# Chapter 13. Test Doubles

*Written by Andrew Trenk and Dillon Bly*

*Edited by Tom Manshreck*

Unit tests are a critical tool for keeping developers productive and reducing defects in code. Although they can be easy to write for simple code, writing them becomes difficult as code becomes more complex.

For example, imagine trying to write a test for a function that sends a request to an external server and then stores the response in a database. Writing a handful of tests might be doable with some effort. But if you need to write hundreds or thousands of tests like this, your test suite will likely take hours to run, and could become flaky due to issues like random network failures or tests overwriting one another's data.

Test doubles come in handy in such cases. A *test double* is an object or function that can stand in for a real implementation in a test, similar to how a stunt double can stand in for an actor in a movie. The use of test doubles is often referred to as *mocking*, but we avoid that term in this chapter because, as we'll see, that term is also used to refer to more specific aspects of test doubles.

Perhaps the most obvious type of test double is a simpler implementation of an object that behaves similarly to the real implementation, such as an in-memory database. Other types of test doubles can make it possible to validate specific details of your system, such as by making it easy to trigger a rare error condition, or ensuring a heavyweight function is called without actually executing the function's implementation.

The previous two chapters introduced the concept of *small tests* and discussed why they should comprise the majority of tests in a test suite. However, production code often doesn't fit within the constraints of small tests due to communication across multiple processes or machines. Test doubles can be

much more lightweight than real implementations, allowing you to write many small tests that execute quickly and are not flaky.

## The Impact of Test Doubles on Software Development

The use of test doubles introduces a few complications to software development that require some trade-offs to be made. The concepts introduced here are discussed in more depth throughout this chapter:

*Testability*

To use test doubles, a codebase needs to be designed to be *testable*—it should be possible for tests to swap out real implementations with test doubles. For example, code that calls a database needs to be flexible enough to be able to use a test double in place of a real database. If the codebase isn't designed with testing in mind and you later decide that tests are needed, it can require a major commitment to refactor the code to support the use of test doubles.

*Applicability*

Although proper application of test doubles can provide a powerful boost to engineering velocity, their improper use can lead to tests that are brittle, complex, and less effective. These downsides are magnified when test doubles are used improperly across a large codebase, potentially resulting in major losses in productivity for engineers. In many cases, test doubles are not suitable and engineers should prefer to use real implementations instead.

*Fidelity*

*Fidelity* refers to how closely the behavior of a test double resembles the behavior of the real implementation that it's replacing. If the behavior of a test double significantly differs from the real implementation, tests that use the test double likely wouldn't provide much value—for example, imagine trying to write a test with a test double for a database that ignores any data added to the database and always returns empty results. But perfect fidelity might not be feasible; test doubles often need to be vastly simpler than the real implementation in order to be suitable for use in tests. In many situations, it is appropriate to use a test double even without perfect

fidelity. Unit tests that use test doubles often need to be supplemented by larger-scope tests that exercise the real implementation.

## Test Doubles at Google

At Google, we've seen countless examples of the benefits to productivity and software quality that test doubles can bring to a codebase, as well as the negative impact they can cause when used improperly. The practices we follow at Google have evolved over time based on these experiences. Historically, we had few guidelines on how to effectively use test doubles, but best practices evolved as we saw common patterns and antipatterns arise in many teams' codebases.

One lesson we learned the hard way is the danger of overusing mocking frameworks, which allow you to easily create test doubles (we will discuss mocking frameworks in more detail later in this chapter). When mocking frameworks first came into use at Google, they seemed like a hammer fit for every nail—they made it very easy to write highly focused tests against isolated pieces of code without having to worry about how to construct the dependencies of that code. It wasn't until several years and countless tests later that we began to realize the cost of such tests: though these tests were easy to write, we suffered greatly given that they required constant effort to maintain while rarely finding bugs. The pendulum at Google has now begun swinging in the other direction, with many engineers avoiding mocking frameworks in favor of writing more realistic tests.

Even though the practices discussed in this chapter are generally agreed upon at Google, the actual application of them varies widely from team to team. This variance stems from engineers having inconsistent knowledge of these practices, inertia in an existing codebase that doesn't conform to these practices, or teams doing what is easiest for the short term without thinking about the long-term implications.

## Basic Concepts

Before we dive into how to effectively use test doubles, let's cover some of the basic concepts related to them. These build the foundation for best practices that we will discuss later in this chapter.

## An Example Test Double

Imagine an ecommerce site that needs to process credit card payments. At its core, it might have something like the code shown in Example 13-1.

**Example 13-1. A credit card service**

```
class PaymentProcessor {
  private CreditCardService creditCardService;
  ...
  boolean makePayment(CreditCard creditCard, Money amou
    if (creditCard.isExpired()) { return false; }
    boolean success =
        creditCardService.chargeCreditCard(creditCard,
    return success;
  }
}
```

It would be infeasible to use a real credit card service in a test (imagine all the transaction fees from running the test!), but a test double could be used in its place to *simulate* the behavior of the real system. The code in Example 13-2 shows an extremely simple test double.

