**Testing Overview**

Testing has always been a part of programming.

An equally important reason why you want to test your software is to support the ability to change.

How testing can help you change?

Whether you’re

* 1. adding new features,
  2. doing a refactoring focused on code health,
  3. or undertaking a larger redesign,

Automated testing can quickly catch mistakes, and this makes it possible to change software with confidence.

Writing automated tests forces you to confront these issues early in the development cycle.

Doing so generally leads to more modular software that enables greater flexibility later.

**Why Do We Write Tests**?

* Hundreds or thousands of simple tests (usually called a *test suite*).
* Creating and maintaining a healthy test suite takes real effort.
* A bad test suite can be worse than no test suite at all.
* At Google, we have determined that testing cannot be an afterthought. we have built testing into the heart of our engineering culture.
* (At one point, more than 80% of production pushes contained user-affecting bugs that had to be rolled back.)
* (TL) of GWS decided to institute a policy of engineer-driven, automated testing.
* Within a year of instituting this policy, the number of emergency pushes *dropped by half*. This drop occurred even though the project was seeing a record number of new changes every quarter.
* 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.

**Testing at the Speed of Modern Development**

The ability for humans to manually validate every behaviour 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 the functionality of Google Search, like finding flights, movie times.

**Write, Run, React**

 What really makes a testing process effective is how it addresses test failures.

**Benefits of Testing Code**

* + Less debugging
  + Increased confidence in changes
  + Improved documentation
  + Simpler reviews
  + Thoughtful design
  + Fast, high-quality releases

**Designing a Test Suite**

* + Google hasn’t always been so large, and the foundations of their approach were laid long ago.
  + How to approach the design and execution of a test suite, often by making mistakes and cleaning up afterward.
  + Develop smaller and smaller tests, which turned out to be faster, more stable, and generally less painful.
  + 2 distinct dimensions for every test case:
  + **Size** => the resources that are required to run a test case: things like memory, processes, and time.
  + **Scope** => the specific code paths we are verifying.

**Test** (**Size**)

* We 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, & it’s allowed to do?
* Small tests run in a single process; medium tests run on a single machine & large tests run wherever they want.
* The most important qualities we want from our test suite are speed and determinism.
* **Small** tests, regardless of the scope, are almost always faster and more deterministic.
* **Medium** tests have more flexibility but also more risk of nondeterminism.
* **Larger** tests are saved for only the most complex and difficult testing scenarios.

**Small tests**

* most constrained & must run in a single process/ single thread.
* code performing the test must run in the same process as the code being tested.
* The purpose of small test restrictions is to ensure that small tests don’t have access to the main sources of test slowness or nondeterminism.
* Tracking down the cause of failing a suite of small tests nondeterministically, becomes a serious drain on productivity.

**Medium tests**

\* Medium tests can span multiple processes, use threads, and can make blocking calls, including network calls,

to localhost & test must be contained within a single machine.

\* With increased flexibility comes increased potential for tests to become slow and nondeterministic.

when writing medium tests, the “safety” is off, and engineers need to be much more careful.

**Large tests**

- They span across multiple machines. For example, the test might run against a system in a remote cluster.

- And that increases the chance of slowness and nondeterminism.

- Large test is more about validating configuration than pieces of code.

- We can limit the impact of flaky tests by automatically rerunning them when they fail. This is effectively trading CPU - Cycles for engineering time. & rerunning a test is only delaying the need to address the root cause of flakiness.

- As you approach 1% flakiness, the tests begin to lose value.

- In most cases, flakes are because of nondeterministic. Which sources are (clock time, thread scheduling, network latency), and more.

- A good, automated test infrastructure should help engineers identify and mitigate any nondeterministic behaviour.

**Properties of all test sizes**

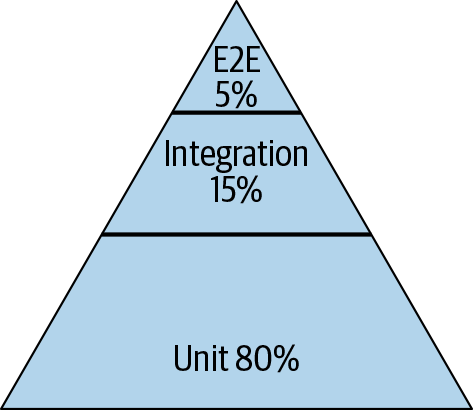
* A test should contain all the information necessary to set up, execute, and tear down its environment.
* Clear code also aids in diagnosing failure when they fail.
* Code is read far more than it is written, so make sure you write the test you’d like to read!

