8 Software Testing: Test-Driven Development

Sir Maurice Wilkes (1913–2010) received the 1967 Turing Award for designing and building EDSAC in 1949, one of the first stored-program computers.

It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent finding errors in my own programs.

—Maurice Wilkes, Memoirs of a Computer Pioneer, 1985

Prerequisites and Concepts

The big concepts of this chapter are test creation, test coverage, and levels of testing.

Concepts:

Agile and Plan-and-Document differ sharply in their approaches to testing: when test writing starts, the order in which different kinds of tests are written, and even who does the testing.

Testing in the Agile lifecycle, which follows ***Test-Driven Development*** (***TDD***), is largely the responsibility of the developers rather than a separate Quality Assurance team. Agile testing follows these steps:

- Starting from the acceptance and integration tests derived from ***User stories***, write failing ***unit tests*** that test the nonexistent code you wish you had.
- Write just enough code to pass one such unit test and look for opportunities to ***refactor*** the code before continuing with the next test. Since many test frameworks display failing test output in red and passing test output in green, the iterative sequence of this step and the previous one is called Red–Green–Refactor.
- To isolate the behavior of the code you're testing from the behavior of other classes or methods on which it depends, use ***test doubles***: "stunt doubles" that stand in for real objects in tests, but whose behavior you can closely control. Test doubles are examples of seams, or places where you can change program behavior during testing without changing the source code itself.
- Construct your tests so that they are Fast, Independent, Repeatable, Self-checking, and Timely (FIRST).
- Capture and inspect ***code coverage*** metrics to help determine which parts of your code need more testing.

For the Plan-and-Document lifecycle, you use some of the same concepts in a quite different order and even with different people:

- The program manager assigns programming tasks based on the SRS, so ***unit testing*** starts after coding. ***Quality-Assurance*** (***QA***) testers take over from the developers to perform the higher level tests.
- Top-down, Bottom-up, and Sandwich are options on how to combine the resulting code to perform ***integration testing***. The testing plan and results are documented, such as by following IEEE Standard 829-2008.
- After integration testing, the QA team performs a ***system test*** before releasing it to the customer. Testing stops when a specified level of coverage is reached, such as "95% statement coverage."

- An alternative to testing, used for small critical software, is ***formal methods***. They use formal specifications of correct program behavior that are automatically verified by theorem provers or by exhaustive state search, both of which can go beyond what conventional testing can do.

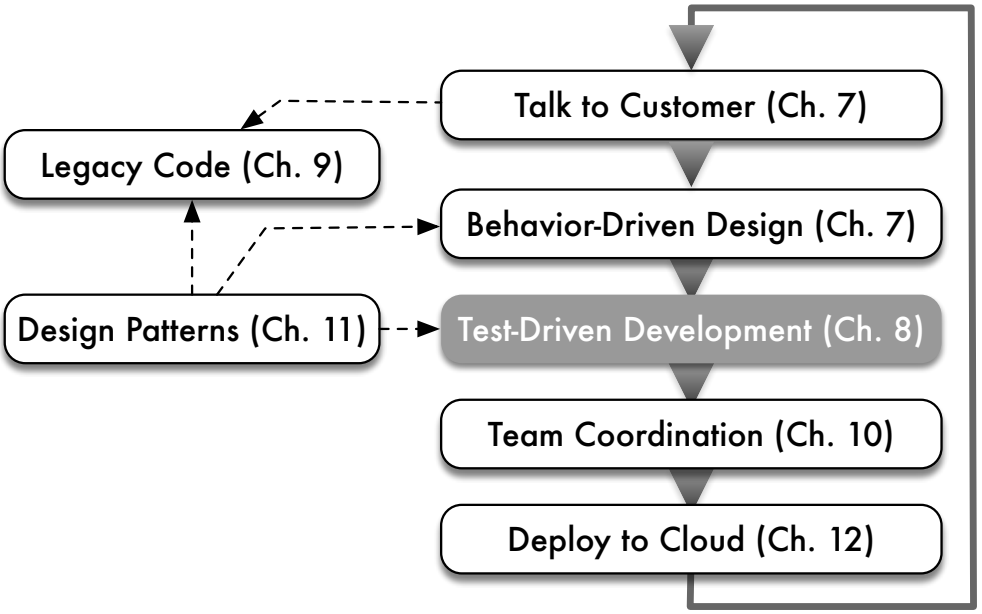

Figure 8.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes unit testing as part of Test-Driven Development.

8.1 FIRST, TDD, and Red–Green–Refactor

Chapter 1 introduced the Agile lifecycle and distinguished two aspects of software assurance: validation ("Did you build the right thing?") and verification ("Did you build the thing right?"). In this chapter, we focus on verification—building the thing right—via software testing as part of the Agile lifecycle. Figure 8.1 highlights the portion of the Agile lifecycle covered in this chapter.

Although testing is only one technique used for verification, we focus on it because its role is often misunderstood, and as a result it doesn't get as much attention as other parts of the software lifecycle. In addition, as we will see, approaching software construction from a test-centric perspective often improves the software's readability and maintainability. In other words, *testable code tends to be clear code, and vice versa.* This insight may take a while to sink in if you are new to TDD, because practicing TDD may feel alien to you. We ask you again to be patient and have faith in the process!

In Agile development, developers do not "toss their code over the wall" to the ***Quality Assurance (QA)*** team, nor do QA engineers extensively exercise the software manually and file bug reports. Instead, Agile developers bear far more responsibility for testing their own code and participating in reviews, while Agile QA responsibilities focus on improving the testing tools infrastructure, helping developers make their code more testable, and verifying that customer-reported bugs are reproducible, as we'll discuss further in Chapter 10. Furthermore, in the vast majority of tests you will write, the test code itself can determine whether the code being tested works or not, without requiring a human to manually check test output or interact with the software.

Even though Agile developers are expected to write their own tests, and those tests are

expected to be automated, there is often a role for some manual testing. For example, ***user acceptance testing*** observes actual users (or QA engineers acting as "typical" users) using the product to determine whether you "built the right thing," and operational acceptance testing may manually try additional scenarios to ensure you "built the thing right." Both can uncover bugs that were previously undetected, some of which can then have automated tests created for them. And some visual aspects of the design, such as whether particular elements on the page render in a visually appealing way, require manual inspection. But in general, modern software quality assurance is the shared responsibility of a whole team following good processes, rather than compartmentalized in a separate group.

In this section we introduce two key ideas that underpin ***Test-driven development*** (TDD): Red–Green–Refactor and making tests FIRST. TDD advocates the use of tests to *drive* the development of code. When TDD is used to create new code, as in this chapter, it is sometimes referred to as *test-first development.* The basic TDD workflow, repeated for each created test, is known as Red–Green–Refactor and proceeds as follows.

When TDD is used to extend or modify legacy code, as in Chapter 9, new tests may be created for code that already exists.

1. Before you write any code, write a test for *one* aspect of the behavior you *expect* the new code will have. Since the code being tested doesn't exist yet, writing the test forces you to think about how you *wish* the code would behave and interact with its collaborators if it did exist. We call this "exercising the code you wish you had."

2. **Red** step: Run the test, and verify that it fails because you haven't yet implemented the code necessary to make it pass (that is, the code you wish you had).

3. **Green** step: Write the *simplest possible* code that causes *this* test to pass without breaking any existing tests.

4. **Refactor** step: Look for opportunities to ***refactor*** either your code or your tests—changing the code's structure to eliminate redundancy or repetition that may have arisen as a result of adding the new code. The tests ensure that your refactoring doesn't introduce bugs.

How do you know when you have completed all necessary tests? If you are using BDD (Chapter 7) to drive your application development, the new code being written is presumably necessary to make one or more Cucumber scenario steps pass. When all steps in a scenario pass, you're done.

Although TDD may feel strange at first, it tends to result in code that is not only well tested, but also more modular and easier to read than code developed separately from tests. While TDD is certainly not the only way to achieve those goals, it is difficult to end up with seriously deficient code if TDD is used correctly.

What about the tests themselves? Five principles for creating good tests are summarized by the acronym FIRST: **F**ast, **I**ndependent, **R**epeatable, **S**elf-checking, and **T**imely.

- **F**ast: it should be easy and quick to run the subset of test cases relevant to your current coding task, to avoid interfering with your train of thought.

- **I**ndependent: The order in which tests run shouldn't matter. More precisely, if no test relies on preconditions created by other tests, we can prioritize running only a subset of tests that cover recent code changes.

- **R**epeatable: test behavior should not depend on external factors such as today's date or on "magic constants" that will break the tests if their values change, as occurred with many 1960s programs when the year 2000 arrived due to the ***Y2K problem***.
- **S**elf-checking: each test should be able to determine on its own whether it passed or failed, rather than relying on humans to check its output.
- **T**imely: tests should be created or updated at the same time as the code being tested. As we'll see, with test-driven development the tests are written *immediately before* the code.