**Example 13-2. A trivial test double**

```
class TestDoubleCreditCardService implements CreditCarc
 @Override
 public boolean chargeCreditCard(CreditCard creditCard,
   return true;
 }
}
```

Although this test double doesn't look very useful, using it in a test still allows us to test some of the logic in the `makePayment()` method. For example, in Example 13-3, we can validate that the method behaves properly when the credit card is expired because the code path that the test exercises doesn't rely on the behavior of the credit card service.

**Example 13-3. Using the test double**

```
@Test public void cardIsExpired_returnFalse() {
  boolean success = paymentProcessor.makePayment(EXPIRE
  assertThat(success).isFalse();
}
```

The following sections in this chapter will discuss how to make use of test doubles in more complex situations than this one.

## Seams

Code is said to be *testable* if it is written in a way that makes it possible to write unit tests for the code. A *seam* is a way to make code testable by allowing for the use of test doubles—it makes it possible to use different dependencies for the system under test rather than the dependencies used in a production environment.

*Dependency injection* is a common technique for introducing seams. In short, when a class utilizes dependency injection, any classes it needs to use (i.e., the class's *dependencies*) are passed to it rather than instantiated directly, making it possible for these dependencies to be substituted in tests.

Example 13-4 shows an example of dependency injection. Rather than the constructor creating an instance of `CreditCardService`, it accepts an instance as a parameter.

**Example 13-4. Dependency injection**

```
class PaymentProcessor {
  private CreditCardService creditCardService;

  PaymentProcessor(CreditCardService creditCardService)
    this.creditCardService = creditCardService;
  }
  ...
}
```

The code that calls this constructor is responsible for creating an appropriate `CreditCardService` instance. Whereas the production code can pass in an implementation of `CreditCardService` that communicates with an external server, the test can pass in a test double, as demonstrated in Example 13-5.

**Example 13-5. Passing in a test double**

```
PaymentProcessor paymentProcessor =
    new PaymentProcessor(new TestDoubleCreditCardServic
```

To reduce boilerplate associated with manually specifying constructors, automated dependency injection frameworks can be used for constructing object graphs automatically. At Google, Guice and Dagger are automated dependency injection frameworks that are commonly used for Java code.

With dynamically typed languages such as Python or JavaScript, it is possible to dynamically replace individual functions or object methods. Dependency injection is less important in these languages because this capability makes it possible to use real implementations of dependencies in tests while only overriding functions or methods of the dependency that are unsuitable for tests.

Writing testable code requires an upfront investment. It is especially critical early in the lifetime of a codebase because the later testability is taken into account, the more difficult it is to apply to a codebase. Code written without testing in mind typically needs to be refactored or rewritten before you can add appropriate tests.

## Mocking Frameworks

A *mocking framework* is a software library that makes it easier to create test doubles within tests; it allows you to replace an object with a *mock*, which is a test double whose behavior is specified inline in a test. The use of mocking frameworks reduces boilerplate because you don't need to define a new class each time you need a test double.

Example 13-6 demonstrates the use of Mockito, a mocking framework for Java. Mockito creates a test double for `CreditCardService` and instructs

it to return a specific value.

**Example 13-6. Mocking frameworks**

```
class PaymentProcessorTest {
  ...
  PaymentProcessor paymentProcessor;

  // Create a test double of CreditCardService with jus
  @Mock CreditCardService mockCreditCardService;
  @Before public void setUp() {
    // Pass in the test double to the system under test
    paymentProcessor = new PaymentProcessor(mockCreditC
  }
  @Test public void chargeCreditCardFails_returnFalse()
    // Give some behavior to the test double: it will r
    // anytime the chargeCreditCard() method is called.
    // "any()" for the method's arguments tells the tes
    // return false regardless of which arguments are p
    when(mockCreditCardService.chargeCreditCard(any(),
        .thenReturn(false);
    boolean success = paymentProcessor.makePayment(CRED
    assertThat(success).isFalse();
  }
}
```

Mocking frameworks exist for most major programming languages. At Google, we use Mockito for Java, the googlemock component of Googletest for C++, and unittest.mock for Python.

Although mocking frameworks facilitate easier usage of test doubles, they come with some significant caveats given that their overuse will often make a codebase more difficult to maintain. We cover some of these problems later in this chapter.

# Techniques for Using Test Doubles

There are three primary techniques for using test doubles. This section presents a brief introduction to these techniques to give you a quick overview of what they are and how they differ. Later sections in this chapter go into more details on how to effectively apply them.

An engineer who is aware of the distinctions between these techniques is more likely to know the appropriate technique to use when faced with the need to use a test double.

## Faking

A *fake* is a lightweight implementation of an API that behaves similar to the real implementation but isn't suitable for production; for example, an in-memory database. Example 13-7 presents an example of faking.

**Example 13-7. A simple fake**

```
// Creating the fake is fast and easy.
AuthorizationService fakeAuthorizationService =
    new FakeAuthorizationService();
AccessManager accessManager = new AccessManager(fakeAut

// Unknown user IDs shouldn't have access.
assertFalse(accessManager.userHasAccess(USER_ID));

// The user ID should have access after it is added to
// the authorization service.
fakeAuthorizationService.addAuthorizedUser(new User(USE
assertThat(accessManager.userHasAccess(USER_ID)).isTrue
```

Using a fake is often the ideal technique when you need to use a test double, but a fake might not exist for an object you need to use in a test, and writing one can be challenging because you need to ensure that it has similar behavior to the real implementation, now and in the future.

