Chapter 12. Unit Testing

Written by Erik Kuefler

Edited by Tom Manshreck

The previous chapter introduced two of the main axes along which Google classifies tests: *size* and *scope*. To recap, size refers to the resources consumed by a test and what it is allowed to do, and scope refers to how much code a test is intended to validate. Though Google has clear definitions for test size, scope tends to be a little fuzzier. We use the term *unit test* to refer to tests of relatively narrow scope, such as of a single class or method. Unit tests are usually small in size, but this isn't always the case.

After preventing bugs, the most important purpose of a test is to improve engineers' productivity. Compared to broader-scoped tests, unit tests have many properties that make them an excellent way to optimize productivity:

- They tend to be small according to Google's definitions of test size. Small tests are fast and deterministic, allowing developers to run them frequently as part of their workflow and get immediate feedback.
- They tend to be easy to write at the same time as the code they're testing, allowing engineers to focus their tests on the code they're working on without having to set up and understand a larger system.
- They promote high levels of test coverage because they are quick and easy to write. High test coverage allows engineers to make changes with confidence that they aren't breaking anything.
- They tend to make it easy to understand what's wrong when they fail because each test is conceptually simple and focused on a particular part of the system.
- They can serve as documentation and examples, showing engineers how to use the part of the system being tested and how that system is intended to work.

Due to their many advantages, most tests written at Google are unit tests, and as a rule of thumb, we encourage engineers to aim for a mix of about 80% unit

tests and 20% broader-scoped tests. This advice, coupled with the ease of writing unit tests and the speed with which they run, means that engineers run a *lot* of unit tests—it's not at all unusual for an engineer to execute thousands of unit tests (directly or indirectly) during the average workday.

Because they make up such a big part of engineers' lives, Google puts a lot of focus on *test maintainability*. Maintainable tests are ones that "just work": after writing them, engineers don't need to think about them again until they fail, and those failures indicate real bugs with clear causes. The bulk of this chapter focuses on exploring the idea of maintainability and techniques for achieving it.

The Importance of Maintainability

Imagine this scenario: Mary wants to add a simple new feature to the product and is able to implement it quickly, perhaps requiring only a couple dozen lines of code. But when she goes to check in her change, she gets a screen full of errors back from the automated testing system. She spends the rest of the day going through those failures one by one. In each case, the change introduced no actual bug, but broke some of the assumptions that the test made about the internal structure of the code, requiring those tests to be updated. Often, she has difficulty figuring out what the tests were trying to do in the first place, and the hacks she adds to fix them make those tests even more difficult to understand in the future. Ultimately, what should have been a quick job ends up taking hours or even days of busywork, killing Mary's productivity and sapping her morale.

Here, testing had the opposite of its intended effect by draining productivity rather than improving it while not meaningfully increasing the quality of the code under test. This scenario is far too common, and Google engineers struggle with it every day. There's no magic bullet, but many engineers at Google have been working to develop sets of patterns and practices to alleviate these problems, which we encourage the rest of the company to follow.

The problems Mary ran into weren't her fault, and there was nothing she could have done to avoid them: bad tests must be fixed before they are checked in, lest they impose a drag on future engineers. Broadly speaking, the issues she encountered fall into two categories. First, the tests she was

working with were *brittle*: they broke in response to a harmless and unrelated change that introduced no real bugs. Second, the tests were *unclear*: after they were failing, it was difficult to determine what was wrong, how to fix it, and what those tests were supposed to be doing in the first place.

Preventing Brittle Tests

As just defined, a brittle test is one that fails in the face of an unrelated change to production code that does not introduce any real bugs.[1] Such tests must be diagnosed and fixed by engineers as part of their work. In small codebases with only a few engineers, having to tweak a few tests for every change might not be a big problem. But if a team regularly writes brittle tests, test maintenance will inevitably consume a larger and larger proportion of the team's time as they are forced to comb through an increasing number of failures in an ever-growing test suite. If a set of tests needs to be manually tweaked by engineers for each change, calling it an "automated test suite" is a bit of a stretch!

Brittle tests cause pain in codebases of any size, but they become particularly acute at Google's scale. An individual engineer might easily run thousands of tests in a single day during the course of their work, and a single large-scale change (see Chapter 22) can trigger hundreds of thousands of tests. At this scale, spurious breakages that affect even a small percentage of tests can waste huge amounts of engineering time. Teams at Google vary quite a bit in terms of how brittle their test suites are, but we've identified a few practices and patterns that tend to make tests more robust to change.

Strive for Unchanging Tests

Before talking about patterns for avoiding brittle tests, we need to answer a question: just how often should we expect to need to change a test after writing it? Any time spent updating old tests is time that can't be spent on more valuable work. Therefore, *the ideal test is unchanging*: after it's written, it never needs to change unless the requirements of the system under test change.

What does this look like in practice? We need to think about the kinds of changes that engineers make to production code and how we should expect

tests to respond to those changes. Fundamentally, there are four kinds of changes:

Pure refactorings

When an engineer refactors the internals of a system without modifying its interface, whether for performance, clarity, or any other reason, the system's tests shouldn't need to change. The role of tests in this case is to ensure that the refactoring didn't change the system's behavior. Tests that need to be changed during a refactoring indicate that either the change is affecting the system's behavior and isn't a pure refactoring, or that the tests were not written at an appropriate level of abstraction. Google's reliance on large-scale changes (described in Chapter 22) to do such refactorings makes this case particularly important for us.

