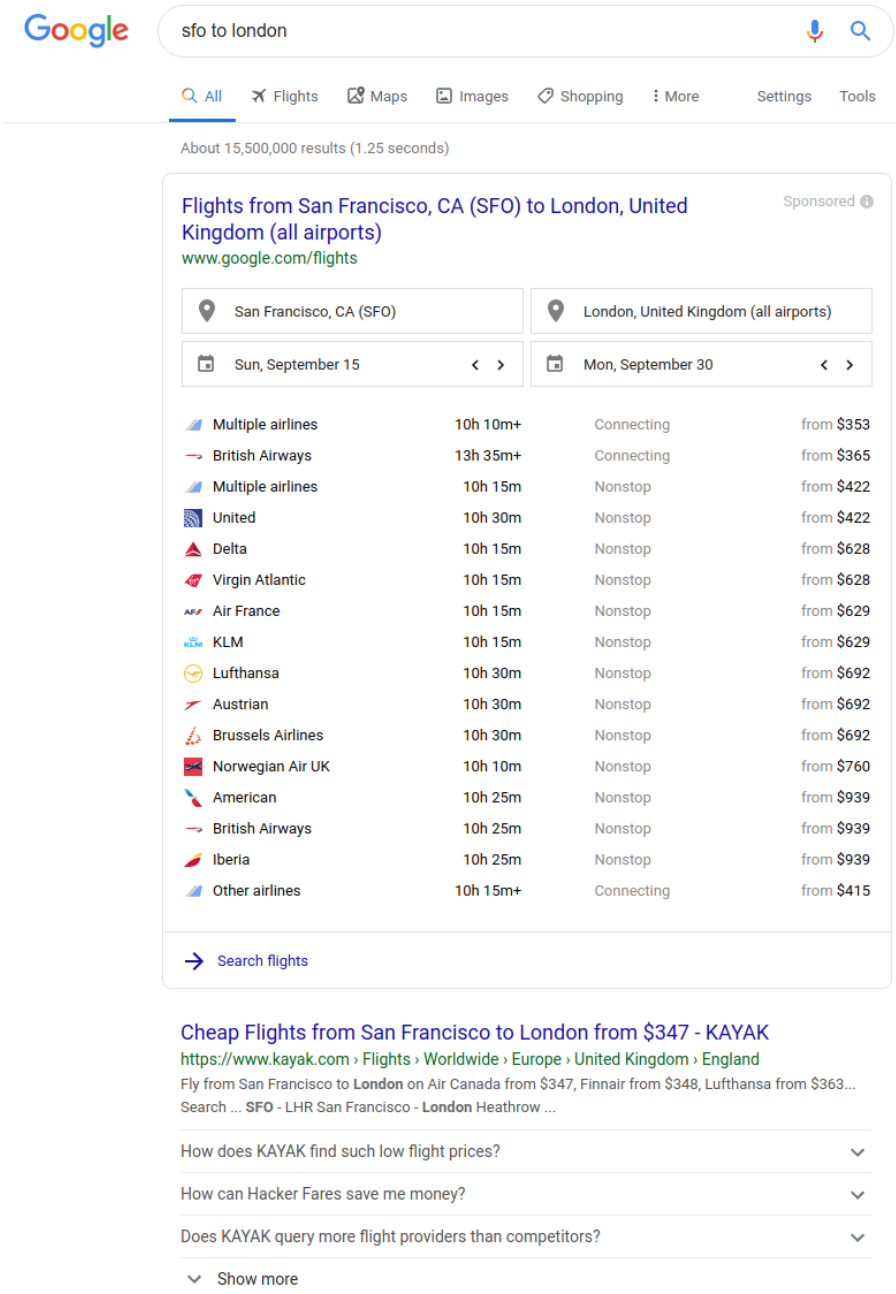

Figure 11-1. Screenshots of two complex Google search results

Write, Run, React

In its purest form, automating testing consists of three activities: writing tests, running tests, and reacting to test failures. An automated test is a small bit of code, usually a single function or method, that calls into an isolated part of a larger system that you want to test. The test code sets up an expected environment, calls into the system, usually with a known input, and verifies the result. Some of the tests are very small, exercising a single code path; others are much larger and can involve entire systems like a mobile operating system or web browser.

Figure 11-2 presents a deliberately simple test in Java using no frameworks or testing libraries. This is not how you would write an entire test suite, but at its core every automated test looks similar to this very simple example.