## Stubbing

*Stubbing* is the process of giving behavior to a function that otherwise has no behavior on its own—you specify to the function exactly what values to return (that is, you *stub* the return values).

Example 13-8 illustrates stubbing. The `when(...).thenReturn(...)` method calls from the Mockito mocking framework specify the behavior of the `lookupUser()` method.

**Example 13-8. Stubbing**

```
// Pass in a test double that was created by a mocking
AccessManager accessManager = new AccessManager(mockAut

// The user ID shouldn't have access if null is returne
when(mockAuthorizationService.lookupUser(USER_ID)).ther
assertThat(accessManager.userHasAccess(USER_ID)).isFals

// The user ID should have access if a non-null value i
when(mockAuthorizationService.lookupUser(USER_ID)).ther
assertThat(accessManager.userHasAccess(USER_ID)).isTrue
```

Stubbing is typically done through mocking frameworks to reduce boilerplate that would otherwise be needed for manually creating new classes that hardcode return values.

Although stubbing can be a quick and simple technique to apply, it has limitations, which we'll discuss later in this chapter.

## Interaction Testing

*Interaction testing* is a way to validate *how* a function is called without actually calling the implementation of the function. A test should fail if a function isn't called the correct way—for example, if the function isn't called at all, it's called too many times, or it's called with the wrong arguments.

Example 13-9 presents an instance of interaction testing. The `verify(...)` method from the Mockito mocking framework is used to validate that `lookupUser()` is called as expected.

**Example 13-9. Interaction testing**

```
// Pass in a test double that was created by a mocking
AccessManager accessManager = new AccessManager(mockAut
accessManager.userHasAccess(USER_ID);

// The test will fail if accessManager.userHasAccess(US
// mockAuthorizationService.lookupUser(USER_ID).
verify(mockAuthorizationService).lookupUser(USER_ID);
```

Similar to stubbing, interaction testing is typically done through mocking frameworks. This reduces boilerplate compared to manually creating new classes that contain code to keep track of how often a function is called and which arguments were passed in.

Interaction testing is sometimes called *mocking*. We avoid this terminology in this chapter because it can be confused with mocking frameworks, which can be used for stubbing as well as for interaction testing.

As discussed later in this chapter, interaction testing is useful in certain situations but should be avoided when possible because overuse can easily result in brittle tests.

## Real Implementations

Although test doubles can be invaluable testing tools, our first choice for tests is to use the real implementations of the system under test's dependencies; that is, the same implementations that are used in production code. Tests have higher fidelity when they execute code as it will be executed in production, and using real implementations helps accomplish this.

At Google, the preference for real implementations developed over time as we saw that overuse of mocking frameworks had a tendency to pollute tests with repetitive code that got out of sync with the real implementation and made refactoring difficult. We'll look at this topic in more detail later in this chapter.

Preferring real implementations in tests is known as *classical testing*. There is also a style of testing known as *mockist testing*, in which the preference is to use mocking frameworks instead of real implementations. Even though some people in the software industry practice mockist testing (including the creators of the first mocking frameworks), at Google, we have found that this style of testing is difficult to scale. It requires engineers to follow strict guidelines when designing the system under test, and the default behavior of most engineers at Google has been to write code in a way that is more suitable for the classical testing style.

## Prefer Realism Over Isolation

Using real implementations for dependencies makes the system under test more realistic given that all code in these real implementations will be executed in the test. In contrast, a test that utilizes test doubles isolates the system under test from its dependencies so that the test does not execute code in the dependencies of the system under test.

We prefer realistic tests because they give more confidence that the system under test is working properly. If unit tests rely too much on test doubles, an engineer might need to run integration tests or manually verify that their feature is working as expected in order to gain this same level of confidence. Carrying out these extra tasks can slow down development and can even allow bugs to slip through if engineers skip these tasks entirely when they are too time consuming to carry out compared to running unit tests.

Replacing all dependencies of a class with test doubles arbitrarily isolates the system under test to the implementation that the author happens to put directly into the class and excludes implementation that happens to be in different classes. However, a good test should be independent of implementation—it should be written in terms of the API being tested rather than in terms of how the implementation is structured.

Using real implementations can cause your test to fail if there is a bug in the real implementation. This is good! You *want* your tests to fail in such cases because it indicates that your code won't work properly in production. Sometimes, a bug in a real implementation can cause a cascade of test failures because other tests that use the real implementation might fail, too. But with good developer tools, such as a Continuous Integration (CI) system, it is usually easy to track down the change that caused the failure.

#### CASE STUDY: @DONOTMOCK

At Google, we've seen enough tests that over-rely on mocking frameworks to motivate the creation of the `@DoNotMock` annotation in Java, which is available as part of the ErrorProne static analysis tool. This annotation is a way for API owners to declare, "this type should not be mocked because better alternatives exist."