New features

When an engineer adds new features or behaviors to an existing system, the system's existing behaviors should remain unaffected. The engineer must write new tests to cover the new behaviors, but they shouldn't need to change any existing tests. As with refactorings, a change to existing tests when adding new features suggest unintended consequences of that feature or inappropriate tests.

Bug fixes

Fixing a bug is much like adding a new feature: the presence of the bug suggests that a case was missing from the initial test suite, and the bug fix should include that missing test case. Again, bug fixes typically shouldn't require updates to existing tests.

Behavior changes

Changing a system's existing behavior is the one case when we expect to have to make updates to the system's existing tests. Note that such changes tend to be significantly more expensive than the other three types. A system's users are likely to rely on its current behavior, and changes to that behavior require coordination with those users to avoid confusion or breakages. Changing a test in this case indicates that we're breaking an explicit contract of the system, whereas changes in the previous cases indicate that we're breaking an unintended contract. Low-level libraries will often invest significant effort in avoiding the need to ever make a behavior change so as not to break their users.

The takeaway is that after you write a test, you shouldn't need to touch that test again as you refactor the system, fix bugs, or add new features. This understanding is what makes it possible to work with a system at scale: expanding it requires writing only a small number of new tests related to the change you're making rather than potentially having to touch every test that has ever been written against the system. Only breaking changes in a system's behavior should require going back to change its tests, and in such situations, the cost of updating those tests tends to be small relative to the cost of updating all of the system's users.

Test via Public APIs

Now that we understand our goal, let's look at some practices for making sure that tests don't need to change unless the requirements of the system being tested change. By far the most important way to ensure this is to write tests that invoke the system being tested in the same way its users would; that is, make calls against its public API rather than its implementation details. If tests work the same way as the system's users, by definition, change that breaks a test might also break a user. As an additional bonus, such tests can serve as useful examples and documentation for users.

Consider Example 12-1, which validates a transaction and saves it to a database.

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction)
  if (isValid(transaction)) {
    saveToDatabase(transaction);
  }
}

private boolean isValid(Transaction t) {
  return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
  String s = t.getSender() + "," + t.getRecipient() + '
  database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int b
```

```
    // Write the balance to the database directly
  }

  public void getAccountBalance(String accountName) {
    // Read transactions from the database to determine t
  }
```

A tempting way to test this code would be to remove the “private” visibility modifiers and test the implementation logic directly, as demonstrated in Example 12-2.

Example 12-2. A naive test of a transaction API’s implementation

```
  @Test
  public void emptyAccountShouldNotBeValid() {
    assertThat(processor.isValid(newTransaction().setSenc
        .isFalse();
  }

  @Test
  public void shouldSaveSerializedData() {
    processor.saveToDatabase(newTransaction()
        .setId(123)
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));
    assertThat(database.get(123)).isEqualTo("me,you,100")
  }
```

This test interacts with the transaction processor in a much different way than its real users would: it peers into the system’s internal state and calls methods that aren’t publicly exposed as part of the system’s API. As a result, the test is brittle, and almost any refactoring of the system under test (such as renaming its methods, factoring them out into a helper class, or changing the serialization format) would cause the test to break, even if such a change would be invisible to the class’s real users.

Instead, the same test coverage can be achieved by testing only against the class’s public API, as shown in Example 12-3.[2]

Example 12-3. Testing the public API

```
@Test
public void shouldTransferFunds() {
  processor.setAccountBalance("me", 150);
  processor.setAccountBalance("you", 20);

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));

  assertThat(processor.getAccountBalance("me")).isEqual
  assertThat(processor.getAccountBalance("you")).isEqua
}

@Test
public void shouldNotPerformInvalidTransactions() {
  processor.setAccountBalance("me", 50);
  processor.setAccountBalance("you", 20);

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));

  assertThat(processor.getAccountBalance("me")).isEqual
  assertThat(processor.getAccountBalance("you")).isEqua
}
```

Tests using only public APIs are, by definition, accessing the system under test in the same manner that its users would. Such tests are more realistic and less brittle because they form explicit contracts: if such a test breaks, it implies that an existing user of the system will also be broken. Testing only these contracts means that you're free to do whatever internal refactoring of the system you want without having to worry about making tedious changes to tests.

It's not always clear what constitutes a "public API," and the question really gets to the heart of what a "unit" is in unit testing. Units can be as small as an individual function or as broad as a set of several related packages/modules. When we say "public API" in this context, we're really talking about the API

exposed by that unit to third parties outside of the team that owns the code. This doesn't always align with the notion of visibility provided by some programming languages; for example, classes in Java might define themselves as "public" to be accessible by other packages in the same unit but are not intended for use by other parties outside of the unit. Some languages like Python have no built-in notion of visibility (often relying on conventions like prefixing private method names with underscores), and build systems like Bazel can further restrict who is allowed to depend on APIs declared public by the programming language.

Defining an appropriate scope for a unit and hence what should be considered the public API is more art than science, but here are some rules of thumb:

- If a method or class exists only to support one or two other classes (i.e., it is a "helper class"), it probably shouldn't be considered its own unit, and its functionality should be tested through those classes instead of directly.
- If a package or class is designed to be accessible by anyone without having to consult with its owners, it almost certainly constitutes a unit that should be tested directly, where its tests access the unit in the same way that the users would.
- If a package or class can be accessed only by the people who own it, but it is designed to provide a general piece of functionality useful in a range of contexts (i.e., it is a "support library"), it should also be considered a unit and tested directly. This will usually create some redundancy in testing given that the support library's code will be covered both by its own tests and the tests of its users. However, such redundancy can be valuable: without it, a gap in test coverage could be introduced if one of the library's users (and its tests) were ever removed.

At Google, we've found that engineers sometimes need to be persuaded that testing via public APIs is better than testing against implementation details. The reluctance is understandable because it's often much easier to write tests focused on the piece of code you just wrote rather than figuring out how that code affects the system as a whole. Nevertheless, we have found it valuable to encourage such practices, as the extra upfront effort pays for itself many times over in reduced maintenance burden. Testing against public APIs won't completely prevent brittleness, but it's the most important thing you can do to ensure that your tests fail only in the event of meaningful changes to your system.

Test State, Not Interactions

Another way that tests commonly depend on implementation details involves not which methods of the system the test calls, but how the results of those calls are verified. In general, there are two ways to verify that a system under test behaves as expected. With *state testing*, you observe the system itself to see what it looks like after invoking with it. With *interaction testing*, you instead check that the system took an expected sequence of actions on its collaborators in response to invoking it. Many tests will perform a combination of state and interaction validation.

Interaction tests tend to be more brittle than state tests for the same reason that it's more brittle to test a private method than to test a public method: interaction tests check *how* a system arrived at its result, whereas usually you should care only *what* the result is. Example 12-4 illustrates a test that uses a test double (explained further in Chapter 13) to verify how a system interacts with a database.

Example 12-4. A brittle interaction test