Y2K bug in action This photo was taken on Jan. 3, 2000. (Wikimedia Commons)

Summary

- Besides ensuring software correctness, another reason to use test-centric software construction is that it often improves readability and maintainability: *testable code tends to be clear code, and vice versa.*
- Software QA is a shared responsibility of the whole team: developers write most of their own tests, with QA staff improving the testing environment and handling some kinds of testing that are hard to automate.
- The TDD cycle of Red–Green–Refactor begins with writing a test that fails because the *subject code* being tested doesn't exist yet (Red), then adding the minimum code necessary to pass just that one example (Green), and finally DRYing out and cleaning up the test code (Refactor).
- Tests should be **F**ast to run, with results **I**ndependent of the order in which they are run, thus **R**epeatably giving the same result. Each test should be **S**elf-checking (the test code itself knows whether the test passed or failed), and should be developed in a **T**imely way with respect to the code it tests. Indeed, TDD suggests developing the tests *before* writing the code.

Self-Check 8.1.1

Suppose step 1 in your Cucumber scenario is passing, but step 2 is failing because the code needed is not yet written. If you are practicing strict BDD and TDD, explain why you will necessarily go through one or more cycles of Red–Green–Refactor before step 2 passes.

◇ If the code for step 2 does not yet exist, strict TDD says you should develop that code by *first* writing a focused test for one aspect of the code's behavior, watching that test fail, *then* writing the code to make it pass. ■

8.2 Anatomy of a Test Case: Arrange, Act, Assert

We begin with a few definitions. Following the terminology in the fairly widely used xUnit Test Patterns[1] (Meszaros 2007), we refer to the object being tested as the ***system under test*** (SUT), whether that "object" is a single method, a group of methods, or even the entire application. That is, SUT is defined from the point of view of the test. The goal of a single ***test***

On a value	Example in RSpec
Value equality	`expect(x).to eq('Ruby')`; `expect(x).not_to eq('Ruby')`
Boolean	`expect(x).to be_truthy` # i.e., non-false/non-nil
Regular expression	`expect(s).to match(/YourRegexpHere/)`
Object properties	`expect(a).to be_a_kind_of(Array)`
	`expect(a).to respond_to(:[])`
On a block	**Example in RSpec**
Exception	`expect { Math.sqrt(-1) }.to raise_error(Math::DomainError)`
Side effect	`expect { Review.first.destroy }.to change { Review.count }.by(-1)`

Figure 8.2: A few examples of the kinds of assertions allowed by RSpec. One can invert the sense of any assertion ("Expect `x` not to equal `50`"), as in the first line of the table, but as Chapter 7 warned, negative assertions should be used with caution, since there are many ways for a program *not* to satisfy a particular condition while still not behaving correctly. The use of braces rather than parentheses for the Exception example shows that `expect` can take either an expression, like `x`, or a callable block.

case for some SUT is to check that some specific behavior happens (for example, the return value from a function matches an expected result) or doesn't happen (for example, passing an empty string to a string comparison function doesn't result in an error or exception). A collection of test cases is called a ***test suite***. A code base usually has several test suites, corresponding to different kinds of tests, as we describe later in Section 8.7.

The ISTQB (International Software Testing Qualifications Board) defines[2] *test object* as the thing being tested and SUT as a test object that is a system, but the xUnit terminology is more widely used among Agile developers.

In this section we focus on ***unit tests***, the finest-grained test cases, for which the SUT is a single method. In particular, if the method being tested does not call any other methods to help do its job, we say it is a ***leaf method***. Since even a leaf method may have multiple testable behaviors, a single method may be the subject of multiple test cases.

A unit test is conceptually simple: call a method, and verify some aspect of its behavior. But even leaf methods usually require establishing some preconditions before exercising the code. For example, when testing a method that combines two lists into a single sorted list, we need to create the two lists that will be provided as input. We then exercise the SUT, and finally check whether the particular behavior we were looking for was correctly exhibited.

In general, then, every test case in a suite follows the same structure of *Arrange, Act, Assert*:

1. Arrange: create any necessary preconditions for the test case, such as setting values of variables that affect the behavior of the SUT.
2. Act: exercise the SUT.
3. Assert: verify that the result or behavior matches what was expected.

The general form of an ***assertion*** or expectation is "Expect `expression` to satisfy `predicate`". An example of a simple predicate is an equality check: "Expect the return value of `Math.sqrt(49)` to equal `7`". As Figure 8.2 shows, other kinds of assertions deal with both inspecting output values and checking non-output-value-related behaviors.

The easiest unit tests to write are those for which the SUT is a method that is a ***pure function***—one that has no side effects and whose return value is always the same for the same arguments. The only thing a test case needs to do is choose some inputs, call the method, and check the returned value. For example, consider a hypothetical method `leap?` that accepts an integer and returns a truthy value if and only if that integer corresponds to a leap year (that is, it is either a multiple of 400, or a multiple of 4 but not 100). So, for

Category	Test value
1. A number that is not a multiple of 4 or 100 (and therefore not a multiple of 400)	1973
2. A number that is a multiple of 4, but not of 100 (and therefore not a multiple of 400)	2008
3. A number that is not a multiple of 400, but is a multiple of 100	1900
4. A number that is a multiple of 400	2000

Figure 8.3: A set of categories that completely covers the space of possible nonnegative numeric inputs for a leap year detector, and a possible test value representative of each category. The calculation of a leap year depends only on which category a value is in, not the value itself.

example, 2000 and 2004 are leap years, but 1900 is not. Since exhaustive testing (trying every possible input) is clearly infeasible, how do we choose which input values to use for our test cases?

A good guideline in such cases is to use input values that would cause the calculation performed in the method to follow different code paths. Given the above rule for leap years, by inspection we can deduce four categories, as Figure 8.3 shows.

A "pure" TDD workflow for developing a function that detects leap years might therefore proceed as follows. Choose any value in category 1 above, and write a test case that asserts that the return value of `leap?` is falsy when called with that value. The test fails because `leap?` doesn't yet exist, so write just enough of `leap?` to make that case pass. Next, choose a value in category 2, write the corresponding test, and when it fails, modify `leap?` so that now both tests pass. Continue until all categories are covered.

You might object that `leap?` is such a simple leaf method, and its functionality so well-circumscribed, that you might as well write the entire method at once along with the four test cases, rather than going through the motions of developing each test case incrementally. That's not an unreasonable objection, and as with other Agile practices, "pure" TDD is an ideal to strive for even if you do not always follow it to the letter. But with methods that have more complex behaviors, TDD is a valuable way to proceed methodically.

Chapter 9 considers how to write tests after the fact for code you didn't write or can't easily inspect.

Summary:

- The system under test (SUT), which may be as small as a single method or as large as the whole app, is the subject of a particular test case. A test suite is a full set of tests, which usually includes unit, integration, and perhaps other types of tests.
- The finest-grained tests are unit tests, which test the code in a single method. The simplest unit tests are those for pure leaf functions: deterministic, no side effects, no collaborators or helper methods called. Hence, it is worth structuring your code to expose as much functionality as possible in pure leaf functions.
- One way to select input values for unit tests is to test critical points (values that may influence the code path in the SUT) and values from each noncritical set (set within which the choice of any particular value does not affect control flow).
- Each unit test checks just one behavior, so, for example, each category of input values would get its own test case. Most testing frameworks provide some way to group together examples that test related behaviors and share common setup or teardown phases.
- Each test case follows the same structure: arrange (set up preconditions), act (stimulate the SUT), assert (verify the expected results). The assertion step makes each test **S**elf-checking, eliminating the need for a human programmer to inspect test results.
- Common assertions are checks on values (equality, betweenness, and so on) and checks on behavior (is an exception raised or not).

Self-Check 8.2.1

In a typical unit test, what are the usual boundaries of the System Under Test (SUT): (a) a class, (b) a single method of a class, (c) a single behavior within a single method of a class?

◇ A single method of a class is the typical SUT for a unit test, although if that method has multiple behaviors, each behavior probably merits its own unit test. ■

Competency 8.2.2

For a pure function about whose control paths you have some general knowledge, identify at least one critical value and at least one range of noncritical values that could be used to construct a unit test case.

8.3 Isolating Code: Doubles and Seams

We can distinguish three characteristics (which may occur individually or together) that complicate unit tests:

- The SUT has one or more dependencies, such as other methods it calls to help do its work. Test cases should isolate the SUT from those dependencies.

- The SUT has side effects when executed; that is, it causes a change in application state visible outside the test code itself. Test cases should verify that the correct side effect occurred, which involves inspecting application state outside the SUT.
- The SUT is not a pure function, because its output depends not only on its input but other implicit factors, such as the time of day or a random event. Test cases should control the values of these factors to force the SUT to traverse predictable code paths.