If an engineer attempts to use a mocking framework to create an instance of a class or interface that has been annotated as `@DoNotMock`, as demonstrated in Example 13-10, they will see an error directing them to use a more suitable test strategy, such as a real implementation or a fake. This annotation is most commonly used for value objects that are simple enough to use as-is, as well as for APIs that have well-engineered fakes available.

**Example 13-10. The @DoNotMock annotation**

```
@DoNotMock("Use SimpleQuery.create() instead of mocking
public abstract class Query {
  public abstract String getQueryValue();
}
```

Why would an API owner care? In short, it severely constrains the API owner's ability to make changes to their implementation over time. As we'll explore later in the chapter, every time a mocking framework is used for stubbing or interaction testing, it duplicates behavior provided by the API.

When the API owner wants to change their API, they might find that it has been mocked thousands or even tens of thousands of times throughout Google's codebase! These test doubles are very likely to exhibit behavior that violates the API contract of the type being mocked—for instance, returning null for a method that can never return null. Had the tests used the real implementation or a fake, the API owner could make changes to their implementation without first fixing thousands of flawed tests.

## How to Decide When to Use a Real Implementation

A real implementation is preferred if it is fast, deterministic, and has simple dependencies. For example, a real implementation should be used for a *value*

*object*. Examples include an amount of money, a date, a geographical address, or a collection class such as a list or a map.

However, for more complex code, using a real implementation often isn't feasible. There might not be an exact answer on when to use a real implementation or a test double given that there are trade-offs to be made, so you need to take the following considerations into account.

### Execution time

One of the most important qualities of unit tests is that they should be fast—you want to be able to continually run them during development so that you can get quick feedback on whether your code is working (and you also want them to finish quickly when run in a CI system). As a result, a test double can be very useful when the real implementation is slow.

How slow is too slow for a unit test? If a real implementation added one millisecond to the running time of each individual test case, few people would classify it as slow. But what if it added 10 milliseconds, 100 milliseconds, 1 second, and so on?

There is no exact answer here—it can depend on whether engineers feel a loss in productivity, and how many tests are using the real implementation (one second extra per test case may be reasonable if there are five test cases, but not if there are 500). For borderline situations, it is often simpler to use a real implementation until it becomes too slow to use, at which point the tests can be updated to use a test double instead.

Parellelization of tests can also help reduce execution time. At Google, our test infrastructure makes it trivial to split up tests in a test suite to be executed across multiple servers. This increases the cost of CPU time, but it can provide a large savings in developer time. We discuss this more in Chapter 18.

Another trade-off to be aware of: using a real implementation can result in increased build times given that the tests need to build the real implementation as well as all of its dependencies. Using a highly scalable build system like Bazel can help because it caches unchanged build artifacts.

### Determinism

A test is *deterministic* if, for a given version of the system under test, running the test always results in the same outcome; that is, the test either always passes or always fails. In contrast, a test is *nondeterministic* if its outcome can change, even if the system under test remains unchanged.

Nondeterminism in tests can lead to flakiness—tests can occasionally fail even when there are no changes to the system under test. As discussed in Chapter 11, flakiness harms the health of a test suite if developers start to distrust the results of the test and ignore failures. If use of a real implementation rarely causes flakiness, it might not warrant a response, because there is little disruption to engineers. But if flakiness happens often, it might be time to replace a real implementation with a test double because doing so will improve the fidelity of the test.

A real implementation can be much more complex compared to a test double, which increases the likelihood that it will be nondeterministic. For example, a real implementation that utilizes multithreading might occasionally cause a test to fail if the output of the system under test differs depending on the order in which the threads are executed.

A common cause of nondeterminism is code that is not hermetic; that is, it has dependencies on external services that are outside the control of a test. For example, a test that tries to read the contents of a web page from an HTTP server might fail if the server is overloaded or if the web page contents change. Instead, a test double should be used to prevent the test from depending on an external server. If using a test double is not feasible, another option is to use a hermetic instance of a server, which has its life cycle controlled by the test. Hermetic instances are discussed in more detail in the next chapter.

Another example of nondeterminism is code that relies on the system clock given that the output of the system under test can differ depending on the current time. Instead of relying on the system clock, a test can use a test double that hardcodes a specific time.

### Dependency construction

When using a real implementation, you need to construct all of its dependencies. For example, an object needs its entire dependency tree to be constructed: all objects that it depends on, all objects that these dependent objects depend on, and so on. A test double often has no dependencies, so constructing a test double can be much simpler compared to constructing a real implementation.

As an extreme example, imagine trying to create the object in the code snippet that follows in a test. It would be time consuming to determine how to construct each individual object. Tests will also require constant maintenance because they need to be updated when the signature of these objects' constructors is modified:

```
Foo foo = new Foo(new A(new B(new C()), new D()), new E
```

It can be tempting to instead use a test double because constructing one can be trivial. For example, this is all it takes to construct a test double when using the Mockito mocking framework:

```
@Mock Foo mockFoo;
```

Although creating this test double is much simpler, there are significant benefits to using the real implementation, as discussed earlier in this section. There are also often significant downsides to overusing test doubles in this way, which we look at later in this chapter. So, a trade-off needs to be made when considering whether to use a real implementation or a test double.