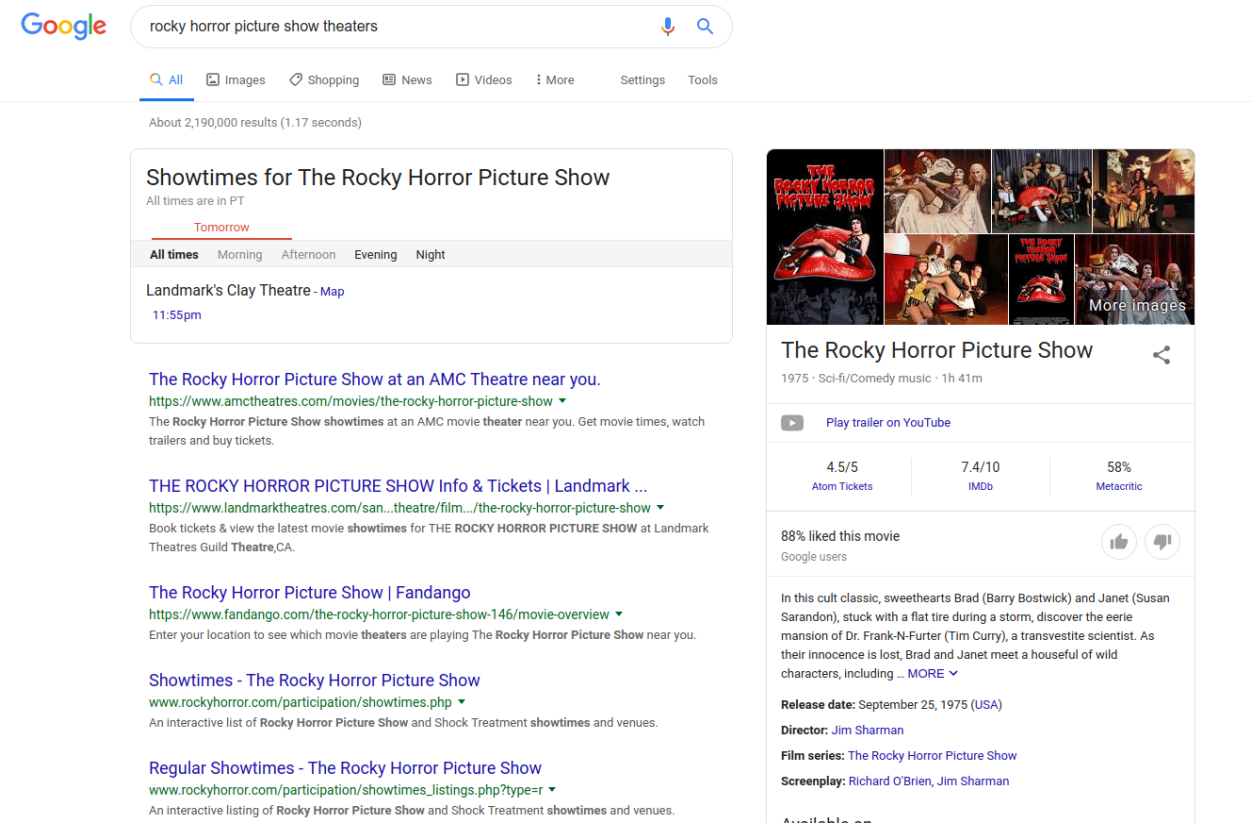

Figure 11-2. An example test

Unlike the QA processes of yore, in which rooms of dedicated software testers poured over new versions of a system exercising every possible behavior, the engineers who build systems today play an active and integral role in writing and running automated tests for their own code. Even in companies where QA is a prominent organization, developer-written tests are commonplace. At the speed and scale that today's systems are being developed, the only way to keep up is by sharing the development of tests around the entire engineering staff.

Of course, writing tests is different from writing *good tests*. It can be quite difficult to train tens of thousands of engineers to write good tests. We will discuss what we have learned about writing good tests in the chapters that follow.

Writing tests is only the first step in the process of automated testing. After you have written tests, you need to run them. Frequently. At its core, automated testing consists of repeating the same action over and over, only requiring human attention when something breaks. We will discuss this

Continuous Integration (CI) and testing in Chapter 23. By expressing tests as code instead of a manual series of steps, we can run them every time the code changes; easily thousands of times per day. Unlike human testers, machines never grow tired or bored.

Another benefit of having tests expressed as code is that it is easy to modularize them for execution in various environments. Testing the behavior of Gmail in Firefox requires no more effort than doing so in Chrome, provided you have configurations for both of these systems.[3] Running tests for a user interface (UI) in Japanese or German can be done using the same test code as for English.

Products and services under active development will inevitably experience test failures. What really makes a testing process effective is how it addresses test failures. Allowing failing tests to pile up quickly defeats any value they were providing so it is imperative not to let that happen. Teams that prioritize fixing a broken test within minutes of a failure are able to keep confidence high, failure isolation fast, and therefore derive more value out of their tests.

In summary, a healthy automated testing culture encourages everyone to share the work of writing tests. Such a culture also ensures that tests are run regularly. Last, and perhaps most important, it places an emphasis on fixing broken tests quickly so as to maintain high confidence in the process.

Benefits of Testing Code

To developers coming from organizations that don't have a strong testing culture, the idea of writing tests as a means of improving productivity and velocity might seem antithetical. After all, the act of writing tests can take just as long (if not longer!) than implementing a feature would take in the first place. On the contrary, at Google we've found that investing in software tests provides several key benefits to developer productivity:

Less debugging

> As you would expect, tested code has fewer defects when it is submitted. Critically, it also has fewer defects throughout its existence; most of them will be caught before the code is submitted. A piece of code at Google is expected to be modified dozens of times in its lifetime. It will be changed by other teams and even automated code maintenance systems. A test written once continues to pay dividends and prevent costly defects and annoying debugging sessions

through the lifetime of the project. Changes to a project, or the dependencies of a project, that break a test can be quickly detected by test infrastructure and rolled back before the problem is ever released to production.

Increased confidence in changes

All software changes. Teams with good tests can review and accept changes to their project with confidence because all important behaviors of their project are continuously verified. Such projects encourage refactoring. Changes that refactor code while preserving existing behavior should (ideally) require no changes to existing tests.

Improved documentation

Software documentation is notoriously unreliable. From outdated requirements, to missing edge cases, it is common for documentation to have a tenuous relationship to the code. Clear, focused tests that exercise one behavior at a time function as executable documentation. If you want to know what the code does in a particular case, look at the test for that case. Even better, when requirements change and new code breaks an existing test, we get a clear signal that the "documentation" is now out of date. Note that tests work best as documentation only if care is taken to keep them clear and concise.

Simpler reviews

All code at Google is reviewed by at least one other engineer before it can be submitted (see Chapter 9 for more details). A code reviewer spends less effort verifying code works as expected if the code review includes thorough tests that demonstrate code correctness, edge cases, and error conditions. Instead of the tedious effort needed to mentally walk each case through the code, the reviewer can verify that each case has a passing test.

Thoughtful design

Writing tests for new code is a practical means of exercising the API design of the code itself. If new code is difficult to test, it is often because the code being tested has too many responsibilities or difficult-to-manage dependencies. Well-designed code should be modular, avoiding tight coupling and focusing on specific

responsibilities. Fixing design issues early often means less rework later.

Fast, high-quality releases

With a healthy automated test suite, teams can release new versions of their application with confidence. Many projects at Google release a new version to production every day—even large projects with hundreds of engineers and thousands of code changes being submitted every day. This would not be possible without automated testing.