```
@Test
public void shouldWriteToDatabase() {
  accounts.createUser("foobar");
  verify(database).put("foobar");
}
```

The test verifies that a specific call was made against a database API, but there are a couple different ways it could go wrong:

- If a bug in the system under test causes the record to be deleted from the database shortly after it was written, the test will pass even though we would have wanted it to fail.
- If the system under test is refactored to call a slightly different API to write an equivalent record, the test will fail even though we would have wanted it to pass.

It's much less brittle to directly test against the state of the system, as demonstrated in Example 12-5.

Example 12-5. Testing against state

```
@Test
public void shouldCreateUsers() {
  accounts.createUser("foobar");
  assertThat(accounts.getUser("foobar")).isNotNull();
}
```

This test more accurately expresses what we care about: the state of the system under test after interacting with it.

The most common reason for problematic interaction tests is an over reliance on mocking frameworks. These frameworks make it easy to create test doubles that record and verify every call made against them, and to use those doubles in place of real objects in tests. This strategy leads directly to brittle interaction tests, and so we tend to prefer the use of real objects in favor of mocked objects, as long as the real objects are fast and deterministic.

NOTE

For a more extensive discussion of test doubles and mocking frameworks, when they should be used, and safer alternatives, see Chapter 13.

Writing Clear Tests

Sooner or later, even if we've completely avoided brittleness, our tests will fail. Failure is a good thing—test failures provide useful signals to engineers, and are one of the main ways that a unit test provides value.

Test failures happen for one of two reasons:[3]

- The system under test has a problem or is incomplete. This result is exactly what tests are designed for: alerting you to bugs so that you can fix them.
- The test itself is flawed. In this case, nothing is wrong with the system under test, but the test was specified incorrectly. If this was an existing test rather than one that you just wrote, this means that the test is brittle. The

previous section discussed how to avoid brittle tests, but it's rarely possible to eliminate them entirely.

When a test fails, an engineer's first job is to identify which of these cases the failure falls into and then to diagnose the actual problem. The speed at which the engineer can do so depends on the test's *clarity*. A clear test is one whose purpose for existing and reason for failing is immediately clear to the engineer diagnosing a failure. Tests fail to achieve clarity when their reasons for failure aren't obvious or when it's difficult to figure out why they were originally written. Clear tests also bring other benefits, such as documenting the system under test and more easily serving as a basis for new tests.

Test clarity becomes significant over time. Tests will often outlast the engineers who wrote them, and the requirements and understanding of a system will shift subtly as it ages. It's entirely possible that a failing test might have been written years ago by an engineer no longer on the team, leaving no way to figure out its purpose or how to fix it. This stands in contrast with unclear production code, whose purpose you can usually determine with enough effort by looking at what calls it and what breaks when it's removed. With an unclear test, you might never understand its purpose, since removing the test will have no effect other than (potentially) introducing a subtle hole in test coverage.

In the worst case, these obscure tests just end up getting deleted when engineers can't figure out how to fix them. Not only does removing such tests introduce a hole in test coverage, but it also indicates that the test has been providing zero value for perhaps the entire period it has existed (which could have been years).

For a test suite to scale and be useful over time, it's important that each individual test in that suite be as clear as possible. This section explores techniques and ways of thinking about tests to achieve clarity.

Make Your Tests Complete and Concise

Two high-level properties that help tests achieve clarity are completeness and conciseness. A test is *complete* when its body contains all of the information a reader needs in order to understand how it arrives at its result. A test is *concise* when it contains no other distracting or irrelevant information. Example 12-6 shows a test that is neither complete nor concise:

Example 12-6. An incomplete and cluttered test

```
@Test
public void shouldPerformAddition() {
  Calculator calculator = new Calculator(new RoundingSt
      "unused", ENABLE_COSINE_FEATURE, 0.01, calculusEr
  int result = calculator.calculate(newTestCalculation(
  assertThat(result).isEqualTo(5); // Where did this nu
}
```

The test is passing a lot of irrelevant information into the constructor, and the actual important parts of the test are hidden inside of a helper method. The test can be made more complete by clarifying the inputs of the helper method, and more concise by using another helper to hide the irrelevant details of constructing the calculator, as illustrated in Example 12-7.

Example 12-7. A complete, concise test