Rather than manually constructing the object in tests, the ideal solution is to use the same object construction code that is used in the production code, such as a factory method or automated dependency injection. To support the use case for tests, the object construction code needs to be flexible enough to be able to use test doubles rather than hardcoding the implementations that will be used for production.

# Faking

If using a real implementation is not feasible within a test, the best option is often to use a fake in its place. A fake is preferred over other test double techniques because it behaves similarly to the real implementation: the system under test shouldn't even be able to tell whether it is interacting with a real implementation or a fake. Example 13-11 illustrates a fake file system.

**Example 13-11. A fake file system**

```
// This fake implements the FileSystem interface. This
// used by the real implementation.
public class FakeFileSystem implements FileSystem {
  // Stores a map of file name to file contents. The fi
  // memory instead of on disk since tests shouldn't ne
  private Map<String, String> files = new HashMap<>();
  @Override
  public void writeFile(String fileName, String content
    // Add the file name and contents to the map.
    files.add(fileName, contents);
  }
  @Override
  public String readFile(String fileName) {
    String contents = files.get(fileName);
    // The real implementation will throw this exceptic
    // file isn't found, so the fake must throw it too.
    if (contents == null) { throw new FileNotFoundExcep
    return contents;
  }
}
```

## Why Are Fakes Important?

Fakes can be a powerful tool for testing: they execute quickly and allow you to effectively test your code without the drawbacks of using real implementations.

A single fake has the power to radically improve the testing experience of an API. If you scale that to a large number of fakes for all sorts of APIs, fakes can provide an enormous boost to engineering velocity across a software organization.

At the other end of the spectrum, in a software organization where fakes are rare, velocity will be slower because engineers can end up struggling with using real implementations that lead to slow and flaky tests. Or engineers might resort to other test double techniques such as stubbing or interaction testing, which, as we'll examine later in this chapter, can result in tests that are unclear, brittle, and less effective.

## When Should Fakes Be Written?

A fake requires more effort and more domain experience to create because it needs to behave similarly to the real implementation. A fake also requires maintenance: whenever the behavior of the real implementation changes, the fake must also be updated to match this behavior. Because of this, the team that owns the real implementation should write and maintain a fake.

If a team is considering writing a fake, a trade-off needs to be made on whether the productivity improvements that will result from the use of the fake outweigh the costs of writing and maintaining it. If there are only a handful of users, it might not be worth their time, whereas if there are hundreds of users, it can result in an obvious productivity improvement.

To reduce the number of fakes that need to be maintained, a fake should typically be created only at the root of the code that isn't feasible for use in tests. For example, if a database can't be used in tests, a fake should exist for the database API itself rather than for each class that calls the database API.

Maintaining a fake can be burdensome if its implementation needs to be duplicated across programming languages, such as for a service that has client libraries that allow the service to be invoked from different languages. One solution for this case is to create a single fake service implementation and have tests configure the client libraries to send requests to this fake service. This approach is more heavyweight compared to having the fake written entirely in memory because it requires the test to communicate across processes. However, it can be a reasonable trade-off to make, as long as the tests can still execute quickly.

## The Fidelity of Fakes

Perhaps the most important concept surrounding the creation of fakes is *fidelity*; in other words, how closely the behavior of a fake matches the behavior of the real implementation. If the behavior of a fake doesn't match the behavior of the real implementation, a test using that fake is not useful—a test might pass when the fake is used, but this same code path might not work properly in the real implementation.

Perfect fidelity is not always feasible. After all, the fake was necessary because the real implementation wasn't suitable in one way or another. For example, a fake database would usually not have fidelity to a real database in terms of hard drive storage because the fake would store everything in memory.

Primarily, however, a fake should maintain fidelity to the API contracts of the real implementation. For any given input to an API, a fake should return the same output and perform the same state changes of its corresponding real implementation. For example, for a real implementation of `database.save(itemId)`, if an item is successfully saved when its ID does not yet exist but an error is produced when the ID already exists, the fake must conform to this same behavior.

One way to think about this is that the fake must have perfect fidelity to the real implementation, but *only from the perspective of the test*. For example, a fake for a hashing API doesn't need to guarantee that the hash value for a given input is exactly the same as the hash value that is generated by the real implementation—tests likely don't care about the specific hash value, only that the hash value is unique for a given input. If the contract of the hashing API doesn't make guarantees of what specific hash values will be returned, the fake is still conforming to the contract even if it doesn't have perfect fidelity to the real implementation.

Other examples where perfect fidelity typically might not be useful for fakes include latency and resource consumption. However, a fake cannot be used if you need to explicitly test for these constraints (e.g., a performance test that verifies the latency of a function call), so you would need to resort to other mechanisms, such as by using a real implementation instead of a fake.

A fake might not need to have 100% of the functionality of its corresponding real implementation, especially if such behavior is not needed by most tests (e.g., error handling code for rare edge cases). It is best to have the fake fail fast in this case; for example, raise an error if an unsupported code path is executed. This failure communicates to the engineer that the fake is not appropriate in this situation.