Designing a Test Suite

Today, Google operates at a massive scale, but we haven't always been so large and the foundations of our approach were laid long ago. Over the years, as our codebase has grown, we have learned a lot about how to approach the design and execution of a test suite, often by making mistakes and cleaning up afterward.

One of the lessons we learned fairly early on is that engineers favored writing larger, system-scale tests, but that these tests were slower, less reliable, and more difficult to debug than smaller tests. Engineers, fed up with debugging the system-scale tests, asked themselves, "Why can't we just test one server at a time?" or, "Why do we need to test a whole server at once? We could test smaller modules individually." Eventually, the desire to reduce pain led teams to develop smaller and smaller tests, which turned out to be faster, more stable, and generally less painful.

This led to a lot of discussion around the company about the exact meaning of "small." Does small mean unit test? What about integration tests, what size are those? We have come to the conclusion that there are two distinct dimensions for every test case: size and scope. Size refers to the resources that are required to run a test case: things like memory, processes, and time. Scope refers to the specific code paths we are verifying. Note that executing a line of code is different from verifying that it worked as expected. Size and scope are interrelated but distinct concepts.

Test Size

At Google, we classify every one of our tests into a size and encourage engineers to always write the smallest possible test for a given piece of

functionality. A test's size is determined not by its number of lines of code, but by how it runs, what it is allowed to do, and how many resources it consumes. In fact, in some cases our definitions of small, medium, and large are actually encoded as constraints the testing infrastructure can enforce on a test. We go into the details in a moment, but in brief, *small tests* run in a single process, *medium tests* run on a single machine, and *large tests* run wherever they want, as demonstrated in Figure 11-3.[4]

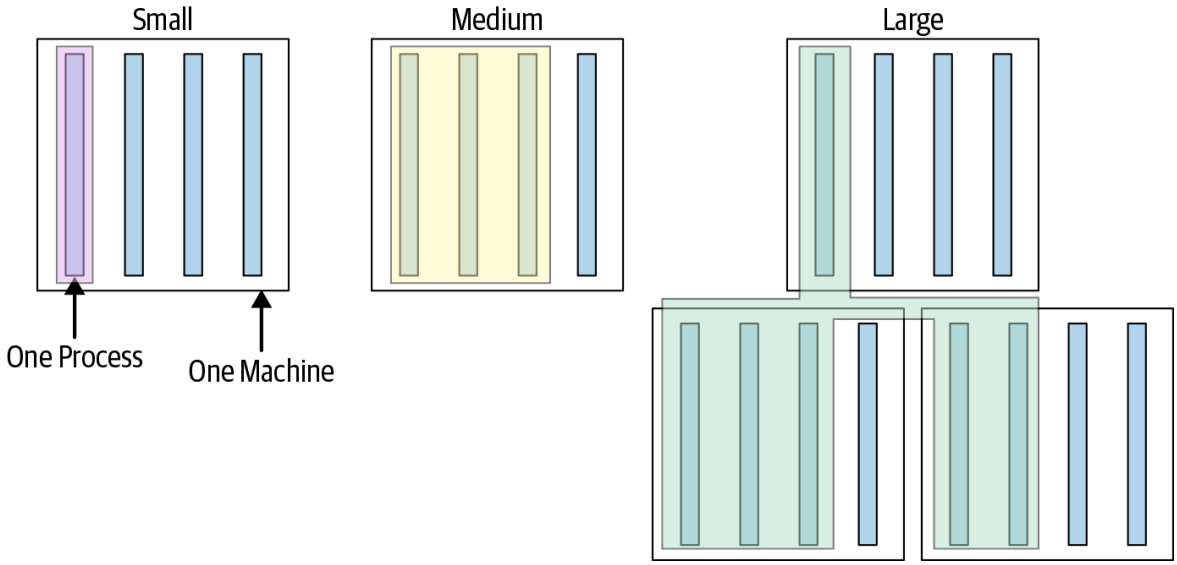

Figure 11-3. Test sizes

We make this distinction, as opposed to the more traditional "unit" or "integration," because the most important qualities we want from our test suite are speed and determinism, regardless of the scope of the test. Small tests, regardless of the scope, are almost always faster and more deterministic than tests that involve more infrastructure or consume more resources. Placing restrictions on small tests makes speed and determinism much easier to achieve. As test sizes grow, many of the restrictions are relaxed. Medium tests have more flexibility but also more risk of nondeterminism. Larger tests are saved for only the most complex and difficult testing scenarios. Let's take a closer look at the exact constraints imposed on each type of test.

SMALL TESTS

Small tests are the most constrained of the three test sizes. The primary constraint is that small tests must run in a single process. In many languages we restrict this even further to say that they must run on a single thread. This means that the code performing the test must run in the same process as the code being tested. You can't run a server and have a separate test process connect to it. It also means that you can't run a third-party program such as a database as part of your test.

The other important constraints on small tests are that they aren't allowed to sleep, perform I/O operations,[5] or make any other blocking calls. This means that small tests aren't allowed to access the network or disk. Testing code that relies on these sorts of operations requires the use of test doubles (see Chapter 13) to replace the heavyweight dependency with a lightweight, in-process dependency.

The purpose of these restrictions is to ensure that small tests don't have access to the main sources of test slowness or nondeterminism. A test that runs on a single process and never makes blocking calls can effectively run as fast as the CPU can handle. It's difficult (but certainly not impossible) to accidentally make such a test slow or nondeterministic. The constraints on small tests provide a sandbox that prevents engineers from shooting themselves in the foot.

These restrictions might seem excessive at first, but consider a modest suite of a couple hundred small test cases running throughout the day. If even a few of them fail nondeterministically (often called flaky tests) tracking down the cause becomes a serious drain on productivity. At Google's scale, such a problem could grind our testing infrastructure to a halt.

At Google, we encourage engineers to try to write small tests whenever possible, regardless of the scope of the test because it keeps the entire test suite running fast and reliably. For more discussion on small versus unit tests see Chapter 12.