```
@Test
public void shouldPerformAddition() {
  Calculator calculator = newCalculator();
  int result = calculator.calculate(newCalculation(2, C
  assertThat(result).isEqualTo(5);
}
```

Ideas we discuss later, especially around code sharing, will tie back to completeness and conciseness. In particular, it can often be worth violating the DRY (Don't Repeat Yourself) principle if it leads to clearer tests. Remember: a *test's body should contain all of the information needed to understand it without containing any irrelevant or distracting information*.

Test Behaviors, Not Methods

The first instinct of many engineers is to try to match the structure of their tests to the structure of their code such that every production method has a corresponding test method. This pattern can be convenient at first, but over time it leads to problems: as the method being tested grows more complex, its test also grows in complexity and becomes more difficult to reason about. For

example, consider the snippet of code in Example 12-8, which displays the results of a transaction.

Example 12-8. A transaction snippet

```
public void displayTransactionResults(User user, Transa
  ui.showMessage("You bought a " + transaction.getItemN
  if (user.getBalance() < LOW_BALANCE_THRESHOLD) {
    ui.showMessage("Warning: your balance is low!");
  }
}
```

It wouldn't be uncommon to find a test covering both of the messages that might be shown by the method, as presented in Example 12-9.

Example 12-9. A method-driven test

```
@Test
public void testDisplayTransactionResults() {
  transactionProcessor.displayTransactionResults(
      newUserWithBalance(
          LOW_BALANCE_THRESHOLD.plus(dollars(2))),
      new Transaction("Some Item", dollars(3)));

  assertThat(ui.getText()).contains("You bought a Some
  assertThat(ui.getText()).contains("your balance is lo
}
```

With such tests, it's likely that the test started out covering only the first method. Later, an engineer expanded the test when the second message was added (violating the idea of unchanging tests that we discussed earlier). This modification sets a bad precedent: as the method under test becomes more complex and implements more functionality, its unit test will become increasingly convoluted and grow more and more difficult to work with.

The problem is that framing tests around methods can naturally encourage unclear tests because a single method often does a few different things under the hood and might have several tricky edge and corner cases. There's a better way: rather than writing a test for each method, write a test for each *behavior*.[4] A behavior is any guarantee that a system makes about how it will respond to

a series of inputs while in a particular state.[5] Behaviors can often be expressed using the words "given," "when," and "then": "*Given* that a bank account is empty, *when* attempting to withdraw money from it, *then* the transaction is rejected." The mapping between methods and behaviors is many-to-many: most nontrivial methods implement multiple behaviors, and some behaviors rely on the interaction of multiple methods. The previous example can be rewritten using behavior-driven tests, as presented in Example 12-10.

Example 12-10. A behavior-driven test

```
@Test
public void displayTransactionResults_showsItemName() {
  transactionProcessor.displayTransactionResults(
      new User(), new Transaction("Some Item"));
  assertThat(ui.getText()).contains("You bought a Some
}

@Test
public void displayTransactionResults_showsLowBalanceWa
  transactionProcessor.displayTransactionResults(
      newUserWithBalance(
          LOW_BALANCE_THRESHOLD.plus(dollars(2))),
      new Transaction("Some Item", dollars(3)));
  assertThat(ui.getText()).contains("your balance is lc
}
```

The extra boilerplate required to split apart the single test is more than worth it, and the resulting tests are much clearer than the original test. Behavior-driven tests tend to be clearer than method-oriented tests for several reasons. First, they read more like natural language, allowing them to be naturally understood rather than requiring laborious mental parsing. Second, they more clearly express cause and effect because each test is more limited in scope. Finally, the fact that each test is short and descriptive makes it easier to see what functionality is already tested and encourages engineers to add new streamlined test methods instead of piling onto existing methods.

Structure tests to emphasize behaviors

Thinking about tests as being coupled to behaviors instead of methods significantly affects how they should be structured. Remember that every behavior has three parts: a "given" component that defines how the system is

set up, a “when” component that defines the action to be taken on the system, and a “then” component that validates the result.[6] Tests are clearest when this structure is explicit. Some frameworks like Cucumber and Spock directly bake in given/when/then. Other languages can use whitespace and optional comments to make the structure stand out, such as that shown in Example 12-11.

Example 12-11. A well-structured test

```
@Test
public void transferFundsShouldMoveMoneyBetweenAccounts
  // Given two accounts with initial balances of $150 a
  Account account1 = newAccountWithBalance(usd(150));
  Account account2 = newAccountWithBalance(usd(20));

  // When transferring $100 from the first to the secor
  bank.transferFunds(account1, account2, usd(100));

  // Then the new account balances should reflect the t
  assertThat(account1.getBalance()).isEqualTo(usd(50));
  assertThat(account2.getBalance()).isEqualTo(usd(120)
}
```

This level of description isn’t always necessary in trivial tests, and it’s usually sufficient to omit the comments and rely on whitespace to make the sections clear. However, explicit comments can make more sophisticated tests easier to understand. This pattern makes it possible to read tests at three levels of granularity:

1. A reader can start by looking at the test method name (discussed below) to get a rough description of the behavior being tested.
2. If that’s not enough, the reader can look at the given/when/then comments for a formal description of the behavior.
3. Finally, a reader can look at the actual code to see precisely how that behavior is expressed.

This pattern is most commonly violated by interspersing assertions among multiple calls to the system under test (i.e., combining the “when” and “then” blocks). Merging the “then” and “when” blocks in this way can make the test less clear because it makes it difficult to distinguish the action being performed from the expected result.

When a test does want to validate each step in a multistep process, it's acceptable to define alternating sequences of when/then blocks. Long blocks can also be made more descriptive by splitting them up with the word "and." Example 12-12 shows what a relatively complex, behavior-driven test might look like.

Example 12-12. Alternating when/then blocks within a test