## Fakes Should Be Tested

A fake must have its *own* tests to ensure that it conforms to the API of its corresponding real implementation. A fake without tests might initially provide realistic behavior, but without tests, this behavior can diverge over time as the real implementation evolves.

One approach to writing tests for fakes involves writing tests against the API's public interface and running those tests against both the real implementation and the fake (these are known as *contract tests*). The tests that run against the real implementation will likely be slower, but their downside is minimized because they need to be run only by the owners of the fake.

## What to Do If a Fake Is Not Available

If a fake is not available, first ask the owners of the API to create one. The owners might not be familiar with the concept of fakes, or they might not realize the benefit they provide to users of an API.

If the owners of an API are unwilling or unable to create a fake, you might be able to write your own. One way to do this is to wrap all calls to the API in a single class and then create a fake version of the class that doesn't talk to the API. Doing this can also be much simpler than creating a fake for the entire API because often you'll need to use only a subset of the API's behavior anyway. At Google, some teams have even contributed their fake to the owners of the API, which has allowed other teams to benefit from the fake.

Finally, you could decide to settle on using a real implementation (and deal with the trade-offs of real implementations that are mentioned earlier in this chapter), or resort to other test double techniques (and deal with the trade-offs that we will mention later in this chapter).

In some cases, you can think of a fake as an optimization: if tests are too slow using a real implementation, you can create a fake to make them run faster. But if the speedup from a fake doesn't outweigh the work it would take to create and maintain the fake, it would be better to stick with using the real implementation.

# Stubbing

As discussed earlier in this chapter, stubbing is a way for a test to hardcode behavior for a function that otherwise has no behavior on its own. It is often a quick and easy way to replace a real implementation in a test. For example, the code in Example 13-12 uses stubbing to simulate the response from a credit card server.

**Example 13-12. Using stubbing to simulate responses**

```
@Test public void getTransactionCount() {
  transactionCounter = new TransactionCounter(mockCredi
  // Use stubbing to return three transactions.
  when(mockCreditCardServer.getTransactions()).thenRetu
      newList(TRANSACTION_1, TRANSACTION_2, TRANSACTION
  assertThat(transactionCounter.getTransactionCount()).
}
```

## The Dangers of Overusing Stubbing

Because stubbing is so easy to apply in tests, it can be tempting to use this technique anytime it's not trivial to use a real implementation. However, overuse of stubbing can result in major losses in productivity for engineers who need to maintain these tests.

### Tests become unclear

Stubbing involves writing extra code to define the behavior of the functions being stubbed. Having this extra code detracts from the intent of the test, and this code can be difficult to understand if you're not familiar with the implementation of the system under test.

A key sign that stubbing isn't appropriate for a test is if you find yourself mentally stepping through the system under test in order to understand why certain functions in the test are stubbed.

### Tests become brittle

Stubbing leaks implementation details of your code into your test. When implementation details in your production code change, you'll need to update your tests to reflect these changes. Ideally, a good test should need to change only if user-facing behavior of an API changes; it should remain unaffected by changes to the API's implementation.

### Tests become less effective

With stubbing, there is no way to ensure the function being stubbed behaves like the real implementation, such as in a statement like that shown in the following snippet that hardcodes part of the contract of the `add()` method (*"If 1 and 2 are passed in, 3 will be returned"*):

```
when(stubCalculator.add(1, 2)).thenReturn(3);
```

Stubbing is a poor choice if the system under test depends on the real implementation's contract because you will be forced to duplicate the details of the contract, and there is no way to guarantee that the contract is correct (i.e., that the stubbed function has fidelity to the real implementation).

Additionally, with stubbing there is no way to store state, which can make it difficult to test certain aspects of your code. For example, if you call `database.save(item)` on either a real implementation or a fake, you might be able to retrieve the item by calling `database.get(item.id())` given that both of these calls are accessing internal state, but with stubbing, there is no way to do this.

### An example of overusing stubbing

Example 13-13 illustrates a test that overuses stubbing.

**Example 13-13. Overuse of stubbing**

```
@Test public void creditCardIsCharged() {
  // Pass in test doubles that were created by a mockir
  paymentProcessor =
      new PaymentProcessor(mockCreditCardServer, mockTr
  // Set up stubbing for these test doubles.
  when(mockCreditCardServer.isServerAvailable()).thenRe
  when(mockTransactionProcessor.beginTransaction()).the
  when(mockCreditCardServer.initTransaction(transactior
  when(mockCreditCardServer.pay(transaction, creditCarc
      .thenReturn(false);
  when(mockTransactionProcessor.endTransaction()).thenF
  // Call the system under test.
  paymentProcessor.processPayment(creditCard, Money.dol
  // There is no way to tell if the pay() method actual
  // transaction, so the only thing the test can do is
  // pay() method was called.
  verify(mockCreditCardServer).pay(transaction, creditC
}
```

Example 13-14 rewrites the same test but avoids using stubbing. Notice how the test is shorter and that implementation details (such as how the transaction processor is used) are not exposed in the test. No special setup is needed because the credit card server knows how to behave.