MEDIUM TESTS

The constraints placed on small tests can be too restrictive for many interesting kinds of tests. The next rung up the ladder of test sizes is the medium test. Medium tests can span multiple processes, use threads, and can make blocking calls, including network calls to `localhost`. The only remaining restriction is that medium tests aren't allowed to make network calls to any system other than `localhost`. In other words, the test must be contained within a single machine.

The ability to run multiple processes opens up a lot of possibilities. For example, you could run a database instance to validate that the code you're testing integrates correctly in a more realistic setting. Or you could test a combination of web UI and server code. Tests of web applications often involve tools like WebDriver that start a real browser and control it remotely via the test process.

Unfortunately, with increased flexibility comes increased potential for tests to become slow and nondeterministic. Tests that span processes or are allowed to make blocking calls are dependent on the operating system and third-party processes to be fast and deterministic, which isn't something we can guarantee in general. Medium tests still provide a bit of protection by preventing access to remote machines via the network, which is far and away the biggest source of slowness and nondeterminism in most systems. Still, when writing medium tests, the "safety" is off, and engineers need to be much more careful.

LARGE TESTS

Finally, we have large tests. Large tests remove the `localhost` restriction imposed on medium tests, allowing the test and the system being tested to span across multiple machines. For example, the test might run against a system in a remote cluster.

As before, increased flexibility comes with increased risk. Having to deal with a system that spans multiple machines and the network connecting them increases the chance of slowness and nondeterminism significantly compared to running on a single machine. We mostly reserve large tests for full-system end-to-end tests that are more about validating configuration than pieces of code, and for tests of legacy components for which it is impossible to use test doubles. We'll talk more about use cases for large tests in Chapter 14. Teams at Google will frequently isolate their large tests from their small or medium tests, running them only during the build and release process so as not to impact developer workflow.

FLAKY TESTS ARE EXPENSIVE

If you have a few thousand tests, each with a very tiny bit of nondeterminism, running all day, occasionally one will probably fail (flake). As the number of tests grows, statistically so will the number of flakes. If each test has even a 0.1% of failing when it should not, and you run 10,000 tests per day, you will be investigating 10 flakes per day. Each investigation takes time away from something more productive that your team could be doing.

In some cases, you can limit the impact of flaky tests by automatically rerunning them when they fail. This is effectively trading CPU cycles for engineering time. At low levels of flakiness, this trade-off makes sense. Just keep in mind that rerunning a test is only delaying the need to address the root cause of flakiness.

If test flakiness continues to grow, you will experience something much worse than lost productivity: a loss of confidence in the tests. It doesn't take needing to

investigate many flakes before a team loses trust in the test suite. After that happens, engineers will stop reacting to test failures, eliminating any value the test suite provided. Our experience suggests that as you approach 1% flakiness the tests begin to lose value. At Google as our flaky rate hovers around 0.15%, which implies thousands of flakes every day. We fight hard to keep flakes in check including actively investing engineering hours to fix them.

In most cases, flakes appear because of nondeterministic behavior in the tests themselves. Software provides many sources of nondeterminism: clock time, thread scheduling, network latency, and more. Learning how to isolate and stabilize the effects of randomness is not easy. Sometimes, effects are tied to low-level concerns like hardware interrupts or browser rendering engines. A good automated test infrastructure should help engineers identify and mitigate any nondeterministic behavior.

PROPERTIES COMMON TO ALL TEST SIZES

All tests should strive to be hermetic: a test should contain all of the information necessary to set up, execute, and tear down its environment. Tests should assume as little as possible about the outside environment, such as the order in which the tests are run. For example, they should not rely on a shared database. This constraint becomes more challenging with larger tests, but effort should still be made to ensure isolation.

A test should contain *only* the information required to exercise the behavior in question. Keeping tests clear and simple aids reviewers in verifying that the code does what it says it does. Clear code also aids in diagnosing failure when they fail. We like to say that "a test should be obvious upon inspection." Because there are no tests for the tests themselves, they require manual review as an important check on correctness. As a corollary to this, we also strongly discourage the use of control flow statements like conditionals and loops in a test. More complex test flows risk containing bugs themselves, and make it more difficult to determine the cause of a test failure.

Remember that tests are often revisited only when something breaks. When you are called to fix a broken test that you have never seen before, you will be thankful someone took the time to make it easy to understand. Code is read far more than it is written, so make sure you write the test you'd like to read!

Test sizes in practice

Having precise definitions of test sizes has allowed us to create tools to enforce them. Enforcement enables us to scale our test suites and still make certain guarantees about speed, resource utilization, and stability. The extent to which these definitions are enforced at Google varies by language. For example, we run all Java tests using a custom security manager that will cause all tests tagged as small to fail if they attempt to do something prohibited, such as establish a network connection.

Test Scope

Though we at Google put a lot of emphasis on test size, another important property to consider is test scope. Test scope refers to how much code is being validated by a given test. Narrow-scoped tests (commonly called "unit tests") are designed to validate the logic in a small, focused part of the codebase, like an individual class or method. Medium-scoped tests (commonly called *integration tests*) are designed to verify interactions between a small number of components; for example, between a server and its database. Large-scoped tests (commonly referred to by names like *functional tests*, *end-to-end* tests, or *system tests*) are designed to validate the interaction of several distinct parts of the system, or emergent behaviors that aren't expressed in a single class or method.

It's important to note that when we talk about unit tests as being narrowly scoped, we're referring to the code that is being *validated*, not the code that is being *executed*. It's quite common for a class to have many dependencies or other classes it refers to, and these dependencies will naturally be invoked while testing the target class. Though some other testing strategies make heavy use of test doubles (fakes or mocks) to avoid executing code outside of the system under test, at Google we prefer to keep the real dependencies in place when it is feasible to do so. Chapter 13 discusses this issue in more detail.

Narrow-scoped tests tend to be small, and broad-scoped tests tend to be medium or large, but this isn't always the case. For example, it's possible to write a broad-scoped test of a server endpoint that covers all of its normal parsing, request validation, and business logic, which is nevertheless small because it uses doubles to stand in for all out-of-process dependencies like a database or filesystem. Similarly, it's possible to write a narrow-scoped test of a single method that must be medium sized. For example, modern web frameworks often bundle HTML and JavaScript together, and testing a UI

component like a Date Picker often requires running an entire browser, even to validate a single code path.