```
@Test
public void shouldTimeOutConnections() {
  // Given two users
  User user1 = newUser();
  User user2 = newUser();

  // And an empty connection pool with a 10-minute time
  Pool pool = newPool(Duration.minutes(10));

  // When connecting both users to the pool
  pool.connect(user1);
  pool.connect(user2);

  // Then the pool should have two connections
  assertThat(pool.getConnections()).hasSize(2);

  // When waiting for 20 minutes
  clock.advance(Duration.minutes(20));

  // Then the pool should have no connections
  assertThat(pool.getConnections()).isEmpty();

  // And each user should be disconnected
  assertThat(user1.isConnected()).isFalse();
  assertThat(user2.isConnected()).isFalse();
}
```

When writing such tests, be careful to ensure that you're not inadvertently testing multiple behaviors at the same time. Each test should cover only a single behavior, and the vast majority of unit tests require only one "when" and one "then" block.

Name tests after the behavior being tested

Method-oriented tests are usually named after the method being tested (e.g., a test for the `updateBalance` method is usually called `testUpdateBalance`). With more focused behavior-driven tests, we have a lot more flexibility and the chance to convey useful information in the test's name. The test name is very important: it will often be the first or only token visible in failure reports, so it's your best opportunity to communicate the problem when the test breaks. It's also the most straightforward way to express the intent of the test.

A test's name should summarize the behavior it is testing. A good name describes both the actions that are being taken on a system and the expected outcome. Test names will sometimes include additional information like the state of the system or its environment before taking action on it. Some languages and frameworks make this easier than others by allowing tests to be nested within one another and named using strings, such as in Example 12-13, which uses Jasmine.

Example 12-13. Some sample nested naming patterns

```
describe("multiplication", function() {
  describe("with a positive number", function() {
    var positiveNumber = 10;
    it("is positive with another positive number", func
      expect(positiveNumber * 10).toBeGreaterThan(0);
    });
    it("is negative with a negative number", function()
      expect(positiveNumber * -10).toBeLessThan(0);
    });
  });
  describe("with a negative number", function() {
    var negativeNumber = 10;
    it("is negative with a positive number", function()
      expect(negativeNumber * 10).toBeLessThan(0);
    });
    it("is positive with another negative number", func
      expect(negativeNumber * -10).toBeGreaterThan(0);
    });
  });
});
```

Other languages require us to encode all of this information in a method name, leading to method naming patterns like that shown in Example 12-14.

Example 12-14. Some sample method naming patterns

```
multiplyingTwoPositiveNumbersShouldReturnAPositiveNumbe
multiply_postiveAndNegative_returnsNegative
divide_byZero_throwsException
```

Names like this are much more verbose than we'd normally want to write for methods in production code, but the use case is different: we never need to write code that calls these, and their names frequently need to be read by humans in reports. Hence, the extra verbosity is warranted.

Many different naming strategies are acceptable so long as they're used consistently within a single test class. A good trick if you're stuck is to try starting the test name with the word "should." When taken with the name of the class being tested, this naming scheme allows the test name to be read as a sentence. For example, a test of a `BankAccount` class named `shouldNotAllowWithdrawalsWhenBalanceIsEmpty` can be read as "BankAccount should not allow withdrawals when balance is empty." By reading the names of all the test methods in a suite, you should get a good sense of the behaviors implemented by the system under test. Such names also help ensure that the test stays focused on a single behavior: if you need to use the word "and" in a test name, there's a good chance that you're actually testing multiple behaviors and should be writing multiple tests!

Don't Put Logic in Tests

Clear tests are trivially correct upon inspection; that is, it is obvious that a test is doing the correct thing just from glancing at it. This is possible in test code because each test needs to handle only a particular set of inputs, whereas production code must be generalized to handle any input. For production code, we're able to write tests that ensure complex logic is correct. But test code doesn't have that luxury—if you feel like you need to write a test to verify your test, something has gone wrong!

Complexity is most often introduced in the form of *logic*. Logic is defined via the imperative parts of programming languages such as operators, loops, and

conditionals. When a piece of code contains logic, you need to do a bit of mental computation to determine its result instead of just reading it off of the screen. It doesn't take much logic to make a test more difficult to reason about. For example, does the test in Example 12-15 look correct to you?

Example 12-15. Logic concealing a bug

```
@Test
public void shouldNavigateToAlbumsPage() {
  String baseUrl = "http://photos.google.com/";
  Navigator nav = new Navigator(baseUrl);
  nav.goToAlbumPage();
  assertThat(nav.getCurrentUrl()).isEqualTo(baseUrl + '
}
```

There's not much logic here: really just one string concatenation. But if we simplify the test by removing that one bit of logic, a bug immediately becomes clear, as demonstrated in Example 12-16.

Example 12-16. A test without logic reveals the bug