As an example, consider testing a Rails controller action. By design, as we have seen, controller actions shouldn't contain "business logic"—instead they manage communication with the model, calling model methods to do the real work and setting up variables to display information in the view. To make our example relevant to SaaS, consider a hypothetical SaaS app that allows the user to look up a movie in another service's movie database, and display the movie info so the user can write a review. Here is how our hypothetical app works:

1. The `Movie` model has a class (static) method `find_in_tmdb` that makes a call to the API of the external service The Movie Database (TMDb)[3] and returns an array of `Movie` objects, which may be empty if there were no matches.
2. If there are no matches, the controller action should redirect the user back to the search page with an appropriate message.
3. If there is exactly one match, the controller should render a view that allows the user to enter a review for that movie.
4. If there is more than one match, the controller should render a different view that allows the user to specify which movie they want to review.
5. Because the model method relies on calling an external service, the call might fail if the service doesn't respond for some reason. In that case, we assume `Movie.find_in_tmdb` will raise the exception `Movie::ConnectionError`.

Figure 8.4 shows what the above controller action might look like.

How would we unit-test this controller action? The Arrange step consists of preparing `params` to hold some search string. The Act step consists of calling the controller action with that search string. But the Assert step depends on whether the call to `find_in_tmdb` returns an empty array, an array of exactly one match, an array containing more than one match, or raises an exception because of an error communicating with The Movie Database. Indeed, as items 2–5 in the list above show, there are really four test cases required here, and to test each of them, we essentially need to be able to control the *behavior of the call* to `find_in_tmdb`.

Michael Feathers (Feathers 2004) defines a *seam* as "a place where you can alter behavior in your program without editing in that place." In our case, we want to alter (control) the behavior of `find_in_tmdb` but without changing the source code of the controller action. Recall that one ability afforded by metaprogramming is being able to modify code while a program is running. In this case, the strategy would be to *temporarily* modify `find_in_tmdb` so that *instead of calling the real method, it calls a "fake" method whose behavior we control* and can change for each test case.

Such a construction is called a ***method stub***, and is easy to implement in languages that support metaprogramming. The RSpec testing framework provides direct support for this, as

https://gist.github.com/ae42df0d8365b35af8b67f2698d74c3c

```
class MoviesController < ApplicationController
  def review_movie
    search_string = params[:search]
    begin
      matches = Movie.find_in_tmdb(search_string)
      if matches.empty? # nothing was found
        redirect_to review_movie_path, :alert => "No matches."
      elsif matches.length == 1
        @movie = matches[0]
        render 'review_movie'
      else # more than 1 match
        @movies = matches
        render 'select_movie'
      end
    rescue Movie::ConnectionError => err
      redirect_to review_movie_path, :alert => "Error contacting TMDb: #{err.
          message}"
    end
  end
end
```

Figure 8.4: A simple controller method that tries to search a remote database for one or more matches to a movie title. The call to the remote service happens from within the `find_in_tmdb` class method.

https://gist.github.com/94f973152cfb2080ed5875ab1685acaf

```
describe MoviesController do
  describe 'looking up movie' do
    it 'redirects to search page if no match' do
      allow(Movie).to receive(:find_in_tmdb).and_return( [] )
      post 'review_movie', {'search_string' => 'I Am Big Bird'}
      expect(response).to redirect_to(review_movie_path)
    end
  end
end
```

Figure 8.5: This RSpec example (test case) stubs `Movie.find_in_tmdb` to isolate the controller action from its collaborators for the purposes of unit testing.

Figure 8.5 shows: the Arrange part of a test now includes setting up a stub for the method, and specifying that when the stub is called, it should return an empty array, ensuring that `matches.empty?` in line 6 of Figure 8.4 will be true, causing line 7 to be executed next. As is typical for a testing framework, RSpec "un-registers" any stubs after each example (test case) is run, making the stub visible only to that test case and thereby keeping tests **I**ndependent. Later we will show how to group together sets of examples that rely on the same precondition setup, so that tests can be DRY as well.

Keeping in mind that every Ruby function call is a method call on an object, line 4 of Figure 8.5 can be read as follows: "Allow the `Movie` class (which is itself an object) to receive a call to its (class) method `find_in_tmdb`, and return an empty array as the return value of that call." Note that it is *not an error* for `find_in_tmdb` not to be called: the stub setup only specifies what should happen *if* that method is called. If we wanted to express the test condition that the method *must* be called, we would replace `allow` with `expect`. In that case, line 4 would be both an Arrange step defining a stub and an Assert step specifying that the test should fail if the stub isn't actually called. RSpec automatically verifies `expect...to receive` assertions at the end of each example, so the test wouldn't need an extra line to check if the stub was called—simply using `expect` rather than `allow` to set up the stub distinguishes the two cases.

In this case, `receive()` creates a seam by overriding a method in place, without us having to edit the file containing the original method (although in this case, the original method doesn't even exist yet). Seams are also important when it comes to adding new code to your application, but in the rest of this chapter we will see many more examples of seams in testing. Seams are useful in testing because they let us break dependencies between a piece of code we want to test and its collaborators, allowing the collaborators to behave differently under test than they would in real life.

The kind of seam we just described is called a ***method stub*** or simply *stub*, because it is a piece of code that replaces the real method's code with a controllable or fixed behavior for testing purposes. A ***mock object*** or simply *mock* is a simplified "stunt double" of an object that can only mimic a few fixed behaviors of the object, such as returning fixed values for specific attributes. Mocks are useful when a real object would be complex to instantiate because it has other dependencies, yet only a few specific properties of the object are necessary for the SUT to work properly. The term ***test double*** generically covers these and a few other types of seams. Figure 8.6 summarizes typical strategies for using these doubles in various unit-testing scenarios, and Figure 8.7 shows examples of each strategy using RSpec.

Spies are similar to stubs, but they allow a call to the real method to proceed while "recording" the arguments and return values for later inspection.

System under test (SUT)	Testing strategy
Pure leaf function—no side effects, no collaborator methods or classes, same inputs yield same outputs	Assert correct output results for critical values and for arbitrary values in noncritical regions
Relies on results from calling collaborator methods or invoking behaviors on collaborator objects	In Arrange phase, create doubles that "force" the desired behavior by returning prearranged values, raising an exception, and so on
Nondeterministic or time-dependent behavior	In code under test, isolate the nondeterminism in a method call that can be stubbed using a double in the Arrange phase
Has side effects	In Arrange phase, observe the relevant state before executing test code; in Assert phase, observe it again and check for side effect

Figure 8.6: Strategy to properly isolate the SUT when it is not a pure function, not a leaf method, or both.

https://gist.github.com/7ba2377b49261a1d97c3109b99a4dbf0

```
# 1. Pure leaf function: test critical values and noncritical regions
it 'occurs when multiple of 4 but not 100' do
  expect(leap?(2008)).to be_truthy
end
it 'does not occur when multiple of 400' do
  expect(leap?(2000)).to be_falsy
end

# 2. Using doubles for explicit dependencies such as collaborators
#    UI.background() calls Defcon.level() to determine display color
it 'colors the UI red if Defcon is 2 or lower' do
  # Arrange: stub Defcon to return 2
  allow(Defcon).to receive(:level).and_return(2)
  expect(UI.background).to eq('red')        # Act and Assert
end

# 3. Has implicit dependencies such as time
it 'runs backups on Tuesdays' do
  # Arrange: stub Date.today to return Tues 2020-02-04
  allow(Date).to receive(:today).and_return(Time.local(2020,2,4))
  expect(run_backups_today?()).to be_truthy  # Act and Assert
end

# 4. Has side effects (verbose version)
it 'lowers Defcon level by 1' do
  # Arrange: check previous value of state
  before = Defcon.level()
  post_alert("Hostile craft detected")    # Act
  expect(Defcon.level()).to eq(before - 1) # Asset
end

# Shortcut version passing a callable to `expect`
it 'lowers Defcon level by 1' do
  expect { post_alert("Hostile craft detected") }.
    to change { Defcon.level() }.by(-1)
end
```

Figure 8.7: RSpec examples corresponding to Figure 8.6.

Summary

- When testing a method that has external dependencies, for example calling other methods or consuming other objects, we use test doubles to "stand in" for the real methods or objects and allow the test to tightly control the SUT's behavior.
- Test doubles are set up in the Arrange phase of a test case. Stubs are set up to control the return values from collaborator methods, while mocks are set up to mimic just those behaviors of the collaborator object used by the SUT.
- Depending on what behavior is being tested, a test case can specify whether a particular stub *must* be called, that is, if the stub not being called signals a bug.