**Example 13-14. Refactoring a test to avoid stubbing**

```
@Test public void creditCardIsCharged() {
  paymentProcessor =
      new PaymentProcessor(creditCardServer, transactic
  // Call the system under test.
  paymentProcessor.processPayment(creditCard, Money.dol
  // Query the credit card server state to see if the p
  assertThat(creditCardServer.getMostRecentCharge(credi
      .isEqualTo(500);
}
```

We obviously don't want such a test to talk to an external credit card server, so a fake credit card server would be more suitable. If a fake isn't available, another option is to use a real implementation that talks to a hermetic credit

card server, although this will increase the execution time of the tests. (We explore hermetic servers in the next chapter.)

### When Is Stubbing Appropriate?

Rather than a catch-all replacement for a real implementation, stubbing is appropriate when you need a function to return a specific value to get the system under test into a certain state, such as Example 13-12 that requires the system under test to return a non-empty list of transactions. Because a function's behavior is defined inline in the test, stubbing can simulate a wide variety of return values or errors that might not be possible to trigger from a real implementation or a fake.

To ensure its purpose is clear, each stubbed function should have a direct relationship with the test's assertions. As a result, a test typically should stub out a small number of functions because stubbing out many functions can lead to tests that are less clear. A test that requires many functions to be stubbed can be a sign that stubbing is being overused, or that the system under test is too complex and should be refactored.

Note that even when stubbing is appropriate, real implementations or fakes are still preferred because they don't expose implementation details and they give you more guarantees about the correctness of the code compared to stubbing. But stubbing can be a reasonable technique to use, as long as its usage is constrained so that tests don't become overly complex.

## Interaction Testing

As discussed earlier in this chapter, interaction testing is a way to validate how a function is called without actually calling the implementation of the function.

Mocking frameworks make it easy to perform interaction testing. However, to keep tests useful, readable, and resilient to change, it's important to perform interaction testing only when necessary.

## Prefer State Testing Over Interaction Testing

In contrast to interaction testing, it is preferred to test code through *state testing*.

With state testing, you call the system under test and validate that either the correct value was returned or that some other state in the system under test was properly changed. Example 13-15 presents an example of state testing.

**Example 13-15. State testing**

```
@Test public void sortNumbers() {
  NumberSorter numberSorter = new NumberSorter(quicksor
  // Call the system under test.
  List sortedList = numberSorter.sortNumbers(newList(3,
  // Validate that the returned list is sorted. It does
  // sorting algorithm is used, as long as the right re
  assertThat(sortedList).isEqualTo(newList(1, 2, 3));
}
```

Example 13-16 illustrates a similar test scenario but instead uses interaction testing. Note how it's impossible for this test to determine that the numbers are actually sorted, because the test doubles don't know how to sort the numbers—all it can tell you is that the system under test tried to sort the numbers.

**Example 13-16. Interaction testing**

```
@Test public void sortNumbers_quicksortIsUsed() {
  // Pass in test doubles that were created by a mockir
  NumberSorter numberSorter =
      new NumberSorter(mockQuicksort, mockBubbleSort);

  // Call the system under test.
  numberSorter.sortNumbers(newList(3, 1, 2));

  // Validate that numberSorter.sortNumbers() used quic
  // will fail if mockQuicksort.sort() is never called
  // mockBubbleSort is used) or if it's called with the
  verify(mockQuicksort).sort(newList(3, 1, 2));
}
```

At Google, we've found that emphasizing state testing is more scalable; it reduces test brittleness, making it easier to change and maintain code over time.

The primary issue with interaction testing is that it can't tell you that the system under test is working properly; it can only validate that certain functions are called as expected. It requires you to make an assumption about the behavior of the code; for example, "*If `database.save(item)` is called, we assume the item will be saved to the database.*" State testing is preferred because it actually validates this assumption (such as by saving an item to a database and then querying the database to validate that the item exists).

Another downside of interaction testing is that it utilizes implementation details of the system under test—to validate that a function was called, you are exposing to the test that the system under test calls this function. Similar to stubbing, this extra code makes tests brittle because it leaks implementation details of your production code into tests. Some people at Google jokingly refer to tests that overuse interaction testing as *change-detector tests* because they fail in response to any change to the production code, even if the behavior of the system under test remains unchanged.

## When Is Interaction Testing Appropriate?

There are some cases for which interaction testing is warranted:

- You cannot perform state testing because you are unable to use a real implementation or a fake (e.g., if the real implementation is too slow and no fake exists). As a fallback, you can perform interaction testing to validate that certain functions are called. Although not ideal, this does provide some basic level of confidence that the system under test is working as expected.
- Differences in the number or order of calls to a function would cause undesired behavior. Interaction testing is useful because it could be difficult to validate this behavior with state testing. For example, if you expect a caching feature to reduce the number of calls to a database, you can verify that the database object is not accessed more times than expected. Using Mockito, the code might look similar to this:

```
verify(databaseReader, atMostOnce()).selectRecords()
```

Interaction testing is not a complete replacement for state testing. If you are not able to perform state testing in a unit test, strongly consider supplementing your test suite with larger-scoped tests that do perform state testing. For instance, if you have a unit test that validates usage of a database through interaction testing, consider adding an integration test that can perform state testing against a real database. Larger-scope testing is an important strategy for risk mitigation, and we discuss it in the next chapter.