Just as we encourage tests of smaller size, at Google we also encourage engineers to write tests of narrower scope. As a very rough guideline, we tend to aim to have a mix of around 80% of our tests being narrow-scoped unit tests that validate the majority of our business logic; 15% medium-scoped integration tests that validate the interactions between two or more components; and 5% end-to-end tests that validate the entire system. Figure 11-3 depicts how we can visualize this as a pyramid.

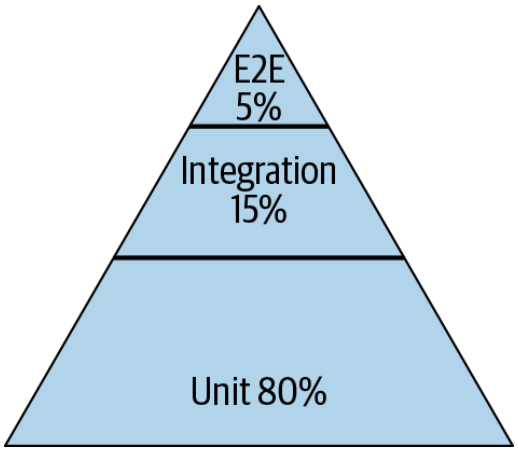

Figure 11-4. Google's version of Mike Cohn's test pyramid;[6] percentages are by test case count, and every team's mix will be a little different

Unit tests form an excellent base because they are fast, stable, and dramatically narrow the scope and reduce the cognitive load required to identify all the possible behaviors a class or function has. Additionally, they make failure diagnosis quick and painless. Two antipatterns to be aware of are the "ice cream cone" and the "hourglass," as illustrated in Figure 11-5.

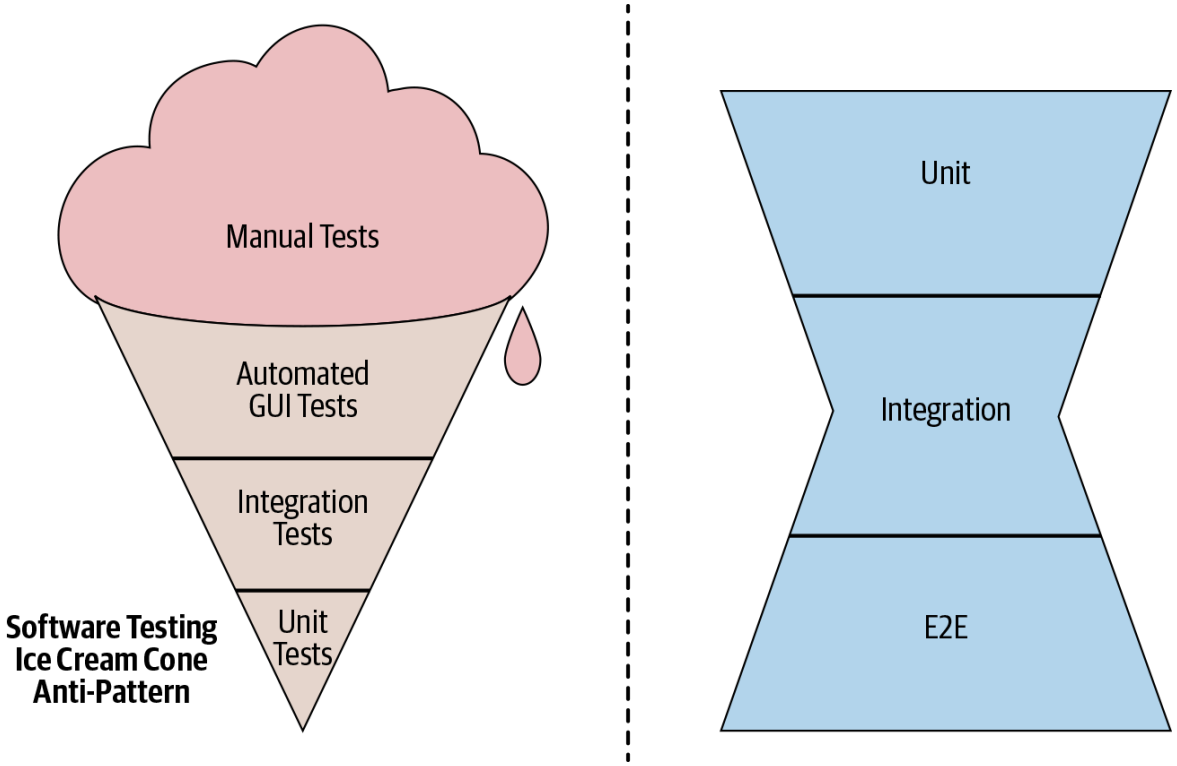

Figure 11-5. Test suite antipatterns

With the ice cream cone, engineers write many end-to-end tests but few integration or unit tests. Such suites tend to be slow, unreliable, and difficult to work with. This pattern often appears in projects that start as prototypes and are quickly rushed to production, never stopping to address testing debt.

The hourglass involves many end-to-end tests and many unit tests but few integration tests. It isn't quite as bad as the ice cream cone, but it still results in many end-to-end test failures that could have been caught quicker and more easily with a suite of medium-scope tests. The hourglass pattern occurs when tight coupling makes it difficult to instantiate individual dependencies in isolation.

Our recommended mix of tests is determined by our two primary goals: engineering productivity and product confidence. Favoring unit tests gives us high confidence quickly, and early in the development process. Larger tests act as sanity checks as the product develops; they should not be viewed as a primary method for catching bugs.

When considering your own mix, you might want a different balance. If you emphasize integration testing, you might discover that your test suites take longer to run but catch more issues between components. When you emphasize unit tests, your test suites can complete very quickly, and you will catch many common logic bugs. But, unit tests cannot verify the interactions between components, like a contract between two systems developed by

different teams. A good test suite contains a blend of different test sizes and scopes that are appropriate to the local architectural and organizational realities.