```
@Test
public void shouldNavigateToPhotosPage() {
  Navigator nav = new Navigator("http://photos.google.c
  nav.goToPhotosPage();
  assertThat(nav.getCurrentUrl())
      .isEqualTo("http://photos.google.com//albums"); /
}
```

When the whole string is written out, we can see right away that we're expecting two slashes in the URL instead of just one. If the production code made a similar mistake, this test would fail to detect a bug. Duplicating the base URL was a small price to pay for making the test more descriptive and meaningful (see the discussion of DAMP versus DRY tests later in this chapter).

If humans are bad at spotting bugs from string concatenation, we're even worse at spotting bugs that come from more sophisticated programming constructs like loops and conditionals. The lesson is clear: in test code, stick to straight-line code over clever logic, and consider tolerating some duplication

when it makes the test more descriptive and meaningful. We'll discuss ideas around duplication and code sharing later in this chapter.

Write Clear Failure Messages

One last aspect of clarity has to do not with how a test is written, but with what an engineer sees when it fails. In an ideal world, an engineer could diagnose a problem just from reading its failure message in a log or report without ever having to look at the test itself. A good failure message contains much the same information as the test's name: it should clearly express the desired outcome, the actual outcome, and any relevant parameters.

Here's an example of a bad failure message:

```
Test failed: account is closed
```

Did the test fail because the account was closed, or was the account expected to be closed and the test failed because it wasn't? A better failure message clearly distinguishes the expected from the actual state and gives more context about the result:

```
Expected an account in state CLOSED, but got account:
    <{name: "my-account", state: "OPEN"}
```

Good libraries can help make it easier to write useful failure messages. Consider the assertions in Example 12-17 in a Java test, the first of which uses classical JUnit asserts, and the second of which uses Truth, an assertion library developed by Google:

Example 12-17. An assertion using the Truth library

```
Set<String> colors = ImmutableSet.of("red", "green", "b
assertTrue(colors.contains("orange"));  // JUnit
assertThat(colors).contains("orange");  // Truth
```

Because the first assertion only receives a Boolean value, it is only able to give a generic error message like "expected <true> but was <false>," which isn't very informative in a failing test output. Because the second assertion

explicitly receives the subject of the assertion, it is able to give a much more useful error message: AssertionError: <[red, green, blue]> should have contained <orange>."

Not all languages have such helpers available, but it should always be possible to manually specify the important information in the failure message. For example, test assertions in Go conventionally look like Example 12-18.

Example 12-18. A test assertion in Go

```
result := Add(2, 3)
if result != 5 {
  t.Errorf("Add(2, 3) = %v, want %v", result, 5)
}
```

Tests and Code Sharing: DAMP, Not DRY

One final aspect of writing clear tests and avoiding brittleness has to do with code sharing. Most software attempts to achieve a principle called DRY—"Don't Repeat Yourself." DRY states that software is easier to maintain if every concept is canonically represented in one place and code duplication is kept to a minimum. This approach is especially valuable in making changes easier because an engineer needs to update only one piece of code rather than tracking down multiple references. The downside to such consolidation is that it can make code unclear, requiring readers to follow chains of references to understand what the code is doing.

In normal production code, that downside is usually a small price to pay for making code easier to change and work with. But this cost/benefit analysis plays out a little differently in the context of test code. Good tests are designed to be stable, and in fact you usually *want* them to break when the system being tested changes. So DRY doesn't have quite as much benefit when it comes to test code. At the same time, the costs of complexity are greater for tests: production code has the benefit of a test suite to ensure that it keeps working as it becomes complex, whereas tests must stand by themselves, risking bugs if they aren't self-evidently correct. As mentioned earlier, something has gone

wrong if tests start becoming complex enough that it feels like they need their own tests to ensure that they're working properly.

Instead of being completely DRY, test code should often strive to be DAMP—that is, to promote "Descriptive And Meaningful Phrases." A little bit of duplication is OK in tests so long as that duplication makes the test simpler and clearer. To illustrate, Example 12-19 presents some tests that are far too DRY.

Example 12-19. A test that is too DRY

```
@Test
public void shouldAllowMultipleUsers() {
  List<User> users = createUsers(false, false);
  Forum forum = createForumAndRegisterUsers(users);
  validateForumAndUsers(forum, users);
}

@Test
public void shouldNotAllowBannedUsers() {
  List<User> users = createUsers(true);
  Forum forum = createForumAndRegisterUsers(users);
  validateForumAndUsers(forum, users);
}

// Lots more tests...

private static List<User> createUsers(boolean... bannec
  List<User> users = new ArrayList<>();
  for (boolean isBanned : banned) {
    users.add(newUser()
        .setState(isBanned ? State.BANNED : State.NORMA
        .build());
  }
  return users;
}

private static Forum createForumAndRegisterUsers(List<U
  Forum forum = new Forum();
  for (User user : users) {
    try {
      forum.register(user);
    } catch(BannedUserException ignored) {}
  }
```

```
    return forum;
  }

  private static void validateForumAndUsers(Forum forum,
    assertThat(forum.isReachable()).isTrue();
    for (User user : users) {
      assertThat(forum.hasRegisteredUser(user))
          .isEqualTo(user.getState() == State.BANNED);
    }
  }
```

The problems in this code should be apparent based on the previous discussion of clarity. For one, although the test bodies are very concise, they are not complete: important details are hidden away in helper methods that the reader can't see without having to scroll to a completely different part of the file. Those helpers are also full of logic that makes them more difficult to verify at a glance (did you spot the bug?). The test becomes much clearer when it's rewritten to use DAMP, as shown in Example 12-20.

Example 12-20. Tests should be DAMP