## Best Practices for Interaction Testing

When performing interaction testing, following these practices can reduce some of the impact of the aforementioned downsides.

### Prefer to perform interaction testing only for state-changing functions

When a system under test calls a function on a dependency, that call falls into one of two categories:

*State-changing*

: Functions that have side effects on the world outside the system under test. Examples: `sendEmail()`, `saveRecord()`, `logAccess()`.

*Non-state-changing*

: Functions that don't have side effects; they return information about the world outside the system under test and don't modify anything. Examples: `getUser()`, `findResults()`, `readFile()`.

In general, you should perform interaction testing only for functions that are state-changing. Performing interaction testing for non-state-changing functions is usually redundant given that the system under test will use the return value of the function to do other work that you can assert. The interaction itself is not an important detail for correctness, because it has no side effects.

Performing interaction testing for non-state-changing functions makes your test brittle because you'll need to update the test anytime the pattern of

interactions changes. It also makes the test less readable given that the additional assertions make it more difficult to determine which assertions are important for ensuring correctness of the code. By contrast, state-changing interactions represent something useful that your code is doing to change state somewhere else.

Example 13-17 demonstrates interaction testing on both state-changing and non-state-changing functions.

**Example 13-17. State-changing and non-state-changing interactions**

```
@Test public void grantUserPermission() {
  UserAuthorizer userAuthorizer =
      new UserAuthorizer(mockUserService, mockPermissic
  when(mockPermissionService.getPermission(FAKE_USER)).

  // Call the system under test.
  userAuthorizer.grantPermission(USER_ACCESS);

  // addPermission() is state-changing, so it is reasor
  // interaction testing to validate that it was callec
  verify(mockPermissionDatabase).addPermission(FAKE_USE

  // getPermission() is non-state-changing, so this lir
  // needed. One clue that interaction testing may not
  // getPermission() was already stubbed earlier in thi
  verify(mockPermissionDatabase).getPermission(FAKE_USE
}
```

## Avoid overspecification

In Chapter 12, we discuss why it is useful to test behaviors rather than methods. This means that a test method should focus on verifying one behavior of a method or class rather than trying to verify multiple behaviors in a single test.

When performing interaction testing, we should aim to apply the same principle by avoiding overspecifying which functions and arguments are validated. This leads to tests that are clearer and more concise. It also leads to tests that are resilient to changes made to behaviors that are outside the scope

of each test, so fewer tests will fail if a change is made to a way a function is called.

Example 13-18 illustrates interaction testing with overspecification. The intention of the test is to validate that the user's name is included in the greeting prompt, but the test will fail if unrelated behavior is changed.

**Example 13-18. Overspecified interaction tests**

```
@Test public void displayGreeting_renderUserName() {
  when(mockUserService.getUserName()).thenReturn("Fake
  userGreeter.displayGreeting(); // Call the system unc

  // The test will fail if any of the arguments to setT
  verify(userPrompt).setText("Fake User", "Good morning

  // The test will fail if setIcon() is not called, eve
  // behavior is incidental to the test since it is not
  // validating the user name.
  verify(userPrompt).setIcon(IMAGE_SUNSHINE);
}
```

Example 13-19 illustrates interaction testing with more care in specifying relevant arguments and functions. The behaviors being tested are split into separate tests, and each test validates the minimum amount necessary for ensuring the behavior it is testing is correct.

**Example 13-19. Well-specified interaction tests**

```
@Test public void displayGreeting_renderUserName() {
  when(mockUserService.getUserName()).thenReturn("Fake
  userGreeter.displayGreeting(); // Call the system unc
  verify(userPrompter).setText(eq("Fake User"), any(),
}
@Test public void displayGreeting_timeIsMorning_useMorr
  setTimeOfDay(TIME_MORNING);
  userGreeter.displayGreeting(); // Call the system unc
  verify(userPrompt).setText(any(), eq("Good morning!")
  verify(userPrompt).setIcon(IMAGE_SUNSHINE);
}
```

# Conclusion

We've learned that test doubles are crucial to engineering velocity because they can help comprehensively test your code and ensure that your tests run fast. On the other hand, misusing them can be a major drain on productivity because they can lead to tests that are unclear, brittle, and less effective. This is why it's important for engineers to understand the best practices for how to effectively apply test doubles.

There is often no exact answer regarding whether to use a real implementation or a test double, or which test double technique to use. An engineer might need to make some trade-offs when deciding the proper approach for their use case.

Although test doubles are great for working around dependencies that are difficult to use in tests, if you want to maximize confidence in your code, at some point you still want to exercise these dependencies in tests. The next chapter will cover larger-scope testing, for which these dependencies are used regardless of their suitability for unit tests; for example, even if they are slow or nondeterministic.

# TL;DRs

- A real implementation should be preferred over a test double.
- A fake is often the ideal solution if a real implementation can't be used in a test.
- Overuse of stubbing leads to tests that are unclear and brittle.
- Interaction testing should be avoided when possible: it leads to tests that are brittle because it exposes implementation details of the system under test.