The Beyoncé Rule

We are often asked, when coaching new hires, which behaviors or properties actually need to be tested? The straightforward answer is: test everything that you don't want to break. In other words, if you want to be confident that a system exhibits a particular behavior, the only way to be sure it will is to write an automated test for it. This includes all of the usual suspects like testing performance, behavioral correctness, accessibility, and security. It also includes less obvious properties like testing how a system handles failure.

We have a name for this general philosophy: we call it the Beyoncé Rule. Succinctly, it can be stated as follows: "If you liked it, then you shoulda put a test on it". The Beyoncé Rule is often invoked by infrastructure teams that are responsible for making changes across the entire codebase. If unrelated infrastructure changes pass all of your tests but still break your team's product, you are on the hook for fixing it and adding the additional tests.

TESTING FOR FAILURE

One of the most important situations a system must account for is failure. Failure is inevitable, but waiting for an actual catastrophe to find out how well a system responds to a catastrophe is a recipe for pain. Instead of waiting for a failure, write automated tests that simulate common kinds of failures. This includes simulating exceptions or errors in unit tests and injecting Remote Procedure Call (RPC) errors or latency in integration and end-to-end tests. It can also include much larger disruptions that affect the real production network using techniques like Chaos Engineering. A predictable and controlled response to adverse conditions is a hallmark of a reliable system.

A Note on Code Coverage

Code coverage is a measure of which lines of feature code are exercised by which tests. If you have 100 lines of code and your tests execute 90 of them, you have 90% code coverage.[7] Code coverage is often held up as the gold standard metric for understanding test quality, and that is somewhat unfortunate. It is possible to exercise a lot of lines of code with a few tests, never checking that each line is doing anything useful. That's because code coverage only measures that a line was invoked, not what happened as a

result (we recommend only measuring coverage from small tests to avoid coverage inflation that occurs when executing larger tests).

An even more insidious problem with code coverage is that, like other metrics, it quickly becomes a goal unto itself. It is common for teams to establish a bar for expected code coverage; for instance, 80%. At first, that sounds eminently reasonable; surely you want to have at least that much coverage. In practice, what happens is that instead of treating 80% like a floor, engineers treat it like a ceiling. Soon, changes begin landing with no more than 80% coverage. After all, why do more work than the metric requires?

A better way to approach the quality of your test suite is to think about the behaviors that are tested. Do you have confidence that everything your customers expect to work, will work? Do you feel confident you can catch breaking changes in your dependencies? Are your tests stable and reliable? Questions like these are a more holistic way to think about a test suite. Every product and team is going to be different; some will have difficult-to-test interactions with hardware, some involve massive datasets. Trying to answer the question "do we have enough tests?" with a single number ignores a lot of context and is unlikely to be useful. Code coverage can provide some insight into untested code, but it is not a substitute for thinking critically about how well your system is tested.

Testing at Google Scale

Much of the guidance to this point can be applied to codebases of almost any size. However, we should spend some time on what we have learned testing at our very large scale. To understand how testing works at Google, you need an understanding of our development environment. The most important fact about which, is that most of Google's code is kept in a single monolithic repository (monorepo). Almost every line of code for every product and service we operate is all stored in one place. We have more than two billion lines of code in the repository today.

Google's codebase experiences close to 25 million lines of change every week. Roughly half of them are made by the tens of thousands of engineers working in our monorepo, and the other half by our automated systems, in the form of configuration updates, or large-scale changes (Chapter 22). Many of those changes are initiated from outside the immediate project. We don't place many limitations on the ability of engineers to reuse code.

The openness of our codebase encourages a level of co-ownership that lets everyone take responsibility for the codebase. One benefit of such openness is the ability to directly fix bugs in a product or service you use (subject to approval, of course) instead of complaining about it. This also implies that many people will make changes in a part of the codebase owned by someone else.

Another thing that makes Google a little different is that almost no teams use repository branching. All changes are committed to the repository head and are immediately visible for everyone to see. Furthermore, all software builds are performed using the last committed change that our testing infrastructure has validated. When a product or service is built, almost every dependency required to run it is also built from source, also from the head of the repository. Google manages testing at this scale by use of a CI system. One of the key components of our CI system is our Test Automated Platform (TAP).

NOTE

For more information on TAP and our CI philosophy, see Chapter 23.

Whether you are considering our size, our monorepo, or the number of products we offer, Google's engineering environment is complex. Every week it experiences millions of changing lines, billions of test cases being run, tens of thousands of binaries being built, and hundreds of products being updated—talk about complicated!

The Pitfalls of a Large Test Suite

As a codebase grows you will inevitably need to make changes to existing code. When poorly written, automated tests can make it more difficult to make those changes. Brittle tests—those which over-specify expected outcomes or rely on extensive and complicated boilerplate—can actually resist change. These poorly written tests can fail even when unrelated changes are made.

If you have ever made a five-line change to a feature only to find dozens of unrelated, broken tests, you have felt the friction of brittle tests. Over time, this friction can make a team reticent to perform necessary refactoring to keep a codebase healthy. The subsequent chapters will cover strategies that you can use to improve the robustness and quality of your tests.

Some of the worst offenders of brittle tests come from the misuse of mock objects. Google's codebase has suffered so badly from an abuse of mocking frameworks that it has led some engineers to declare "no more mocks!" Although that is a strong statement, understanding the limitations of mock objects can help you avoid misusing them.

NOTE

For more information on working effectively with mock objects, see Chapter 13.

In addition to the friction caused by brittle tests, a larger suite of tests will be slower to run. The slower a test suite, the less frequently it will be run, and the less benefit it provides. We use a number of techniques to speed up our test suite, including parallelizing execution and using faster hardware. However, these kinds of tricks are eventually swamped by a large number of individually slow test cases.

Tests can become slow for many reasons like booting significant portions of a system, firing up an emulator before execution, processing large datasets, or waiting for disparate systems to synchronize. Tests often start fast enough but slow down as the system grow. For example, maybe you have an integration test exercising a single dependency that takes five seconds to respond, but over the years you grow to depend on a dozen services and now the same tests take five minutes.