■ ***Elaboration: Seams in other languages***

In non-object-oriented languages such as C, seams are hard to create. Since all method calls are resolved at link time, usually the developer creates a library containing the "fake" (test double) version of a desired method, and carefully controls library link order to ensure the test-double version is used. Similarly, since C data structures are accessed by reading directly from memory rather than calling accessor methods, data structure seams (mocks) are usually created by using preprocessor directives such as `#ifdef TESTING` to compile the code differently for testing vs. production use.

In statically-typed OO languages like Java, since method calls are resolved at runtime, one way to create seams is to create a subclass of the class under test and override certain methods when compiling against the test harness. Mocking objects is also possible, though the mock object must satisfy the compiler's expectations for a fully-implemented "real" object, even if the mock is doing only a small part of the work that a real object would. The JMock website[4] shows some examples of inserting testing seams in Java.

In dynamic OO languages like Ruby that let you modify classes at runtime, we can create a seam almost anywhere and anytime. RSpec exploits this ability in allowing us to create just the specific mocks and stubs needed by each test, which makes tests easy to write.

Self-Check 8.3.1

Name two likely violations of FIRST that arise when unit tests actually call an external service as part of testing.

◇ The test may no longer be Fast, since it takes much longer to call an external service than to compute locally. The test may no longer be Repeatable, since circumstances beyond our control could affect its outcome, such as the temporary unavailability of the external service. ■

Competency 8.3.2

When unit-testing a method that calls another ("helper") method as part of its method code, identify what test double(s) would be needed to isolate the SUT from the implementation or behavior of the helper method.

Competency 8.3.3

When unit-testing a method that consumes another object as part of its method code, identify what test double(s) would be needed to isolate the SUT from the implementation

or behavior of the needed object.

8.4 Stubbing the Internet

When testing a method that makes a call to an external service via an API, there are many reasons we almost certainly *don't* want to make a real call to that API. One reason is abuse of the service's terms. Another is that making real calls might prevent the test from being **R**epeatable depending on how the remote service responds, and would almost certainly prevent the test from being **F**ast.

Several years ago, a website that hosted academic papers threatened to cut off access from a major US university because a student-authored SaaS app at that university repeatedly made "test" API calls against the real website.

In fact, when testing our own app, all that we really care about is whether the API calls it *would* make are correctly formed—analogous to checking a call to a method stub to make sure the arguments are correct. So the more general question is: Where should we stub external methods when testing an app that makes calls to an external service? In Figure 8.5 we chose to stub the model and mimic the results of the gem's calls to TMDb, but a more robust integration testing approach would instead place the stub "closer" to the remote service. In particular, we could create fixtures—files containing the JSON content returned by actual calls to the service—and arrange to intercept calls to the remote service and return the contents of those fixture files instead. The Webmock[5] gem does exactly this: it stubs out the entire Web except for particular URIs that return a canned response when accessed from a Ruby program. (You can think of Webmock as `allow(...).to receive(...).and_return` for the whole Web.) There's even a companion gem VCR[6] that automates getting a response from the real service, saving the response data in a fixture file, and then "replaying" the fixture when your tests cause the remote service to be "called" by intercepting low-level calls in the Ruby HTTP library.

VCR (for ***Videocassette Recorder***) was an analog-tape video-recording device popular in the 1980s but made obsolete by DVDs in the early 2000s. The `vcr` gem even uses the term *cassette* to refer to the stored server responses that are replayed during tests.

From an integration-testing standpoint, Webmock is the most realistic way to test interactions with a remote service, because the stubbed behavior is "farthest away"—we are stubbing as late as possible in the flow of the request. Therefore, when creating Cucumber scenarios to test external service integration, Webmock is usually the appropriate choice. From a unit testing point of view (as we've adopted in this chapter) it's less compelling, since we are concerned with the correct behavior of specific class methods, and we don't mind stubbing "close by" in order to observe those behaviors in a controlled environment.

Summary:

- To create **F**ast and **R**epeatable test cases for code that communicates with an external service, we use stubs to mimic the service's behavior. `context` blocks can group specs that test different behaviors of the remote service, using `before` blocks to set up necessary stubs or other preconditions to simulate each behavior.
- The question of "where to stub" an external service depends on the purpose of the test. Stubbing "far away" using Webmock is more realistic and appropriate for functional or integration tests; stubbing "close by" in a gem or library that communicates with the remote service is often adequate for low-level unit tests.

Self-Check 8.4.1

Is "stubbing the Internet" in conflict with the advice of Chapter 7 that one should avoid

mocks or stubs in full-system Cucumber scenarios?
◇ Full-system testing should avoid "faking" certain parts of it as we have done using seams in most of this chapter. However, if the "full system" includes interacting with outside services we don't control, such as the interaction with TMDb in this example, we do need a way to "fake" their behavior for testing. ■

8.5 CHIPS: Intro to RSpec on Rails

CHIPS 8.5: Intro to RSpec on Rails

`https://github.com/saasbook/hw-tdd-rspec`
In this assignment, you'll learn to use RSpec and other tools to support test-driven development.

8.6 Fixtures and Factories

Doubles are appropriate when you need a stand-in with a small amount of functionality to isolate the code under test from its dependencies. But suppose you were testing a new instance method of class `Movie` called `name_with_rating` that returns a nicely formatted string showing a movie's title and rating. Clearly, such a method would have to access the `title` and `rating` attributes of a `Movie` instance. You could create a double that knows all that information, and pass that double:

https://gist.github.com/040ab61ab111ebfe71e9c42ff9dc726b

```
1 fake_movie = double('Movie')
2 allow(fake_movie).to receive(:title).and_return('Casablanca')
3 allow(fake_movie).to receive(:rating).and_return('PG')
4 expect(fake_movie.name_with_rating).to eq 'Casablanca (PG)'
```

But since the instance method being tested is part of the `Movie` class itself, it makes sense to use a real object here, since this isn't a case of isolating the test code from collaborator classes.

Where can we get a real `Movie` instance to use in such a test? Most testing frameworks for object-oriented languages support the use of ***factories***—bits of code or declarative descriptions of objects designed to allow rapid creation of full-featured objects (rather than mocks) at testing time. The goal of a factory is to quickly create valid instances of a class using some default attributes that you can selectively override for testing. For example, if you were testing some code that allows a user to write a review for a movie, you might need a valid movie instance to pass to that code. In the above scenario of testing a title-and-rating formatter, you don't care what the movie's release date is, or who directed it; you just need a movie object that is valid and whose title and rating you do know. So you would ask the factory to produce a movie instance whose title and rating you specify, but whose other attributes you don't care about as long as they are valid values.

Don't confuse this use of the term "factory" with the Abstract Factory Pattern discussed in Chapter 11.

You might think this seems like more work than just creating a movie instance directly by calling its constructor. In our simple example, that may be true. But there are two cases in which factories really shine. The first is when the object to be created has many attributes that must be initialized at creation time, even though any particular test case may only care about the specific values of a few of them. For example, the app that manipulates `Movie`

https://gist.github.com/81860a06858f0233566e76324228b185

```
# spec/factories/movie.rb

FactoryBot.define do
  factory :movie do
    title 'A Fake Title' # default values
    rating 'PG'
    release_date { 10.years.ago }
  end
end
```

https://gist.github.com/b12c0c67a9859877324ec85dc7dee37a

```
# in spec/models/movie_spec.rb
describe Movie do
  it 'should include rating and year in full name' do
    # 'build' creates but doesn't save object; 'create' also saves it
    movie = FactoryBot.build(:movie, :title => 'Milk', :rating => 'R')
    expect(movie.name_with_rating).to eq 'Milk (R)'
  end
end
```

Figure 8.8: Using factories rather than fixtures preserves Independence among tests. Frameworks such as FactoryBot (`gem 'factory_bot_rails'` in Gemfile) make factory creation easy.

objects may have validations requiring a movie to have a valid release date or other fields meeting specific criteria, yet the test above doesn't care about the values of those other fields. In such cases, you can ask the factory to create an object in which certain attribute values are specified but others are filled in with valid defaults. The second case is when objects you need to create have has-many or belongs-to relationships with other objects, as Chapter 5 describes. For example, if a Review belongs to a Movie, and you are creating a set of tests to check various behaviors of a Review, you literally cannot create a valid Review instance without creating a Movie instance for it to belong to, even if the tests you are writing don't care about the movie itself. In this case, the Review factory can be configured so that creating a Review also creates a valid Movie to which it belongs. Again, you can either specify a particular Movie object you've created, or let the factory create one with valid default values. Then in your test you can simply ask for a Review object to be created, without having the details of the parent relationship clutter your test code.

The Ruby gem FactoryBot[7] lets you define a factory for any kind of model in your app and create just the objects you need quickly for each test, selectively overriding only certain attributes, as Figure 8.8 shows.