```
  @Test
  public void shouldAllowMultipleUsers() {
    User user1 = newUser().setState(State.NORMAL).build()
    User user2 = newUser().setState(State.NORMAL).build()

    Forum forum = new Forum();
    forum.register(user1);
    forum.register(user2);

    assertThat(forum.hasRegisteredUser(user1)).isTrue();
    assertThat(forum.hasRegisteredUser(user2)).isTrue();
  }

  @Test
  public void shouldNotRegisterBannedUsers() {
    User user = newUser().setState(State.BANNED).build();

    Forum forum = new Forum();
    try {
      forum.register(user);
    } catch(BannedUserException ignored) {}
```

```
    assertThat(forum.hasRegisteredUser(user)).isFalse();
  }
```

These tests have more duplication, and the test bodies are a bit longer, but the extra verbosity is worth it. Each individual test is far more meaningful and can be understood entirely without leaving the test body. A reader of these tests can feel confident that the tests do what they claim to do and aren't hiding any bugs.

DAMP is not a replacement for DRY; it is complementary to it. Helper methods and test infrastructure can still help make tests clearer by making them more concise, factoring out repetitive steps whose details aren't relevant to the particular behavior being tested. The important point is that such refactoring should be done with an eye toward making tests more descriptive and meaningful, and not solely in the name of reducing repetition. The rest of this section will explore common patterns for sharing code across tests.

Shared Values

Many tests are structured by defining a set of shared values to be used by tests and then by defining the tests that cover various cases for how these values interact. Example 12-21 illustrates what such tests look like.

Example 12-21. Shared values with ambiguous names

```
  private static final Account ACCOUNT_1 = Account.newBui
      .setState(AccountState.OPEN).setBalance(50).build()

  private static final Account ACCOUNT_2 = Account.newBui
      .setState(AccountState.CLOSED).setBalance(0).build(

  private static final Item ITEM = Item.newBuilder()
      .setName("Cheeseburger").setPrice(100).build();

  // Hundreds of lines of other tests...

  @Test
  public void canBuyItem_returnsFalseForClosedAccounts()
    assertThat(store.canBuyItem(ITEM, ACCOUNT_1)).isFalse
  }

  @Test
```

```
public void canBuyItem_returnsFalseWhenBalanceInsuffici
  assertThat(store.canBuyItem(ITEM, ACCOUNT_2)).isFalse
}
```

This strategy can make tests very concise, but it causes problems as the test suite grows. For one, it can be difficult to understand why a particular value was chosen for a test. In Example 12-21, the test names fortunately clarify which scenarios are being tested, but you still need to scroll up to the definitions to confirm that `ACCOUNT_1` and `ACCOUNT_2` are appropriate for those scenarios. More descriptive constant names (e.g., `CLOSED_ACCOUNT` and `ACCOUNT_WITH_LOW_BALANCE`) help a bit, but they still make it more difficult to see the exact details of the value being tested, and the ease of reusing these values can encourage engineers to do so even when the name doesn't exactly describe what the test needs.

Engineers are usually drawn to using shared constants because constructing individual values in each test can be verbose. A better way to accomplish this goal is to construct data using helper methods (see Example 12-22) that require the test author to specify only values they care about, and setting reasonable defaults[7] for all other values. This construction is trivial to do in languages that support named parameters, but languages without named parameters can use constructs such as the *Builder* pattern to emulate them (often with the assistance of tools such as AutoValue):

Example 12-22. Shared values using helper methods

```
# A helper method wraps a constructor by defining arbit
# each of its parameters.
def newContact(
    firstName="Grace", lastName="Hopper", phoneNumber='
  return Contact(firstName, lastName, phoneNumber)

# Tests call the helper, specifying values for only the
# care about.
def test_fullNameShouldCombineFirstAndLastNames(self):
  def contact = newContact(firstName="Ada", lastName="L
  self.assertEqual(contact.fullName(), "Ada Lovelace")

// Languages like Java that don't support named paramet
// by returning a mutable "builder" object that represe
// construction.
private static Contact.Builder newContact() {
```

```
    return Contact.newBuilder()
      .setFirstName("Grace")
      .setLastName("Hopper")
      .setPhoneNumber("555-123-4567");
  }

  // Tests then call methods on the builder to overwrite
  // that they care about, then call build() to get a rea
  // builder.
  @Test
  public void fullNameShouldCombineFirstAndLastNames() {
    Contact contact = newContact()
        .setFirstName("Ada")
        .setLastName("Lovelace")
        .build();
    assertThat(contact.getFullName()).isEqualTo("Ada Love
  }
```

Using helper methods to construct these values allows each test to create the exact values it needs without having to worry about specifying irrelevant information or conflicting with other tests.

Shared Setup

A related way that tests shared code is via setup/initialization logic. Many test frameworks allow engineers to define methods to execute before each test in a suite is run. Used appropriately, these methods can make tests clearer and more concise by obviating the repetition of tedious and irrelevant initialization logic. Used inappropriately, these methods can harm a test's completeness by hiding important details in a separate initialization method.

The best use case for setup methods is to construct the object under tests and its collaborators. This is useful when the majority of tests don't care about the specific arguments used to construct those objects and can let them stay in their default states. The same idea also applies to stubbing return values for test doubles, which is a concept that we explore in more detail in Chapter 13.

One risk in using setup methods is that they can lead to unclear tests if those tests begin to depend on the particular values used in setup. For example, the test in Example 12-23 seems incomplete because a reader of the test needs to go hunting to discover where the string "Donald Knuth" came from.

Example 12-23. Dependencies on values in setup methods

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
  nameService = new NameService();
  nameService.set("user1", "Donald Knuth");
  userStore = new UserStore(nameService);
}

// [... hundreds of lines of tests ...]

@Test
public void shouldReturnNameFromService() {
  UserDetails user = userStore.get("user1");
  assertThat(user.getName()).isEqualTo("Donald Knuth");
}
```

Tests like these that explicitly care about particular values should state those values directly, overriding the default defined in the setup method if need be. The resulting test contains slightly more repetition, as shown in Example 12-24, but the result is far more descriptive and meaningful.