Tests can also become slow due to unnecessary speed limits introduced by functions like `sleep()` and `setTimeout()`. Calls to these functions are often used as naive heuristics before checking the result of non-deterministic behavior. Sleeping for half a second here or there doesn't seem too dangerous at first, however if a "wait-and-check" is embedded in a widely used utility, pretty soon you have added minutes of idle time to every run of your test suite. A better solution is to actively poll for a state transition with a frequency closer to microseconds. You can combine this with a timeout value in case a test fails to reach a stable state.

Failing to keep a test suite deterministic and fast ensures it will become roadblock to productivity. At Google, engineers who encounter these tests have found ways to work around slowdowns, with some going as far as to skip the tests entirely when submitting changes. Obviously, this is a risky practice and should be discouraged, but if a test suite is causing more harm than good, eventually engineers will find a way to get their job done, tests or no tests.

The secret to living with a large test suite is to treat it with respect. Incentivize engineers to care about their tests; reward them as much for having rock-solid tests as you would for having a great feature launch. Set appropriate performance goals and refactor slow or marginal tests. Basically, treat your tests like production code. When simple changes begin taking nontrivial time, spend effort making your tests less brittle.

In addition to developing the proper culture, invest in your testing infrastructure by developing linters, documentation, or other assistance that makes it more difficult to write bad tests. Reduce the number of frameworks and tools you need to support to increase the efficiency of the time you invest to improve things.[8] If you don't invest in making it easy to manage your tests, eventually engineers will decide it isn't worth having them at all.

History of Testing at Google

Now that we've discussed how Google approaches testing, it might be enlightening to learn how we got here. As mentioned previously, Google's engineers didn't always embrace the value of automated testing. In fact, until 2005, testing was closer to a curiosity than a disciplined practice. Most of the testing was done manually, if it was done at all. However, from 2005 to 2006 a testing revolution occurred and changed the way we approach software engineering. Its effects continue to reverberate within the company to this day.

The experience of the GWS project, which we discussed at the opening of this chapter, acted as a catalyst. It made it clear how powerful automated testing could be. Following the improvements to GWS in 2005, the practices began spreading across the entire company. The tooling was primitive. However, the volunteers, who came to be known as the Testing Grouplet, didn't let that slow them down.

Three key initiatives helped usher automated testing into the company's consciousness: Orientation Classes, the Test Certified program, and Testing on the Toilet. Each one had influence in a completely different way, and together they reshaped Google's engineering culture.

Orientation Classes

Even though much of the early engineering staff at Google eschewed testing, the pioneers of automated testing at Google knew that at the rate the company was growing, new engineers would quickly outnumber existing

team members. If they could reach all the new hires in the company, it could be an extremely effective avenue for introducing cultural change. Fortunately, there was, and still is, a single choke point that all new engineering hires pass through: orientation.

Most of Google's early orientation program concerned things like medical benefits and how Google Search worked, but starting in 2005 it also began including an hour-long discussion of the value of automated testing.[9] The class covered the various benefits of testing such as increased productivity, better documentation, and support for refactoring. It also covered how to write a good test. For many Nooglers (new Googlers) at the time, such a class was their first exposure to this material. Most important, all of these ideas were presented as though they were standard practice at the company. The new hires had no idea that they were being used as trojan horses to sneak this idea into their unsuspecting teams.

As Nooglers joined their teams following orientation, they began writing tests and questioning those on the team who didn't. Within only a year or two, the population of engineers who had been taught testing outnumbered the pretesting culture engineers. As a result, many new projects started off on the right foot.

Testing has now become more widely practiced in the industry so most new hires arrive with the expectations of automated testing firmly in place. Nonetheless, orientation classes continue to set expectations about testing and connect what Nooglers know about testing outside of Google to the challenges of doing so in our very large and very complex codebase.

Test Certified

Initially, the larger and more complex parts of our codebase appeared resistant to good testing practices. Some projects had such poor code quality that they were almost impossible to test. To give projects a clear path forward, the Testing Grouplet devised a certification program that they called Test Certified. Test Certified aimed to give teams a way to understand the maturity of their testing processes, and more critically, cookbook instructions on how to improve it.

The program was organized into five levels, and each level required some concrete actions to improve the test hygiene on the team. The levels were designed in such a way that each step up could be accomplished within a quarter, which made it a convenient fit for Google's internal planning cadence.

Test Certified Level 1 covered the basics: set up a continuous build; start tracking code coverage; classify all your tests as small, medium, or large; identify (but don't necessarily fix) flaky tests; and create a set of fast (not necessarily comprehensive) tests that can be run quickly. Each subsequent level added more challenges like "no releases with broken tests" or "remove all nondeterministic tests." By Level 5, all tests were automated, fast tests were running before every commit, all nondeterminism had been removed, and every behavior was covered. An internal dashboard applied social pressure by showing the level of every team. It wasn't long before teams were competing with one another to climb the ladder.

By the time the Test Certified program was replaced by an automated approach in 2015 (more on pH later) it had helped more than 1,500 projects improve their testing culture.

Testing on the Toilet

Of all the methods the Testing Grouplet used to try to improve testing at Google, perhaps none was more off-beat than Testing on the Toilet (TotT). The goal of TotT was fairly simple: actively raise awareness about testing across the entire company. The question is, what's the best way to do that in a company with employees scattered around the world?

The Testing Grouplet considered the idea of a regular email newsletter, but given the heavy volume of email everyone deals with at Google, it was likely to become lost in the noise. After a little bit of brainstorming, someone proposed the idea of posting flyers in the restroom stalls as a joke. We quickly recognized the genius in it: the bathroom is one place that everyone must visit at least once each day, no matter what. Joke or not, the idea was cheap enough to implement that it had to be tried.

In April 2006, a short writeup covering how to improve testing in Python appeared in restroom stalls across Google. This first episode was posted by a small band of volunteers. To say the reaction was polarized is an understatement; some saw it as an invasion of personal space, and they objected strongly. Mailing lists lit up with complaints, but the TotT creators were content: the people complaining were still talking about testing.