In database-backed MVC apps, one other source of "real" objects for use in tests is ***fixtures***—a set of objects whose existence is guaranteed and fixed, and can be assumed by all test cases. The term *fixture* comes from the manufacturing world: a test fixture is a device that holds or supports the item under test. Since all state in Rails SaaS apps is kept in the database, a fixture file defines a set of objects that is automatically loaded into the test database before tests are run, so you can use those objects in your tests without first setting them up. Rails looks for fixtures in a file containing objects expressed in ***YAML*** (a recursive acronym for YAML Ain't Markup Language), as Figure 8.9 shows. Following convention over configuration, the fixtures for the `Movie` model are loaded from `spec/fixtures/movies.yml`, and are available to your specs via their symbolic names, as Figure 8.9 shows.

But unless used carefully, fixtures can interfere with tests being **I**ndependent, as every test now depends implicitly on the fixture state, so changing the fixtures might change the behavior of tests. In addition, although any given test probably relies on only one or two

https://gist.github.com/ab3223cfdfc9270c8391ea63f5f66571

```
# spec/fixtures/movies.yml
milk_movie:
  id: 1
  title: Milk
  rating: R
  release_date: 2008-11-26

documentary_movie:
  id: 2
  title: Food, Inc.
  release_date: 2008-09-07
```

https://gist.github.com/697fb4ec76a1f1845a72bf1d2341b972

```
# spec/models/movie_spec.rb:

require 'rails_helper.rb'

describe Movie do
  fixtures :movies
  it 'includes rating and year in full name' do
    movie = movies(:milk_movie)
    expect(movie.name_with_rating).to eq('Milk (R)')
  end
end
```

Figure 8.9: Fixtures declared in YAML files (top) are automatically loaded into the test database before each spec is executed (bottom). After each example runs, the database is cleared out and the fixtures reloaded.

fixtures, the union of fixtures required by all tests can become unwieldy. Therefore, fixtures should be used very sparingly if at all, and primarily for truly fixed data that, in production, would not be expected to change while the app is running but needs to be present in order for it to work. For example, at deployment time the app might allow setting the timezone or language in which it operates and storing the preferences in the database, and many aspects of the app might rely on these values being set to a legal value. Having a fixture that "hardwires" some values suitable for testing is reasonable in this case. As a rule of thumb, *use factories for kinds of data that normally change while the app is running, and consider fixtures for data that doesn't change but must be present for the app to work at all.*

Whether you use factories or fixtures, the test framework itself (in our case, RSpec) is responsible for restoring the state of the world to look "pristine" before the next test case runs, just as with doubles. Specifically, the database is completely erased, and any fixtures are then reloaded. Doing this *test teardown* before every single example keeps tests **I**ndependent.

Summary

- When a test needs to operate on a real object rather than a mock, the real object can be created on the fly by a factory or preloaded as a fixture. But beware that fixtures can create subtle interdependencies between tests, breaking **I**ndependence, so best practice is to avoid them except for fixed data that must be present for the app to run at all.
- Tests are a form of internal documentation. RSpec exploits Ruby language features to let you write exceptionally readable test code. Like application code, test code is there for humans, not for the computer, so taking the time to make your tests readable not only deepens your understanding of them but also documents your thoughts more effectively for those who will work with the code after you've moved on.

Self-Check 8.6.1

Suppose a test suite contains a test that adds a model object to a table and then expects to find a certain number of model objects in the table as a result. Explain how the use of fixtures may affect the Independence of the tests in this suite, and how the use of factories can remedy this problem.

◇ If the fixtures file is ever changed so that the number of items initially populating that table changes, this test may suddenly start failing because its assumptions about the initial state of the table no longer hold. In contrast, a factory can be used to quickly create only those objects needed for each test or example group on demand, so no test needs to depend on any global "initial state" of the database. ■

8.7 Coverage Concepts and Types of Tests

How much testing is enough? A poor but unfortunately widely-given answer is "As much as you can before the release deadline." A very coarse-grained alternative is the *code-to-test ratio*, the number of non-comment lines of code divided by number of lines of tests of all types. In production systems, this ratio is usually less than 1, that is, there are more lines of test than lines of app code. The command `rake stats` issued in the root directory of a Rails app computes this ratio based on the number of lines of RSpec tests and Cucumber scenarios.

Another widely-used metric that is more conservative is "when the rate of new bug reports falls below some threshold." This formulation acknowledges that while code can never be proven bug-free, bugs are getting harder to find. But a more precise way to approach the question is to combine such metrics with ***code coverage***. Since the goal of testing is to exercise the subject code in at least the same ways it would be exercised in production, what fraction of those possibilities is actually exercised by the test suite? Surprisingly, measuring coverage is not as straightforward as you might suspect. Figure 8.12 shows a simple fragment of code that we will use to illustrate the definitions of several commonly-used coverage terms.

Sometimes written with a subscript, S_0.

- S0 or Method coverage: Is every method executed at least once by the test suite? Satisfying S0 requires calling `foo` and `bar` at least once each.

Structure of test cases:

- `before(:each) do...end`
  Set up preconditions executed before each spec (use `before(:all)` to do just once, at your own risk)
- `it 'does something' do...end`
  A single example (test case) for one behavior
- `describe 'collection of behaviors' do...end`
  Groups a set of related examples

Mocks and stubs:

- `m=double('movie')`
  Creates a mock object with no predefined methods
- `allow(m).to receive(:rating).and_return('R')`
  Replaces the existing `rating` method on `m`, or defines a new `rating` method if none exists, that returns the canned response `'R'`
- `m=double('movie', :rating=>'R')`
  Shortcut that combines the 2 previous examples
- `allow(Movie).to receive(:find).and_return(@fake_movie)`
  If `Movie.find` is called, `@fake_movie` will be returned; if not called, no error

Useful methods and objects for controller specs: Your specs must be in the `spec/controllers` subdirectory for these methods to be available.

- `post '/movies/create', {title: 'Milk', rating: 'R'}`
  Causes a POST request to `/movies/create` and passes the given hash as the value of `params`. `get`, `put`, `delete` also available.
- `expect(response).to render_template('show')`
  Checks that the controller action renders the `show` template for this controller's model
- `expect(response).to redirect_to(controller: 'movies', action: 'new')`
  Checks that the controller action redirects to `MoviesController#new` rather than rendering a view

Figure 8.10: Some of the most useful RSpec methods introduced in this chapter. See the `rspec.info` documentation site for details and additional methods not listed here.

Assertions on method calls: can also negate by using either `to_not` or `not_to` (whichever reads better) in place of `to`

- `expect(Movie).to receive(:find).exactly(2).times`
  Stubs `Movie.find` and ensures it's called exactly twice. Omit `exactly` if you don't care how many calls; `at_least()` and `at_most()` also available
- `expect(Movie).to receive(:find).with('Milk','R')`
  Checks that `Movie.find` is called with exactly 2 arguments having these values
- `expect(Movie).to receive(:find).with(anything(),anything())`
  Checks that `Movie.find` is called with 2 arguments whose values aren't checked
- `expect(Movie).to receive(:find).with(hash_including title: 'Milk')`
  Checks that `Movie.find` is called with 1 argument that must be a hash (or something that quacks like one) that includes the key `:title` with the value `'Milk'`
- `expect(Movie).to receive(:find).with(no_args())`
  Checks that `Movie.find` is called with zero arguments

Matchers:

- `expect(greeting).to eq 'bonjour'`
  Compares its argument for equality with receiver of assertion
- `expect(value).to be >= 7`
  Compares its argument with the given value; syntactic sugar for `expect(value).to(be.>=(7))`
- `expect(num).to be_within(delta).of(value)`
  Test whether a numeric expression is within a threshold of some numeric value (useful for floating-point calculations)
- `expect(str).to match(/`*regexp*`/)`
  Assert that the string matches the given regexp
- `expect(result).to be_remarkable`
  Asserts that calling `remarkable?` (note question mark) on `result` returns a non-nil value

Figure 8.11: Continuation of summary of useful RSpec methods introduced in this chapter.

https://gist.github.com/811c0b0c8f91fbbb8a4233898450267a

```
1  class MyClass
2    def foo(x,y,z)
3      if x
4        if (y && z) then bar(0) end
5      else
6        bar(1)
7      end
8    end
9    def bar(x) ; @w = x ; end
10 end
```

Figure 8.12: A simple code example to illustrate basic coverage concepts.