**Test Scope** 3 scales/ranges.

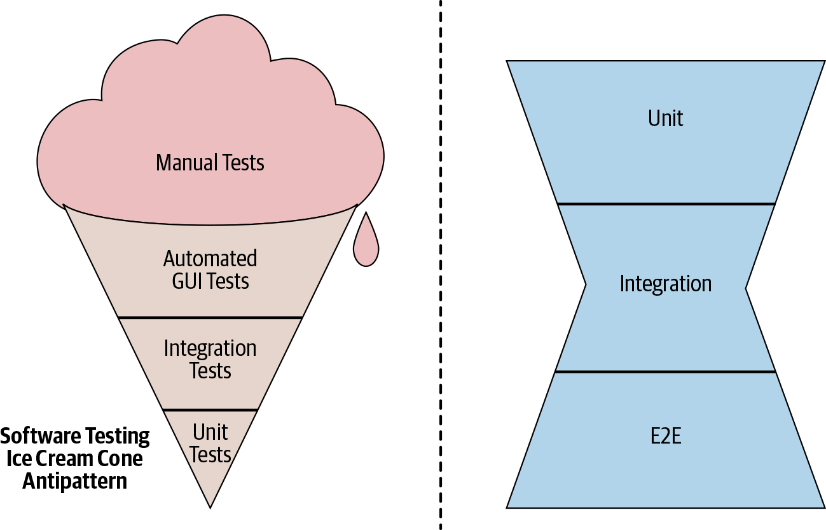
it’s possible to write a broad-scoped test of a server endpoint.

it’s possible to write a narrow-scoped test of a single method that must be medium sized.

We encourage engineers to write tests of narrower scope.



Unit tests are fast, stable & identify all the possible behaviours. - make failure diagnosis quick and painless.



**Ice cream:** test suites tend to be slow, unreliable, and difficult to work with

i.e. (projects quickly rushed to production)

**Hourglass:** End-to-End test failures because of lack of medium scope.

occurs when tight coupling makes it difficult to instantiate individual dependencies in isolation.

Mix of tests is determined by our two primary goals: **engineering productivity** and **product confidence.**

If you emphasize [integration] => take longer to run but catch more issues between components.

If you emphasize [unit] => quickly & logic bugs. But cannot verify the interactions between components.

A good test suite is an appropriate blend.

**The Beyoncé Rule “**If you liked it, then you shoulda put a test on it.”

1) Test everything that you don’t want to break**. -** system exhibits a particular behaviour.

2) A predictable and controlled response to adverse conditions is a hallmark of a reliable system.

**NOTE on Code coverage.**

* which lines of feature code are exercised by which tests. - Gold standard metric for understanding test quality.
* Code coverage only measures that a line was invoked, not what happened as a result.
* why do more work than the metric requires?
* 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.**

1. Google’s code is kept in a single, monolithic repository ([monorepo](https://oreil.ly/qSihi)).
2. We don’t place many limitations on the ability of engineers to reuse code.
3. Changes are committed to the repository head and are immediately visible for everyone to see.
4. 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).

**The Pitfalls of a Large Test Suite**

* Inevitable need for change - Brittle tests (can fail even when unrelated changes are made) actually resist change.
* The slower a test suite, the less frequently it will be run, and the less benefit it provides. + Reasons & Techniques
* Tests often start fast enough but slow down as the system grows.
* [ grow to depend on a dozen services] + [ sleep () – setTimeout() ]
* Failing to keep a test suite deterministic and fast ensures it will become roadblock to productivity.
* Like production, when changes begin taking nontrivial time, spend effort making your tests less brittle.
* invest in making it easy to manage your tests. => Not to => having them at all.

## History of Testing at Google

GWS as catalyst =>how powerful automated testing could be**.**

3 usher automated testing into the company’s consciousness.

## Orientation Classes

1. class covered the various benefits of testing, such as increased productivity, better documentation, and support for refactoring.
2. Testing has now become more widely practiced in the industry, so most new hires arrive with the expectations of automated testing firmly in place.

## Test Certified As a replacement for Test Certified,

\* Some projects had such poor code quality that they were almost impossible to test.

\* Test Certified aimed to give teams a way to understand the maturity of testing processes and how to improve it.

\* The program was organized into five levels.

Test Certified Level 1

set up a continuous build, start tracking code coverage; classify, identify flaky tests & create a set of fast tests.

By Level 5

all tests were automated, fast, all nondeterminisms were removed, and every behaviour was covered.

## Testing on the Toilet (posting flyers in the restroom stalls)

# The was actively raise awareness about testing across the entire company.

# 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

- Every code (including [ feature code & tests]) change must go through code review.

- Reviewers are expected to review the quality and correctness of both.

- It’s reasonable to block a change if it is missing tests.

- The pH tool gathers metrics on the health of a project, (test coverage & test latency).

- If engineers wrote tests on their own, they are 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.
* We often use human judgment to evaluate the performance of telephony or video-calling systems.
* There are certain creative assessments at which humans excel [ searching for complex security vulnerabilities is something that humans do better] => Exploratory Testing.
* Using automated testing to cover well-understood behaviours 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.