Example 12-24. Overriding values in setup mMethods

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
  nameService = new NameService();
  nameService.set("user1", "Donald Knuth");
  userStore = new UserStore(nameService);
}

@Test
public void shouldReturnNameFromService() {
  nameService.set("user1", "Margaret Hamilton");
  UserDetails user = userStore.get("user1");
```

```
    assertThat(user.getName()).isEqualTo("Margaret Hamilt
  }
```

Shared Helpers and Validation

The last common way that code is shared across tests is via "helper methods" called from the body of the test methods. We already discussed how helper methods can be a useful way for concisely constructing test values—this usage is warranted, but other types of helper methods can be dangerous.

One common type of helper is a method that performs a common set of assertions against a system under test. The extreme example is a `validate` method called at the end of every test method, which performs a set of fixed checks against the system under test. Such a validation strategy can be a bad habit to get into because tests using this approach are less behavior driven. With such tests, it is much more difficult to determine the intent of any particular test and to infer what exact case the author had in mind when writing it. When bugs are introduced, this strategy can also make them more difficult to localize because they will frequently cause a large number of tests to start failing.

More focused validation methods can still be useful, however. The best validation helper methods assert a *single conceptual fact* about their inputs, in contrast to general-purpose validation methods that cover a range of conditions. Such methods can be particularly helpful when the condition that they are validating is conceptually simple but requires looping or conditional logic to implement that would reduce clarity were it included in the body of a test method. For example, the helper method in Example 12-25 might be useful in a test covering several different cases around account access.

Example 12-25. A conceptually simple test

```
  private void assertUserHasAccessToAccount(User user, Ac
    for (long userId : account.getUsersWithAccess()) {
      if (user.getId() == userId) {
        return;
      }
    }
```

```
    fail(user.getName() + " cannot access " + account.get
  }
```

Defining Test Infrastructure

The techniques we've discussed so far cover sharing code across methods in a single test class or suite. Sometimes, it can also be valuable to share code across multiple test suites. We refer to this sort of code as *test infrastructure*. Though it is usually more valuable in integration or end-to-end tests, carefully designed test infrastructure can make unit tests much easier to write in some circumstances.

Custom test infrastructure must be approached more carefully than the code sharing that happens within a single test suite. In many ways, test infrastructure code is more similar to production code than it is to other test code given that it can have many callers that depend on it and can be difficult to change without introducing breakages. Most engineers aren't expected to make changes to the common test infrastructure while testing their own features. Test infrastructure needs to be treated as its own separate product, and accordingly, *test infrastructure must always have its own tests*.

Of course, most of the test infrastructure that most engineers use comes in the form of well-known third-party libraries like JUnit. A huge number of such libraries are available, and standardizing on them within an organization should happen as early and universally as possible. For example, Google many years ago mandated Mockito as the only mocking framework that should be used in new Java tests and banned new tests from using other mocking frameworks. This edict produced some grumbling at the time from people comfortable with other frameworks, but today, it's universally seen as a good move that made our tests easier to understand and work with.

Conclusion

Unit tests are one of the most powerful tools that we as software engineers have to make sure that our systems keep working over time in the face of unanticipated changes. But with great power comes great responsibility, and careless use of unit testing can result in a system that requires much more

effort to maintain and takes much more effort to change without actually improving our confidence in said system.

Unit tests at Google are far from perfect, but we've found tests that follow the practices outlined in this chapter to be orders of magnitude more valuable than those that don't. We hope they'll help you to improve the quality of your own tests!

TL;DRs

- Strive for unchanging tests.
- Test via public APIs.
- Test state, not interactions.
- Make your tests complete and concise.
- Test behaviors, not methods.
- Structure tests to emphasize behaviors.
- Name tests after the behavior being tested.
- Don't put logic in tests.
- Write clear failure messages.
- Follow DAMP over DRY when sharing code for tests.

1 Note that this is slightly different from a *flaky test*, which fails nondeterministically without any change to production code.

2 This is sometimes called the "Use the front door first principle."

3 These are also the same two reasons that a test can be "flaky." Either the system under test has a nondeterministic fault, or the test is flawed such that it sometimes fails when it should pass.

4 See *https://testing.googleblog.com/2014/04/testing-on-toilet-test-behaviors-not.html* and *https://dannorth.net/introducing-bdd*.

5 Furthermore, a *feature* (in the product sense of the word) can be expressed as a collection of behaviors.

6 These components are sometimes referred to as "arrange," "act," and "assert."

7 In many cases, it can even be useful to slightly randomize the default values returned for fields that aren't explicitly set. This helps to ensure that two different instances

won’t accidentally compare as equal, and makes it more difficult for engineers to hardcode dependencies on the defaults.