- S1 or Call coverage or Entry/Exit coverage: Has each method been called from every place it could be called? Satisfying S1 requires calling `bar` from both line 4 and line 6.
- C0 or Statement coverage: Is every statement of the source code executed at least once by the test suite, counting both branches of a conditional as a single statement? In addition to calling `bar`, satisfying C0 would require calling `foo` at least once with `x` true (otherwise the statement in line 4 will never be executed), and at least once with `y` false.
- C1 or Branch coverage: Has each branch been taken in each direction at least once? Satisfying C1 would require calling `foo` with both false and true values of `x` and with values of `y` and `z` such that `y && z` in line 4 evaluates once to true and once to false. A more stringent condition, *decision coverage*, requires that each *subexpression* that independently affects a conditional expression be evaluated to true and false. In this example, a test would additionally have to separately set `y` and `z` so that the condition `y && z` fails once for `y` being false and once for `z` being false.
- C2 or Path coverage: Has every possible route through the code been executed? In this simple example, where `x,y,z` are treated as booleans, there are 8 possible paths.
- Modified Condition/Decision Coverage (MCDC) combines a subset of the above levels: Every point of entry and exit in the program has been invoked at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome.

Achieving C0 coverage is relatively straightforward, and a goal of 100% C0 coverage is not unreasonable. Achieving C1 coverage is more difficult since test cases must be constructed more carefully to ensure each branch is taken at least once in each direction. C2 coverage is most difficult of all, and not all testing experts agree on the additional value of achieving 100% path coverage. Therefore, code coverage statistics are most valuable to the extent that they highlight undertested or untested parts of the code and show the overall comprehensiveness of your test suite. The SimpleCov gem[8] is easy to configure and measures and displays the C0 and C1 coverage of your specs, allowing you to browse file-by-file to see which lines of your app were exercised by your test suites. If you have multiple suites, such as a set of Cucumber features as well as a set of specs, you must decide whether you need to know only whether a particular line of your app is exercised by *some* test, which may be either a Cucumber scenario or an RSpec example, or whether you need to know *which type* of test exercised it. SimpleCov does the former by default, but its instructions tell you how to do the latter.

This chapter, and the above discussion of coverage, have focused on unit tests. Chapter 7 explained how user stories could become automated acceptance tests; we can think of a Cucumber scenario as both a ***system test***, because it exercises code in many different parts of the application in the same ways a user would, as well as an ***acceptance test***, because a properly-written scenario reflects and verifies the behavior the user said they wanted. In SaaS, such tests may also be called *full-stack tests*, since a typical scenario exercises every part of the app from the browser-based UI to the database. Unlike unit tests, system tests rarely rely on test doubles to isolate behavior; on the contrary, the goal is to simulate real users as closely as possible.

	Unit	Functional	System/Integration
What is tested	One method/class	Several methods/classes	Large chunks of system
Running time	Very fast	Fairly fast	Slow
Error localization	Excellent	Moderate	Poor
Coverage	Excellent	Moderate	Poor
Use of doubles	Frequently	Occasionally	Rarely/never

Figure 8.13: Summary of the differences among unit tests, functional tests, and integration or whole-system tests.

Any test that covers more than one method but is not a full-stack test is generically an ***integration test***. For example, an RSpec test of a controller action would probably stub out calls to the database and bypass the routing mechanism, neither of which is central to testing the controller action itself, but would probably include interactions with mechanisms such as parsing form input, which clearly are outside the controller action.

System and integration tests are important, but insufficient. Their resolution is poor: if an integration test fails, it is harder to pinpoint the cause since the test touches many parts of the code. Especially for system tests, coverage also tends to be poor because even though a single scenario touches many classes, it executes only a few code paths in each class. For the same reason, system and integration tests also tend to take longer to run. On the other hand, while unit tests run quickly and can isolate the subject code with great precision (improving both coverage resolution and error localization), because they rely on fake objects to isolate the subject code, they may mask problems that would only arise in integration tests. In other words, high assurance requires both good coverage and a mix of all three kinds of tests. Figure 8.13 summarizes the relative strengths and weaknesses of different types of tests.

We have focused on testing for correctness ("did you build the thing right"), but in practice, other flavors of tests are part of any comprehensive test suite:

- A ***smoke test*** consists of a minimal attempt to operate the software, to see whether anything is obviously wrong before running the rest of the test suite. For example, if a low-level coding error prevents a SaaS app from displaying its home page or accepting logins, there is no point in running further tests.
- ***Compatibility testing*** is less prominent in SaaS since the app developers control the server environment, but may still be important for testing the app's UI in different browsers. For example, Sauce Labs[9] supports running SaaS integration tests on a variety of browsers and operating systems to check correct client behavior, and even captures a screencast of each run so you can visually check behaviors such as whether the same fonts look good in different browsers.

- ***Regression testing*** ensures that previously-fixed bugs do not reappear. We return to regression tests in Section 10.6.
- ***Performance***, ***stress***, and ***security*** testing are types of ***non-functional testing*** that ensure the software meets these operational criteria, which are particularly important for SaaS. We return to these in Chapter 12.
- ***Accessibility testing*** ensures that the software is usable by persons with disabilities. In SaaS, accessibility testing focuses primarily on the client-side user experience.

Summary

- Static and dynamic measures of coverage, including code-to-test ratio (reported by `rake stats`), C0 or C1 coverage (reported by SimpleCov), and C2 coverage, measure the extent to which your test suite exercises different paths in your code. SimpleCov provides one way to measure coverage for Ruby code, including Rails apps.
- Rather than setting "hard targets" for coverage levels, use coverage reports to identify under-tested parts of your app so you can enhance the test suite accordingly.
- Unit, integration, and system/acceptance tests differ in terms of their running time, resolution (ability to localize errors), ability to exercise a variety of code paths, and ability to perform a "reasonableness check" or so-called ***smoke test*** on the whole application. All three are vital to software assurance.
- In addition to ***functional tests*** that check correctness, we also need ***non-functional tests*** for accessibility, compatibility, security, and performance.

Self-Check 8.7.1

Why does high test coverage not necessarily imply a well-tested application?

◇ Coverage says nothing about the quality of the tests. However, low coverage certainly implies a poorly-tested application. ■

Self-Check 8.7.2

What is the difference between C0 code coverage and code-to-test ratio?

◇ C0 coverage is a *dynamic* measurement of what fraction of all statements are executed by a test suite. Code-to-test ratio is a *static* measurement comparing the total number of lines of code to the total number of lines of tests. ■

8.8 Other Testing Approaches and Terminology

The field of software testing is as broad and long-lived as software engineering and has its own literature. Its range of techniques includes formalisms for proving things about coverage, empirical techniques for selecting which tests to create, and directed-random testing. Depending on an organization's "testing culture," you may hear different terminology than we've used in this chapter. Ammann and Offutt's *Introduction to Software Testing* (Ammann and Offutt 2008) is one of the best comprehensive references on the subject. Their approach is to divide a piece of code into ***basic blocks***, each of which executes from the beginning to the end with no possibility of branching, and then join these basic blocks into a graph in which conditionals in the code result in graph nodes with multiple out-edges. We can then think of testing as "covering the graph": each test case tracks which nodes in the graph it visits, and the fraction of all nodes visited at the end of the test suite is the test coverage. Ammann and Offutt go on to analyze various structural aspects of software from which such graphs can be extracted, and present systematic automated techniques for achieving and measuring coverage of those graphs.

One insight that emerges from this approach is that the levels of testing described in the previous section refer to *control flow coverage*, since they are only concerned with whether

specific parts of the code are executed or not. Another important coverage criterion is *define–use coverage* or *DU-coverage*: given a variable `x` in some program, if we consider every place that `x` is assigned a value and every place that the value of `x` is used, DU-coverage asks what fraction of all *pairs* of define and use sites are exercised by a test suite. This condition is weaker than all-paths coverage but can find errors that control-flow coverage alone would miss.

Another testing term distinguishes ***black-box tests***, whose design is based solely on the software's external specifications, from ***white-box tests*** (also called *glass-box tests*), whose design reflects knowledge about the software's implementation that is not implied by external specifications. For example, the external specification of a hash table might just state that when we store a key/value pair and later read that key, we should get back the stored value. A black-box test would specify a random set of key/value pairs to test this behavior, whereas a white-box test might exploit knowledge about the hash function to construct worst-case test data that results in many hash collisions. Similarly, white-box tests might focus on boundary values—parameter values likely to exercise different parts of the code.

Mutation testing, invented by Ammann and Offutt, is a test-automation technique in which small but syntactically legal changes are automatically made to the program's source code, such as replacing `a+b` with `a-b` or replacing `if (c)` with `if (!c)`. Most such changes should cause at least one test to fail, so a mutation that causes *no* test to fail indicates either a lack of test coverage or a very strange program.