Ultimately, the uproar subsided and TotT quickly became a staple of Google culture. To date, engineers from across the company have produced several hundred episodes, covering almost every aspect of testing imaginable (in addition to a variety of other technical topics). New episodes are eagerly anticipated and some engineers even volunteer to post the episodes around

their own buildings. We intentionally limit each episode to exactly one page, challenging authors to focus on the most important and actionable advice. A good episode contains something an engineer can take back to the desk immediately and try.

Ironically for a publication that appears in one of the more private locations, TotT has had an outsized public impact. Most external visitors see an episode at some point in their visit, and such encounters often lead to funny conversations about how Googlers always seem to be thinking about code. Additionally, TotT episodes make great blog posts, something the original TotT authors recognized early on. They began publishing lightly edited versions publicly, helping to share our experience with the industry at large.

Despite starting as a joke, TotT has had the longest run, and the most profound impact of any of the testing initiatives started by the Testing Grouplet.

Testing Culture Today

Testing culture at Google today has come a long way from 2005. Nooglers still attend orientation classes on testing, and TotT continues to be distributed almost weekly. However, the expectations of testing have more deeply embedded themselves in the daily developer workflow.

Every code change at Google is required to go through code review. And every change is expected to include both the feature code and tests. Reviewers are expected to review the quality and correctness of both. In fact, it is perfectly reasonable to block a change if it is missing tests.

As a replacement for Test Certified, one of our engineering productivity teams recently launched a tool called Project Health (pH). The pH tool continuously gathers dozens of metrics on the health of a project, including test coverage and test latency, and makes them available internally. pH is measured on a scale of one (worst) to five (best). A pH-1 project is seen as a problem for the team to address. Almost every team that runs a continuous build automatically get a pH score.

Over time, testing has become an integral part of Google's engineering culture. We have myriad ways to reinforce its value to engineers across the company. Through a combination of training, gentle nudges, mentorship, and, yes, even a little friendly competition, we have created the clear expectation that testing is everyone's job.

Why didn't we start by mandating the writing of tests?

The Testing Grouplet had considered asking for a testing mandate from senior leadership but quickly decided against it. Any mandate on how to develop code would be seriously counter to Google culture and likely slow the progress, independent of the idea being mandated. The belief was that successful ideas would spread, so the focus became demonstrating success.

If engineers were deciding to write tests on their own, it meant that they had fully accepted the idea and were likely to keep doing the right thing—even if no one was compelling them to.

The Limits of Automated Testing

Automated testing is not suitable for all testing tasks. For example, testing the quality of search results often involves human judgement. We conduct targeted, internal, studies using Search Quality Raters who execute real queries and record their impressions. Similarly, it is difficult to capture the nuances of audio and video quality in an automated test, so we often use human judgement to evaluate the performance of telephony or video-calling systems.

In addition to qualitative judgements, there are certain creative assessments for which humans excel. For example, searching for complex security vulnerabilities is something that humans do better than automated systems. After a human has discovered and understood a flaw, it can be added to an automated security testing system like Google's Cloud Security Scanner where it can be run continuously and at scale.

A more generalized term for this technique is Exploratory Testing. Exploratory Testing is a fundamentally creative endeavor in which someone treats the application under test as a puzzle to be broken, maybe by executing an unexpected set of steps or by inserting unexpected data. When conducting an exploratory test, the specific problems to be found are unknown at the start. They are gradually uncovered by probing commonly overlooked code paths or unusual responses from the application. As with the detection of security vulnerabilities, as soon as an exploratory test discovers an issue, an automated test should be added to prevent future regressions.

Using automated testing to cover well-understood behaviors enables the expensive and qualitative efforts of human testers to focus on the parts of

your products for which they can provide the most value—and avoid boring them to tears in the process.

Conclusion

The adoption of developer-driven automated testing has been one of the most transformational software engineering practices at Google. It has enabled us to build larger systems with larger teams, faster than we ever thought possible. It has helped us keep up with the increasing pace of technological change. Over the past 15 years, we have successfully transformed our engineering culture to elevate testing into a cultural norm. Despite the company growing by a factor of almost 100 times since the journey began, our commitment to quality and testing is stronger today than it has ever been.

This chapter has been written to help orient you to how Google thinks about testing. In the next few chapters we are going to dive even deeper into some key topics that have helped shape our understanding of what it means to write good, stable, and reliable tests. We will discuss the what, why, and how of unit tests, the most common kind of test at Google. We will wade into the debate on how to effectively use test doubles in tests through techniques such as faking, stubbing, and interaction testing. Finally, we will discuss the challenges with testing larger and more complex systems, like many of those we have at Google.

At the conclusion of these three chapters you should have a much deeper and clearer picture of the testing strategies we use and, more important, why we use them.

TL;DRs

- Testing is as much about catching bugs as alerting you to changes
- For tests to scale, they must be automated
- A balanced test suite is necessary for maintaining healthy test coverage
- "If you liked it, you should have put a test on it"
- Changing the testing culture in organizations take time

1 "Defect Prevention: Reducing Costs and Enhancing Quality"

2 "Failure at Dhahran"

3 Getting the behavior right across different browsers and languages is a different story! But, ideally, the end-user experience should be the same for everyone.

4 Technically we have four sizes of test at Google: small, medium, large, and *enormous*. The internal difference between medium and large is actually subtle and historical; so, in this book, most descriptions of large actually apply to our notion of enormous.

5 There is a little wiggle room in this policy. Tests are allowed to access a filesystem if they use a hermetic, in-memory implementation.

6 Succeeding with Agile, by Mike Cohn, 2009.

7 Keep in mind that there are different kinds of coverage (line, path, branch, etc.) and each says something different about which code has been tested. In this simple example, line coverage is being used.

8 Each supported language at Google has one standard test framework and one standard mocking/stubbing library. One set of infrastructure runs most tests in all languages across the entire codebase.

9 This class was so successful that an updated version is still taught today. In fact, it is one of the longest-running orientation classes in the company's history.