Fuzz testing consists of throwing random data at your application and seeing what breaks. In 2014, Google engineers reported[10] that over a 2-year period, fuzz testing had helped find over 1,000 bugs in the open source video-processing utility `ffmpeg`. Fuzz testing has been particularly useful for finding security vulnerabilities that are missed by both manual code inspection and formal analysis, including stack and buffer overflows and unchecked null pointers. While such memory bugs do not arise in interpreted languages like Ruby and Python or in type-safe and memory-safe compiled languages such as Rust, fuzz testing can still find interesting bugs in SaaS apps. *Random* or *black box fuzzing* either generates completely random data or randomly mutates valid input data, such as changing certain bytes of metadata in a JPEG image to test the robustness of the image decoder. *Smart fuzzing* incorporates knowledge about the app's structure and possibly a way to specify how to construct "realistic but fake" fuzz data. For example, smart-fuzzing SaaS might include randomizing the variables and values occurring in form postings or URIs, or attempting various cross-site scripting or SQL injection attacks, which we'll discuss in Chapter 12. Finally, *white-box fuzzing* uses ***symbolic execution***, which simulates execution of a program observing the conditions under which each branch is taken or not, then generates fuzzed inputs to exercise the branch paths not taken during the simulated execution. White-box fuzzing requires no explicit knowledge of the app's structure and can theoretically provide C2 (all paths) coverage, but in practice the size of the search space is huge, and white-box fuzzing relies on a diverse set of "seed inputs" to be effective. This combination of formal analysis and random directed testing is representative of the current state of the art in thorough software testing. For a short contemporary survey of fuzz testing, see Godefroid's article in Communications of the ACM (Godefroid 2020).

Summary of other testing approaches: We can think of testing as "covering a graph" of possible software behaviors. The graph can represent control flow (basic block coverage), variable assignment and usage (DU-coverage), a space of random inputs (fuzz testing), or a space of possible tests with respect to specific errors in the code (mutation testing). The different approaches are complementary and tend to catch different types of bugs.

Self-Check 8.8.1

The Microsoft Zune music player had an infamous bug that caused all Zunes to "lock up" on December 31, 2008. Later analysis showed that the bug would be triggered on the last day of any leap year. What kinds of tests—black-box, glass-box, mutation, or fuzz—would have been likely to catch this bug?

◇ A glass-box test for the special code paths used for leap years would have been effective. Fuzz testing might have been effective: since the bug occurs roughly once in every 1460 days, a few thousand fuzz tests would likely have found it. ■

8.9 CHIPS: The Acceptance Test/Unit Test Cycle

8.10 The Plan-And-Document Perspective on Testing

The project manager takes the Software Requirements Specification from the requirements planning phase and divides it into the individual program units. Developers then write the code for each unit, and then perform unit tests to make sure they work. In many organizations, quality assurance staff performs the rest of the higher-level tests, such as module, integration, system, and acceptance tests.

There are three options on how to integrate the units and perform integration tests:

1. *Top-down integration* starts with the top of the tree structure showing the dependency among all the units. The advantage of top-down is that you quickly get some of the high level functions working, such as the user interface, which allows stakeholders to offer feedback for the app in time to make changes. The downside is that you have to create many stubs to get the app to limp along in this nascent form.

2. *Bottom-up integration* starts at the bottom of the dependency tree and works up. There is no need for stubs, as you can integrate all the pieces you need for a module. Alas, you don't get an idea how the app will look until you get all the code written and integrated.

3. *Sandwich integration*, not surprisingly, tries to get the best of both worlds by integrating from both ends simultaneously. Thus, you try to reduce the number of stubs by selectively integrating some units bottom-up and try to get the user interface operational sooner by selectively integrating some units top-down.

The next step for the QA testers after integration tests is the system test, as the full app should work. This is the last step before showing it to customers for them to try out. Note that system tests cover both non-functional requirements, such as performance, and functional requirements of features found in the SRS.

One question for plan-and-document is how to decide when testing is complete. Typically, an organization will enforce a standard level of testing coverage before a product is

ready for the customer. Examples might be statement coverage (all statements executed at least once), or all user input opportunities are tested with both good input and problematic input.

In the plan and document process, the final test is for the customers to try the product in their environment to decide whether they will accept the product or not. That is, the aim is validation, not just verification. In Agile development, the customer is involved in trying prototypes of the app early in the process, so there is no separate system test before running the acceptance tests.

As you should expect from the plan-and-document process, documentation plays an important role in testing. Figure 8.14 gives an outline for a test plan based on IEEE Standard 829-2008.

While testing is fundamental to software engineering, quoting another Turing Award winner:

Edsger W. Dijkstra (1930–2002) received the 1972 Turing Award for fundamental contributions to developing programming languages.

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> —Edsger W. Dijkstra

Thus, there has been a great deal of research investigating approaches to verification beyond testing. Collectively, these techniques are known as ***formal methods***. The general strategy is to start with a formal specification and prove that the behavior of the code follows the behavior of that spec. These are mathematical proofs, either done by a person or done by a computer. The two options are ***automatic theorem proving*** or ***model checking***. Theorem proving uses a set of inference rules and a set of logical axioms to produce proofs from scratch. Model checking verifies selected properties by exhaustive search of all possible states that a system could enter during execution.

Because formal methods are so computationally intensive, they tend to be used only when the cost to repair errors is very high, the features are very hard to test, and the item being verified is not too large. Examples include vital parts of hardware like network protocols or safety critical software systems like medical equipment. For formal methods to actually work, the size of the design must be limited: the largest formally verified software to date is an operating system kernel that is less than 10,000 lines of code, and its verification cost about $500 per line of code (Klein et al. 2010).

To put the cost of formal methods in perspective, NASA spent $35M per year to maintain 420,000 lines of code[11] for the space shuttle, or about $80 per line of code per year.

Hence, formal methods are *not* good matches to high-function software that changes frequently, as is generally the case for Software as a Service.

Top-Level Test Plan Outline
1. Introduction
1.1. Document identifier
1.2. Scope
1.3. References
1.4. System overview and key features
1.5. Test overview
1.5.1 Organization
1.5.2 Overall test schedule
1.5.3 Integrity level schema
1.5.4 Resources summary
1.5.5 Responsibilities
1.5.6 Tools, techniques, methods, and metrics

2. Details of the Top-Level Test Plan
2.1. Test processes including definition of test levels
2.1.1 Process: Management
2.1.1.1 Activity: Management of test effort
2.1.2 Process: Acquisition
2.1.2.1 Activity: Acquisition support test
2.1.3 Process: Supply
2.1.3.1 Activity: Planning test
2.1.4 Process: Development
2.1.4.1 Activity: Concept
2.1.4.2 Activity: Requirements
2.1.4.3 Activity: Design
2.1.4.4 Activity: Implementation
2.1.4.5 Activity: Test
2.1.4.6 Activity: Installation/checkout
2.1.5 Process: Operation
2.1.5.1 Activity: Operational test
2.1.6 Process: Maintenance
2.1.6.1 Activity: Maintenance test
2.2. Test documentation requirements
2.3. Test administration requirements
2.4. Test reporting requirements

3. General
3.1. Glossary
3.2. Document change procedures and history

Figure 8.14: Outline of Top-Level Test Plan Documentation that follows the IEEE Standard 829-2008.

Tasks	*In Plan-and-Document*	*In Agile*
Test Plan and Documentation	Software Test Documentation such as IEEE Standard 829-2008	User stories
Order of Coding and Testing	1. Code units 2. Unit test 3. Module test 4. Integration test 5. System test 6. Acceptance test	1. Acceptance test 2. Integration test 3. Module test 4. Unit test 5. Code units
Testers	Developers for unit tests; QA testers for module, integration, system, and acceptance tests	Developers
When Testing Stops	Company policy (e.g., statement coverage, happy and sad user inputs)	All tests pass (green)

Figure 8.15: The relationship between the testing tasks of Plan-and-Document versus Agile methodologies.

Summary: Testing and formal methods reduce the risks of errors in designs.

- Unlike BDD/TDD, the plan-and-document process starts with writing code before you write the tests.
- Developers then perform unit tests.
- Especially in large projects, different people perform the higher-level tests. The integration tests options of putting the units together are top-down, bottom-up, or sandwich.
- Testers do a separate system test to ensure the product passes both functional and non-functional requirements before exposing it to customers for the final acceptance test.
- ***Formal methods*** rely on formal specifications and automated proofs or exhaustive state search to verify more than what testing can do, but they are so expensive to perform that today they are only applicable to small, stable, critical portions of hardware or software.
- Figure 8.15 shows the different test tasks for plan-and-document versus Agile processes.

Self-Check 8.10.1

Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.

◇ Top-down needs stubs to perform the tests, but it lets stakeholders get a feeling for how the app works. Bottom-up does not need stubs, but needs potentially everything written before stakeholders see it work. Sandwich integration works from both ends to try to get both benefits. ■

8.11 Fallacies and Pitfalls

***Fallacy:* 100% test coverage with all tests passing means no bugs.**

There are many reasons this statement can be false. Complete test coverage says nothing about the quality of the individual tests. As well, some bugs may require passing a certain value as a method argument (for example, to trigger a divide-by-zero error), and control flow testing often cannot reveal such a bug. There may be bugs in the interaction between your app and an external service such as TMDb; stubbing out the service so you can perform local testing might mask such bugs.

***Pitfall:* Dogmatically insisting on 100% test coverage all passing (green) before you ship.**

As we saw above, 100% test coverage is not only difficult to achieve at levels higher than C1, but gives no guarantees of bug-freedom even if you do achieve it. Test coverage is a useful tool for estimating the overall comprehensiveness of your test suite, but high confidence requires a variety of testing methods—integration as well as unit, fuzzing as well as hand-constructing test cases, define-use coverage as well as control-flow coverage, mutation testing to expose additional holes in the test strategy, and so on. Indeed, in Chapter 12 we will discuss operational issues such as security and performance, which call for additional testing strategies beyond the correctness-oriented ones described in this chapter.

***Fallacy:* You don't need much test code to be confident in the application.**

While insisting on 100% coverage may be counterproductive, so is going to the other extreme. The *code-to-test ratio* in production systems (lines of noncomment code divided by lines of tests of all types) is usually less than 1, that is, there are more lines of test than lines of app code. As an extreme example, the SQLite database included with Rails contains over 1200 times as much test code as application code[12] because of the wide variety of ways in which it can be used and the wide variety of different kinds of systems on which it must work properly! While there is controversy over how useful a measure the code-to-test ratio is, given the high productivity of Ruby and its superior facilities for DRYing out your test code, a `rake stats` ratio between 0.2 and 0.5 is a reasonable target.

***Pitfall:* Relying too heavily on just one kind of test (unit, functional, integration).**

Unit and functional tests are useful for covering rare corner cases and code paths. They also tell you how well-factored or modular your code is: a module or method that is easy to test has well-circumscribed external dependencies, which in turn reinforces that it can be well tested in isolation. On the other hand, because of that very isolation, even 100% unit test coverage tells you nothing about interactions *among* classes or modules. That's where integration-level tests such as the Cucumber scenarios of Chapter 7 are useful. Such tests touch only a tiny fraction of all possible application paths and exercise only a few behaviors in each method, but they do test the interfaces and interactions among modules. One rule of thumb used at Google and elsewhere (Whittaker et al. 2012) is "70–20–10": 70% short and focused unit tests, 20% functional tests that touch multiple classes, 10% full-stack or integration tests. See Chapter 7 for the complementary pitfall of over-reliance on integration

tests.

Pitfall: **Undertested integration points due to over-stubbing.**

Mocking and stubbing confer many benefits, but they can also hide potential problems at integration points—places where one class or module interacts with another. Suppose `Movie` has some interactions with another class `Moviegoer`, but for the purposes of unit testing `Movie`, all calls to `Moviegoer` methods are stubbed out, and vice versa. Because stubs are written to "fake" the behavior of the collaborating class(es), we no longer know if `Movie` "knows how to talk to" `Moviegoer` correctly. Good coverage with functional and integration tests, which don't stub out all calls across class boundaries, avoids this pitfall.

Pitfall: **Writing tests after the code rather than before.**

Thinking about "the code we wish we had" from the perspective of a test for that code tends to result in code that is testable. This seems like an obvious tautology until you try writing the code first without testability in mind, only to discover that surprisingly often you end up with mock trainwrecks (see next pitfall) when you do try to write the test.

In addition, in the traditional Waterfall lifecycle described in Chapter 1, testing comes after code development, but with SaaS that can be in "public beta" for months, no one would suggest that testing should only begin after the beta period. Writing the tests first, whether for fixing bugs or creating new features, eliminates this pitfall.

Pitfall: **Mock Trainwrecks.**

Mocks exist to help isolate your tests from their collaborators, but what about the collaborators' collaborators? Suppose our `Movie` object has a `pics` attribute that returns a list of images associated with the movie, each of which is a `Picture` object that has a `format` attribute. You're trying to mock a `Movie` object for use in a test, but you realize that the method to which you're passing the `Movie` object is going to expect to call methods on its `pics`, so you find yourself doing something like this:

https://gist.github.com/87f04dad610b1da187de8da024bf02af

```
1 movie = double('Movie', :pics => [double('Picture', :format => 'gif')])
2 expect(Movie.count_pics(movie)).to eq 1
```

This is called a *mock trainwreck*, and it's a sign that the method under test (`count_pics`) has excessive knowledge of the innards of a `Picture`. In Chapters 9 and 11 we'll encounter a set of additional guidelines to help you detect and resolve such ***code smells***.

Pitfall: **Inadvertently creating dependencies regarding the order in which specs are run, for example by using `before(:all)`.**

If you specify actions to be performed only once for a whole group of test cases, you may introduce dependencies among those test cases without noticing. For example, if a `before :all` block sets a variable and test example A changes the variable's value, test example B could come to rely on that change if A is usually run before B. Then B's behavior in the future might suddenly be different if B is run first. (RSpec and most testing tools provide an option for deliberately scrambling the order in which tests are run to help detect such problems.) Therefore it's best to use `before :each` and `after :each` whenever possible.

Pitfall: **Forgetting to re-prep the test database when the schema changes.**

Remember that tests run against a separate copy of the database, not the database used in development (Section 4.2). Therefore, whenever you modify the schema by applying a migration, you must also run `rake db:test:prepare` to apply those changes to the test database; otherwise your tests may fail because the test code doesn't match the schema.

8.12 Concluding Remarks: TDD vs. Conventional Debugging

In this chapter we've used RSpec to develop a method using TDD with unit tests. Although TDD may feel strange at first, most people who try it quickly realize that they already use the unit-testing techniques it calls for, but in a different workflow. Often, a typical developer will write some code, assume it probably works, test it by running the whole application, and hit a bug. As an MIT programmer lamented at the first software engineering conference in 1968: "We build systems like the Wright brothers built airplanes—build the whole thing, push it off a cliff, let it crash, and start over again."

Once a bug has been hit, if inspecting the code doesn't reveal the problem, the typical developer would next try inserting print statements around the suspect area to print out the values of relevant variables or indicate which path of a conditional was followed. The TDD developer would instead write assertions using `expect`.

If the bug still can't be found, the typical developer might isolate part of the code by carefully setting up conditions to skip over method calls they don't care about or change variable values to force the code to go down the suspected buggy path. For example, they might do this by setting a breakpoint using a debugger and manually inspecting or manipulating variable values before continuing past the breakpoint. In contrast, the TDD developer would isolate the suspect code path using stubs and mocks to control what happens when certain methods are called and which direction conditionals will go.

By now, the typical developer is absolutely convinced that he'll certainly find the bug and won't have to repeat this tedious manual process, though this usually turns out to be wrong. The TDD developer has isolated each behavior in its own spec, so repeating the process just means re-running the spec.

In other words: If we write the code first and have to fix bugs, we end up using the same techniques required in TDD, but less efficiently and more manually, hence less productively.

But if we use TDD, bugs can be spotted immediately as the code is written. If our code works the first time, using TDD still gives us a regression test to catch bugs that might creep into this part of the code in the future.

- *How Google Tests Software* (Whittaker et al. 2012) is a rare glimpse into how Google has scaled up and adapted the techniques described in this chapter to instill a culture of testing that is widely admired by its competitors.
- The online RSpec documentation[13] gives complete details and additional features used in advanced testing scenarios.
- *The RSpec Book* (Chelimsky et al. 2010) is the definitive published reference to RSpec and includes examples of features, mechanisms and best practices that go far beyond this introduction.

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 0521880386.

D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesøy. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends (The Facets of Ruby Series)*. Pragmatic Bookshelf, 2010. ISBN 1934356379.

M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 9780131177055.

P. Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, Feb 2020. doi: 10.1145/3363824.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. ISBN 0131495054. URL `https://www.amazon.com/xUnit-Test-Patterns-Refactoring-Code/dp/0131495054?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0131495054`.

J. A. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. Addison-Wesley Professional, 2012. ISBN 0321803027.

Notes

[1] `https://xunitpatterns.com`
[2] `https://glossary.istqb.org/en/search`
[3] `https://themoviedb.org`
[4] `http://jmock.org/getting-started.html`
[5] `https://github.com/bblimke/webmock`
[6] `http://github.com/vcr`
[7] `https://github.com/thoughtbot/factory_bot_rails`
[8] `https://github.com/colszowka/simplecov`
[9] `https://saucelabs.com`
[10] `https://security.googleblog.com/2014/01/ffmpeg-and-thousand-fixes.html`
[11] `http://www.fastcompany.com/magazine/06/writestuff.html`
[12] `http://www.sqlite.org/testing.html`
[13] `http://rspec.info`